CR 115514

Final Report

Volume II.

Contract NAS 9-11778

ADVANCED SOFTWARE TECHNIQUES FOR DATA MANAGEMENT SYSTEMS

February 1972

INTERMETRICS

OFFICE OF PRIME RESPONSIBILITY

ED5

N72-21203
CR115514

Final Report

Volume II.

Contract NAS 9-11778

ADVANCED SOFTWARE TECHNIQUES FOR DATA MANAGEMENT SYSTEMS

February 1972

INTERMETRICS

OFFICE OF PRIME RESPONSIBILITY

ED5

Final Report

Volume II.

Contract NAS 9-11778

ADVANCED SOFTWARE TECHNIQUES FOR DATA MANAGEMENT SYSTEMS

February 1972

SPACE SHUTTLE FLIGHT EXECUTIVE SYSTEM: FUNCTIONAL DESIGN

Prepared by:

James T. Pepe

INTERMETRICS INCORPORATED · 701 CONCORD AVENUE · CAMBRIDGE, MASSACHUSETTS 02138 · (617) 661-1840

FOREWORD

This document is the final report on the functional design of a flight executive system for the Space Shuttle mission. The study was sponsored by the Manned Spacecraft Center, Houston, Texas, under Contract NAS-9-11778. It was performed by Intermetrics, Inc., Cambridge, Massachusetts, under the technical direction of Mr. Joseph A. Saponaro, to whom the author is indebted for his many helpful contributions to the design of this executive system and to the format of this report.

The study program covered the period from June 16, 1971 through February 16, 1972. The Technical Monitor for the Manned Spacecraft Center was Mr. Donald Barron.

The publication of this report does not constitute approval by the NASA of the findings or recommendations contained therein.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Scope

This document presents a top level functional design of a software executive system for the Space Shuttle avionics computer. The design task was accomplished as part of a study entitled Advanced Software Techniques for Data Management Systems. Three primary functions of the executive are emphasized in the design: task management, I/O management and configuration management.

The executive system organization is based on the applications software and configuration requirements established during the Phase B definition of the Space Shuttle program. Although the primary features of the executive system architecture were derived from Phase B requirements, it has been specified for implementation with the IBM 4 Pi EP aerospace computer and ultimately is expected to be incorporated into a breadboard data management computer system at NASA Manned Spacecraft Center's Information Systems Division. Accordingly, the executive system has been structured for internal operation on the IBM 4 Pi EP system with its external configuration and applications software assumed to be characteristic of the centralized quad-redundant avionics systems defined in Phase B.

1.2 Executive System Overview

The major areas of the executive system designed during the course of this study are briefly summarized below with the major characteristics defined.

1.2.1 Control Structure

The executive system is based on a combined synchronous/ asynchronous control structure with priority dispatching for processor allocation and task execution. Cyclic computations are operated at high priority in a synchronous mode under the supervision of a cyclic control executive function. It is initiated by a timer interrupt at a fixed frequency, currently 20 msec, with the scheduling and sequencing of each computation in a minor cycle predetermined and specified via control sequencing tables. The total running time of the synchronous mode or "foreground" is constrained to be at maximum less than a percentage of the minor cycle frequency, the percentage to be established during implementation. After completing the execution of the cyclic computations each minor cycle, the executive dispatches the processor to one of the "ready" tasks in the executive ready queue on the basis of priority. A total of three priority levels have been established for application programs.

1.2.2 Interrupt and Task Dispatching

All external interrupts within the configuration are fielded and serviced by the executive as in any real time system, allowing a multiprogrammed task environment in the background. The concept termed "segmented dispatching" is however employed for background tasks. That is, although interrupts are immediately serviced by the executive and entries are made in appropriate queues, the interrupted task is resumed and continued until it either ends or until it reaches a segment dispatch point. Only then is a higher priority background task activated by the executive dispatcher. In this way long duration tasks can be organized into reasonable execution segments with task swapping or interruption points more predictable. The dispatching of the cyclic task controller each minor cycle is however an exception and is executed immediately at the occurrence of the minor cycle clock interrupt. This exception is made as a reasonable tradeoff to provide the timing and response characteristics needed for cyclic computations ultimately assigned in the synchronous mode. This subject is discussed more fully in Chapter 2.

1.2.3 Task and Event Scheduling

Any executing task may request the executive to schedule another task on the occurrence of an event or a specified time. Events are system defined in scope and may be posted or deposted by application tasks via the executive.

1.2.4 Memory Organization and Allocation

All application software program modules are known to the executive via a program module directory. Programs are defined as either total mission resident or mission phase resident. Phase resident programs are loaded from the secondary storage device into their assigned portion of the operating memory by the phase initiation function of the executive. Dynamic memory is allocated to each task by the executive, when a task is made ready for execution, out of a subpool of working memory established for each priority level. Dynamic memory requirements are preestablished and defined for each program in the directory. Memory is allocated in continuous blocks within the priority pool and addressing is accomplished via base registers on the EP computer.

A portion of the memory is dedicated to shared data. The common memory pool, the compool, is organized into mission dependent resident data and an overlaid area for phase dependent data. The phase dependent shared memory is initialized with the program load at phase initiation and statically assigned during the phase. All access to the common data is controlled through and by the executive. The executive prevents conflicts in memory utilization by placing the conflicting task in a wait state until the memory is properly released by the task to which it is presently assigned.

1.2.5 I/O Control

Control and execution of all input and output operations are performed by the executive system. Input/output services are performed in two modes: on demand via request by an executing task, or table driven as in the case of cyclic computations in the synchronous mode. Secondary memory management is under the control of the executive. Limited use of the secondary storage device is assumed during any mission phase. The executive is responsible for the maintaining of tables of current status and communication paths to all redundant equipment within the system configuration.

1.2.6 Configuration Management Error Recovery

The executive responds to all system hardware and software detected error conditions and supervises reconfiguration of the system. A standard system error recovery action is defined for each error class. Application tasks may invoke during execution local recovery for a class via specification of a task re-entry point.

1.2.7 Executive Functions as a Summary

The specific functions that the executive performs within the scope of its design to insure the overall integrity and proper execution of application tasks are the following:

a) control allocation of the processor by scheduling and dispatching both periodic and nonperiodic tasks;

b) provide timing and event handling services to insure proper scheduling of tasks;

c) supervise and control all I/O operations;

d) allocate all resources to tasks and avoid conflicts; resources include dynamic memory, secondary storage and shared memory;

e) provide methods for controlling conflicts over shared memory;

f) maintain and update all system queues and tables;

g) provide the means of hardware error recovery and system reconfiguration;

h) provide linkage and common subroutines and executive services in application tasks via controlled simple interfaces.

1.3 Task Objectives and Approach

The executive system design task was accomplished in conjunction with other major tasks of the study. Its primary objectives were threefold:

a) review the Phase B avionics configuration and software requirements and identify major functions of the executive system;

b) analyze and determine key aspects of the executive structure such as: methods of task scheduling and control, external interrupt control techniques, task dispatching algorithms, allocation and sharing schemes, and application program interfaces;

c) develop functional logic and algorithm design for the task management, I/O management and configuration management modules of the executive system. The design is to incorporate definition of application program interfaces to the executive.

The approach taken in this task was based upon several constraints and necessary assumptions about the nature of the Space Shuttle mission.

1) The application software is not completely defined. Hence, specific parameters, such as the amount of dynamic memory needed, can not now be decided. This topic is again discussed in Chapter 8.

2) The software system we are developing is a kernel executive system for use in the Space Shuttle Data Management computer. It is not an operating system for a ground based system.

3) The breadboard data management computer system at NASA Manned Spacecraft Center's Information Systems Division is not at present completely specified. Thus, several assumptions concerning the design are made and pointed out in later chapters.

4) The executive features incorporated in this design are those deemed necessary to execute the application software as far as it is defined in the Phase B Study Reports [1,2].

Page Intentionally Left Blank

Chapter 2

Executive Design Functional Requirements

2.1 Introduction

The fundamental features of an executive system must be based on the requirements of its environment and the application software it controls. Ideally, it should be efficiently tailored to meet the design objectives and operating environment of the total system. Prior to discussing the design chosen, the purpose of this chapter is to review major system requirements impacting on executive system architecture. These topics include: aspects of the avionics system configuration and applications software, and the organization of the host computer system. Finally, several key issues relative to the selection of a particular executive system structure (as it influences task control, resource allocation and interrupt handling) are discussed with respect to the appropriate design considerations.

2.2 Space Shuttle Avionics System

2.2.1 Configuration

The Phase B Space Shuttle avionics systems have been reviewed and are discussed in Volume 1 of this study. Although more than one Phase B design was reviewed, a hypothetical system configuration is briefly described incorporating the important features of the designs to the software executive.

The avionics configuration assumed consists of a centralized data management computer system interfaced to all avionics subsystems via a high speed time multiplexed serial data bus system as illustrated in Figure 2.1. The data management computer system consists of quad redundant computers which operate in a simplex redundant mode.

During critical phases of the mission more than one computer is operating with one of them designated as the prime computer. The prime computer transmits and receives all commands and

and data over the data bus to the avionics subsystems. The standby computers are synchronized with the prime computer via external control and execute the identical software. Outputs from the prime computer are monitored by the standby computers and compared via hardware by its bus control unit in lieu of transmission. The results of the comparison are sent to external control unit and crew operator personnel for voting and switching.

The data bus system consists of a bus control unit (BCU), 4 bus lines and remote interface units (IU) for equipment connection. The BCU functions as a peripheral under command from the computer and controls the transmission of information over the bus. It communicates with the IU which in turn acquires, converts and sends data to and from the subsystems. The bus system operates in a "command response" mode in which data is sent only when requested by the central computer. The operation and control of the bus is described more fully in Appendix A. There is no provision for interrupts from the subsystems. Each bus line carries serial digital data at 1 MBPS. The bus system is quad redundant with each BCU capable of transmitting on each of the four buses; however, each computer interfaces with only one BCU. Redundant subsystems are interfaced to physically separate bus lines via the interface units. The computer system is also interfaced to redundant secondary storage units. These units contain additional programs and data tables for various mission phases. For the purpose of executive design it will be assumed to have limited use during a phase with restricted write access. Also for purposes of executive design, it will be assumed that other external units may be interfaced to the computer directly and not via the data bus such as display and control subsystems.

2.2.2 Application Software

The total onboard software has been estimated (during Phase B) at requiring approximately 50,000 32 bit words of operating memory and a peak rate speed of approximately 200,000 equivalent adds operations per second. For purposes of this discussion the total flight software for the Space Shuttle central computer system may be broadly classified into two areas: the executive and mission applications software. The application software is under the control of the executive and supports all phases of the mission: boost, insertion, orbital operations, coast and powered flight, rendezvous, docking, undocking, entry and landing. The applications software to support these phases comprises the following functional areas:

a) flight control and stabilization

b) guidance

c) navigation

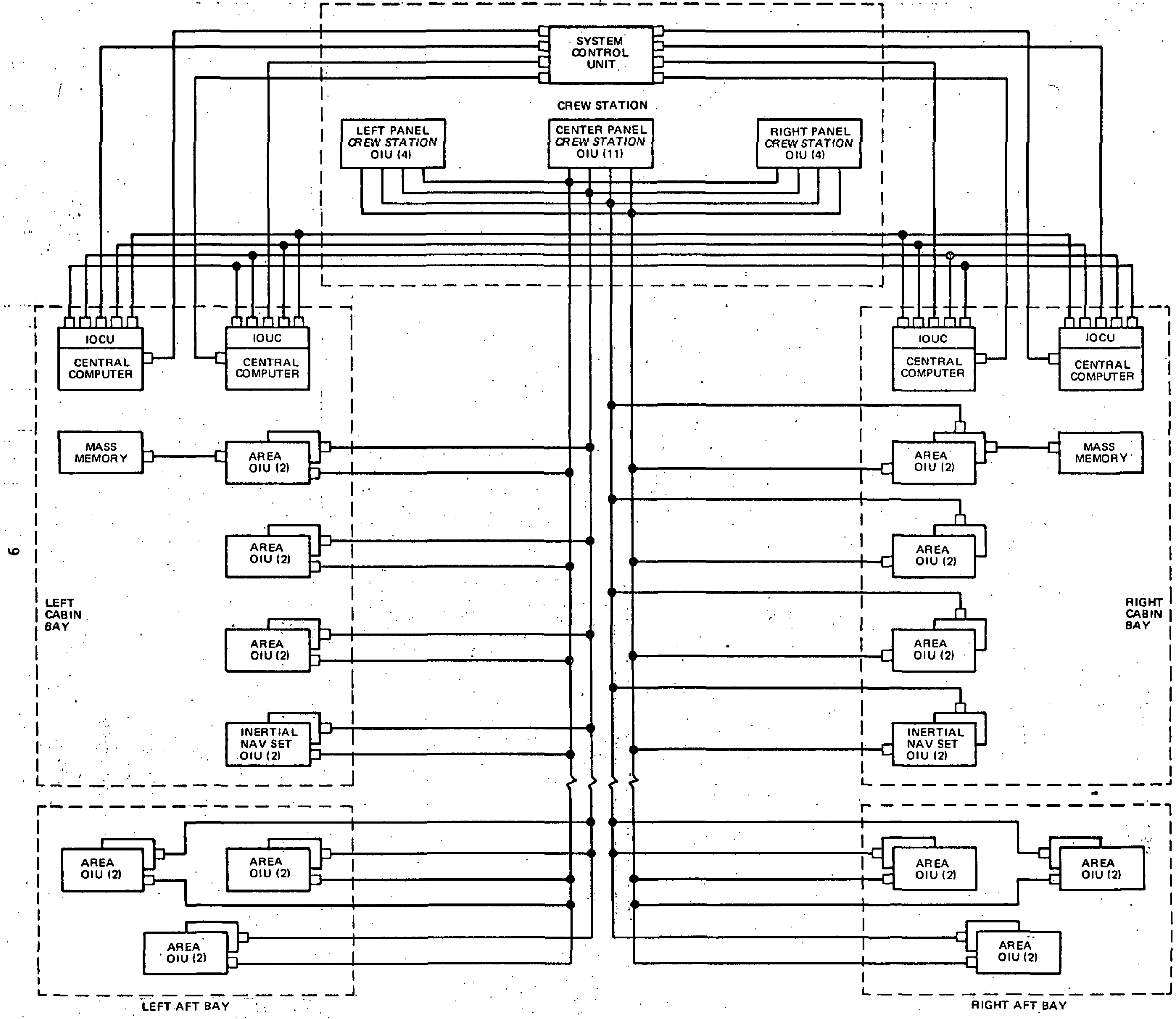


Figure 2.1 Avionics Computer Configuration

d) trajectory targeting

e) crew displays and control

f) onboard checkout and systems monitoring

g) avionics subsystem management and support.

Estimates of size, data requirements and frequency of operation of this software have been estimated during Phase B. The flight control and stabilization function place the highest respond time demands on the system and have been estimated at basic frequency of approximately 20 msec. Subsystem and status monitoring rates are significantly less at 1 sample/sec being the average although the number of such samples and processing loads are greater. Targeting, navigation and guidance schemes are characteristic of more lengthy, iterative mathematical calculations, requiring large CPU utilization.

The full impact of crew interaction via display and control is not completely determined. It is evident however, they will require the capability to interact through the display to: load programs and data, select major program modes for execution, terminate execution, request displays, select control options, configure and reconfigure equipment, and monitor the status of avionics subsystems. The crew will also interact with the computer through other controls such as the rotational hand controller when flying under pilot control.

These requirements indicate the Shuttle software environment to include three types of tasks:

a) cyclic tasks: Tasks which are performed on a periodic basis at varying frequencies.

b) response/request tasks. These are tasks which are performed in response to a pre-selected mode such as the rendezvous mission mode. Generally these tasks are major sequences or functions initiated throughout the mission by the crew.

c) demand tasks: These are tasks which must be performed at the occurrence of a system event or certain time.

2.2.3 Data Rates

In the Phase B avionics design concept, the data bus system provides a communication path between the avionics equipment and the prime computer complex. No general requirement for terminal to terminal communication which cannot, or should not be routed through the computer complex was identified. The exact number and type of subsystems has been continually changing. A representative list provided below is presented to indicate the scope of the system.

1) Primary propulsion subsystem: this system consists of two orbital insertion engines and one orbital maneuvering engine.

2) Reaction control subsystem: at least 20 RCS jets located in the nose, wings and tail for effecting rotation and translation in space.

3) Hydraulic system: hydraulic power generation, distribution, control, and conversion of mechanical energy. It consists of supply lines, gimbals, pumps, aerodynamic surfaces, flaps, wheel controls, etc.

4) Electrical power generation and distribution system: fuel cells and battery, and the auxiliary power units located throughout the Shuttle.

5) Navigation aids/air data: a collection of equipment providing navigation and landing capabilities (ALS, radar altimeter, TACAN, DME, etc.).

6) Environmental control system: the environmental control system provides temperature, pressure, and humidity control of equipment, equipment bays, and personnel compartments.

7) Cryogenic system: contains the hydrogen and oxygen for the primary propulsion, the reaction control system, the fuel cells and the auxiliary power units.

8) Displays and controls: this system is assumed to have local processing capability and accepts dynamic data through the bus for updating of display parameters.

9) Telecommunication: this system consists of various transmitters and receivers including S-band, C-band, VHF, telemetry encoder, EVA communications, air traffic control communications, etc.

10) Guidance, navigation and control: this subsystem is composed of elements necessary to control, stabilize and navigate the Shuttle vehicle during all phases of the mission. It interfaces to the reaction control system, jet engines, aerodynamic control surfaces, and landing gear, etc. It has access to sensors which include the

inertial subsystem, horizon and star trackers, approach landing aids, rendezvous radar, radar altimeter, etc.

Although this list of subsystems may not be complete for the final organization of the avionics system it is meant to be representative. It is estimated that approximately 150 to 250 LRU's are associated with the subsystems listed above.

2.2.4 Data Requirements

The following is a summary of the data requirements abstracted from the various studies of Phase B contractors.

1) Speed. Peak load estimates of data rate for both the Shuttle and orbiter have ranged between 100,000 and 250,000 bits per second, including overhead. Considering an average overhead of approximately 50% for each bus transaction and allowing for a minimum of 100% expansion to the maximum speed, a capability of 10^6 bits per second has been assumed to be an adequate requirement. This speed should allow the computer to acquire data at a rate of approximately 10,000 average transactions per second.

2) Measurements. Estimates have ranged between 4000 and 6500 unique data points to be sampled from the total complement of avionics equipment by the central computer. Data types include:

   digital parallel
   digital serial
   analog
   discrete

   The majority of these data points are measurements input to the computer, and are estimated at approximately 60% to 70% of the traffic on the data bus.

3) Response time/sampling frequency. The maximum sampling frequency of measurements is estimated at fifty samples per second. The average sampling frequency for status information is between two and five samples per second. Very little information was made available on response requirements and load distribution of subsystems.

2.3 Features of the IBM 4 Pi EP Computer System

The hardware features of the computer can directly influence the executive system software design. In this section the most pertinent features of the IBM 4 Pi EP computer assumed in the executive design are presented for review. 4 Pi EP hardware is documented in detail in the IBM Programming Manual for System 4 Pi Model EP [3].

2.3.1 Computer Organization

The EP is a byte addressable computer with two bytes constituting a half word, four bytes a full word, and eight bytes a double word. The EP memory size for the computer in the ISD breadboard is assumed at 24K 32 bit words. An additional 16K multiport buffer memory may be incorporated; yet its status is unknown at this time.

There are 16 general registers (GR) of full word size used for high speed fixed point and logical operations and four floating point registers of (FPR) of double word size used for floating point operations.

The instructions are organized into four classes: register to register (RR), register to indexed storage (RX), register to storage (RS), and storage and immediate operand (SI). A complete list of all instructions may be found in reference [3].

All addressing of core storage within instructions is done relative to a base address stored in one of the general registers, designated the base register. Many instructions' address fields can reference up to 4K bytes beyond a base address by adding a 12 bit displacement to the contents of a base register. RX instructions further extend this addressing capability by also allowing indexed addressing.

2.3.2 Interrupts

There are five classes of interrupts in the EP.

a) I/O interruptions allow the CPU to respond to conditions in the channels and I/O units.

b) Program interruptions signal unusual conditions encountered in a program, e.g., incorrect operands and operand specifications. This class of interrupt may be subdivided into nine subclasses identified by the interruption code generated by the EP. The subclasses are:

1) Operation Exception: operation code unassigned

2) Privileged-Operation Exception: a privileged operation is encountered in the problem state

3) Specification Exception: incorrect operand specification

4) Fixed Point Overflow Exception

5) Fixed Point Divide Exception

6) Exponent Overflow Exception

7) Exponent Underflow Exception

8) Significance Exception: the result of a floating point add or subtract has an all zero fraction

9) Floating Point Divide Exception

c) Supervisor call interruptions result from the execution of a SVC opcode. This interrupt is used to switch from the problem state to the supervisor state in which privileged instructions can be executed.

d) External interruptions allow the CPU to respond to signals from the interruption key on the system control panel and the timer. The timer is a full word in main storage location 80. An external interrupt is generated when the value of the timer goes from positive to negative. A timer is essential to the executive system. The exact details of the timer in the breadboard are not known as of this time.

e) The machine check interruption occurs when a hardware error is encountered. A diagnostic procedure is automatically initiated.

Should several interrupts occur simultaneously they are honored in the following order:

1) machine check;

2) program or supervisor call (mutually exclusive interrupts);

3) external;

4) I/O.

Each of the five interrupts described above has two related program status words (PSW) associated with them in unique main storage locations (see Figure 2.2). An interrupt causes the current PSW to be stored in the "old" position and the PSW in the "new" position to become the current PSW. The old PSW contains all the information necessary to resume the problem program again at the point of interruption, and the new PSW allows executing a routine associated with the interrupt.

As mentioned above the supervisor state (as distinct from the problem state) allows a class of privileged instructions to be executed. The executive uses these instructions to maintain the integrity of the system. Examples of privileged instructions include direct I/O operations, setting system masks, and setting PSWs. To prevent their use by application tasks a program interruption is generated when a privileged instruction is encountered in the problem state.

The supervisor state can also be used to protect the executive from invalid access by application tasks. Hence, SVC operations provide the means for application tasks to correctly use the executive, and they help insure that an application task does not alter the executive.

2.3.3 4 Pi Input/Output Via Standard Channel

Another important EP hardware feature is the structure of the I/O control system. Since the structure of I/O operations depends heavily upon the channel control structure and its operation, I/O management will be one of the most configuration sensitive areas of the executive software. Hence, a clear understanding of the EP's I/O system is necessary.

All I/O operations are initiated by a START I/O instruction. If the channel is free, this instruction is executed, and the CPU continues processing its program. Then the channel, independent of the CPU, selects the I/O device the instruction specifies.

START I/O causes the channel to fetch a channel address word (CAW) from main storage location 72. This word points to the main storage location where the channel program begins. The channel program is a series of chained channel command words (CCW), each of which contains a command code to the channel as well as main memory data addresses and byte counts. See Figures 2.3 and 2.4 for the CAW and CCW formats.

Should an I/O command be rejected during execution of a START I/O (by a program check, busy condition, etc.), the command rejection is indicated in the PSW. The details of the conditions that prevented I/O initiation are given in the channel status word (CSW) which is stored in main storage location 64 when the command is rejected (see Figure 2.5). The CSW is formed or reformed by START I/O, TEST I/O, or an I/O interruption. This word contains information about the termination of an I/O instruction. An error recovery program that is initiated because of an I/O error will depend heavily upon the CSW to determine the cause of the error and whether a system reconfiguration is necessary.

System Mask	Key	AMWP	Interruption Code
ILC	CC	Program Mask	Instruction Address

Figure 2.2 Program Status Word Format

Key	0000	Command Address

Figure 2.3 Channel Address Word Format

<table>
<tr><td colspan="2">Command Code</td><td colspan="2">Data Address</td></tr>
<tr><td>Flags</td><td>0000</td><td>▨</td><td>Count</td></tr>
</table>

Figure 2.4 Channel Command Word Format

<table>
<tr><td>Key</td><td>0000</td><td colspan="2">Command Address</td></tr>
<tr><td colspan="3">Status</td><td>Count</td></tr>
</table>

Figure 2.5 Channel Status Word Format

2.3.4 4 Pi EP Data Bus Input/Output

The 4 Pi computer in the DMS breadboard (Figure 2.6) will be interfaced to a bus system via a stored program data processor (SPDP). The details of this interface and method of operation are currently not known. Accordingly, by direction, the executive design has been based upon interfacing to a Phase B type of bus system described previously. It is anticipated that the functional organization of executive I/O management will remain the same.

2.4 Executive Design Issues

In conjunction with the review of the avionics system requirements, several factors of the basic executive system structure were evaluated. The purpose of this and succeeding sections is to discuss these issues.

Prior to performing the analysis several design goals were established to be used as guidance in selecting an ultimate design approach. Primary considerations of the executive structure analyzed are:

a) synchronous versus asynchronous task control;

b) interrupt handling and task dispatching;

c) resource allocation;

d) shared data;

e) secondary storage management.

The primary objective or goal usually adapted by most executive system designers is the achievement of an "efficient" executive where efficiency is some measure of throughput. Efficiency may be defined by either the fraction of executive overhead time spent doing nonproductive work or in terms of response time. In performing analyses of these issues, efficiency was considered a necessary but not primary factor since it often tends to lead towards complex design resulting in complex testing and verification of software. Ideally, flight software should not only be tailored to meet operational mission requirements but should be structured to enhance software verification and flexibility to adjust to changing needs. Therefore, the following design criteria were used as evaluation of the executive structure.

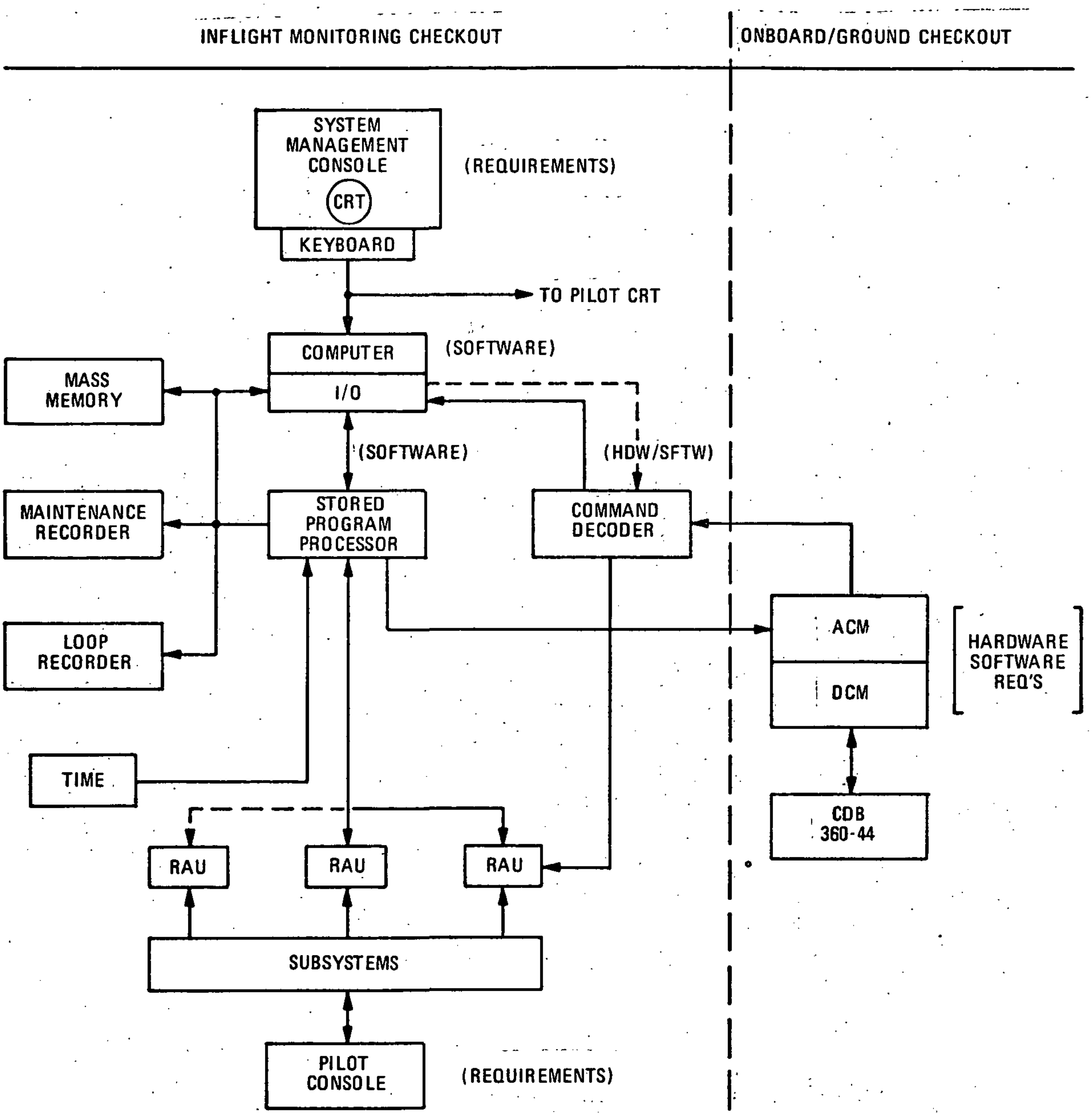


Figure 2.6 ISD Breadboard Data Management System

a) To provide an executive system which will control and allocate resources of the system to satify operational mission requirements (i.e., one that does the job).

b) To establish an executive organization which facilitates verification of application software and reliability of code.

c) To structure an executive enabling flexibility and modularity in incorporation of application software changes over long term maintenance periods.

d) To define simple and well defined application program interfaces to the executive system. It should be structured as a virtual machine to the applications programmer.

e) To develop an executive structure which is both simple and efficient but consistent with other objectives.

2.5 Synchronous versus Asynchronous Task Control

A primary function of the task management portion of the executive is the scheduling, dispatching and control of the allocation of processor to task in the job stream. It is a fundamental feature of the executive system. Most large ground based computer systems incorporate very flexible and general task scheduling and dispatching algorithms to accommodate a varied number and type of users. The Shuttle software on the other hand, is more tailored to its environment. Although Phase B contractors have specified synchronous structured executives, shuttle software requirements do not allow task scheduling to be completely planned in advance. Furthermore, it is our contention that a pure synchronous structure would ultimately be modified to accommodate priority based event handling since it is necessary as a Shuttle software feature. We have chosen a design which accommodates the best features from each control structure. The following presents the advantages and disadvantages of synchronous versus asynchronous control.

2.5.1 Synchronous Structure

A synchronous structured executive is based on a timer-interrupt, fixed schedule, time slice mode of operation. For example, Shuttle baseline designs use a 20 msec interval as a basic reference frame for the system, providing a minor cycle sampling rate of 50 cps. Under this concept jobs are organized by the designer into short routines, and when the executive detects a timer interrupt (i.e., every 20 msec) it examines the "task schedule tables" to determine which set of routines is to be operated during the next program interval. Each 20 msec interval contains all 50/sec tasks and a selection of other lower frequency tasks. The minor cycle is operated every 20 msec, and a percentage of that time is distributed among the tasks that are assigned to each minor cycle. A background job may be run in the slack time before the next minor cycle. Each task is statically structured as a subroutine such that it can be dynamically called and returned to the executive.

Using a command response data bus control concept, scheduling I/O in a synchronous structure is similar to the scheduling of tasks. The I/O requirements for each mission phase or major cycle are predetermined and synchronized with the structure of tasks operated in the major cycle. The I/O request list is assumed to be fixed. Since the I/O requirements will have different frequencies, they are incorporated in each minor cycle in correspondence to load balancing of the processing tasks.

For example, assume all I/O requirements for a particular mission phase are organized into three categories of frequencies: 50 times/sec, 5/sec, and 1/sec. Assume that X, Y, and Z are the number of commands in each category. Assume further that a minor cycle occurs every 20 ms and that a BCU is commanded with a list of I/O requests each minor cycle. The averaged number of I/O operations required to be scheduled each minor cycle are: all of the 50/sec requests, 1/10 of the 5/sec requests, and 1/50 of the 1/sec signals. In a synchronous structure tables of predetermined I/O requests are organized according to sampling frequencies. The appropriate number of I/O entries to command each minor cycle are selected from these tables. The synchronized concept attempts to avoid non-deterministic behavior of I/O, I/O queues, and I/O backlog.

Several types of I/O activity cannot be determined in advance; for example, the command of jets on and off. The I/O scheduler may accomplish this by providing a place for the command in the appropriate list and then causing the BCU to skip the command or incorporate it, depending on the results of the stabilization and control tasks.

2.5.2 Example of a Synchronous Executive

For purposes of illustration the basic functions performed by a synchronously controlled executive include:

a) managing data bus I/O by issuing all I/O requests for the minor cycle;

b) managing task execution by executing all high frequency tasks; deciding what tasks executed at a less frequent rate must now be done and executing these; and doing background and/or housekeeping functions in any slack time.

To enable the executive to perform these tasks with the least amount of overhead, judiciously organized system tables must be used. A description of the contents of the types of tables is presented below.

A cyclic command table (CCT) lists all tasks and the frequency they are to be done in a given mission phase. In other mission phases a different table is used, which can be stored on a mass storage device until it is needed. For example, a typical entry would be

frequency	program module ID	core address

It must also contain pointers to the I/O requests for every minor cycle. That is, in a particular minor cycle all I/O requests are known in advance since a synchronous structure is deterministic. Thus, the executive can issue all I/O requests at once. For example, consider the following CCT entries:

Frequency	Module	Address
every minor cycle	A B C	1000 2000 3000
every other minor cycle	D E	4000 5000
every four minor cycles	F G H	6000 7000 10000

The order of execution of these program modules every four minor cycles would be the following

Minor Cycles	Modules
N	A B C
N+1	A B C D E
N+2	A B C
N+3	A B C D E F G H

Should D take an abnormal exit during the (N+1)st minor cycle, and deschedule E, the order then becomes:

Minor Cycles	Modules
N	A B C
N+1	A B C D
N+2	A B C
N+3	A B C D E F G H

A flowchart of a synchronous executive structure is presented in Chapter 4.

2.5.3. Advantages of Synchronous Structure

a) There is minimal overhead for scheduling and dispatching because all tasks are known to the system in advance, and hence, are prescheduled. The executive knows which fixed set of code to execute in each time slice.

b) The executive design is simple and thus easy to program.

c) The system is not multiprogrammed so no queues of ready and waiting tasks have to be maintained. In other words, more than one task is never in contention at any time for the processor. One fixed set of code is executed in each time slice. Memory conflict problems are also eliminated since core and word areas for all programmers are pre-allocated.

d) The system is deterministic which makes the task of software verification easier. A programmer must divide a long program into segments to evenly distribute over several time slices. The break points can occur at places at which he knows no interrupting program can interfere with his program or data.

e) The computational and I/O load will be balanced over a major loop. Thus, no degraded response can occur because of computational or I/O overload. Response is predictable.

f) The predictability of the system eliminates sharing problems. Programs can be put together in time slices so that no data sharing problems result. This fact eliminates the need for a central update routine for data. Also, the need for reentrant coding, and hence, dynamic storage allocation, is eliminated.

2.5.4 Disadvantages of Synchronous Structure

a) Application programming is more difficult especially for long programs. The programmer must break such a program into segments so that between segments any running program cannot interfere with his program or data. Also, fitting his program segments into time slices with other program segments is difficult. Timing requirements of each segment must be known before these can be fitted together in a time slice. Thus, the programmer has a second constraint, namely, time bounding his segments.

b) Changing application programs or mission programming requirements can be a major redesign. Such a change can require rebalancing of the entire computational load. New requirements can mean having to spread the existing application programs more thinly over the time slices of a major loop, so that the new programs can also be fit. That is, each existing program segment might be restricted to a smaller time bound, and hence, reprogramming will result.

c) Each time slice must accommodate the worst case computational requirement. For example, if the crew is provided the option to display a parameter during a particular mission phase, then the calculation of that parameter will have to be incorporated into the sequence whether or not the crew ever requests it.

This situation is particularly bad if more tasks are added to the system. If in the worst case 80% of the computer's time is being used, a task having a worst case requirement of 25% cannot be accommodated. If it were accommodated, some time slice would have a worst case requirement of over 100%. This situation is unacceptable in a synchronous structure.

d) A synchronous structure does not allow tasks to be run on a time or event basis. In particular, this type of fixed-sequence executive organization does not provide a structure which allows for external interaction by the crew, or which copes with a random job stream. Jobs must be predetermined and assigned to slots in a sequence and must operate within the basic reference framework. It is not clear at this point whether all Shuttle requirements can be so predetermined.

Both Phase B executive designs allow tasks to be scheduled on an event basis. That is, when an event occurs a task can then be scheduled. A scheduler is used to fit the newly scheduled task into the time slices and to deschedule lower priority tasks when necessary. Such an executive cannot be fully synchronous, as defined and described above.

2.5.5 Asynchronous Structure

In an asynchronous control structure scheduling and allocation of the processor are accomplished in real time according to the needs of the operating environment. Under this concept processing tasks are assigned a priority which establishes their relative importance to each other. A task with a given priority runs until a wait is encountered, or the existence of a higher priority task is established.

The distinction between synchronous and asynchronous control structure can be illustrated by the "states" in which a task will exist while operating under each structure. In a synchronous structure, tasks are in one of two states: actively running or not running. At any instant of time only one task is in the running state and all others are not running. The transition to the running state occurs when a task's scheduled time slot arrives.

In an asynchronous structure, a task, while present in the system, will exist in one of four states: running, waiting, ready, or inactive. The executive insures the proper transition of states depending upon either internal or external stimuli. The running state definition is obvious. Note that the running state can only be entered from the ready to run state. This unifies the dispatcher functions. The waiting state is either a voluntary or involuntary state, depending upon its cause. A voluntary wait would be a wait for completion of I/O, or perhaps some external time stimulus. An involuntary wait would be awaiting resources (e.g., memory) to become available. The inactive state occurs when the task is neither running, waiting, or ready.

The ready state can be entered from all other states and indicates that a job has all the facilities available to it to run. The function of the dispatcher is to pick the most appropriate task from the ready queue and start it running.

State changes from wait to ready would occur when the awaited stimulus has occurred. The change from limbo to ready state occurs when a schedule request is issued by some task. The switch from running to ready occurs when a task is preempted by a higher priority task or interrupt.

In summary, an asynchronous structure is one in which one or more tasks may be in the ready state awaiting allocation of the processor. In a simplex computer system this is termed multprogramming, i.e., the concurrent operation of more than one task.

An overview of the operation of a general asynchronous executive is illustrated in Figure 2.7. The scheduler and dispatcher, once in control, should be able to pick a task and run with it. The scheduler assigns or reassigns task priorities, verifies that all the task resources are available, and maintains the overall view of real time events. All task starting is done through the dispatcher.

The scheduling function in a broad sense consists of making appropriate entries in task blocks and priority queues so that the dispatcher need only select jobs from the top of the ready list. If there is a number of tasks to be scheduled, the scheduler treats some as more important than others and executes them first. If the dispatch function occurs at some time other than at the end of a program, then a multiprogrammed environment is a direct result.

The interrupt handler "posts" the event complete, makes the task ready if possible, and then passes control to the scheduler to act on the information it has provided.

The resource allocator is invoked as an executive function by the scheduler to test readiness to run, and if not ready, will inform the scheduler of the requirements for readiness. It may also be invoked to test availability of contention items.

I/O in an asynchronous structure is generally scheduled on a demand basis. An active task requiring I/O schedules its request via an I/O queue. The task is placed into the wait state until completion of the I/O request. The I/O control routines operate on the I/O queue and interface the I/O peripheral (i.e., the bus system) to perform the request. I/O is performed asynchronously with other processing tasks in the system. After acknowledging receipt, initiation or completion of the I/O request, the scheduler is informed via a simulated or actual interrupt. The task awaiting the I/O request is then placed into the ready state and awaits processor assignment. However, demand scheduling may not be easily implemented in the Shuttle software due to the high speed of the BCU as a peripheral and the intended block scheduling.

2.5.6 Advantages of Asynchronous Structure

a) It is able to adapt to a random job stream; i.e., it does not require rebalancing of a computational load, and it can tolerate periodic overload and backlog since it is designed to cope with this problem. Time and event scheduling can easily be accommodated.

b) It is more adaptive to a real time environment. If a task of high priority must be scheduled, it is not necessary to deschedule a lower priority task. The task dispatcher selects this high priority task for execution while lower priority tasks remain in the ready state. Lower overhead results.

c) Application programming is easier. An asynchronous structure does not require long program sequences such as targeting, etc. to be arbitrarily organized into fixed segments to fit in some fixed cycle or sequence.

d) Since it is able to adapt to a random job stream, interface with the crew is easier. If the crew schedules a program of high priority, they can be sure this program will not be spread out over small portions of many time slices but will be executed quickly.

e) It has a greater flexibility for incorporating changes than the fixed sequence approach. A change in mission requirements is not a major programming change for existing programs.

2.5.7 Disadvantages of Asynchronous Structure

a) The multiprograming environment resulting from this type of scheduling is more complex and difficult to test and verify. Programmers no longer know where their programs will be interrupted. Thus, the executive must guarantee data integrity, handle sharing of data, and allow for reentrant coding. It can be made more predictable, however, particularly on the Shuttle where no external interrupts exist.

b) Since all tasks are run through the scheduler and dispatcher, there is an increased overhead for running these programs, queueing ready and waiting tasks, and handling the queues. However, this overhead can be minimized by combining the features of synchronous and as will be explained later.

2.5.8 Need for Asynchronous Features

Since the nature of the Space Shuttle mission requires the computer to respond to unpredictable events, such as the crew altering the job stream, handling emergency situations, reconfiguring because of failed equipment, etc., a fully synchronous executive is insufficient. As mentioned above, both contractors see the need for scheduling tasks on an event basis. Since this fact is a step toward asynchronous structure, the question arises to what degree the executive organization should be asynchronous. Because of the simplicity of a synchronous structure, as many of its advantages as possible must be kept. It is the disadvantages that must be eliminated by allowing some asynchronous features.

The following structure obtains the best features of both. Tasks will be organized into foreground and background categories. The foreground tasks are those tasks run at a fixed frequency by the scheduler in a synchronous manner, as described above. The time needed to execute each of these tasks must be small, i.e., less than a minor cycle. By definition all foreground tasks (synchronously) scheduled in a minor cycle must be totally executed in that cycle. The remainder of the time of the minor cycle can be devoted to executing background tasks. Background tasks have several features. They can be operated on a priority basis; they can be long (i.e., require more than one minor cycle to execute); and they can be scheduled on a time or event basis. The nature of background tasks makes a queue structure necessary. Hence, in a minor cycle there are now two types of scheduling: synchronous and asynchronous. By making as many tasks foreground as possible we eliminate much overhead in scheduling/dispatching background tasks.

The advantages of this structure are:

a) it eases the incorporation of event and time dependent rescheduling of tasks;

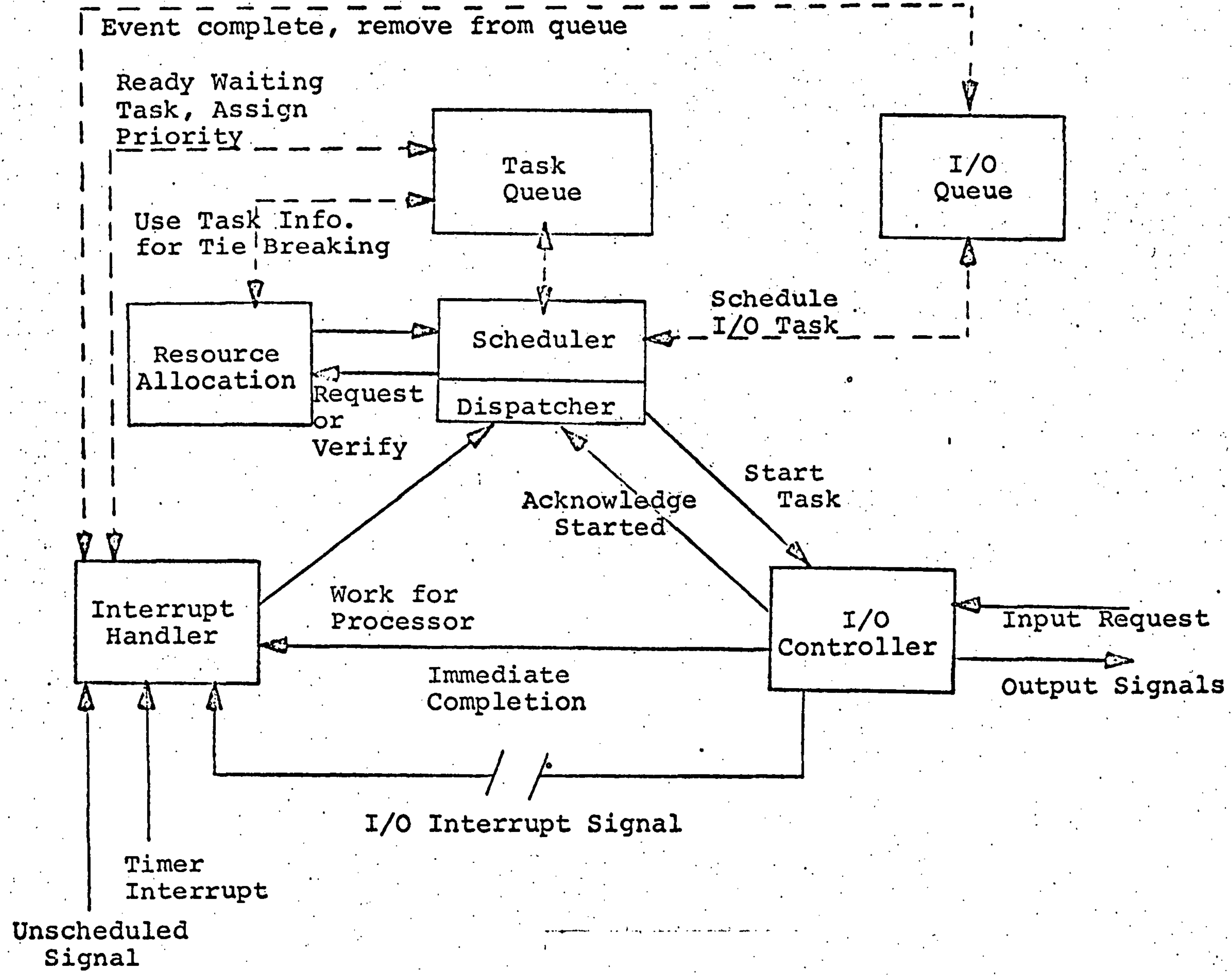


Figure 2.7 System Flow of General Asynchronous Structured Executive

b) asynchronous structure data integrity problem avoided, i.e., programmer control of where interrupts will be serviced;

c) tasks for which precise timing anaylysis is unnecessary or impossible, or which require extreme timing safety factors can be executed on a priority basis rather than on a time slice basis.

Thus, in general, the proposed structure can handle the Shuttle software in a way advantageous to either completely synchronous or completely asynchronous structure.

2.6 Interrupt Handling and Task Dispatching

The interruption of a running program in response to an external signal was introduced into the computer technology to serve two purposes:

a) provide rapid response-time to asynchronous events;

b) eliminate the necessity of polling (and its overhead) to discover whether an awaited event has yet occurred.

In single-processor systems, particularly dedicated systems where most or all of the computation is devoted to a single application, the introduction of interrupt-mode computation raises the hazards associated with multiprocessing: at arbitrary times, an interruption can introduce what appears to be a parallel task which is at least conceivably capable of disrupting the progress of the interrupted task by altering its variables. Thus, methods for masking or inhibiting interruptions were added, and the nature of the functions allowed in interrupt-mode was restricted. Properly and thoroughly applied, these fixes allowed programs to perform properly, although no truly thorough method has been found of proving that the system was actually properly programmed.

There exist therefore, two relevant negative aspects of interruptions: timing response uncertainties, and potential data disruption and conflict. Both can be minimized by causing interrupts to <u>schedule</u> tasks whenever possible, as opposed to <u>performing</u> them. This provision reduces the multiplicity of possible timing situations, since job swapping occurs only at specified intervals.

Accordingly, it is considered desirable to utilize hardware interrupts such that tasks are scheduled and the interrupted task is rapidly resumed. The primary consideration becomes when to dispatch a higher priority task resulting from an interrupt, such that respond time requirements can be satisfied.

When an active task is dispatched into the wait state, another higher priority task is dispatched (made active) from the ready queue. When else does the executive dispatch? The following summarizes various approaches considered.

a) If the executive dispatches at no other time, system response time to high priority tasks cannot be guaranteed since long duration tasks would execute to their end. This appears unacceptable in the Space Shuttle mission unless all lengthy tasks were broken down into separate, sufficiently short, independent tasks.

b) The executive can dispatch whenever a task of higher priority than the active task is scheduled. In this case, interruption of the active task will occur at a random point in the coding and a higher priority task given the CPU. This uncertainty can lead to a program verification problem due to its random nature and non-repeatability.

c) Alternatively, a programmer can inhibit dispatching at dangerous points in his program. Tasks of higher priority would be dispatched when permitted. However, this method does not completely solve the verification problem or prevent a higher priority task being delayed from execution for an unacceptable amount of time. By introducing an onboard "watchdog" timer, it is possible to guarantee a maximum time in which dispatching is inhibited. If a programmer exceeds this maximum time in inhibiting dispatching, the CPU is taken from his program. However, the dispatch will now occur at a random point.

d) Another approach is to require the application task to be organized into short segments in which the dispatcher is requested at the end of each segment. If these segments were fixed at short intervals it would enable system response time to be maintained.

Furthermore, the segment organization of a lengthy program provides visible and controllable evidence to the programmer of the possible points that alternate control paths can occur. Conversely, he is assured that once the segment begins it is non-interruptable until it ends other than by the executive servicing of an interrupt or the task placing itself into a wait state. Similar arguments could be used for the previous approach.

The chosen method involves a modification to approach (d). First, high priority cyclic tasks, operating in a synchronous mode in the foreground, will always be dispatched at the occurrence of the clock interrupt. All other tasks will only be dispatched at the segment points. This will guarantee response time where it is needed and loosen the requirement for segment operating limits.

Secondly, the establishment of segments for lengthy programs can be aided by an assembler or compiler. Given that a procedure oriented higher order language is used for application programming, it can often suggest segment points and make them visible to the programmer. Tentative examples of compiler based segment points are:

a) on all forward GO TO statements;

b) entry or exit from a block;

c) maximum time allowed in a segment exceeded.

The programmer must have a compiler override capability.

2.7 Resource Allocation

A resource may be defined as a facility of a computing system that can be temporarily assigned to tasks to enable them to perform their computations. Examples of resources pertinent to the Shuttle software are core storage, shared data, and data sets on mass memory units. Resource allocation is that function of a computer's operating system that assigns resources, when possible, to the tasks requesting them. In a multiprogrammed system, several tasks can request the exclusive use of a single resource. Since only one task at a given time can be granted its request, the others must wait until these resources are freed. Care must be exercised in resource allocation to minimize the number of transitions of a task from the active to the wait state and to avoid allocation conflicts.

To be specific, several conflicts can result from inefficient resource allocation. These are:

a) deadlock,

b) memory fragmentation,

c) priority conflict..

We will define each of these conditions in the following paragraphs.

2.7.1 Deadlock

Deadlock is a condition in which two (or more) tasks are each waiting for a resource held by the other before either can proceed. Neither task can release the resource it holds, so neither can be taken out of the wait state [16]. For example, suppose task A holds resource R1 and needs R2, but task B holds R2 and needs R1. Since neither task can release its resource, neither can proceed and deadlock results.

Deadlock detection algorithms can be included in an operating system to enable the task performing resource allocation to recognize potentially hazardous situations, and hence, to avoid them. This topic has been discussed extensively by several authors [9-10,13-16]. However, such an algorithm can cost a high overhead in execution time. The Space Shuttle executive should have an alternate way of avoiding deadlock.

Deadlock is the result of incremental resource allocation. That is, it is the result of tasks requesting resources sequentially during execution. By avoiding incremental allocation we can avoid deadlock without costly detection algorithms. More will be said about this topic later as it relates to the Space Shuttle computer.

2.7.2 Memory Fragmentation

Memory fragmentation is a condition in which a task cannot be granted its request for a large block of contiguous core because all available core for dynamic allocation is in small noncontiguous blocks.

When this situation arises in a large ground based computing system having a large secondary memory, part of the contents of core are rolled out temporarily to create a large enough contiguous area of main memory to satisfy dynamic allocation requests. However, on the Space Shuttle computer we seek to minimize the use of any MMU because of its inherent complexity. Thus, most data will be maintained in main memory so that programs can operate at maximum speed. Programs and data are only reloaded into the operating memory at low frequency during the mission, such as at the start of a new mission phase.

2.7.3 Priority Conflict

Finally, an allocation conflict can arise when a low priority task holds a resource that a high priority task requests. Often the resource cannot be released by the former task as in the case of temporary work areas of core storage. Unfortunately, the high priority task must now be placed in the wait state until the low priority task can safely release the resource. The result of this situation is a degradation in the system's response time for high priority computations. For a sufficiently large degradation the effects upon the overall mission can be very serious.

Each of these hazardous situations must be avoided in designing a resource allocation algorithm for the Space Shuttle computer. The following section will present methods of avoiding these problems.

2.8 Allocation of Specific Resources

In the Space Shuttle computer there will be three categories of resource allocation for which provisions must be made. These are:

a) dynamic memory allocation,

b) common data sharing,

c) data set management.

2.8.1 Dynamic Memory Allocation

Dynamic memory allocation occurs when the executive temporarily assigns blocks of core storage to a task requesting this resource. This core is returned to the dynamic core pool either by the task during its execution or by the executive at the end of the task. To avoid deadlock we require that all core requests of a task be satisfied when the task is placed in the ready state. That is, to avoid incremental allocation a task makes all core requests known to the executive via its TCB at schedule time. If the request can be satisfied, the task can be placed in the ready state provided it is not awaiting the allocation of any other resource. If not, the task is placed in the wait state, awaiting the release of a sufficient amount of dynamic core to satisfy its needs. When this core becomes available, the task can be placed in the ready state. Eventually when the task becomes active, it has all the core it will ever need and will not have to be placed back in the wait state during execution for lack of this resource. Hence, deadlock cannot occur because of a conflict in dynamic core allocation.

Although we have avoided deadlock fairly easily, the problem of memory fragmentation is not as readily solved. The reason for this increased difficulty is that several alternative methods of avoiding this problem are available to us, and the specific method chosen depends upon the computational requirements of the mission application programs. So far these requirements are not known in any detail. Hence, we will examine four methods of memory allocation and determine which is optimal with respect to our present knowledge of the program requirements.

2.8.1.1 Fully Static. This method would avoid dynamic storage allocation by permanently assigning to each task all the core storage it needs for the duration of the mission. Memory conflicts are obviously avoided.

If the total amount of core so assigned is small, e.g., 1K bytes, then avoiding the problems of dynamic storage allocation is advantageous since the executive design will be simpler. However, the amount of core needed is likely to be higher than our 1K example above, so the extra cost in the amount of memory needed for static allocation becomes uneconomical.

This is not to say that no task should have its work areas permanently assigned. For example, a computation executed every minor cycle will utilizes its work area for a large percentage of every major cycle. In this case it could be economical to statically assign this task's work area to it. However, for the large amount of tasks run on a less frequent basis the percentage of a major cycle that they utilize their work areas is small. Hence, static storage allocation cannot be the only method of storage allocation in the Space Shuttle computer.

Note that any task having a static work area allocation is by its very nature non-reentrant.

2.8.1.2 Fully Dynamic. A frequently used method of dynamic storage allocation in large scale computing systems is to allow all tasks to compete with each other for all available core. A task can request a block of any size provided it does not exceed the amount of core available. If this block is available, it will be allocated to the task [4].

The disadvantage of fully dynamic allocation is that it does not solve the problem of memory fragmentation.

2.8.1.3 Semi-dynamic. Let dynamic core be divided into blocks of several specific sizes, e.g., 50 bytes, 100 bytes, .5K bytes and 1K bytes. Tasks which request core must be structured so that their request conforms to one of these sizes. Although this method imposes a restriction upon the tasks, the problem of memory fragmentation is now solved.

There still remains the problem of low priority tasks holding core and preventing high priority tasks from executing. The problem can be partially solved by allowing several blocks of each size in dynamic core. This will reduce the probability of all blocks of a given size being simultaneously allocated. However, the number of blocks of each size cannot be too large since this would be as uneconomical as static memory allocation. Program requirements will of course determine how many blocks and what sizes to allow.

2.8.1.4 Priority Subpool Allocation. Dynamic core will be divided into sections called subpools, one corresponding to each possible task priority level. A task requesting core will then receive its allocation only from the subpool corresponding to its priority level. Within a subpool core can be allocated on a fully dynamic or semi-dynamic basis.

If the fully dynamic method were used, fragmentation would occur within each subpool. To avoid this problem we will use semi-dynamic memory allocation (as explained above) within subpools. Each subpool will have several blocks of core of several different sizes. A task is then allocated a block of its requested size when it is placed on the ready queue.

Should a task request a block of core that is unavailable within its subpool because of existing allocations, a block from a lower priority level can be used for allocation. This will prevent a high priority task from having to wait for the release of core while low priority tasks can be scheduled. In addition, tasks of the highest priority will not have to share their subpool with any other tasks. These tasks will have the least interference from other tasks in competing for core.

The sizes of the blocks and the number of each size are determined by the number of tasks and their requirements at the given priority level. Once this algorithm has been implemented size and quantity parameters can be varied for optimization. This is the method selected in the design structure.

2.8.2 Common Data Sharing

In any multiprogramming system a resource allocation problem arises when data in core memory can be simultaneously used by two (or more) tasks. If two tasks only want to read the data, no conflict exists. However, if one of the tasks wants to update before the other has finished reading, a conflict arises.

To illustrate this, consider the examples shown in Figure 2.8. In both examples TASK B interrupts TASK A during the execution of a statement. In Example 1, presume that the interruption occurred while the matrix $\overset{*}{N}$ was being read. When TASK A resumes, the computation of $\overset{*}{M}$ will continue using some "old" $\overset{*}{N}$ data and the "new" $\overset{*}{N}$ data assigned in TASK B. In order to prevent this conflict, initiation of TASK B would have to be stalled until the reading of $\overset{*}{N}$ in TASK A is completed.

In Example 2, presume that the interruption occurs just after the current value of Y is loaded into the accumulator. When TASK A resumes, the "old" value of Y (i.e., not reflecting the update of Y in TASK B) is restored into the accumulator, X is subtracted and the result assigned to Y. In order to prevent this conflict, the initiation of TASK B would have to be stalled until the value of Y is updated in TASK A.

These examples illustrate the fact that accesses to shared data must be controlled to prevent conflicts. One possible way of doing this is by preventing task dispatching at critical times. This method is too restrictive however, especially for high priority tasks needing fast system response. We will investigate alternative approaches to this problem.

a) OS/360 uses the ENQ and DEQ macros to grant tasks access rights to shared data. ENQ will grant a task access rights as long as no other task is using the data. In the latter case, the task requesting access rights is put in the wait state, awaiting the release of this data (DEQ). Upon this release, the next task enqueued for access rights is taken out of the wait state and allowed to proceed. For two tasks that only want to read shared data, this method imposes a needless wait for one while the other has the data enqueued.

b) A second approach to avoid common data sharing conflicts is to use UPDATE blocks as is done in the HAL compiler [6,7]. An UPDATE block is a group of statements within a program providing a controlled environment for the reading and writing of shared data variables. Upon entry into the UPDATE block, read or write locks are established around parts of the compool containing the variables to be referenced. There need not be an individual lock for each variable nor should there be only one lock around the the entire compool. How the compool is organized can be decided at a later time depending upon the programs to be executed and their requirements.

   Should a part of the compool needed by a task be unavailable for locking, the task is placed in the wait state. Any other parts of the compool it has locked are now unlocked so that they can be used by nonwaiting tasks. The requesting task can be placed in the ready state when the scheduler determines that all parts of the compool requested now can be allocated to this task. At this time read or write locks are established around these parts of the compool.

   Three types of locks can be established: read, write, and writing. We say that unlocked data is in state 0 and locked data can be in states 1-3 corresponding to the three types of locks respectively.

   A read lock will enable another task that also wishes to read lock this data to do so. If a write lock is established around a piece of data, a copy of the data is made for the updating task. Upon closing the UPDATE block, the compool is updated as long as no other locks exist around the data to be undated. No writing locks can be put on a given part of the compool, until any read locks already there are removed by all tasks reading this data. If the locks exist, the updating task must wait until the locks are removed.

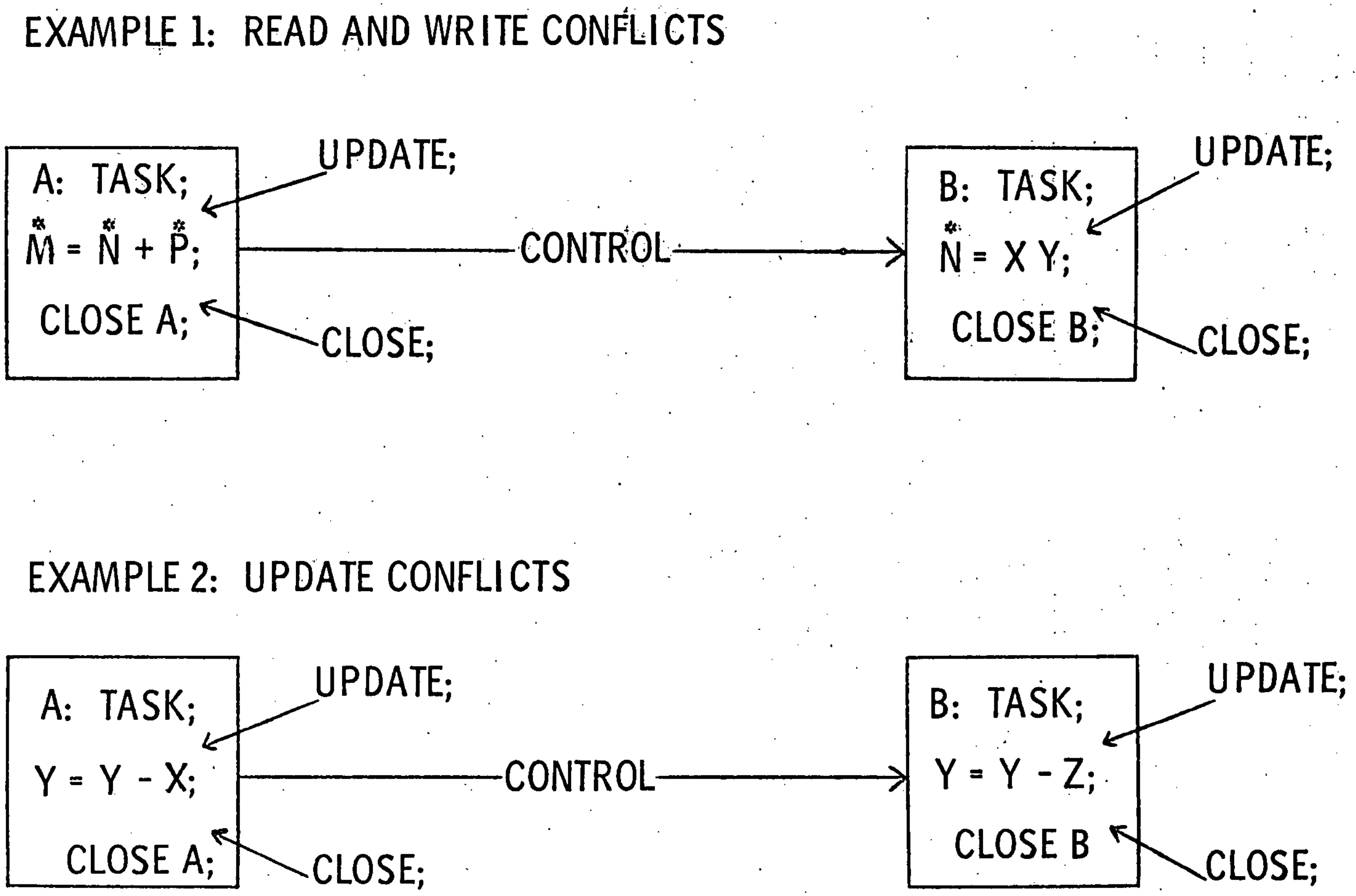
Consider the first example above and suppose that the statements in question (in TASKS A and B) were enclosed within UPDATE blocks. In TASK A a read-lock is established for $\overset{*}{N}$, because it will be read only. After the interruption, a write-lock is established for $\overset{*}{N}$ and TASK B proceeds toward completion using copy-data for $\overset{*}{N}$ rather than active data. At the end of the update block in TASK B, the process stalls because of the read-lock imposed in TASK A. As a result, TASK A is allowed to continue with consistent "old" $\overset{*}{N}$ data. After completion of TASK A, a copy-cycle is effected in TASK B and $\overset{*}{N}$ is updated. All conflicts are eliminated. A table of compool state transitions follows.

Present State / Desired State	Free	Read Locked	Write Locked	Writing Locked
Free	O.K.	O.K.	not applicable	O.K.
Read Locked	O.K.	O.K.	O.K.	Wait
Write Locked	O.K.	O.K.	Wait	Wait
Writing Locked	not applicable	Wait	O.K.	not applicable

To prevent any task from locking a part of the compoool any longer than necessary, no I/O statements and no programmed WAIT statements will be allowed in an UPDATE block. This requirement will prevent a high priority task from having to wait for long time intervals while a lower priority task has data locked.

To economize on the amount of core needed for the compool, part of the compool can be overlaid on transitions to different mission phases. If two tasks that are only executed during a particular mission phase use part of the compool, it is needless to keep this part of the compool in core as long as no other task in another phase will ever again use the data. In this case as new program modules are read into core during a mission phase transition, this part of the compool can be overlaid.

Figure 2.8 Control of Shared Data



NOTES: 1. B "INTERRUPTS" A IN BOTH CASES

2. #1 TASK A RESUMES USING OLD AND NEW VALUES FOR $\overset{*}{N}$

3. #2 TASK RESUMES "CLOBBERING" THE VALUE FOR Y SET BY TASK B

2.8.3 Data Set Management

Data set management is heavily dependent upon the type of mass storage unit used on the Space Shuttle. If a tape drive is used, as in the MDC/TRW study, very little data management capability will be necessary. However, if a random access unit is used, as in the NR/IBM study, more extensive data management facilities will be necessary.

In this report we will assume a random access unit, especially since the ASIL configuration includes an IBM 2311 disk drive. However, the data management system we will present is not as general purpose as in the System/360, for example. It is designed to meet the needs of the Space Shuttle mission. One of the criteria used in designing this part of the executive is the desirability of minimizing use of the random access unit during the Space Shuttle mission. The major anticipated uses of the storage unit are to record flight data, to update the programs in core memory on a per mission phase basis, and to retrieve display skeletons for the visual display application programs. More frequent use of the mass storage unit is unnecessary, based upon the two Phase B study reports [1,2].

There will be two classes of data sets on the random access storage unit, read only and read/write. The former category may be read at any time by any number of tasks without conflict. The latter category, however, can cause access conflicts, and hence, some protection mechanism is necessary.

A directory of each data set on the storage unit and its characteristics will be maintained in core memory (see Figure 3.4). The data set directory entry for a read/write data set will identify only one program module*with writing access rights. Whenever a task requests to write upon a data set, the I/O supervisor will check to see if the data set is indeed read/write, and if the requesting task has access rights. Since only one task can update a given read/write data set, no write conflicts are possible.

A task may also request to read a read/write data set. For example, data recorded in a former mission phase may be important to an executing task. In this case, the I/O supervisor will honor the read request. However, the software must be structured so that the requesting task is not reading part of the data that is presently being updated. The I/O supervisor will not check for this fact. Each task that wishes to read a read/write data set is responsible for knowing the integrity of the data it receives.

* This program module must not be reentrant.

Page Intentionally Left Blank

Chapter 3

Executive System Architecture

3.1 Introduction

Describing the architecture of the executive system consists of more than an explanation of how the various parts of the executive software work. It also consists of an explanation of how these parts dynamically interact with each other to extend the power of the host machine. Furthermore, the hardware structure of this machine plays an additional role in executive system design since particular hardware features, such as I/O channel structure, influence the software design. In a sense we may consider the machine together with its executive software to be the full executive system that enables application programs to be executed.

The executive system is responsible for the control of all computing tasks in the Space Shuttle real time software environment. It must manage the allocation and utilization of all resources of the system including processor, memory, data bus system, secondary memory, timers, and all other devices connected to the computer. The executive system must be organized such that it simply and efficiently allocates system resources to the computing tasks and provides sufficient general services to application programs to enable them to achieve mission requirements.

In order to make the system flexible, it must be structured such that the executive modules are either self-contained or utilize a standardized set of subroutines. It must be possible to make alterations to these modules without jeopardizing the rest of the executive functions.

In order to make the system simple, it is necessary to prevent application programs, regardless of their complexity, from directly performing system control functions. This limits the number of checks and balances necessary in order to assure full system reliability. This does not mean that application programs are denied use of hardware facilities, but rather that the control of such facilities is restricted to one responsible module.

Since the system must support applications which will have real-time inputs and outputs, it will have to be oriented toward being able to guarantee response within some predictable time constraints and yet not be performing supervisory tasks so frequently as to constrict throughput rates, a problem encountered in many highly interactive systems.

This chapter presents a description of the architecture of the executive system selected as a basis for the rest of the design. The structure was derived from the analysis of executive functions and system requirements described in Chapter 2. The major executive queues, directories, control linkage and operating environment are defined.

3.2 Executive and Task Structure

The flight software for the Space Shuttle computer avionics system can be organized into two categories: system software and application software. The executive system is the kernel of systems software which interfaces directly between the computer configuration and the applications software. It should be constructed to appear as part of a virtual machine to the application software programmer. System software can include other functions such as display software, interpreters, or other functions necessary as utilities to application software. In this report, the executive system structure identified is a kernel set of functions necessary to continue and execute application software.

Certain assumptions have been made about the application software, which are necessitated by the characteristics of the executive system. The major structural properties that application tasks must possess are the following:

1) All application tasks communicate with each other and with the executive following a rigid set of conventions which will be described in the following chapters.

2) Application software is block oriented with all the program modules for a given mission phase in main memory during that phase. Application tasks are structured as subroutines dispatched by the executive (analogous to OS/360).

3) There are no direct I/O operations from application tasks. The executive's I/O routines handle all I/O requests.

```
A:  PROCEDURE
          —
          —
          —
          —
          —

    SCHEDULE A IN T
      SECONDS;
    END
```

CORRECT

```
A:  PROCEDURE
    SCHEDULE A IN T
       SECONDS;
          —
          —
          —
          —
          —

    END
```

INCORRECT

Figure 3.1 Correct and incorrect methods of time scheduling of background tasks.

4) A task can request the scheduling of another task.

5) All access to shared data is through the executive.

6) The executive maintains a list of all program modules that can ever be executed by the system during flight.

7) A task can include a local recovery procedure in case of a software error.

8) All tasks' dynamic memory requirements are known to the system preflight.

9) Background tasks, which are repetitively operated, are rescheduled at the end of their execution and not at the beginning as shown in Figure 3.1. This is specified because there might otherwise be insufficient background time to complete the task prior to its next dispatching.

3.2.1 General Description

The executive system is driven by a minor cycle real time interrupt, which causes execution of the cyclic sequencer. The cyclic sequencer is an executive task which performs all functions that are characterized by precise timing sepcifications. It commands all I/O operations done on a periodic basis, supervises execution of all computations to be run on a periodic basis, updates core memory with input received in the last minor cycle, and monitors the status of avionics subsystems. Upon termination of the cyclic sequencer, the dispatcher is called to select a background task for execution.

The dispatcher is at the heart of the executive system. It is this executive function that selects tasks for execution on a priority basis. When a task terminates, it returns to the dispatcher, which calls a terminator routine to insure the release of all system resources held by the task.

While an application task is executing, it may request another task to be scheduled for execution by calling the scheduler. Scheduling can be done unconditionally, on a time basis, or on the occurrence of an event. A function of the scheduler is to put this new task in a state ready for execution. It does so by calling the resource allocator to give the task any resources it may need. Should a resource be unavailable the task must wait for scheduling until this resource is freed. At this time, the resource can then be assigned to the task, and the task is then ready for execution. It competes for CPU time on a priority basis with all other tasks in a similar ready state. The dispatcher will choose the highest priority task that is ready for execution and assign the CPU to this task. A task will continue executing until it ends, or until it voluntarily releases the CPU, or until a system event occurs necessitating the CPU being assigned to another task.

At any time during its execution, a task may request I/O operations to be done and may request its own execution be halted until these I/O operations are completed. It is one of the functions of the executive to supervise and schedule all I/O operations. In addition, the executive must supervise error recovery functions. Should a hardware or software error occur, the executive must provide the capability of running a specific recovery routine depending upon the type of error. A system reconfiguration routine might then have to be executed if a piece of hardware is judged faulty. The faulty equipment will then be switched out, and the system will continue execution.

The execution software to perform all the above functions will be organized in modular fashion. We will now identify the necessary modules.

3.2.2 Identification of Executive Program Modules

a) Cyclic sequencer: performs all services done on a minor cycle basis.

b) Scheduler: puts previously inactive task or waiting task in a status ready for execution.

c) Dispatcher: assigns CPU to a task ready for execution.

d) Resource allocator: assigns system resources to tasks.

e) I/O supervisor: dispatches all I/O requests to channels.

f) Machine check supervisor: diagnostic routines executed when hardware error is detected.

g) Reconfiguration routines: brings up standby equipment when active unit is judged faulty.

h) Timer routine: sets hardware timer and signals events based upon elapsed times.

i) Program check supervisor: provides recovery from detectable software errors, such as division by zero.

j) Supervisor service routines: provide supervisor services for application programs, e.g., enable a task to await an event or to free an assigned resource.

3.2.3 Executive Operating Environment

The executive is not presented with a random stream of tasks, queued upon secondary storage, as is OS/360. Instead there is a fixed set of tasks organized on a mission phase basis. Within a particular phase, task throughput is maximized. Then if core memory must be overlaid with new program modules, they are loaded from secondary storage at the beginning of a new mission phase in order to minimize the use of the mass memory unit. Moreover, since the modules loaded will be known preflight, their loading addresses and relocation constants will be determined at compile time. In other words, fully dynamic loading and binding of program modules is not supported by the executive. This minimal use of the MMU presents a fixed program environment for the executive system.

3.3 Definitions

3.3.1 Task

A task is an executive unit of work shich competes for system resources. A task is created dynamically upon execution of the executive's scheduling function. A task is identified and defined a unique task control block. A task control block (TCB) is a table containing all pertinent control information for a task used by the executive for task management. The TCB is created by the scheduler when it attempts to bring a currently unscheduled program module into the system. Each TCB contains a pointer to a program module which the task executes.

A program module is code executable by the executive. Program modules are started by the executive and return control to the executive END function upon completion. A program module may be associated with more than one task.

The following information is contained in the TCB:

a) task identification;

b) program module entry point;

c) program module characteristics, such as reentrant;

d) an area to save the PSW, 16 GRs and 4 FPRs should the task go on the ready or wait queues;

e) task priority;

f) a flag to denote the task being partially complete;

g) a pointer to the DCD entry for the task's dynamic core;

h) a pointer to the chain of ECBs should the task go into the wait state;

i) the number of events the task awaits to be made ready, supplied when the task goes into the wait state;

h) a pointer to a list of the compool parts the task has locked while it is in an update block;

k) a timer entry indicating the time at which the task can be made ready should it be on the time wait queue;

l) a pointer to any task's TCB that schedules this task by LINK;

m) an entry point for a task specified recovery procedure in case of a program check error;

n) threaded list pointers for the queue and subqueue the TCB is on.

A task control block designed for the EP is illustrated in Figure 3.2. It contains a task ID assigned by EXEC dynamically at schedule time.

A task may be in one of four task states at any time.

a) Active: The task has been allocated the CPU and is executing.

b) Ready: The task has been assigned all its resources and if ready for execution. It only awaits the CPU.

c) Wait: The task is awaiting the occurrence of some event or events in the system. Such an event may be the release of a resource, an elapsed time, or an I/O interruption.

Program module ID		
characteristics	entry point	
priority	completion state	DCD pointer
PSW	(2 full words)	
GR	(16 full words)	
FPR	(8 full words)	
event information	ECB pointer	
Compool lock list pointer		
Timer entry		
Recovery program address		
Pointer to parent task's TCB		
TCB queue pointer		
" " "		
Subqueue pointer		
" "		

← 1 Full Word →

Figure 3.2 Format of a Task Control Block

d) Inactive: The task is not presently known to the scheduler. However, its program module is present in core storage or on a mass memory unit. (Strictly speaking, an inactive task is merely a program module and not a task. A program module is made a task at schedule time, when its TCB is created.)

Our concept of the states of a task is analogous to the MULTICS concept of the states of a process [11,17]. A state transition diagram is shown in Figure 3.3.

3.3.2 Executive Queues

The executive queues are lists used by the executive to associate and control tasks of a similar condition. Task control blocks are linked into lists corresponding to a particular executive queue. A task can only exist in one queue at any instant of time. There are four major executive queues: ready queue, wait queue, time queue and I/O queue.

a) Ready queue: The ready queue is a threaded list whose elements are the TCBs of the tasks ready for execution. These TCBs are organized on a priority basis with the TCBs corresponding to the highest priority tasks occurring at the beginning of the list. An entry is established by the scheduler in the ready queue when a task is brought to the ready state.

b) Wait queue: The wait queue is a threaded list whose elements are the TCBs of the tasks waiting for the occurrence of some event or events. Each TCB on the wait queue points to a list of ECBs, and each ECB on this list corresponds to an event. When all these events or some allowable combination of them have been completed, the task can be put on the ready queue.

c) Time queue: The time queue is a subqueue of the wait queue. The tasks on the time queue are awaiting the occurrence of a timed event. At some multiple of a minor cycle time interval, the executive examines the tasks on this queue, to determine if they can be made ready at the present time. If so, those that can are placed on the ready queue.

d) I/O queue: The I/O queue is a subqueue of the wait queue. The tasks on the I/O queue are awaiting the completion of some I/O operation. When the I/O operation completes, a task awaiting it in this queue can now be placed on the ready queue.

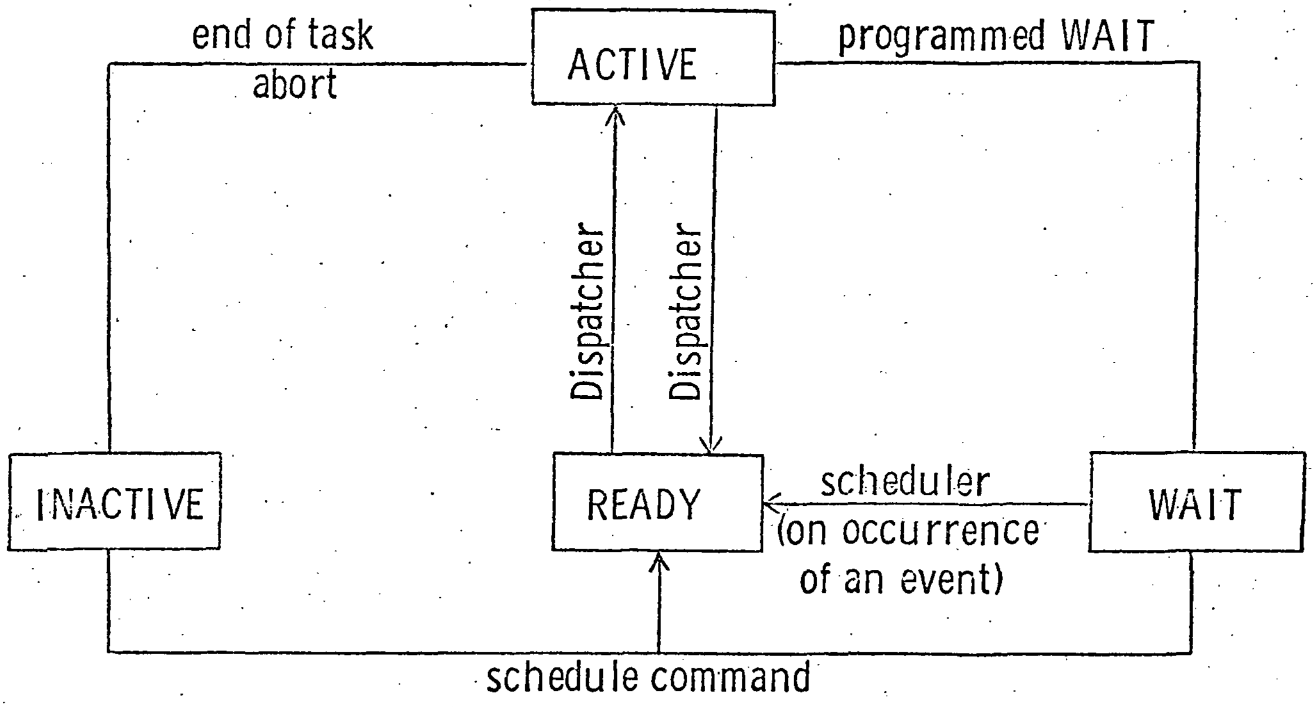


Figure 3.3 Task State Transition Diagram

3.3.3 Common Data Pool

The COMPOOL is an area of operating memory permanently assigned to data variables shared by tasks. All communication between tasks is done through the compool. Data assigned in the compool remains in the system subsequent to a task completion. It is statically assigned as opposed to the dynamic memory assigned to a task for working storage. The compool is organized into two parts: a mission compool and a phase related compool. The data assigned in the mission portion of the compool is permanently resident. Data assigned to the phase dependent portion of the compool exists only during that phase of the mission. It is overlaid with other phase data during subsequent mission phases. When a mission phase is initiated, the phase is loaded from the secondary memory and the phase dependent compool is initialized. Data which is to be retained subsequent to a task completion must be included in the compool. All accesses to data in the compool must be coordinated by the application task through the executive system. The executive prevents conflicts in the use of the data by system tasks. The SECURE, RELEASE and COPY executive functions are provided for compool interaction and are discussed in a succeeding chapter.

3.3.4 I/O Request Block

The I/O request block (IORB) is a table of all pertinent control information for the I/O channel to execute an I/O operation. The format, content ,and use of this control block are discussed in Chapter 5.

3.3.5 Directories

There will be three directories present in core storage for use by the executive task management functions. These directories and their use will now be defined.

3.3.5.1 <u>The Program Module Directory</u>. The program module directory (PMD) is a list of all program modules known to the system; i.e., all program modules both in operating memory and secondary storage. Each entry consists of three full words and has the format shown in Figure 3.4a. It contains the following information:

a) program module identification,

b) where the module is resident,

c) address of module,

d) module characteristics, such as reentrant,

e) dynamic core needs.

This directory is updated when the contents of core change or new program modules are added to the system preflight. Its major purpose is to enable the scheduler to locate a program module and to provide enough information to construct a TCB.

3.3.5.2 The Data Set Directory. The data set directory (DSD) is a list of all data sets residing on the MMU. A data set may be an executable program module or a collection of flight data. An entry in this directory is three full words containing a data set identification word, MMU starting address, logical record length, and data set characteristics (i.e., read only or read/write). In addition, if this data set can be updated, the program module with update rights will be identified in the DSD entry. This information is illustrated in Figure 3.4b.

The DSD enables the I/O supervisor to locate data sets on the MMU for I/O operations.

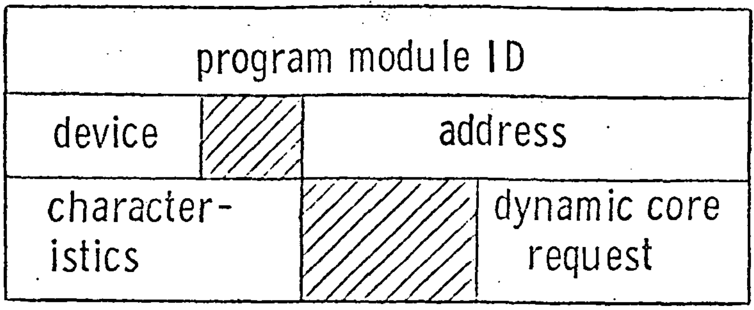
3.3.5.3 The Dynamic Core Directory. The dynamic core directory (DCD) is a list of all blocks of core that can be dynamically assigned to a task. Each entry is two full words containing the address of the block, its byte length, its subpool number, and an assignment bit. The format is given in Figure 3.4c. The DCD enables the executive to dynamically assign core to tasks at schedule time.

3.4 Subroutine Linkage

In order to standardize the way program modules are structured and to avoid conflicts in parameter passing, register usage, and register saving, a method of program module initialization and linkage must be developed. The EP hardware structure, as seen by the programmer, is similar enough to System/360 to make a linkage convention similar to the 360 feasible.

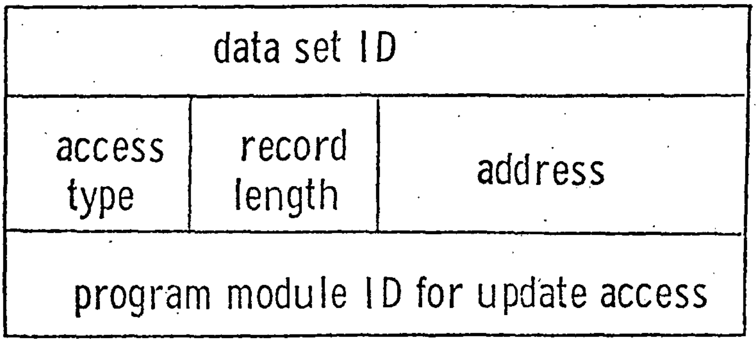
Upon entering a program module the contents of the general registers must be saved so that they can be restored upon task termination. These registers are stored in an area of core called the save area. Each task must provide a save area, pointed to by GR13, which is used by any subtask it calls. The format of the save area is shown in Figure 3.5.

1. Program Module Directory (PMD)



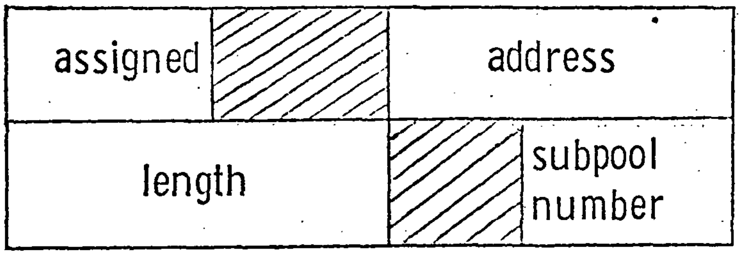
(a)

2. Data Set Directory (DSD)



(b)

3. Dynamic Core Directory (DCD)



(c)

Figure 3.4 System Directory Elements

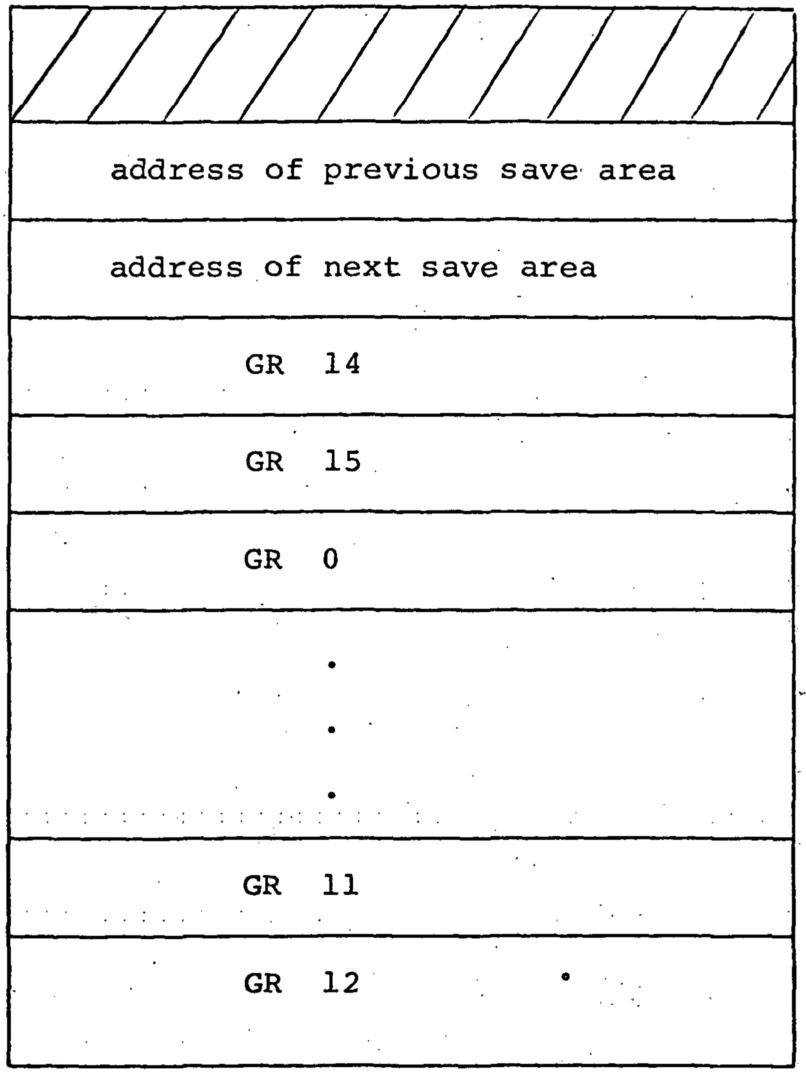


Figure 3.5 Format of a Save Area

After saving the general registers, one or more of these registers can be initialized as base registers. All addressing of core storage in a program module is done with base registers. Finally, a new save area address is put in GR13. An example of this linkage follows.

```
STM       14, 12, 12(13)      save registers in save area.

BALR      12, 0               initialize GR12 as a base
                              register.

USING     *, 12               declare to assembler that
                              GR12 is base register.

LA        2, SAVEAREA         get address of next save
                              area.

ST        13, SAVEAREA +4     store address of previous
                              save area in next save area.

ST        2,  8(13)           store address of next
                              save area in previous save
                              area.

LR        13, 2               load GR13 with address of
                              next save area to complete
                              linkage.
```

When the linkage and initialization are done, a task may now freely use the general registers.

The following assignment of the general registers will be made:

GR0: contains address of dynamic core upon entry to program module,

GR1: used to pass parameters between program modules,

GR2-GR12: may be freely used by tasks,

GR13: points to save area provided by task,

GR14: contains the return address of task that called currently executing task,

GR15: contains entry point address when control is passed to a task and can also contain a return code when a task terminates.

Upon completion of its computation, a task terminates by restoring the content of the general registers it had saved upon entry, setting its return code in GR15, and branching to the return address in GR14.

Example:

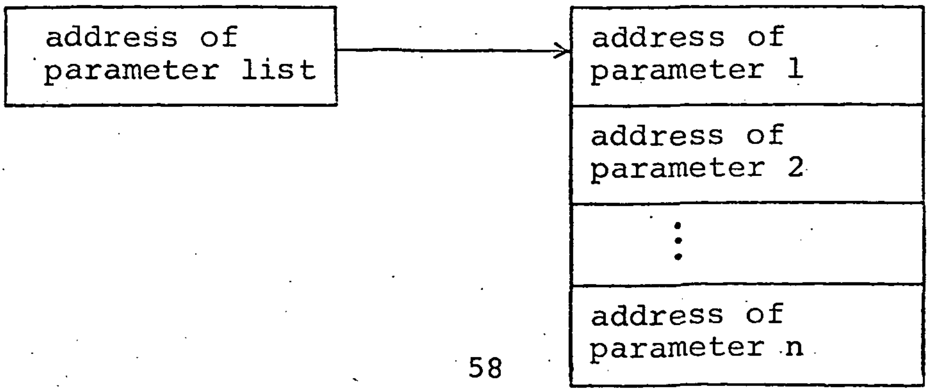
```
        L       13, 4(13)       get address of previous save area,
        LM      14,12,12(13)    restore registers,
        LA      15,4            load return code of 4,
        BR      14              return.
```

As previously mentioned, all communication between tasks is via the compool. Since one task cannot pass another a parameter list, the compool serves as the communication medium. Data variable assignments in the compool are generated at system compile time and do not change during the mission. In other words, no dynamic assignments can be made in the compool. All tasks reference compool data at fixed locations for the duration of the mission.

3.4.1 Common Subroutines

In addition to a task being able to schedule another task, a task may execute a common subroutine. A subroutine is a piece of coding which may be used by several tasks without itself becoming a task. A common subroutine must be reentrant or serially reusable. In the former case the calling task supplies working memory for the subroutine. In the latter case, the subroutine must supply control for preventing multiple simultaneous executions. A software generated event can be used by the subroutine as a semaphore to insure only one user at a time [12]. This topic is further discussed in Section 3.7. Examples of common subroutines are square root, trigonometric functions, and vector/matrix functions.

The calling task may pass parameters to a common subroutine by providing a pointer. This pointer will contain the address of a list of pointers, each pointing to one of the parameters, as illustrated below.



The subroutine may now read each of the passed parameters and return a computed value in one of them. The registers in which the parameter pointer and dynamic core pointer are passed to the common subroutine have been described in the last section.

3.5 Task Priority Levels

In the Space Shuttle computer there will be six priority levels, 0-5, with 0 being the highest priority. Priorities 3, 4, and 5 are used by application tasks.

Priority 2 is reserved for any application task while it is executing an UPDATE block. That is, if a task of priority 3, 4, or 5 is executing an UPDATE block, the task's priority is raised to 2 until the updating of common memory is completed. It then returns to its previous value. Thus in effect, we are limiting dispatching of priority 3-5 tasks while another task executes an UPDATE block. By the nature of the system there will be at most one task at priority 2 at any given time. This places restrictions on the use of an update block in that a task cannot enter the wait state voluntarily under any conditions. It must enter the block, complete the updating of common memory, and exit the block. The high priority cyclic sequencer is allowed to interrupt an update block.

Priority 1 is only used by the cyclic sequencer. It is given priority over any application task because of the time dependent nature of its execution. Should the cyclic sequencer be unable to lock part of the compool, the task at priority 2 is executed until it closes its UPDATE block. Now the cyclic sequencer can lock its required data without interference. The use of priority 2 is specifically designed to enable the cyclic sequencer to execute with the least possible wait due to shared data unavailability.

If a response time equal to a minor cycle is insufficient to handle critical mission functions, a special priority level could be included in the executive system. Priority 0 can be reserved for acyclic tasks that must immediately be executed for the safety of the mission. These tasks are time constrained and must execute in less than 0.5 msec. This rule is enforced by a timer in the hardware. (Although the EP has only one timer, the computer chosen for the Shuttle mission would need at least two, one for the minor cycle interrupt and one for timing critical task events.) Moreover, priority 0 tasks may not use dynamic core or use the compool since by their very nature no wait in their execution can be tolerated.

Examples of priority 0 tasks are computations that must be done during a critical maneuver, engine burn or cutoff, etc. Should one of these tasks require more than 0.5 msec to execute, it may change its priority to 3 or lower during its execution. Should there be no higher priority task scheduled, it will continue execution at this lower priority. Otherwise, it must wait for the CPU. In this way critical functions can immediately be given 0.5 msec of CPU time without seriously interfering with the executive's cyclic functions that must be performed every minor cycle.

Including priority 0 in this executive system would require more hardware interfaces to the computer than we have assumed. There would have to be a method of generating an immediate external interrupt in the CPU from the subsystem or device sending the interrupt condition. However, subsystem requirements have not been sufficiently defined yet to determine whether or not a priority 0 is necessary in this system.

3.6 Assignment of Core Memory

Operating memory will be organized as follows: the lower core addresses will be assigned to the executive, as shown in Figure 3.6. The first locations contain system registers, such as the timer, the PSWs, and the CSW. This assignment is described in the 4 Pi EP Manual [3]. The next block of core contains the executive's program modules, followed by the executive work area. Within this latter area the executive's queues, directories and tables are resident.

There are three types of queues present in this area: TCB queues, ECB queues, and IORB queues. Since each type of control block is a fixed size, the executive can maintain three threaded lists of unused blocks of core storage, each element of which contains enough core for allocation as one of the three types of control blocks, respectively. Thus, when a task requires a control block, the executive can remove an element from the appropriate queue of unused blocks and assign this block to the task to be formatted into a control block. Similarly, when the executive determines a task is finished with a control block, that core that the control block occupied is then returned to the appropriate queue of unused blocks for later allocation.

Sufficient space must be allowed this part of core to hold the maximum number of control blocks that will ever be needed by application tasks at any given time. Should space be unavailable, this is an error condition since more tasks are in the system than its resources can accommodate.

The compool will immediately follow the executive work area and be divided into a mission portion, which is resident in main memory throughout the flight, and a phase portion, which is overlaid when a mission phase transition occurs. A similar feature exists with the application software which follows next in memory. The mission resident part comes first, followed by the phase dependent part.

The protection key feature of the EP assumes each block of 2K bytes of core is assigned a single protection key. No subdivisions of these blocks can be assigned different protection keys. For this reason and the fact that most aerospace computers do not have a protection key feature an alternate method of protecting parts of main memory from illegal access is necessary. To avoid executive overhead in performing this protection function, simulation of the entire software system on a ground based computer must check for illegal accesses from the application tasks. The methods of valid executive access are discussed in later chapters.

3.7 Events

An event is an occurrence of significance to the system. There are a fixed number of events established for the system identified in an event directory. There are five categories of events recognized by the executive, the first four of which are controlled by the executive. These are: time events, I/O completion events, release of shared data, and release of dynamic memory. If other external interrupts are used in the EP system they may also be categorized as an event. The final category of events include those which are controlled via application software and used for task synchronization.

There are two types of events within this last category: latched and unlatched. A latched event has associated with it a binary state either on or off. Latched events may be signalled on (posted) or signalled off (deposted) under application software control via the executive. A latched event maintains its current state until changed via signal command. An important use of latched events is to record the occurrence of an event within the system so that if a task later wishes to use the occurrence of the event as a criterion for performing a function, it can do so without having lost all record of the events occurrence. An unlatched event is only signalled on. It is signalled off immediately after processing by the executive. In a sense an unlatched event is a pulsed event analogous to a hardware interrupt.

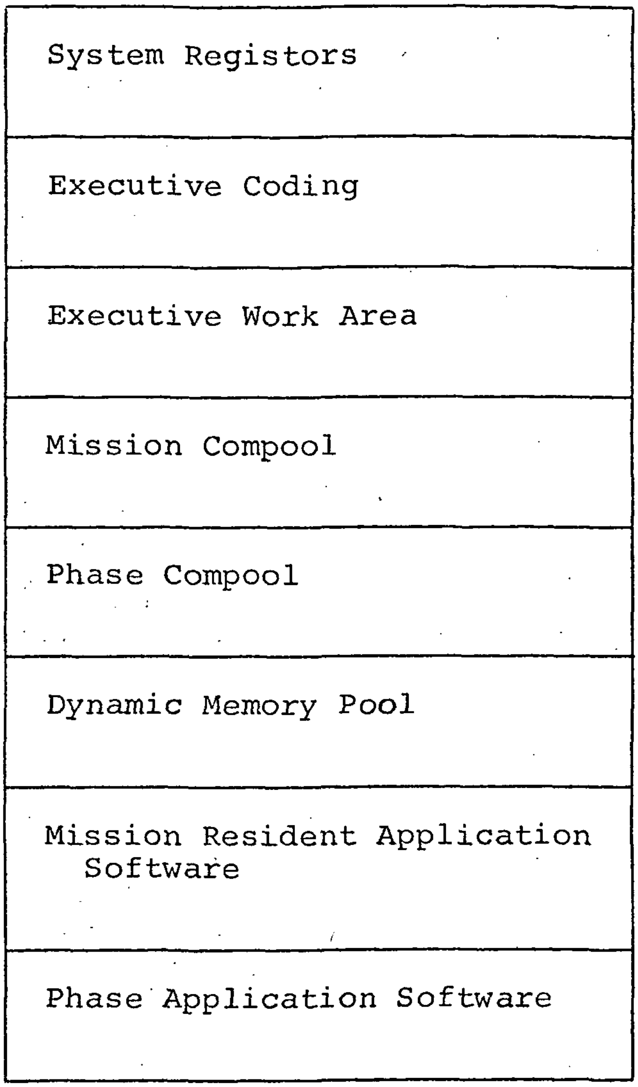


Figure 3.6 Structure of Operating Memory

An event control block (ECB) contains the current status of an event. It is dynamically created by the executive when a task is placed in the wait state. All events have system scope. When the anticipated event occurs, bit 0 of the ECB is set to 1 to record the event for the executive. See Figure 3.7 for the format of an ECB. The ECB contains a bit to denote if the task is awaiting the event, a bit to denote if the event is completed, and two threaded list pointers.

3.7.1 Event Handling

In the Space Shuttle software system there is a close relationship between task management and event handling. Tasks that are placed in the wait state remain there until the anticipated events that they are awaiting occur. Then the event handling facilities of the executive call upon the scheduler to place these tasks in the ready state.

Tasks can be placed in the wait state in two ways. First, a task can voluntarily request the executive to place its TCB on the wait queue until some anticipated event or events occur. Second, when the scheduler attempts to place a task in the ready state, the unavailability of a resource on the nonoccurrence of some event(s) causes the task to wait until the resource is freed or the event(s) occurs.

A TCB in the wait queue is associated with a threaded list of ECBs, each corresponding to an event whose occurrence the task awaits. In addition, each event has an associated event list which contains pointers to all ECBs of tasks awaiting the occurrence of the event. Thus, when an event occurs, each ECB pointed to by the event list can be posted, i.e., record the fact that the event occurred. An illustration of this control structure is given in Figure 3.8.

After the event occurs, the scheduler is called. Its function is to determine if any task awaiting this event can be placed on the ready queue. The criterion for this decision is whether or not all (or some acceptable combination) of the events a task is awaiting have occurred. If so, the task is placed in the ready state by having its TCB moved to the ready queue and having its ECBs deleted. In addition, the scheduler can now delete the event list associated with the event. Tasks can perform a function based upon the occurrence of a single event or upon the occurrence of some combination of several events. In the latter case the allowable combinations are

1) The occurrence of all of the events, or

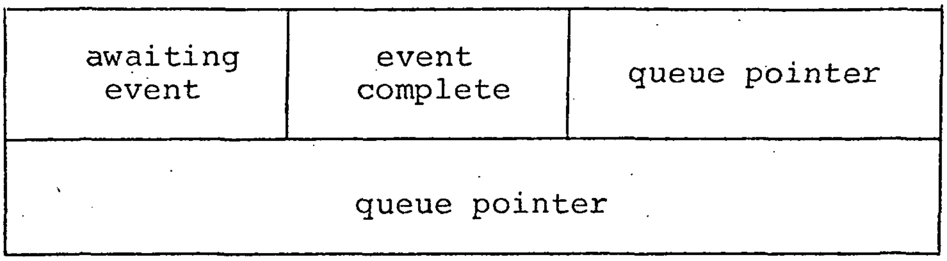


Figure 3.7 Format of Event Control Block

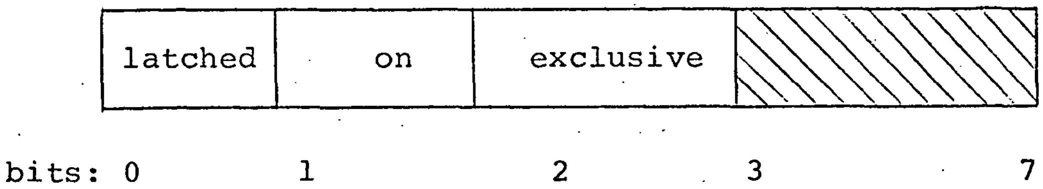


Figure 3.9 Format of Event Descriptor Byte

2) the occurrence of m out of n events, where $m < n$.

Each task awaiting an event in one of the first four categories can only await this one event and not some combination of events. However, software controlled events contain a predefined number of distinct events which may be used individually or in combinations by tasks. Events are not dynamically created by the system. Hence, software generated signals must correspond to events defined at system generation time. Each software generated event contains an event descriptor byte, containing the characteristics and state of the event. Figure 3.9 shows the format of the byte. Bit 0 describes the event as latched or unlatched; bit 1 records whether the event is on or off; and bit 2 describes the event as exclusive or non-exclusive, a distinction we will presently explain.

Within the class of unlatched events we will choose a subset to be exclusive events. An important use of exclusive events is to exclude tasks from use of some serially reusable resource. When an exclusive event is signalled on, only the highest priority task awaiting the event is placed in the ready state. All other tasks awaiting the event remain on the wait queue. When the highest priority task is made ready, the event is then signalled off by the scheduler to be sure no other application task can interfere with the exclusion process. This use of exclusive events is analogous to Dijkstra's concept of semaphores [12].

Note: it is the duty of the programmer to know if the events he is using in his tasks are being used by any other tasks. Without being sure of this fact, tasks can unintendedly interfere with each other's execution and destroy the integrity of their computations.

Also note: in the actual implementation of this executive system, some categories of events will be immediately serviced by the executive upon occurrence of the event, and hence, a record of the event's occurrence will be unnecessary. These events will therefore not need ECBs in their functional implementation. These events include release of dynamic memory and unlocking parts of the compool.

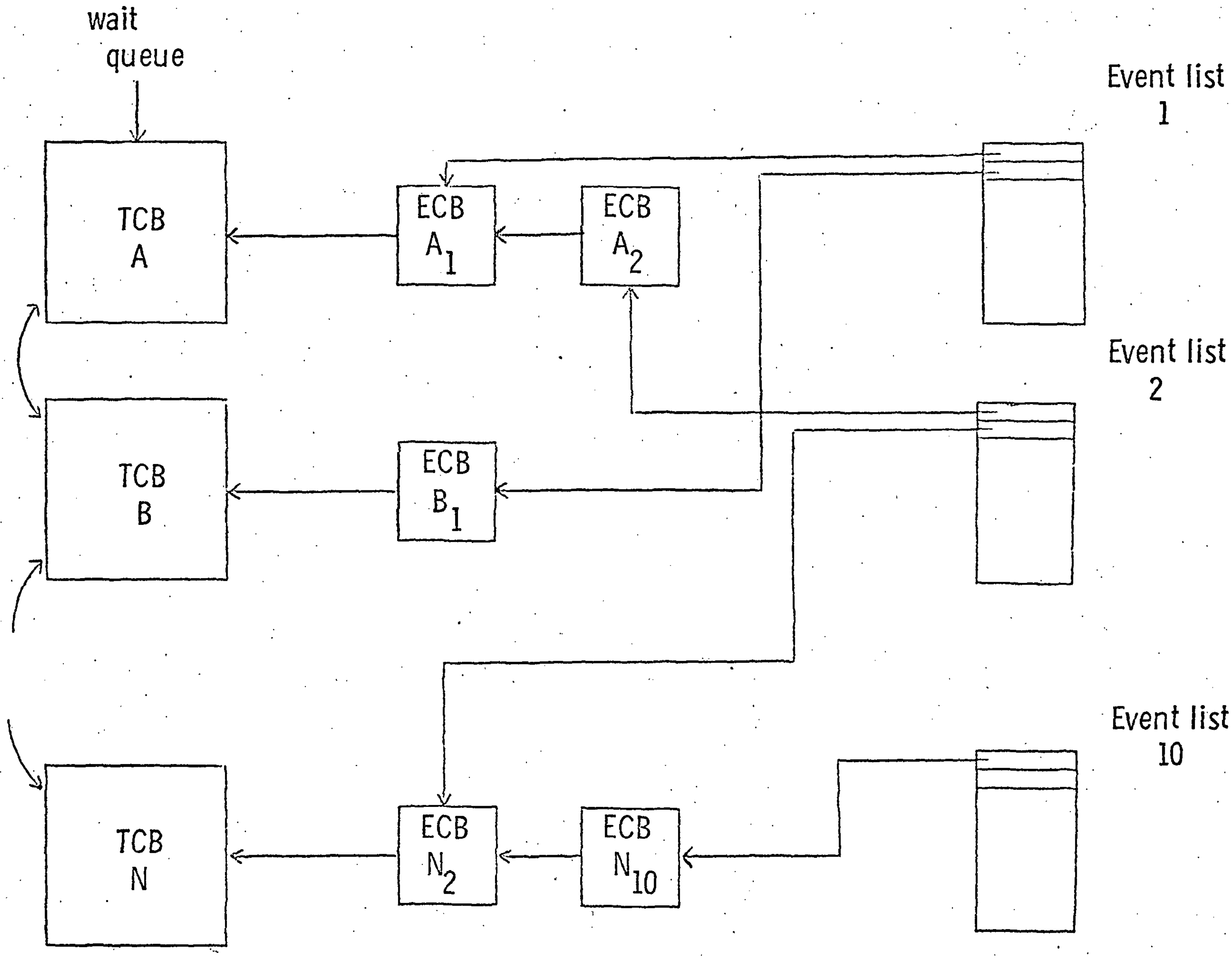

wait
queue
TCB
A
ECB
A_1
ECB
A_2
Event list
1
Event list
2
TCB
B
ECB
B_1
Event list
10
TCB
N
ECB
N_2
ECB
N_{10}

Figure 3.8

3.8 I/O Scheduling

We assume that the data bus system hardware will be mechanized in a way which allows bus operations to continue independently of the CPU once an I/O command is issued to the bus control unit. This means that the processor is only allocated to the I/O function during an I/O channel command and should be reallocated to the computation job stream upon completion of the command. The design question for the software I/O control will be how to schedule the I/O operation: should it be decoupled from the executive program control and maintain its own separate I/O queue, or should it be inserted as an integral part of a fixed sequence? For example, if I/O were operated each minor cycle it would output data from the previous cycle, and input data which is to be processed during the following cycle. With this concept, however, the I/O must be predetermined and fixed, with constraints similar to those for fixed scheduling of computational jobs. Input and output then occurs each cycle, whether it is needed or not. This approach will cause excess data to be put on the bus, reducing its effective bandwidth, and its capability for expanded performance. On the other hand, scheduling I/O as a priority queue based on demand, has many features in common with scheduling jobs (e.g., priority, timing, conflicts, etc.). An effect of the I/O queue on the system is that several jobs may be in a suspended state awaiting I/O completions. Methods are available to avoid such delays, for example, buffering for data in and out, and issuing commands only via a queue. The I/O algorithms presented in Chapter 5 will combine the best features of synchronous and asynchronous control.

3.9 I/O Considerations

At present there are uncertainties concerning the operation of I/O whose resolution overlaps the designs of the shuttle avionics subsystems. Some of these uncertainties are:

a) Does the central computer have to perform echo checking of all common data issued on the bus to ensure that commands are received by the right subsystem; or is this function performed by the bus control hardware, or by the standard interface units?

b) Is data validation in transmission a responsibility of software?

c) Can demand/response really be achieved via the central computer software? This question becomes important if the use of existing hardware is contemplated, because I/O demands may force the computer into an "I/O-bound" condition, or seriously load its processing capability.

d) How are devices such as the hand controller to be incorporated into I/O without interrupts?

e) Where and when should conversion and limit testing be done: in the central software, or at the subsystem?

f) How is telemetry downlink and uplink handled and how does it effect I/O control software?

Chapter 4

Task Management Functions

4.1 Introduction

In this chapter we will present a functional design of the executive software task management functions. Each task management function is defined, and flowcharts are presented. The intention of this chapter is to present a functional design description of each of the task management areas of the executive and not to present a coding level design. For this reason several software error checking features have been incorporated in the algorithms, but yet others have not been since they are more appropriately included on a coding level of design.

4.1.1 Definition of Task Management Functions

The Task Management area of the executive system has the primary function of controlling the sequencing of task execution. It supervises the scheduling and dispatching of the CPU, the allocation of memory resources to application software in accordance with a defined controlling algorithm; and it responds to requests from executing tasks for task and event control. As part of this function an executive routine, called the Cyclic Sequencer, is defined and operates at priority 1. This routine controls the synchronous execution of cyclic application subroutines.

a) The Scheduler is that part of task management which takes a program module from the inactive state, makes it a task, and places it on the ready or wait queue. Moreover, the Scheduler takes tasks from the wait state, and when possible, places them in the ready state.

b) The Dispatcher selects a ready task for execution. It observes a priority algorithm with tasks organized in a FIFO manner within a priority level.

c) The Resource Allocator is called by the Scheduler and tries to give tasks the main memory resources they need for execution.

d) The Cyclic Sequencer manages all tasks and I/O performed at Priority 1.

e) Task Management Service Routines are those executive routines which an application task can call upon to perform some task management function. These routines include:

1. Freemain - release dynamic core held by the active task

2. Secure - lock part of the compool for reading or updating

3. Release - unlock the part of the compool held by the active task

4. Copy - copy part of the compool into the active task's work area

5. Link - schedule a task and wait upon the task's completion

6. End - terminate the currently active task

7. Schedule - schedule a task

8. Wait - place the active task in the wait state

9. Signal - turn a system event on or off

10. Test Event - test a software event to see if it is on or off

11. Change CCT - change an entry in the Cyclic Control Table

Each of these routines is called by a 4 Pi EP Supervisor Call (SVC), explained in Chapter 2.

4.1.2 The Scheduler

The scheduler is functionally organized into two parts; a SCHEDULE processor which responds to supervisor calls to schedule a program module as a task, and an event services processor which is called at the occurrence of system software events, i.e. a software signal.

4.1.2.1 The SCHEDULE SVC Processor. This routine performs the following functions:

1) Search the TCB queues to see if a task is already scheduled using the requested program module. If so, the module's characteristics must be checked to be sure no scheduling conflict exists. Such a conflict can arise if the program module is not reentrant and is scheduled as a task more than once concurrently.

2) Create a TCB for the task from information found in the PMD.

3) If the task is to be scheduled upon some condition, place the task in the wait state and set up the appropriate ECB linkage.

4) For tasks to be scheduled unconditionally, try to allocate any necessary core storage. If it is unavailable, place the task in the wait state.

5) If the task can be made ready, place the task on the ready queue by priority. The TCB becomes the last one at its priority level.

6) Return control to the active task.

When a TCB is inserted into a queue (all of which have a threaded list structure), this process is accomplished merely by pointer manipulation. For example, suppose that task A at priority 3 and task C at priority 5 are on the ready queue, as shown in Figure 4.1a. To place task B on the ready queue at priority 4 new pointers must be established. These priorities are illustrated in Figure 4.1b.

4.1.2.2 Event Services Processor. When the scheduler is called by the software associated with an event, it performs the following functions:

1) For exclusive software signalled events at most, one task can be made ready. Hence, the scheduler finds the highest priority waiting task and tries to put it into the ready state. When a task is put into the ready state, the pointer to its ECB in the event list can now be deleted.

2) For non-exclusive events, the scheduler checks to see if all tasks awaiting the event can be made ready. Those

Ready Queue

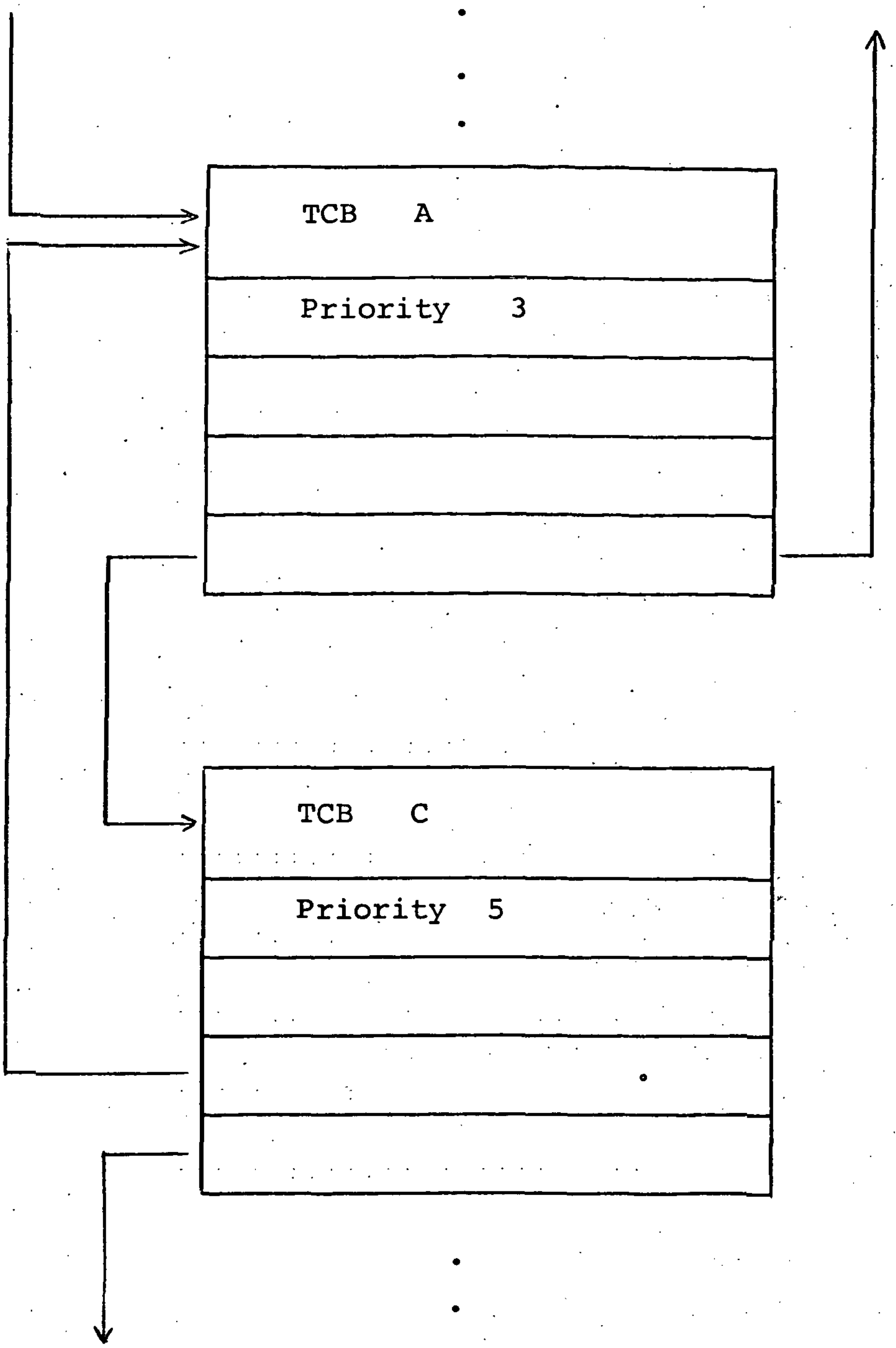


Figure 4.1a Example of TCB queue
Before Entry of TCB B

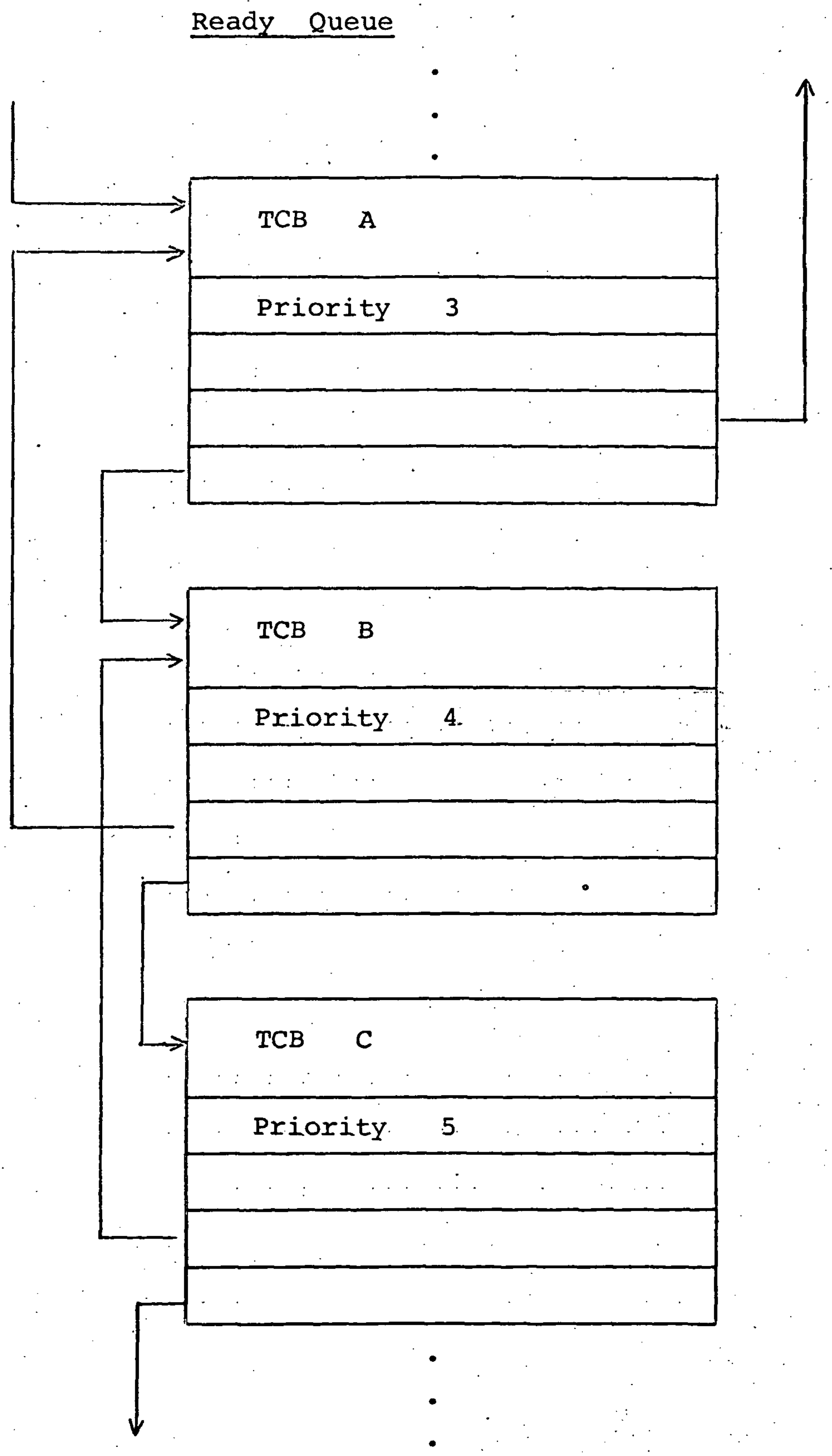


Figure 4.1b Example of TCB queue After Entry of TCB B

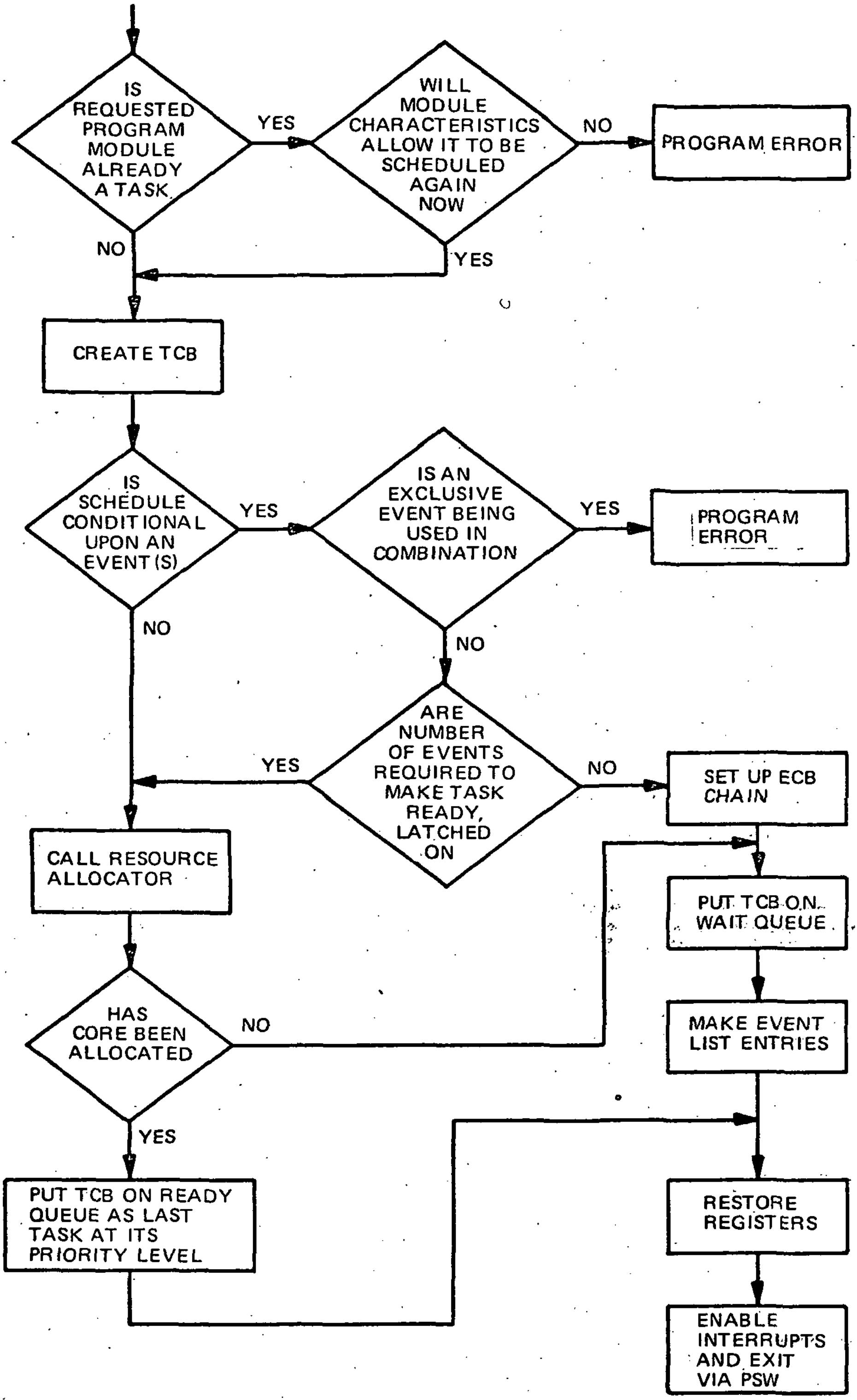
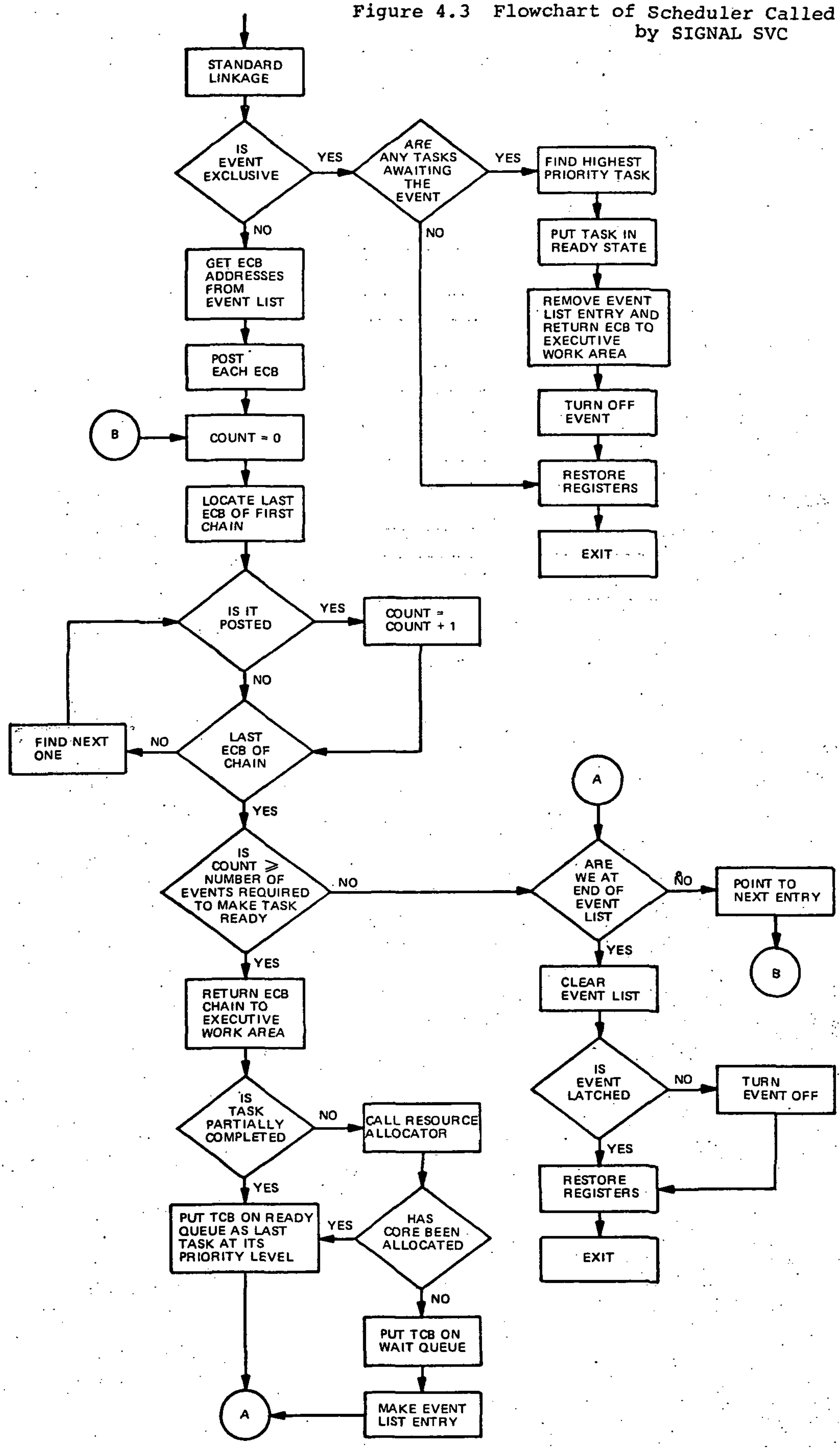

IS REQUESTED PROGRAM MODULE ALREADY A TASK
YES
WILL MODULE CHARACTERISTICS ALLOW IT TO BE SCHEDULED AGAIN NOW
NO
PROGRAM ERROR
NO
YES
CREATE TCB
IS SCHEDULE CONDITIONAL UPON AN EVENT (S)
YES
IS AN EXCLUSIVE EVENT BEING USED IN COMBINATION
YES
PROGRAM ERROR
NO
NO
ARE NUMBER OF EVENTS REQUIRED TO MAKE TASK READY, LATCHED ON
YES
NO
SET UP ECB CHAIN
CALL RESOURCE ALLOCATOR
PUT TCB ON WAIT QUEUE
HAS CORE BEEN ALLOCATED
NO
MAKE EVENT LIST ENTRIES
YES
PUT TCB ON READY QUEUE AS LAST TASK AT ITS PRIORITY LEVEL
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW

Figure 4.2 Flowchart of SCHEDULE SVC

Figure 4.3 Flowchart of Scheduler Called as Subroutine by SIGNAL SVC



that can have their TCBs put on the ready queue, and the core occupied by the ECBs is returned to the executive's work area queues. Those tasks that cannot be made ready, remain on the wait queue. The scheduler can now delete the event's entire list of pointers to the ECBs of tasks awaiting the event.

3) Return control to the event software.

4.1.3 The Dispatcher

Dispatching is the central function of the executive system. The dispatcher initiates all application tasks, and all tasks under normal conditions return to the dispatcher upon termination. At that time, a terminate routine is executed to enable the task to release any system resources it may be holding. This process is illustrated in Figure 4.4.

When there are no ready tasks in the system, the dispatcher places the CPU in the wait state. This feature aids digital simulation requirements. The simulator can be implemented to advance through the wait until the next environmental interrupt is predicted.

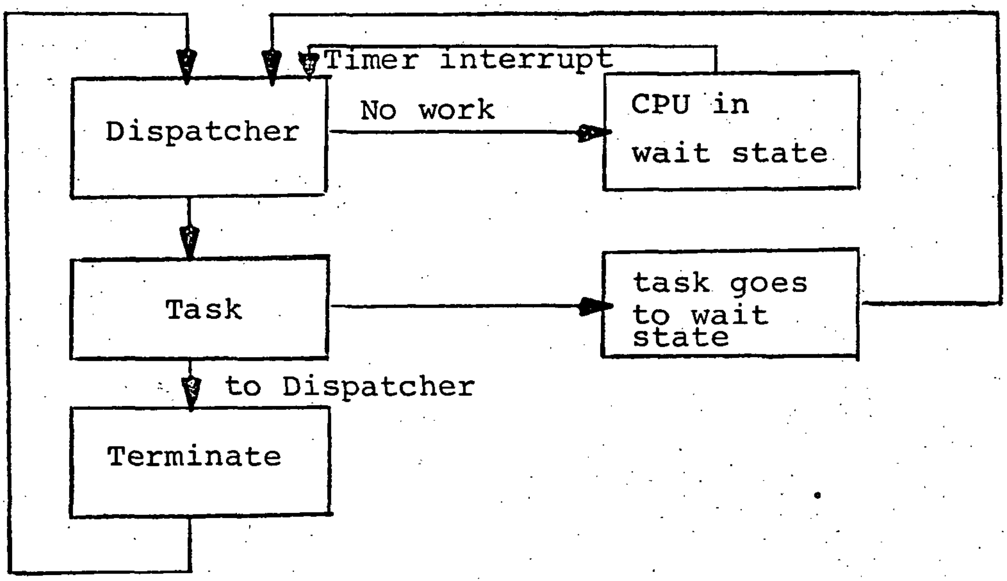


Figure 4.4 Overview of Dispatching and Terminating a Task

The major function of the dispatcher is to select the highest priority task ready for execution and make it active. Within a priority level the oldest ready task is always selected first yielding a FIFO dispatching algorithm. When a task is being initiated by the dispatcher, i.e., when it is not being dispatched in a partially completed state, the dispatcher assigns a save area to the task so that the task can perform its standard linkage operations as described in Chapter 3. The CPU is then assigned to the task making it the system's active task.

As explained in Chapter 2 application software operating in the background is segmented if it has lengthy execution time. Each active task voluntarily requests the dispatcher to check the ready queue to determine if a higher priority task is waiting for the CPU. If so, the higher priority task is made active. These segment points are established at convenient breakpoints to minimize the effect of potential job swaps. These dispatching checks are done with SVCs, inserted in the program with the assistance of an assembler or compiler that generates the object coding. The flowchart of the dispatch check algorithm is presented in Figure 4.6.

The dispatcher is entered at:

1) the end of a task via return linkage;

2) a segment point in a "long" background task via a supervisor call;

3) the active task's going into the wait state;

4) the beginning of a minor cycle via the timer interrupt software.

4.1.4. The Resource Allocator

The resource allocator is a subroutine called by the executive's task management functions to allocate dynamic memory to tasks in the system.

The dynamic memory requirements of each application software module is pre-established at system generation and specified in the PMD. The function of the allocator is to maintain the current status of all of dynamic memory and to service requests made to it by other parts of the executive.

As explained in Chapter 3, a portion of the operating memory is used as dynamic memory. It is organized into blocks

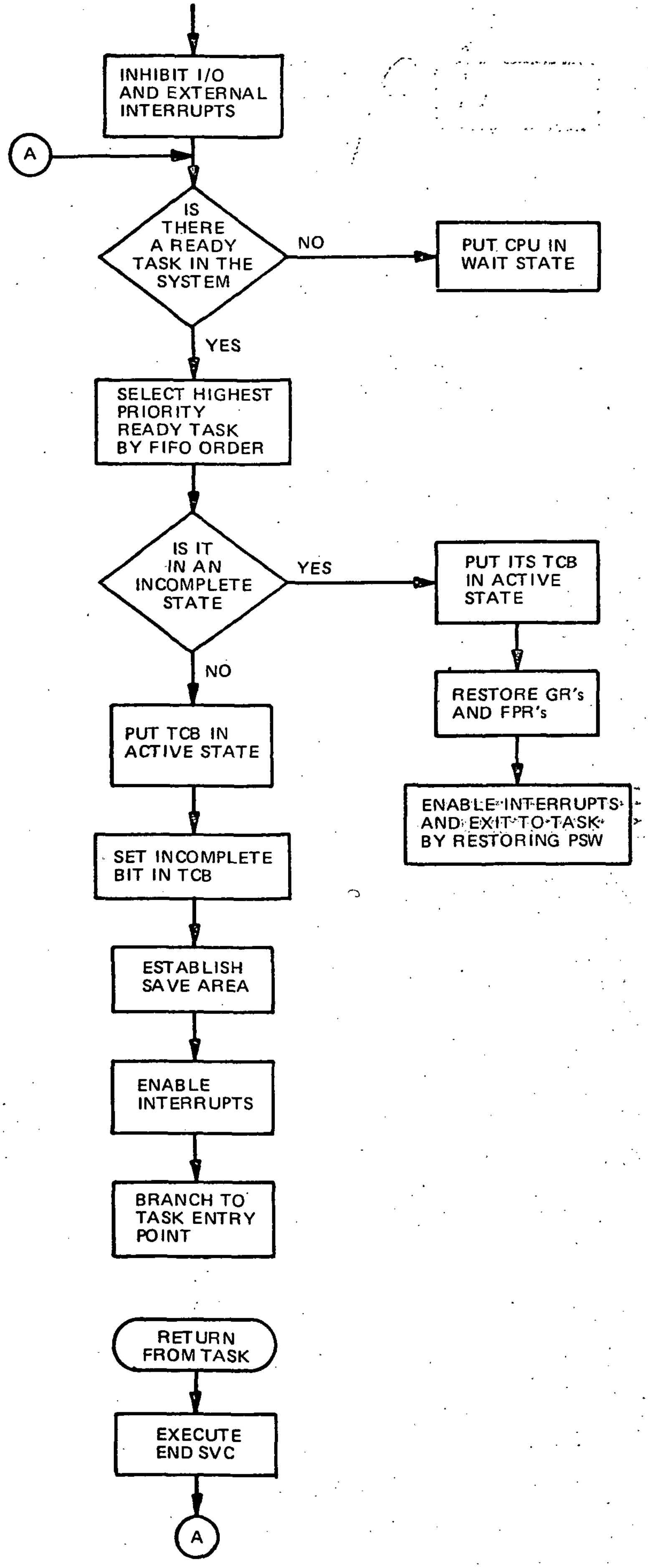

INHIBIT I/O AND EXTERNAL INTERRUPTS
A
IS THERE A READY TASK IN THE SYSTEM
NO
PUT CPU IN WAIT STATE
YES
SELECT HIGHEST PRIORITY READY TASK BY FIFO ORDER
IS IT IN AN INCOMPLETE STATE
YES
PUT ITS TCB IN ACTIVE STATE
RESTORE GR's AND FPR's
ENABLE INTERRUPTS AND EXIT TO TASK BY RESTORING PSW
NO
PUT TCB IN ACTIVE STATE
SET INCOMPLETE BIT IN TCB
ESTABLISH SAVE AREA
ENABLE INTERRUPTS
BRANCH TO TASK ENTRY POINT
RETURN FROM TASK
EXECUTE END SVC
A

Figure 4.5 Flowchart of Dispatcher

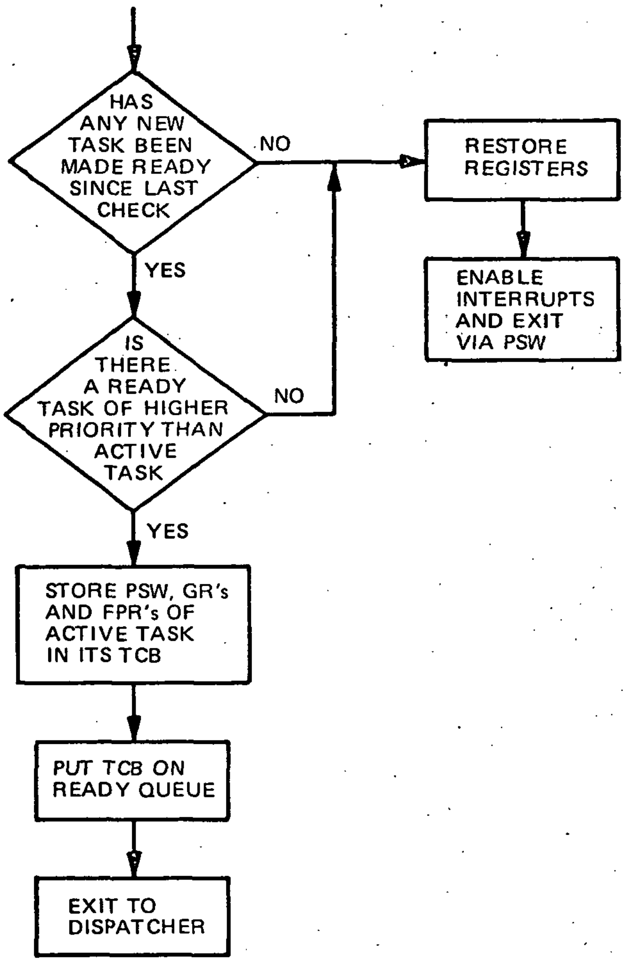

HAS ANY NEW TASK BEEN MADE READY SINCE LAST CHECK
NO
YES
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW
IS THERE A READY TASK OF HIGHER PRIORITY THAN ACTIVE TASK
NO
YES
STORE PSW, GR's AND FPR's OF ACTIVE TASK IN ITS TCB
PUT TCB ON READY QUEUE
EXIT TO DISPATCHER

Figure 4.6 Flowchart of Dispatch Check SVC

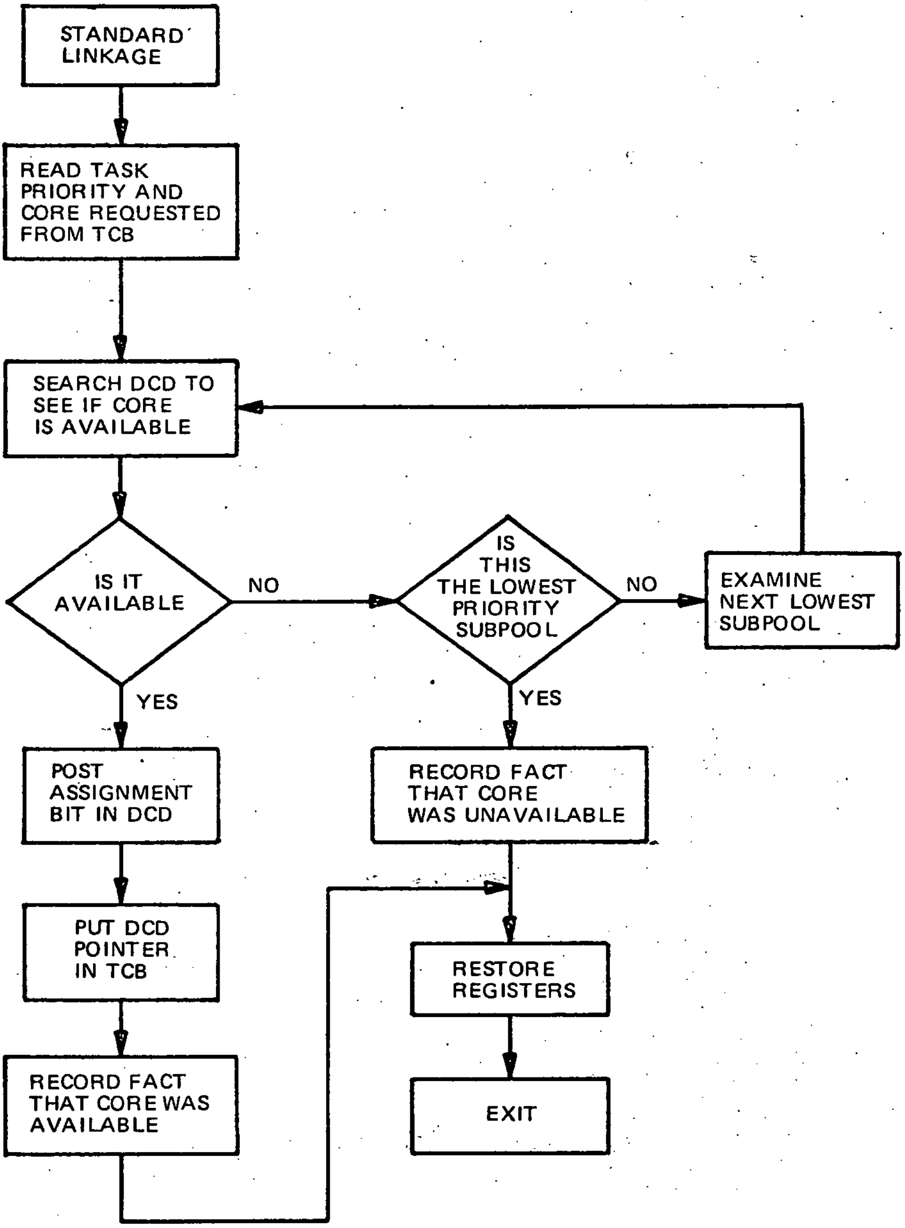

STANDARD LINKAGE
READ TASK PRIORITY AND CORE REQUESTED FROM TCB
SEARCH DCD TO SEE IF CORE IS AVAILABLE
IS IT AVAILABLE
NO
IS THIS THE LOWEST PRIORITY SUBPOOL
NO
EXAMINE NEXT LOWEST SUBPOOL
YES
POST ASSIGNMENT BIT IN DCD
PUT DCD POINTER IN TCB
RECORD FACT THAT CORE WAS AVAILABLE
YES
RECORD FACT THAT CORE WAS UNAVAILABLE
RESTORE REGISTERS
EXIT

Figure 4.7 Flowchart of Resource Allocator

dedicated to each priority level. When the allocator is entered with a request for x words of dynamic memory for task A at priority k, it determines if x continuous words are currently unused in the pool associated with priority k or any lower priority. If the memory is unavailable, the task is put in the wait state pending memory release.

A task may not request additional memory during its execution. All memory allocation is granted to a task only upon initiation. However, this task may release all of its dynamic memory at any time during execution to economize in the use of this resource.

4.1.5 The Cyclic Sequencer

The cyclic sequencer is operated as a task scheduled via the timed wait queue. It is put on the ready minor cycle with priority 1. It contains cyclic control tables (CCT) identifying a list of all cyclic computations and the frequency at which each must be executed. These computations are executed as subroutines of the cyclic sequencer, and hence, their execution time must be fitted to the minor cycle time interval. The cyclic subroutines are considered the system's foreground computations, and in turn, they may schedule other tasks to be executed in the background at priority 3, 4, or 5.

An entry of the CCT is shown in Figure 4.8. It contains the address of the subroutine to be executed; the frequency setting indicating the frequency in an integral number of minor cycles at which the subroutine is to be executed; and a frequency count. The count contains the number of minor cycles since the subroutine was last executed. It is incremented in each minor cycle and zeroed when the subroutine executes. In addition, there are pointers to the I/O commands for each subroutine. Frequency settings may be dynamically changed by the subroutine during flight via a supervisor call.

Upon entry into the cyclic sequencer, each CCT entry is examined. The frequency count is incremented by 1 and compared to the frequency setting. Should these entries be equal, the subroutine must be executed in this minor cycle. In this case, the frequency count is set to 0, and the subroutine's input commands are executed. To make the most efficient use of the channel this process is performed for each CCT entry before any subroutines are executed. Now each subroutine can be

program module address
frequency setting
frequency count
address of input commands
address of output commands

←1 full word→

Figure 4.8 Format of CCT Entry

executed, and at its completion its output commands are executed. This algorithm is presented in Figure 4.9.

During execution a subroutine must wait for its input requests to be completed before continuing its execution. The cyclic sequencer algorithm chosen minimizes the time that the subroutine may have to wait.

The subroutines to be executed in a given minor cycle are run in the order in which they appear in the CCT. Two subroutines executed at the same frequency may be run out of phase by initially biasing their frequency counts. For example, if subroutines A, B, and C are executed every 8 minor cycles, and if A and C are not to be run in the same minor cycle, the CCT entries may be initially set as shown.

	frequency setting	frequency count
A	8	0
B	8	0
C	8	4

The result is that A and B are run in that order every 8 minor cycles. C is also run at that same frequency although it is 4 minor cycles out of phase with A and B.

If a subroutine's execution time is too long, it must be broken into several smaller subroutines so as not to overload the system during any one minor cycle interval. Each of the smaller subroutines runs at the same frequency and must run in successive minor cycles. As in the above example, this can be accomplished by initial biasing of the frequency counts. For example, presume that A must be executed every 4 minor cycles and is organized into 3 parts A_1, A_2, and A_3 with an entry made in the CCT for each piece.

	frequency setting	frequency count
A_1	4	4
A_2	4	3
A_3	4	2

By phasing the frequency count in the initial conditions, computation A is run in 3 successive minor cycles: A_1 in the first, A_2 in the second, and A_3 in the third. Each has a frequency of execution of 4 minor cycles.

To prevent a system overload during a minor cycle some percentage of the CPU and I/O channel's time should be reserved for foreground computations. The remaining time will be devoted to executive overhead and background computations. Should the cyclic sequencer be placed on the ready queue before its previous execution has terminated, a software error condition results because of the overload.

Since foreground subroutines can schedule background tasks, it is necessary to have a method of preventing a program module from being scheduled as a task before its previous execution is finished. This prevention can be accomplished by several methods, one of which is to use event handling. For example; let A be a foreground routine and B a module which is scheduled for execution by A under one condition. Since A is cyclic, caution must be used in the method selected for invoking the execution of B.

B may be scheduled as a task through the use of a SCHEDULE SVC. If B has not completed execution prior to A scheduling B again, it is possible for two tasks to be in the system associated with module B. This condition will occur, for example, if B enters the wait state for a sufficiently long time interval.

As a solution to this problem define Q to be a latched event associated with the condition that B should be executed. Let A be structured to signal event Q on when it detects that B should be scheduled.

At phase initiation the start up routine will schedule B on event Q. This will establish B as a task in the wait state until Q is signalled on. Eventually when A signals Q on, B can be executed.

Task B can then be re-established in two ways:

1) B can avoid termination until mission phase transition by having a structure looping upon itself as shown in Figure 4.10a. Whenever Q is signalled on by A, B is again executed. At phase transition time A and B can be terminated.

2) B can reschedule itself as a task prior to its termination as shown in Figure 4.10b.B remains in the wait state until A signals Q on.

In either case, two concurrent executions of program module B are avoided.

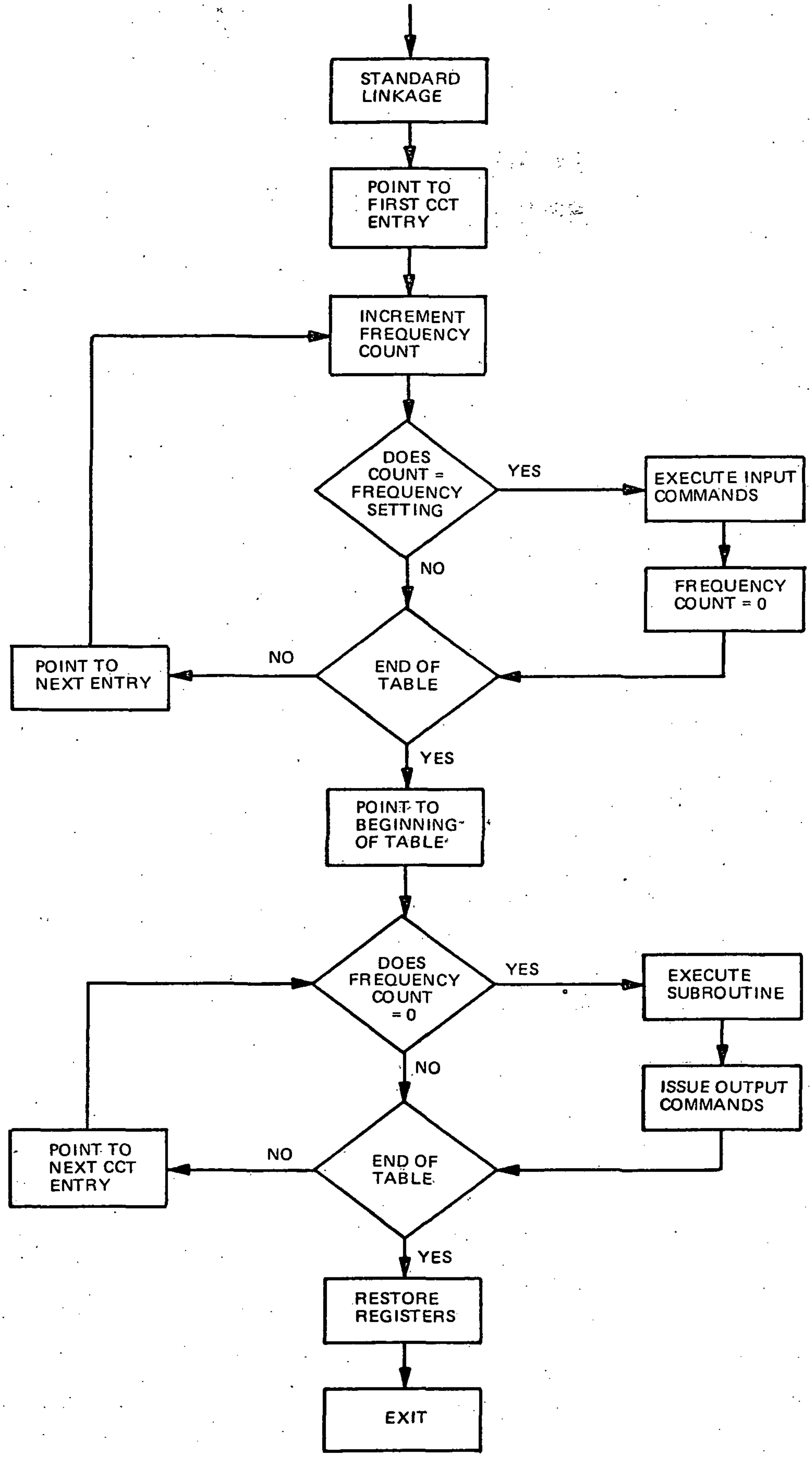


Figure 4.9 Flowchart of Cyclic Sequencer

```
B: PROCEDURE
Begin: Wait for QON;
  Signal Q Off;

         .
         .
         .

  Go to Begin;
       END
```

Figure 4.10a Re-establishing Background Task

```
B:  PROCEDURE
Begin: Signal Q Off;

         .
         .
         .

Schedule B on Event Q;
       END
```

Figure 4.10b Re-establishing Background Task

The cyclic sequencer is the only priority 1 task in the system. Thus, the dynamic core in the priority 1 subpool is not shared with any other tasks and can be considered statically assigned to the cyclic sequencer. To reduce overhead this core should not be allocated through the resource allocator. There need only be subpools for priorities 3, 4, and 5.

4.1.6 Supervisor Service Routines

Upon the execution of a supervisor call, a PWS associated with the supervisor interrupt becomes the new PSW. This PSW will enter a general SVC routine to determine which executive service routine to execute and then to branch to this routine. The flowchart for this process is shown in Figure 4.11. In addition, the flowcharts for the task management supervisor service routines listed in Section 4.1.1 will also be presented in Figs. 4.12-4.21.

Certain SVC's require parameters to be supplied to the executive. For example, SCHEDULE requires the name of the program module that is to be scheduled as a task. A list of the necessary parameters is supplied in Chapter 9.

a) FREEMAIN (SVC 1) - The purpose of this SVC is to enable a task to release all of its dynamic memory during execution.

b) SECURE (SVC 2) - This SVC enables a task to lock parts of the compool for reading or updating. If a copy of parts of the compool are to be created, the task must supply the copy area from its core allocation. It does so by providing a pointer to the copy area as a parameter with the SVC. Should the task have to wait for compool access, it does so in a partially completed state. The PSW and registers stored in the TCB must correspond to a point in the coding at which execution is to continue when the task becomes active again.

c) RELEASE (SVC 3) - To close an update block the locks established by the active task must be released. This SVC does so by referencing the parameter list supplied by the SECURE SVC. The pointer to this list is in the active task's TCB. This list contains the addresses of each lock and the type of lock established by SECURE. Any necessary updating of the compool is done and then all locks released.

d) COPY (SVC 4) - The SVC copies parts of the compool into a part of the active task's work area. It enables the active task to read parts of the compool without having to keep locks established for long time intervals. This SVC would be used instead of an Update block if the active task wanted to use compool data for long periods of time, but yet did not want to prevent other tasks from updating the data.

e) LINK (SVC 5) - The LINK SVC allows a task to create a dependent task and await this task's completion before allowing its own execution to continue. Should the dependent task abort because of an error, the calling task also aborts; and if this latter task was scheduled via a LINK, the task that scheduled it also aborts, etc.

f) END (SVC 6) - Upon termination a task returns to the dispatcher via the return address in GR 14. The dispatcher puts the CPU in the supervisor state by executing the END SVC. The END SVC performs several bookkeeping functions for the executive. It closes any update block that is still open, frees dynamic memory still held by the task, and puts the task in the wait state until any pending I/O requests are completed. Upon termination it returns to the dispatcher.

g) SCHEDULE (SVC 7) - This SVC allows the active task to schedule another task without establishing a task dependence, as in the case of LINK. The schedule can be unconditional or conditional upon some set of criteria. These criteria include:

   1) scheduling on some software event or events occurring;

   2) scheduling at some specific time; and

   3) scheduling after some time interval has elapsed.

   These criteria are analagous to the types of scheduling available within HAL [7,8].

h) WAIT (SVC 8) - The WAIT SVC allows the active task to place itself in the wait state pending the occurrence of some event or group of events. The allowable events are:

   1) waiting for some software event or events being signalled on (posted);

   2) waiting until a specific time occurs; and

   3) waiting for a time interval to elapse [7,8].

i) SIGNAL (SVC 9) - Signal turns an event on or off, depending upon the parameters supplied by the SVC. When an event is turned on,the scheduler is called to place any tasks awaiting the event in the ready state, if possible.

j) TEST EVENT (SVC 10) - The status of the event tested is returned to the active task via a flag which is set by the executive. The active task supplies a pointer to the flag as an SVC parameter.

k) CHANGE CCT (SVC 11) - This SVC enables a task to change the entries in the CCT as mission phase requirements change. Direct updating of the CCT by tasks is illegal and should be checked for during system simulation.

l) DISPATCH CHECK (SVC 12) - This SVC occurs at program segment points. It returns control to the executive to check if a higher priority is waiting for the CPU. If so, the previously active task is put in the ready state, and the new higher priority task is made active (via the dispatcher).

4.2 Timer Interrupt

When the value of the EP timer goes from positive to negative, an external interrupt is generated. This interrupt is used to signal the start of a new minor cycle. The executive coding associated with the timer interrupt will first reset the timer to interrupt at the start of the next minor cycle and then service the mission clock. A check is then made to be sure the cyclic sequencer terminated the last minor cycle. If not, a software overload condition exists, and a program error condition results. The cyclic sequencer's TCB is now formatted and put at the top of the ready queue, and the previously active task is put in the wait state.

All other tasks awaiting a timed event are checked every N minor cycles to see if they can now be made ready. The timer entry in the TCB contains the time at which the task can be put in the ready state. When the system clock equals or exceeds this time, the task can be made ready for execution. The value of N is a system parameter. Its value must be an integer greater than or equal to 1 depending upon the system response time desired.

To expedite checking the time wait queue, TCB entries on the queue will be arranged in terms of increasing time at which they can be made ready. That is, suppose task A can be put on the ready queue at time x, and task B at time y. If x<y, the TCB for task A must preceed the TCB for B on the time wait queue. As a result, it is not necessary to examine the entire queue whenever it is serviced every N minor cycles. Checking of entries can stop when the timer entries in the TCBs exceed the value of the system clock used for comparison. At its completion the timer interrupt routine exits to the dispatcher. The flowchart for the above algorithm is given in Figure 4.22.

4.3 Deadlock Detection

As explained in Chapter 2, the algorithms for resource allocation avoid incremental allocations, and hence avoid deadlock. However, the SIGNAL and WAIT supervisor calls introduce the possibility of deadlocked tasks. For example, if task A contains the SVCs, WAIT M and SIGNAL N, in that order, it goes into the wait state until M occurs. Now suppose task B contains the SVCs, WAIT N and SIGNAL M. It too goes into the wait state, and if a third task does not signal M or N, tasks A and B are deadlocked.

Judicious program design can, of course, avoid this problem. However, an alternative approach is to periodically check for deadlocked tasks. If the time at which each task goes into the wait state is stored in its TCB, a low priority task could periodically check these times. If a certain time criterion was exceeded, the waiting task would be considered deadlocked, and error recovery would commence.

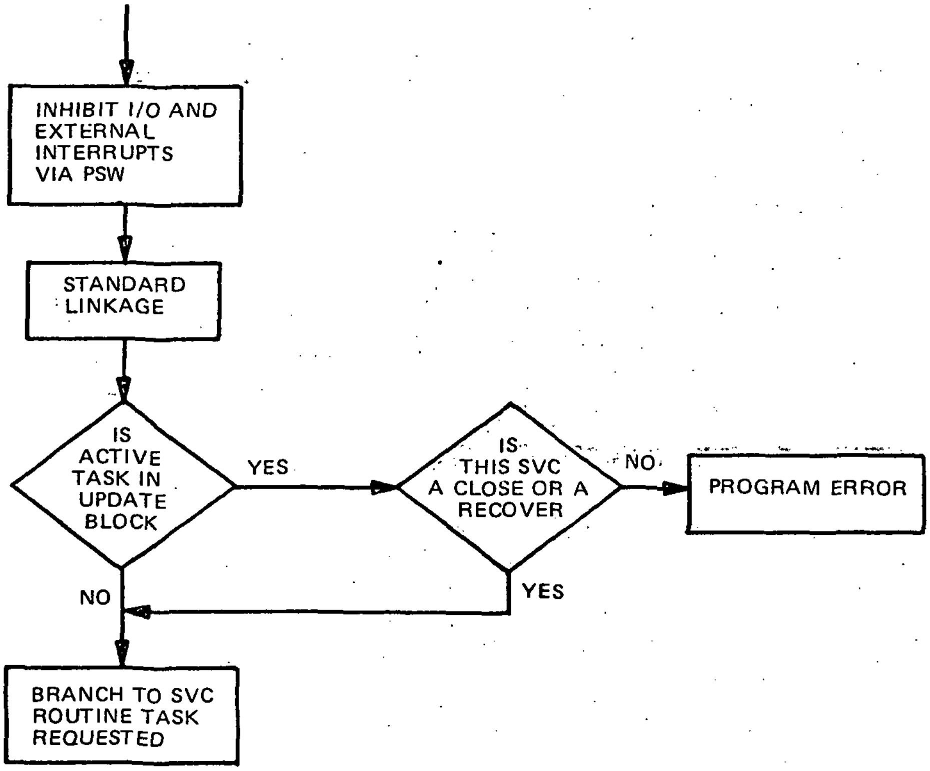

INHIBIT I/O AND EXTERNAL INTERRUPTS VIA PSW
STANDARD LINKAGE
IS ACTIVE TASK IN UPDATE BLOCK
YES
IS THIS SVC A CLOSE OR A RECOVER
NO
PROGRAM ERROR
YES
NO
BRANCH TO SVC ROUTINE TASK REQUESTED

Figure 4.11 Flowchart of SVC Interrupt Routine

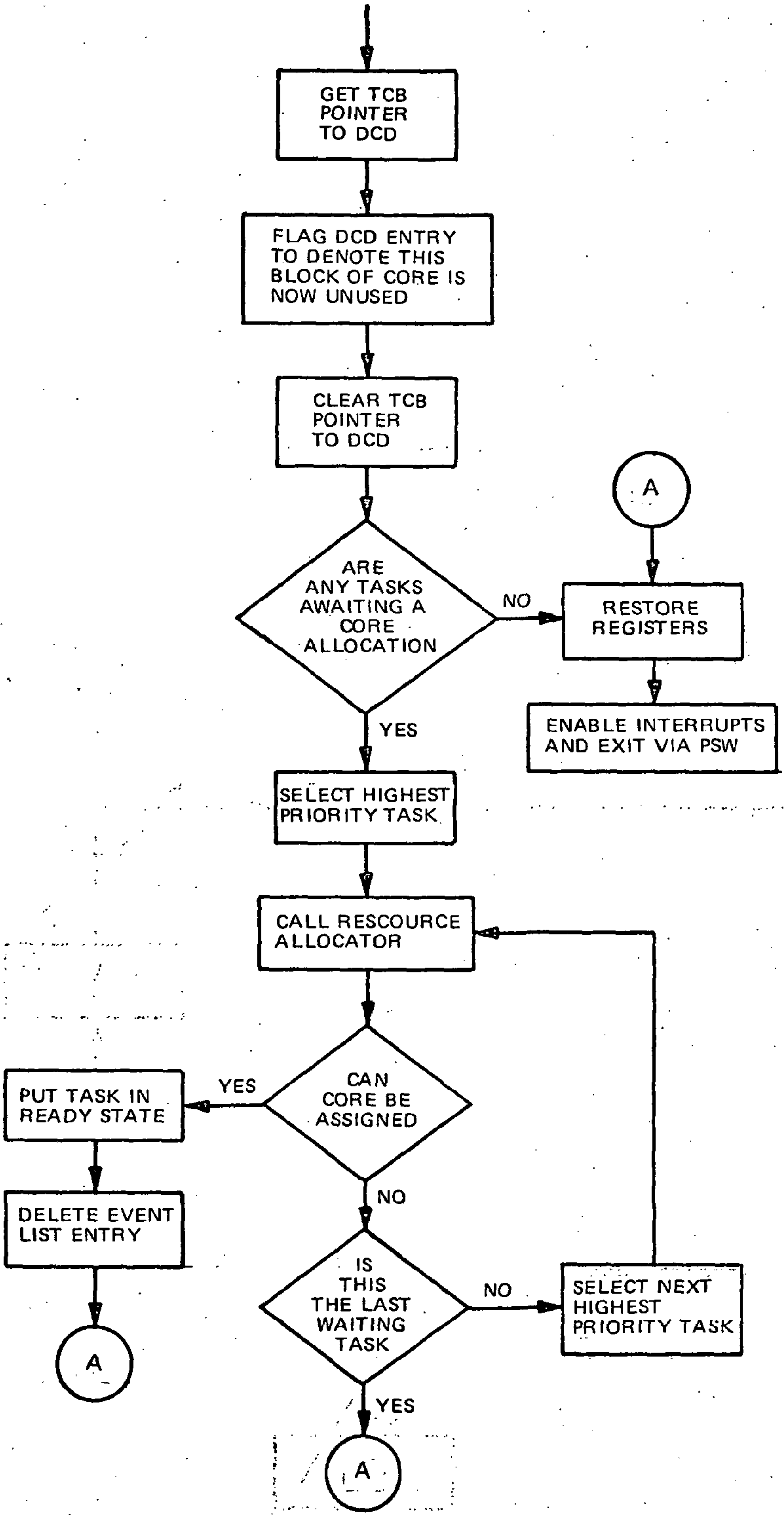

GET TCB POINTER TO DCD
FLAG DCD ENTRY TO DENOTE THIS BLOCK OF CORE IS NOW UNUSED
CLEAR TCB POINTER TO DCD
ARE ANY TASKS AWAITING A CORE ALLOCATION
NO
RESTORE REGISTERS
A
ENABLE INTERRUPTS AND EXIT VIA PSW
YES
SELECT HIGHEST PRIORITY TASK
CALL RESCOURCE ALLOCATOR
CAN CORE BE ASSIGNED
YES
PUT TASK IN READY STATE
DELETE EVENT LIST ENTRY
A
NO
IS THIS THE LAST WAITING TASK
NO
SELECT NEXT HIGHEST PRIORITY TASK
YES
A

Figure 4.12 Flowchart of FREEMAIN SVC

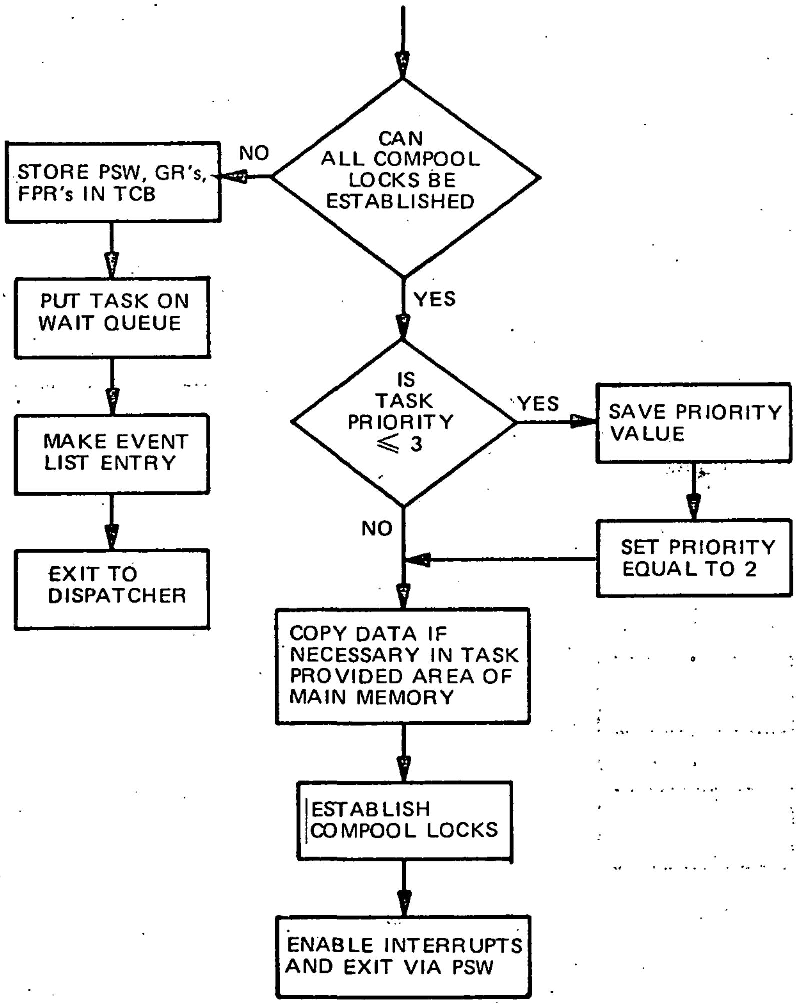

CAN ALL COMPOOL LOCKS BE ESTABLISHED
NO
STORE PSW, GR's, FPR's IN TCB
PUT TASK ON WAIT QUEUE
MAKE EVENT LIST ENTRY
EXIT TO DISPATCHER
YES
IS TASK PRIORITY ≤ 3
YES
SAVE PRIORITY VALUE
SET PRIORITY EQUAL TO 2
NO
COPY DATA IF NECESSARY IN TASK PROVIDED AREA OF MAIN MEMORY
ESTABLISH COMPOOL LOCKS
ENABLE INTERRUPTS AND EXIT VIA PSW

Figure 4.13 Flowchart of SECURE SVC

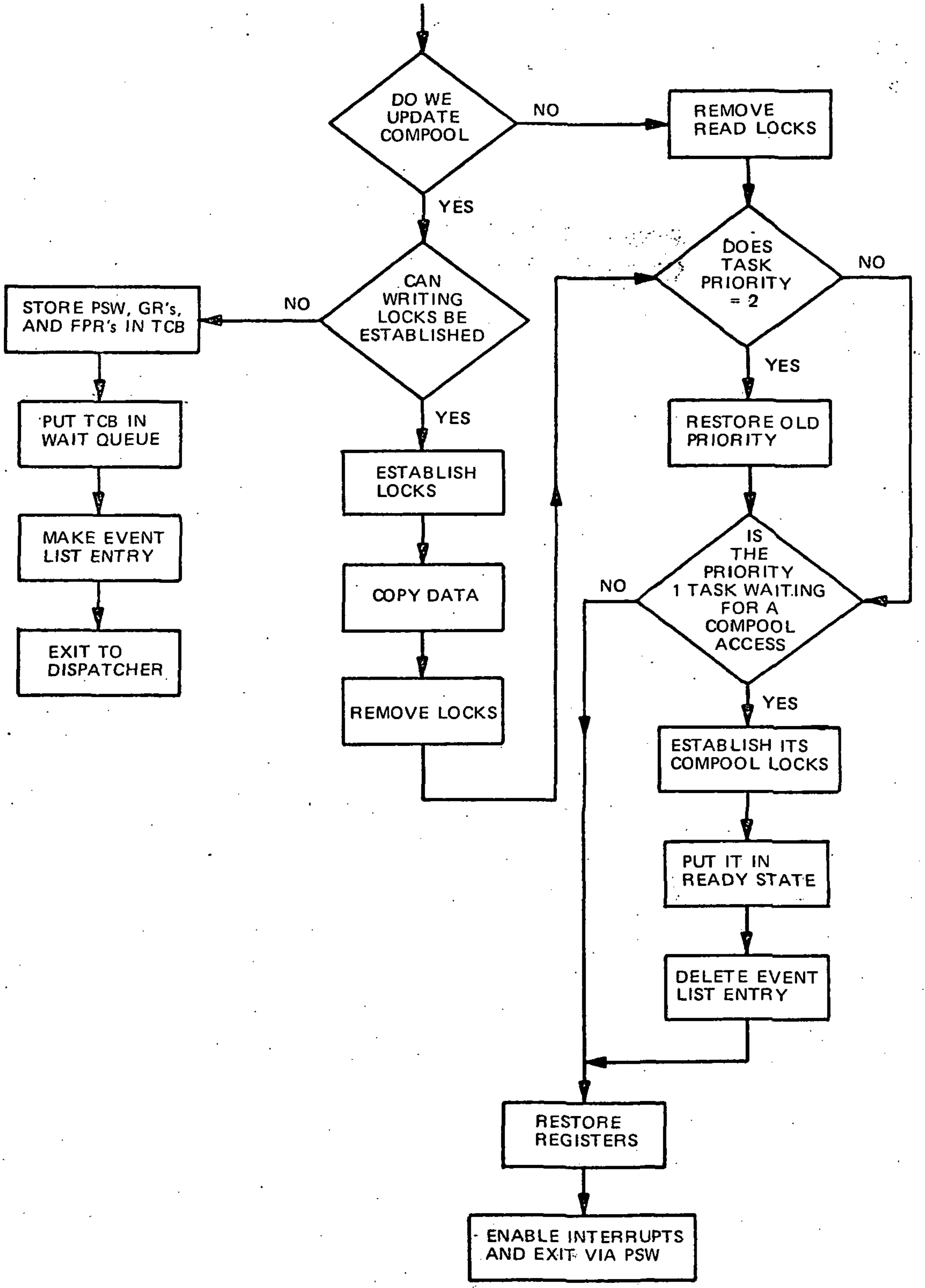


Figure 4.14 Flowchart of RELEASE SVC

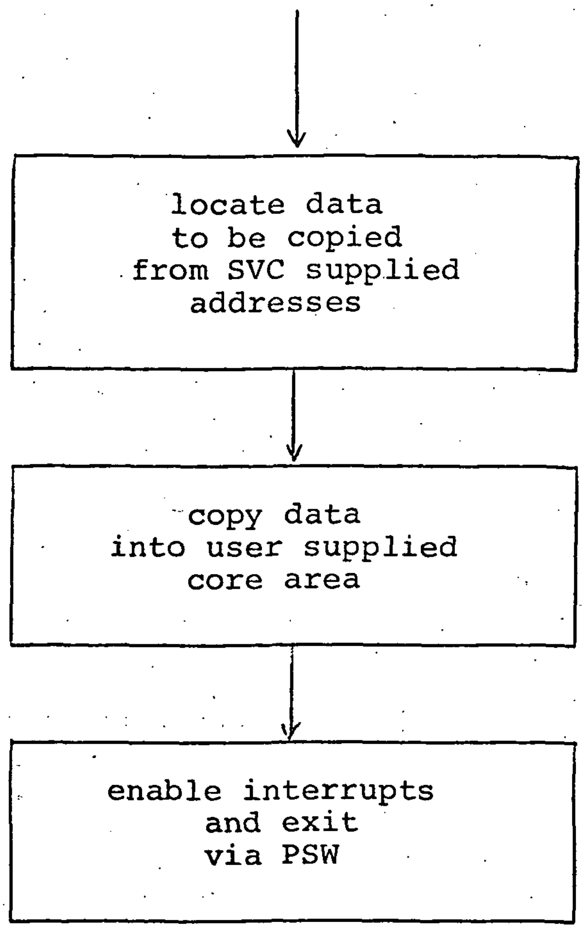


Figure 4.15 Flowchart of COPY SVC

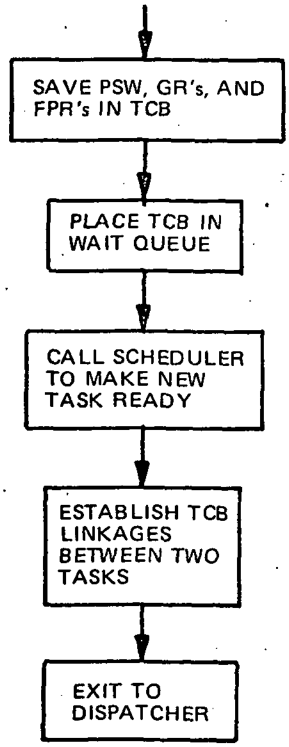

SAVE PSW, GR's, AND FPR's IN TCB
PLACE TCB IN WAIT QUEUE
CALL SCHEDULER TO MAKE NEW TASK READY
ESTABLISH TCB LINKAGES BETWEEN TWO TASKS
EXIT TO DISPATCHER

Figure 4.16 Flowchart of LINK SVC

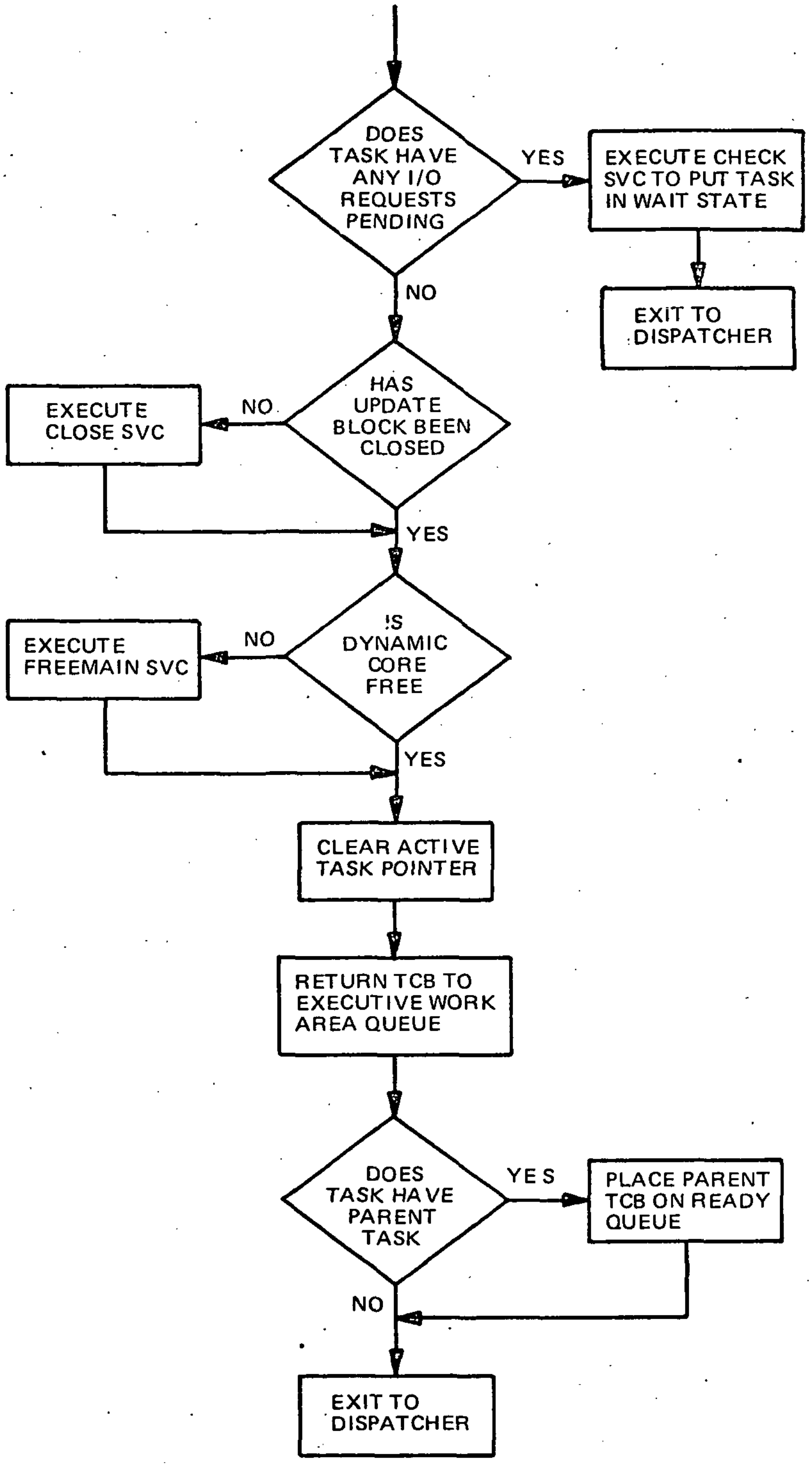

DOES TASK HAVE ANY I/O REQUESTS PENDING
YES
EXECUTE CHECK SVC TO PUT TASK IN WAIT STATE
EXIT TO DISPATCHER
NO
HAS UPDATE BLOCK BEEN CLOSED
NO
EXECUTE CLOSE SVC
YES
IS DYNAMIC CORE FREE
NO
EXECUTE FREEMAIN SVC
YES
CLEAR ACTIVE TASK POINTER
RETURN TCB TO EXECUTIVE WORK AREA QUEUE
DOES TASK HAVE PARENT TASK
YES
PLACE PARENT TCB ON READY QUEUE
NO
EXIT TO DISPATCHER

Figure 4.17 Flowchart of END SVC

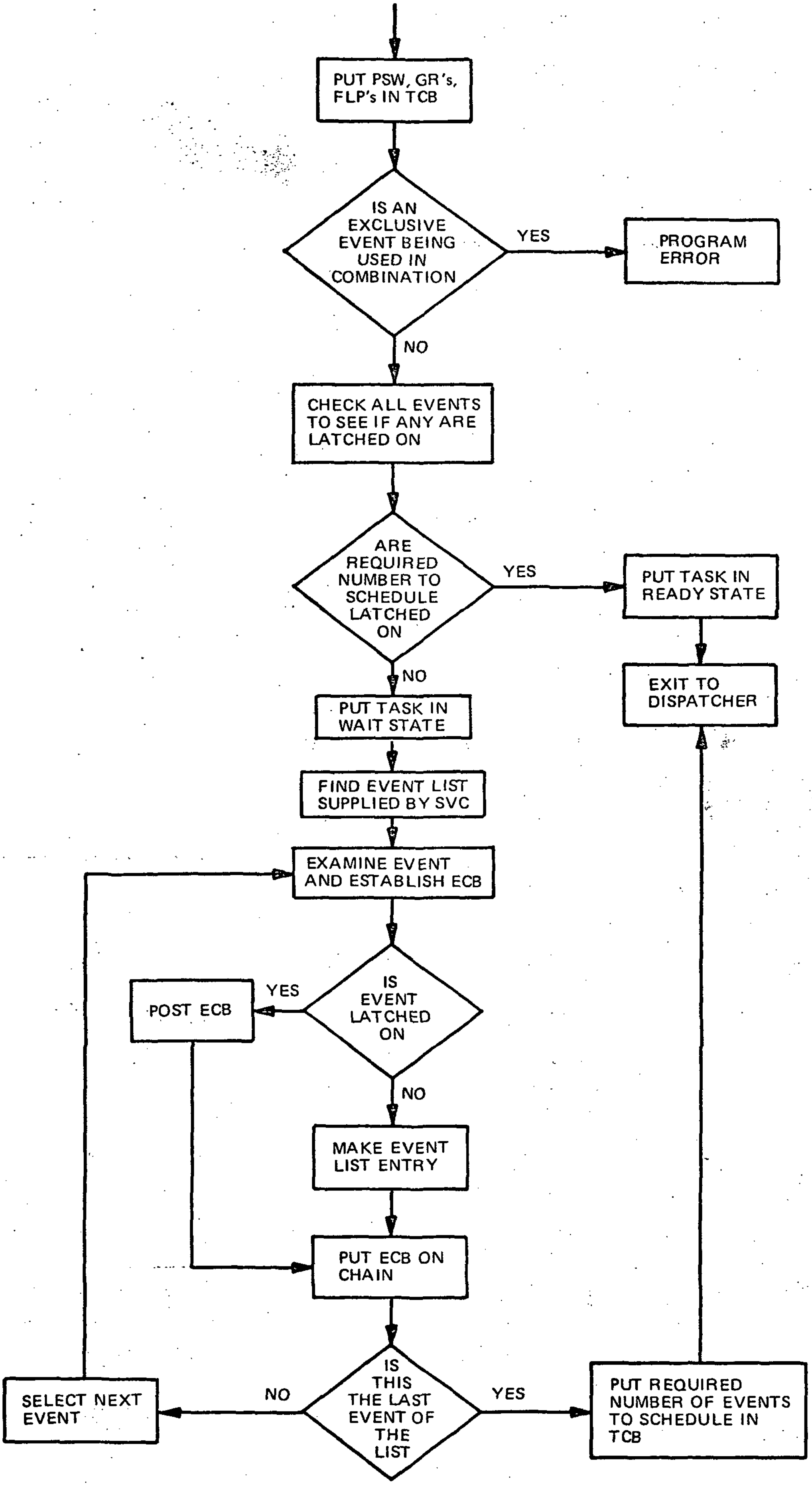

PUT PSW, GR's, FLP's IN TCB
IS AN EXCLUSIVE EVENT BEING USED IN COMBINATION
YES
PROGRAM ERROR
NO
CHECK ALL EVENTS TO SEE IF ANY ARE LATCHED ON
ARE REQUIRED NUMBER TO SCHEDULE LATCHED ON
YES
PUT TASK IN READY STATE
EXIT TO DISPATCHER
NO
PUT TASK IN WAIT STATE
FIND EVENT LIST SUPPLIED BY SVC
EXAMINE EVENT AND ESTABLISH ECB
IS EVENT LATCHED ON
YES
POST ECB
NO
MAKE EVENT LIST ENTRY
PUT ECB ON CHAIN
IS THIS THE LAST EVENT OF THE LIST
NO
SELECT NEXT EVENT
YES
PUT REQUIRED NUMBER OF EVENTS TO SCHEDULE IN TCB

Figure 4.18 Flowchart of WAIT SVC

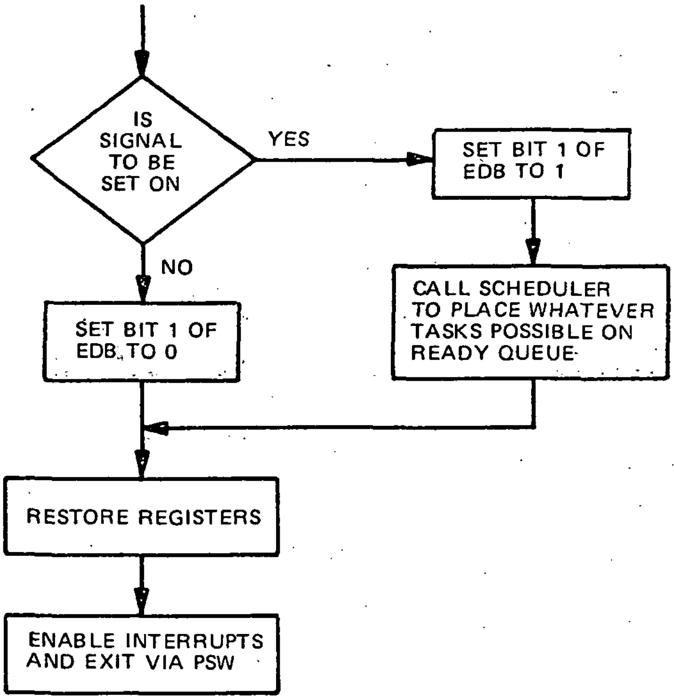

IS SIGNAL TO BE SET ON
YES
NO
SET BIT 1 OF EDB TO 1
CALL SCHEDULER TO PLACE WHATEVER TASKS POSSIBLE ON READY QUEUE
SET BIT 1 OF EDB TO 0
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW

Figure 4.19 Flowchart of SIGNAL SVC

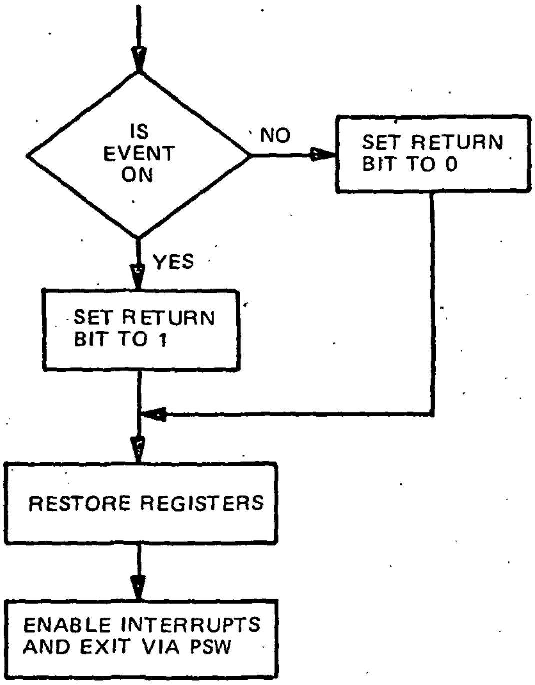


Figure 4.20 Flowchart of TEST EVENT SVC

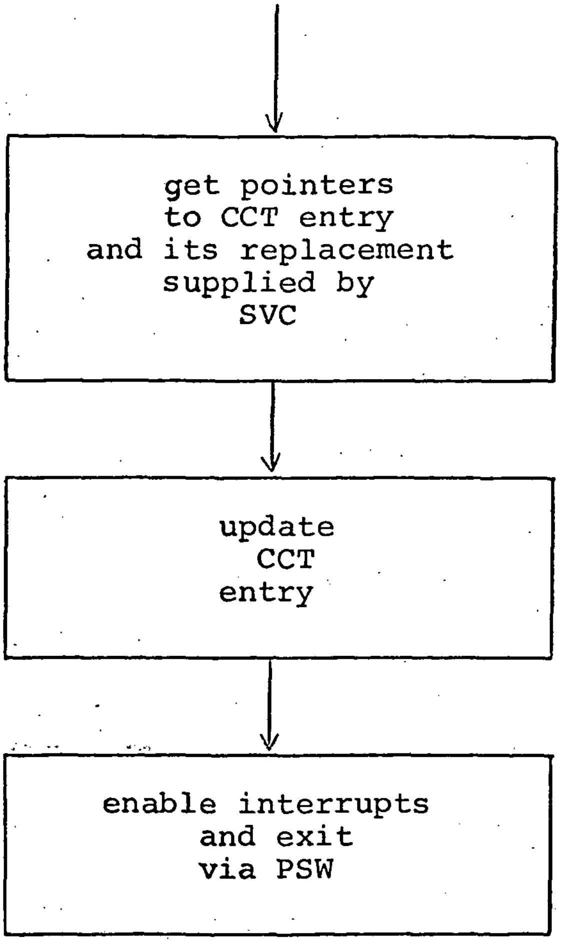

get pointers to CCT entry and its replacement supplied by SVC
update CCT entry
enable interrupts and exit via PSW

Figure 4.21 Flowchart of CHANGE CCT SVC

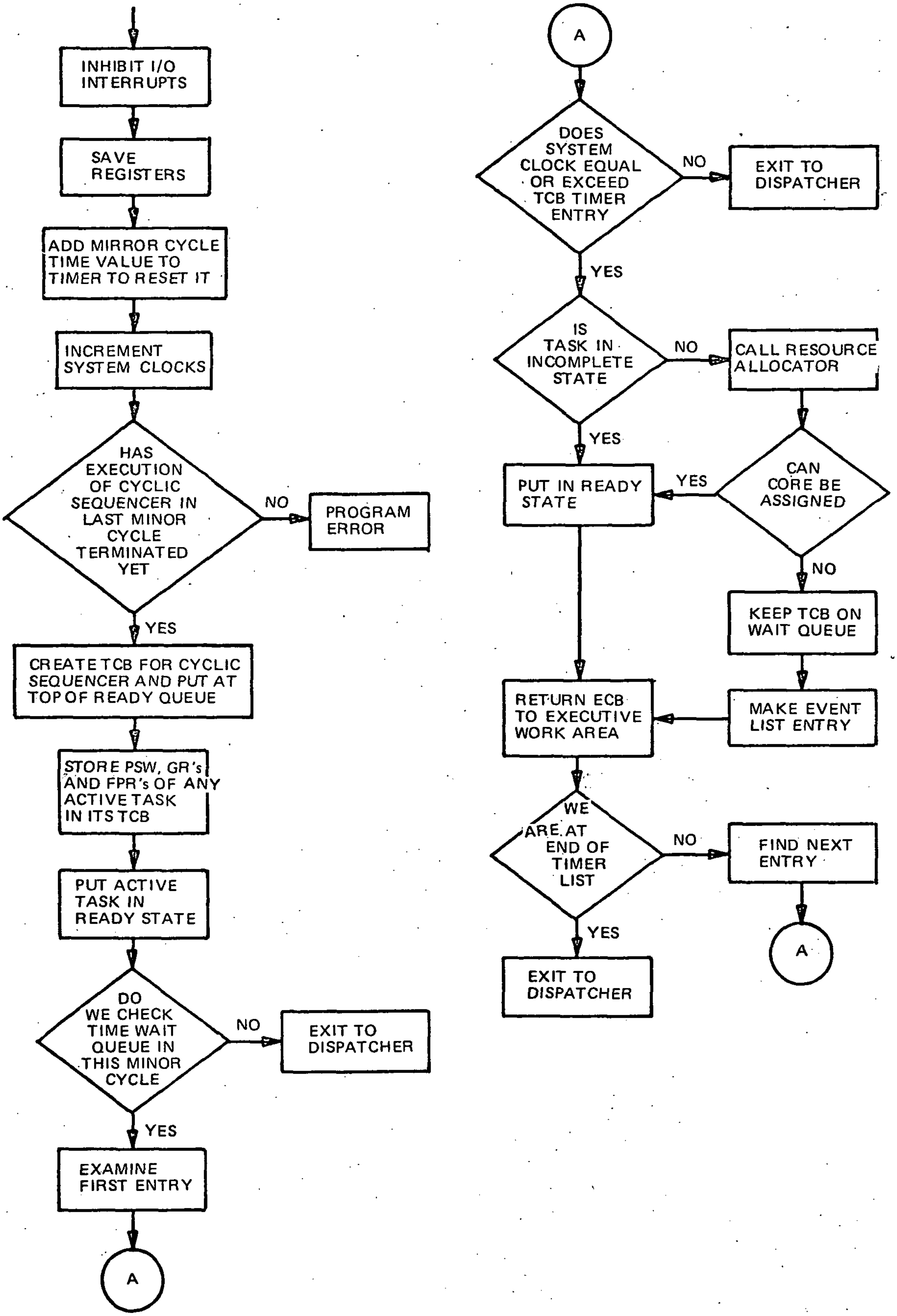

INHIBIT I/O INTERRUPTS
SAVE REGISTERS
ADD MIRROR CYCLE TIME VALUE TO TIMER TO RESET IT
INCREMENT SYSTEM CLOCKS
HAS EXECUTION OF CYCLIC SEQUENCER IN LAST MINOR CYCLE TERMINATED YET
NO
PROGRAM ERROR
YES
CREATE TCB FOR CYCLIC SEQUENCER AND PUT AT TOP OF READY QUEUE
STORE PSW, GR's AND FPR's OF ANY ACTIVE TASK IN ITS TCB
PUT ACTIVE TASK IN READY STATE
DO WE CHECK TIME WAIT QUEUE IN THIS MINOR CYCLE
NO
EXIT TO DISPATCHER
YES
EXAMINE FIRST ENTRY
A
A
DOES SYSTEM CLOCK EQUAL OR EXCEED TCB TIMER ENTRY
NO
EXIT TO DISPATCHER
YES
IS TASK IN INCOMPLETE STATE
NO
CALL RESOURCE ALLOCATOR
YES
PUT IN READY STATE
YES
CAN CORE BE ASSIGNED
NO
KEEP TCB ON WAIT QUEUE
MAKE EVENT LIST ENTRY
RETURN ECB TO EXECUTIVE WORK AREA
WE ARE AT END OF TIMER LIST
NO
FIND NEXT ENTRY
A
YES
EXIT TO DISPATCHER

Figure 4.22 Flowchart of Timer Interrupt Software

Chapter 5

I/O Management Functions

5.1 Introduction

The input/output control function of the executive provides supervision of all I/O operations in the system. The design of this part of the software reflects the specific requirements of the Space Shuttle avionics system.

Current Phase B concepts are based on interfacing the computer to onboard subsystems via a common data bus of up to 10^6 bits per second data rate capability. The computer's I/O section will be connected to a bus control unit whose function is to command the bus system. All subsystems are connected to the main bus through a standard interface unit which supplies standard digital format of data and commands. The bus system will contain redundant paths to achieve the FO/FO/FS requirement. The final design of the data bus system is a crucial aspect of the avionics system design and directly effects the computer software.

Indeed, this part of the executive software design is the most hardware sensitive. We are, of course, directing our design toward the 4 Pi EP computer, and this fact influences our algorithms. In particular, we will make use of the I/O interrupt, channel command, and channel test conventions of the EP in the design.

5.2 Definition of I/O Management Functions

There are three basic I/O management functions. These are: READ, WRITE, and CHECK. These functions are essentially interfaces between application tasks and executive I/O services. As with task management services they are executed by means of SVCs with parameter lists. Further details about the parameters are in Chapter 9.

a) READ (SVC 13) - The purpose of this SVC is to input data into main memory from the MMU or one of the avionics subsystems. It queues an I/O request to the I/O channel and then returns control to the active task.

b) WRITE (SVC 14) - This SVC outputs data from main memory to the MMU or one of the avionics subsystems. As with READ, it queues an I/O request to the I/O channel and then returns control to the active task.

In the case of READ or WRITE, the active task may continue processing and may at some point wait for an I/O operation to be completed by executing CHECK.

c) CHECK (SVC 15) - An active task may place itself in the wait state until a particular I/O operation is completed by executing CHECK. Should the operation have been completed when this SVC is executed, the active task continues processing.

5.3 I/O Queues and Control Blocks

Since the I/O channel may have several requests pending while it is performing some operation, a queue of I/O requests is necessary. The elements of this queue are I/O Request Blocks linked with a threaded list structure. The format of an IORB is shown in Figure 5.1. An IORB contains all the information necessary for the channel to perform the desired I/O operation. This information includes:

a) channel, subchannel and device addresses;

b) task priority;

c) number of bytes of data to be transmitted;

d) device command;

e) if device is MMU, a data address;

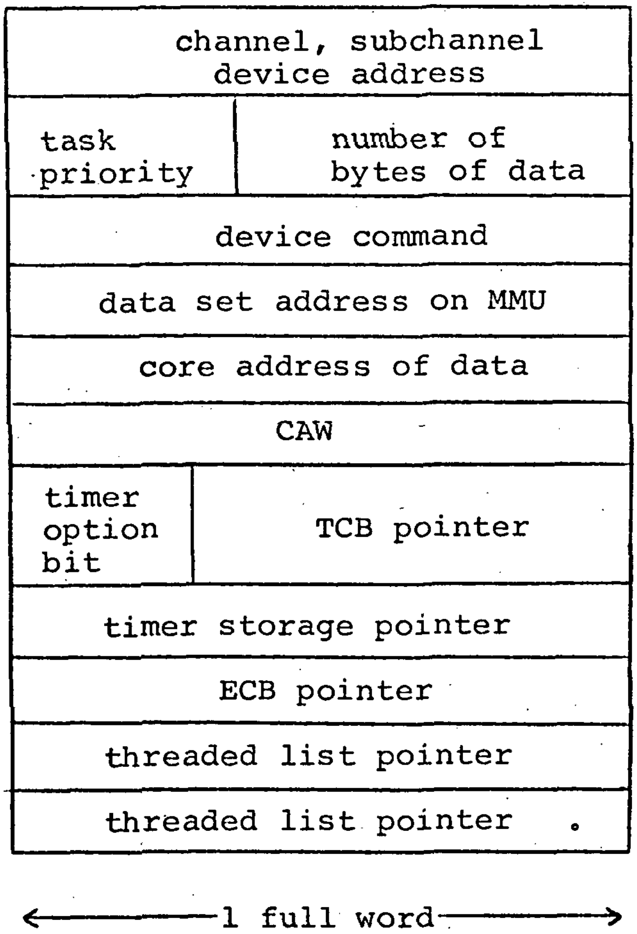


Figure 5.1 Format of an IORB

f) main memory data address;

g) CAW;

h) pointer to the TCB;

i) a flag to be set if the value of the timer is to be returned in a read operation;

j) a pointer to where this timer is to be stored;

k) a pointer to the ECB for the I/O operation;

l) threaded list pointers.

IORBs are dynamically created by the executive when a task performs an I/O operation. The necessary core storage for the IORB is taken from the executive's work area as described in Chapter 3. This control block is then placed on the channel's queue on a priority basis with its priority equal to the priority of the active task. Thus, a high priority task's I/O commands are executed before those of a low priority task.

Associated with each READ or WRITE is an ECB located in the program module's coding or established in the task's dynamic core (in the case of a reentrant module). This ECB is posted upon completion of the I/O operation by the I/O interrupt supervisor. This posting enables CHECK to perform its desired function of determining whether a particular I/O command is completed. In addition, by binding the ECB to the given I/O command, a particular READ or WRITE must be completed before this same command can be executed again. However, should this latter condition occur, the READ or WRITE will place the task in the wait state until its ECB is posted. Then the command can be executed again.

5.4 The I/O Supervisor

The I/O supervisor is an executive routine called when an I/O interrupt occurs. Upon occurrence of the interrupt the current PSW is saved and a PSW associated with the interrupt becomes the new PSW, as explained in Chapter 2. The new PSW gives CPU control to the I/O supervisor.

The I/O supervisor first checks for successful completion of the last I/O operation. If an error occurred, an error recovery routine will be called. The error recovery performed will be a function of the type of error encountered. System reconfiguration might then be necessary. If the operation was successfully completed, any task awaiting the I/O operation is put in the ready state. The next pending I/O request is then selected, the channel program is formatted; and the channel is activated.

A task issuing a read command has the option of having the data time tagged when it is read into core. That is, the value of the timer at input time can also be stored by the I/O supervisor in core in a location the task specified with the read command. This time value is of importance to certain numerical integration algorithms. The I/O supervisor is responsible for returning the timer value to the task (see Figure 5.2).

5.5 I/O Service Routines

The algorithms and flowcharts for the three I/O SVCs mentioned in section 5.1 will be presented here and in Figs. 5.3-5.5.

The read and write routines each format the IORB to be queued to the channel's list of I/O requests. Queueing is done on a priority basis with the priority of the task becoming the priority of the IORB. When the MMU is the device to be read or written upon, a secondary storage routine is called to locate the data set and convert the physical record requested into an actual MMU address, which is put into the IORB. This routine is explained in Chapter 7.

When the channel is not busy, the READ or WRITE SVC takes the IORB, formats the channel program, and activates the channel. Otherwise, channel activation is only done by the I/O supervisor.

If data is to be read into core, the core address specified must be checked to be sure it is not a protected area. For example, an address in the compool is not allowed. This checking of protected addresses requires buffering of data whose involvement with I/O operations can cause conflicts between tasks. When data in the compool is to be inputed or outputed, the requesting task must access the data via the executive and use part of its working core as a buffer. No direct I/O operations are allowed in the compool. In addition, the physical address of the device to be read or written upon is found in a device table maintained by the configuration management routines. This table is called the Redundant Equipment Table and will be described in Chapter 6. Should a device fail and a spare be used to replace it, the new device address is entered in this table for use by the I/O routines. Thus, any system reconfiguration will cause an update of this table.

5.6 Cyclic and Non-Cyclic I/O

A bus I/O transaction once initiated by the computer is independent of the computer software organization. The command/response addressed bus may be directed by a computer with either an asynchronous or synchronous software structure. The main difference will be in the scheduling and dispatching of I/O requests, and in the coordination of I/O with processing.

In the synchronous structure, I/O requests must be preplanned and interleaved with the task processing. I/O requests are dispatched in a list every minor cycle and carried out concurrently with task processing. A synchronous software structure requires a command response bus access method. A polling or contention access method would be difficult to run with a synchronous structure [8]. However, in an asynchronous structure, I/O is scheduled on a demand basis by the processing tasks. These I/O requests may also be carried out concurrent with task processing, but their scheduling and dispatching are non-deterministic.

The major distinction between cyclic and non-cyclic I/O in this executive system is that I/O done by the cyclic sequencer is table driven via the CCT. That is, the cyclic sequencer has tables of how frequently each I/O operation it performs must be done. Because of the high priority of the cyclic sequencer, the read/write routines will insert these requests at the beginning of the IORB queue to insure their completion before the next minor cycle interrupt. In addition, the percent of I/O channel usage by the cyclic sequencer must be limited. Sufficient time must be allowed for the channel to complete all I/O operations generated by cyclic computations before their next execution.

5.7 Configuration Dependent Features

The data bus system we are assuming is a high speed data transmission device which is primarily used for sampling measurements from avionics subsystems and sending computed information back to the subsystems. We are not designing the executive I/O system for devices such as printers or tape drives to be on the data bus.

The EP architecture features we have used in structuring the I/O management functions of the executive system are the following: the I/O interrupt, channel programs consisting of CCWs, the characteristics of the START I/O and TEST I/O instructions, and the CSW.

Computer control of the data bus is accomplished via the I/O channel. Since the EP allows no direct BCU control, the I/O channel sends commands to the BCU and receives returning information. Thus, the channel-BCU interface hardware must transform channel commands into a BCU command format. Other computers, such as the Hughes 230, allow more direct BCU control than the EP. Appendix A of this volume presents a study of operation and control of the data bus with such a computer.

5.8 I/O Error Correction

Upon the detection of an I/O error, via the CSW, the executive must perform several functions. First, the occurrence of the error must be reported in the record of the flight kept on the MMU. Next an indicator is flashed to the pilot, and finally a reconfiguration routine is called. The faulty equipment must be isolated and an inactive spare switched into the configuration to allow the mission to continue.

The BCU hardware can be structured to try an I/O transmission several times when an error is detected before reporting the error to the computer. In other words, the error can be made invisible to the computer and the executive until the BCU determines it cannot correct the error by retransmission of the I/O command. At this point, the BCU reports the error to the I/O channel, and the channel in turn formats the appropriate CSW.

A discussion of data bus error control is presented in Appendix B.

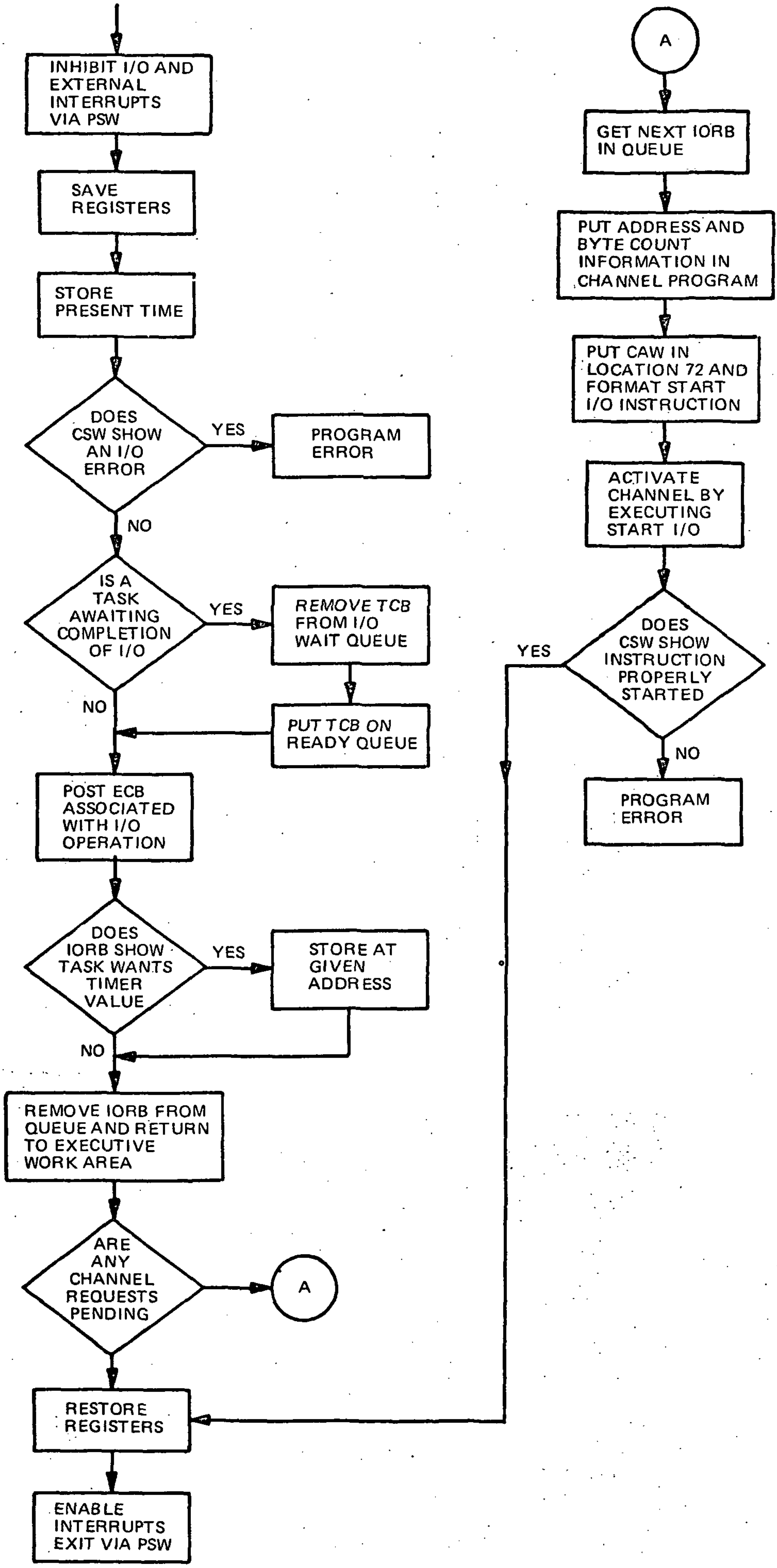

INHIBIT I/O AND EXTERNAL INTERRUPTS VIA PSW
SAVE REGISTERS
STORE PRESENT TIME
DOES CSW SHOW AN I/O ERROR
YES
PROGRAM ERROR
NO
IS A TASK AWAITING COMPLETION OF I/O
YES
REMOVE TCB FROM I/O WAIT QUEUE
PUT TCB ON READY QUEUE
NO
POST ECB ASSOCIATED WITH I/O OPERATION
DOES IORB SHOW TASK WANTS TIMER VALUE
YES
STORE AT GIVEN ADDRESS
NO
REMOVE IORB FROM QUEUE AND RETURN TO EXECUTIVE WORK AREA
ARE ANY CHANNEL REQUESTS PENDING
A
RESTORE REGISTERS
ENABLE INTERRUPTS EXIT VIA PSW
A
GET NEXT IORB IN QUEUE
PUT ADDRESS AND BYTE COUNT INFORMATION IN CHANNEL PROGRAM
PUT CAW IN LOCATION 72 AND FORMAT START I/O INSTRUCTION
ACTIVATE CHANNEL BY EXECUTING START I/O
DOES CSW SHOW INSTRUCTION PROPERLY STARTED
YES
NO
PROGRAM ERROR

Figure 5.2 Flowchart of I/O Interrupt Software

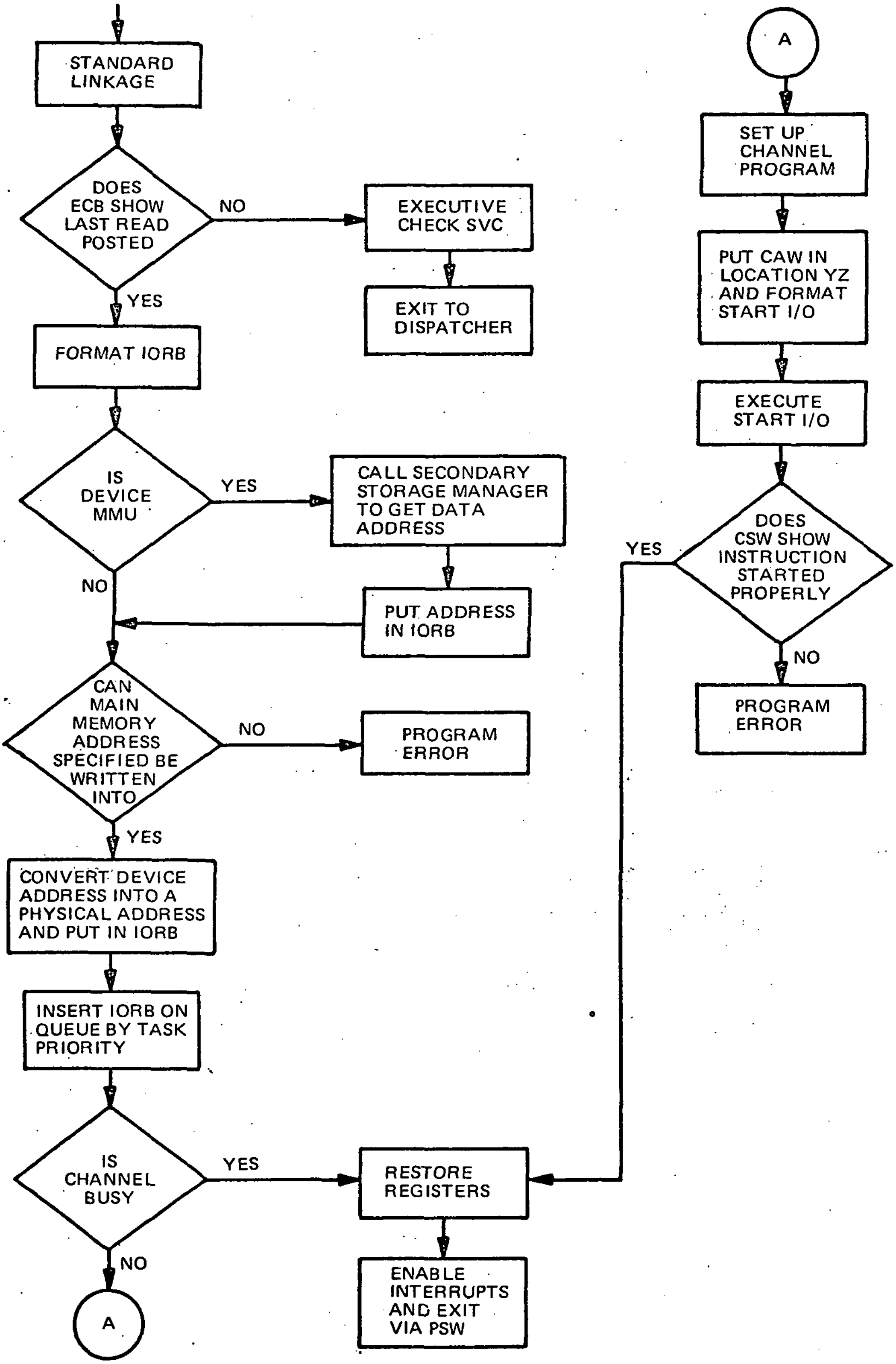

STANDARD LINKAGE
DOES ECB SHOW LAST READ POSTED
NO
EXECUTIVE CHECK SVC
EXIT TO DISPATCHER
YES
FORMAT IORB
IS DEVICE MMU
YES
CALL SECONDARY STORAGE MANAGER TO GET DATA ADDRESS
PUT ADDRESS IN IORB
NO
CAN MAIN MEMORY ADDRESS SPECIFIED BE WRITTEN INTO
NO
PROGRAM ERROR
YES
CONVERT DEVICE ADDRESS INTO A PHYSICAL ADDRESS AND PUT IN IORB
INSERT IORB ON QUEUE BY TASK PRIORITY
IS CHANNEL BUSY
YES
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW
NO
A
A
SET UP CHANNEL PROGRAM
PUT CAW IN LOCATION YZ AND FORMAT START I/O
EXECUTE START I/O
DOES CSW SHOW INSTRUCTION STARTED PROPERLY
YES
NO
PROGRAM ERROR

Figure 5.3 Flowchart of READ SVC

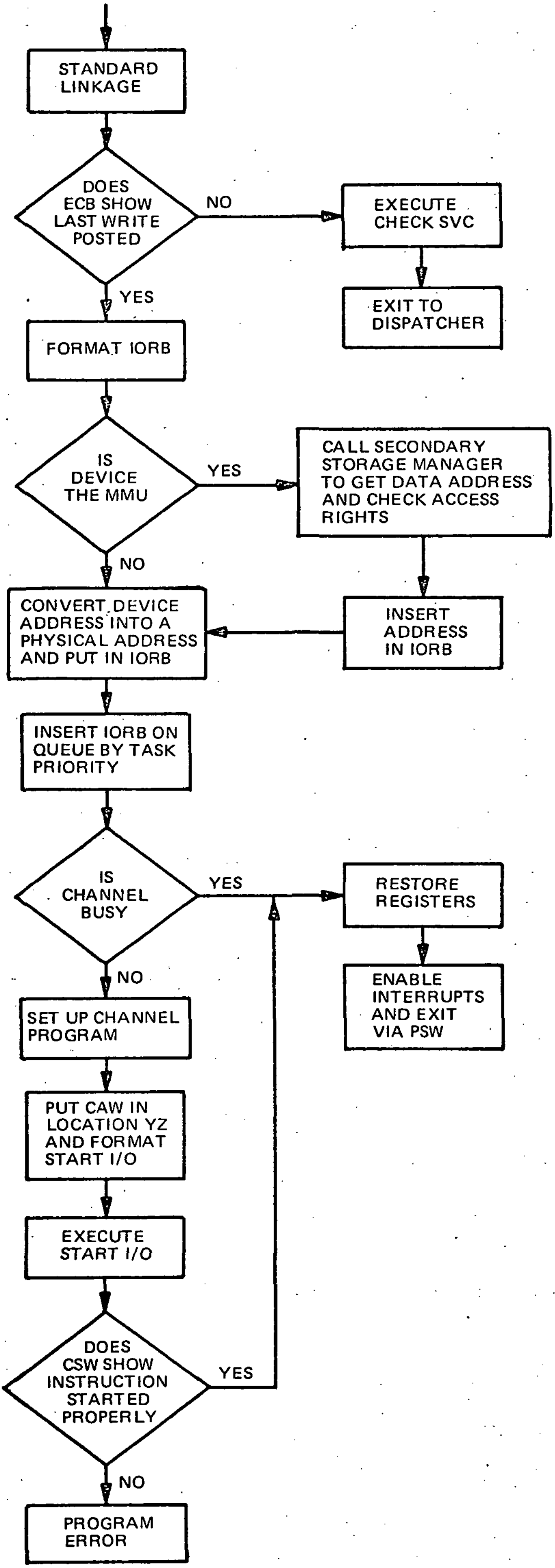

STANDARD LINKAGE
DOES ECB SHOW LAST WRITE POSTED
NO
EXECUTE CHECK SVC
EXIT TO DISPATCHER
YES
FORMAT IORB
IS DEVICE THE MMU
YES
CALL SECONDARY STORAGE MANAGER TO GET DATA ADDRESS AND CHECK ACCESS RIGHTS
NO
CONVERT DEVICE ADDRESS INTO A PHYSICAL ADDRESS AND PUT IN IORB
INSERT ADDRESS IN IORB
INSERT IORB ON QUEUE BY TASK PRIORITY
IS CHANNEL BUSY
YES
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW
NO
SET UP CHANNEL PROGRAM
PUT CAW IN LOCATION YZ AND FORMAT START I/O
EXECUTE START I/O
DOES CSW SHOW INSTRUCTION STARTED PROPERLY
YES
NO
PROGRAM ERROR

Figure 5.4 Flowchart of WRITE SVC

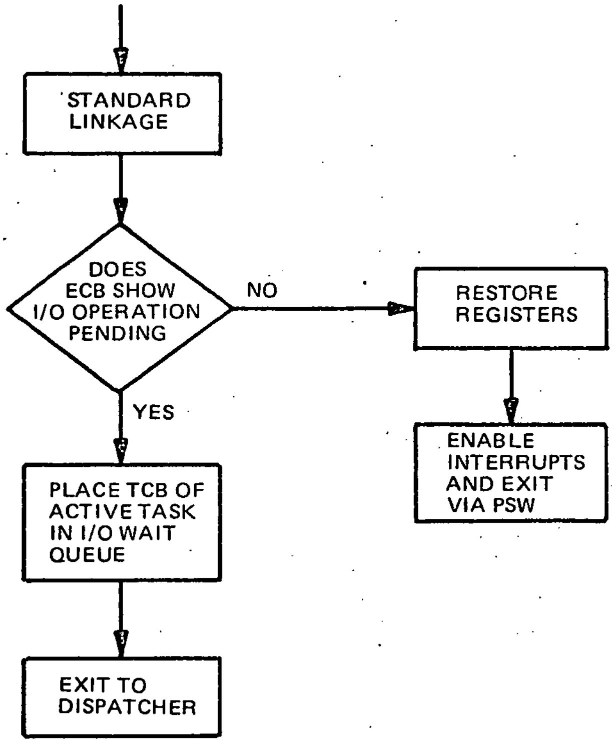

STANDARD LINKAGE
DOES ECB SHOW I/O OPERATION PENDING
NO
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT VIA PSW
YES
PLACE TCB OF ACTIVE TASK IN I/O WAIT QUEUE
EXIT TO DISPATCHER

Figure 5.5 Flowchart of CHECK SVC

Chapter 6

Configuration Management

6.1 Introduction

The topic of configuration management is very extensive, covering many aspects of computer and system design. An adequate discussion of this topic in relation to the Space Shuttle mission must treat the areas of power on initialization, mission phase initialization, error recovery, switching between simplex and redundant modes of operation, and system synchronization. The first three of these topics are pertinent to the 4 Pi EP configuration in the Avionics Systems Integration Laboratory, which will operate in a simplex mode, and hence, these topics will be included in the design of this executive system. The latter two topics are pertinent to the avionics configurations proposed in both Phase B Study Reports and will be treated in this report in a tutorial manner. As we will later see, the configuration management functions are very dependent upon the computer and system architecture assumed.

6.2 Initialization

When the EP computer is powered on, initial program loading (IPL) must be performed. IPL is initiated by the operator pressing a load key. The load is done from an MMU with the unit address taken from switch settings on the console. The first 24 bytes read are placed in main memory locations 0-23. The double words read into locations 8 and 16 are then used as CCWs for subsequent I/O operations.

When the channel ceases its activity, the CPU fetches the double word in location 0 as the PSW and proceeds under its control. The first program module loaded and executed should be a hardware diagnostic routine to insure the computer and subsystems are functioning properly.

Upon successful completion of the diagnostic checks, the mission program for the first phase of flight is loaded. All program module loads are absolute since the mission programs for each flight phase will be preassembled with absolute addresses. At load time the program modules and data need merely be put into main memory at their predefined locations. Furthermore, part of the mission program will be phase independent in the sense that it will be resident in main memory for the entire flight. Examples of this part of the program are the executive, part of the Compool, and some common subroutines, such as sine and cosine. The remainder is phase dependent, changing with the beginning of each flight phase.

The signal to begin a new mission phase can be initiated by the pilot pressing a button. This signal would initiate a priority 0 task which would begin the phase transition. On the other hand, this determination could also be more automated by allowing the computer to determine a phase transition time based upon some set of predetermined criteria. In either case, phase transition involves reloading the phase dependent parts of the computer's main memory.

Phase transition begins by inhibiting the cyclic sequencer subroutines from executing every minor cycle, except for the phase transition subroutine. The background tasks can then execute to completion, or the pilot can examine the TCB queues via the graphic display systems. He can then terminate any background tasks he wishes in order to shorten phase transition time. When all background tasks have terminated, the phase transition subroutine will issue input commands to the MMU to load the phase dependent program modules and data for the next phase. An important part of this load is overlaying the phase dependent entries of the CCT with entries corresponding to the new phase dependent subroutines. The PMD must also be updated to record which program modules are in main memory and which are not. Now at load completion normal processing for the new mission phase can begin. It starts by the timer interrupt occurring, and the cyclic sequencer beginning execution of the subroutines associated with the mission phase.

6.3 Failure Detection and Error Recovery

The area of failure detection in this executive system has two main focal points: internal computer failures and subsystem failures. The former category consists of hardware malfunctions and software errors. The latter consists of the computer's detecting a malfunctioning subsystem by periodically monitoring the status of each. Whenever a failure is detected, a recovery procedure must be invoked.

6.3.1 Hardware Failures

A machine check interrupt is generated in the EP when a hardware malfunction is detected. The PSW associated with the interrupt is given control, and the CPU can then execute a diagnostic routine to determine the cause of the error. An advantage to this procedure is that the CPU can try to restart computation at the point of failure. However, if the diagnostic procedure indicates a persistent machine failure, the EP must be powered down so that the faulty hardware can be replaced. Since the EP is operating in simplex mode, there is no backup computer to take over the computational load. It is almost inconceivable to formulate a recovery procedure for the case where a periodically executed diagnostic test reveals a consistent machine failure, such as an adder error, for which no machine check interrupt is generated. Upon detection, the CPU can be powered down, but tasks which have been running in this environment have probably produced invalid results if this failure condition has existed for some time. Furthermore, the invalid results may have been propagated through the system to an arbitrary degree. Thus, it appears almost mandatory to rely only on instantaneous discovery of error by the hardware.

6.3.2 Software Errors

A software error can be detected two ways: either by the EP hardware generating a program interrupt or by a task determining that an error exists. The program interrupt enables a new PSW to be given control which will invoke a recovery procedure. The standard system recovery procedure will be to terminate the task.

Termination includes releasing dynamic memory and shared data, and removing all of the task's I/O requests from the IORB queue. On the other hand, a task may specify its own recovery procedure to be used instead of the system procedure. The new procedure is specified by an SVC executive in the task. The SVC supplies the address of the recovery procedure, and the executive places this address in the task's TCB. The SVC may be executed several times within a task with a different procedure address specified each time. The flowcharts for these algorithms are given in Figures 6.1 and 6.2. Should a task determine a software error exists by checks within the coding, e.g., by checking an argument for negativity before taking a square root, the task can specify what corrective action to take at that point. It can transfer control to a recovery procedure, or it can immediately terminate. In either case, the executive does not intervene in the recovery process.

6.3.2.1 RECOVER SVC. RECOVER (SVC 16) - The purpose of this supervisor call is to allow a task to specify what corrective action should be taken if a program check interrupt occurs during its execution. (See Figure 6.1)

6.2.3 Subsystem Monitoring

The subsystem monitoring function consists of periodic monitoring of the health of the subsystems which are interfaced to the bus. The objectives are to provide an updated status of the system and to detect errors and failures. Diagnostic routines must be initiated upon detection of an error to provide fault isolation to the functional path or redundant unit level. In conjunction with fault isolation data must be collected periodically to enable trend analysis to be performed as a means of failure prediction.

The cyclic sequencer will periodically request status information from each subsystem. This information is examined by a cyclic subroutine to determine if the subsystems are operating properly. When an error is detected, a fault isolation and reconfiguration procedure must be executed. The procedure will switch out the faulty equipment and replace it with a spare. The spare is chosen from a redundant equipment table (RET) maintained in main memory. A typical entry of this table is illustrated in Figure 6.3. Each entry contains the logical unit number, its physical address and its status. Upon switching active units the formerly active unit is flagged as faulty in the RET, and the new unit is flagged as active. The RET is also used by the I/O routines to determine the physical addresses of logical units for structuring IORBs.

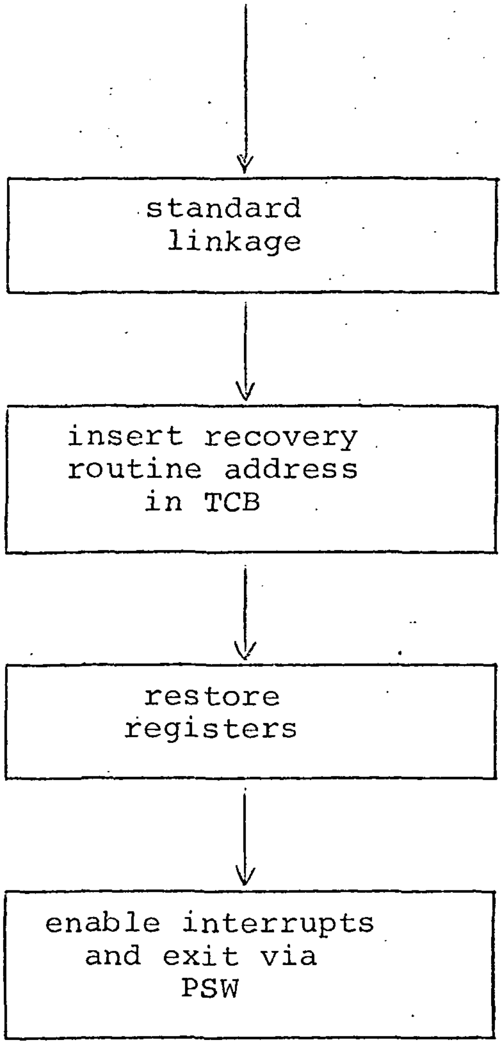

standard linkage
insert recovery routine address in TCB
restore registers
enable interrupts and exit via PSW

Figure 6.2.1: Flowchart of RECOVER

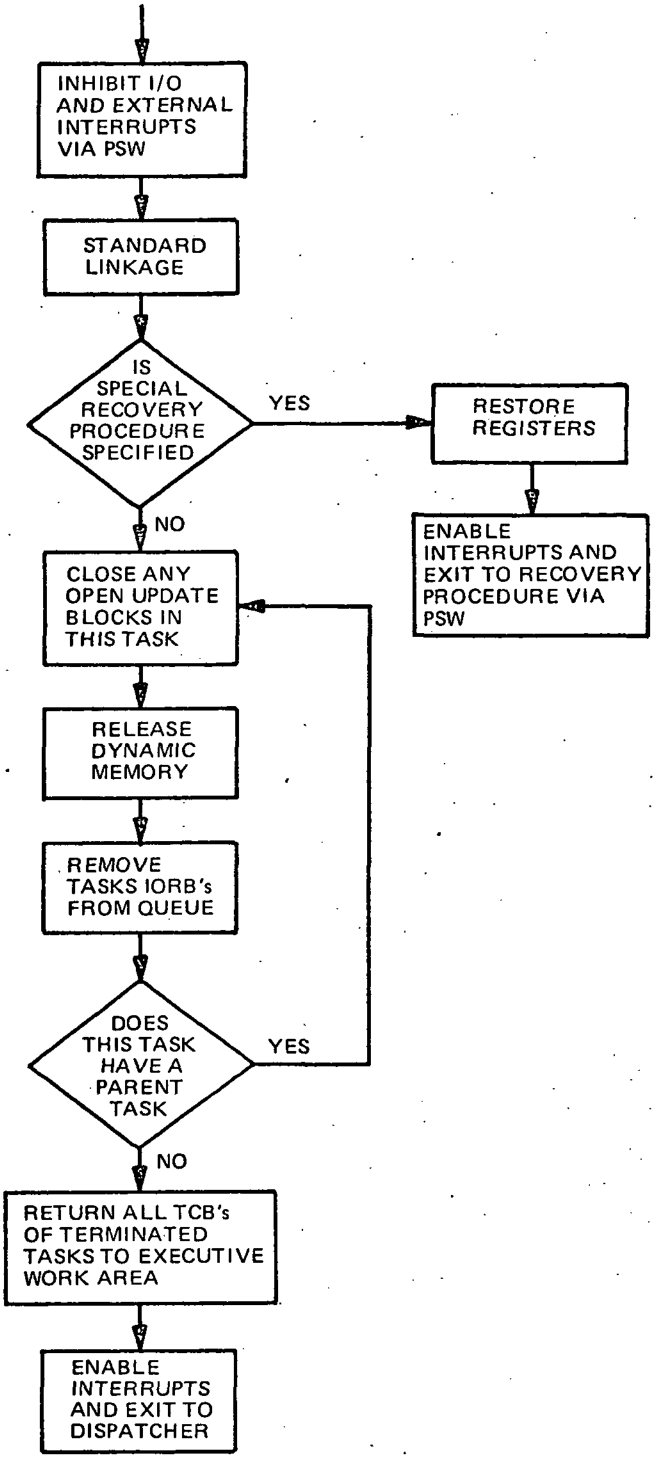

INHIBIT I/O AND EXTERNAL INTERRUPTS VIA PSW
STANDARD LINKAGE
IS SPECIAL RECOVERY PROCEDURE SPECIFIED
YES
NO
RESTORE REGISTERS
ENABLE INTERRUPTS AND EXIT TO RECOVERY PROCEDURE VIA PSW
CLOSE ANY OPEN UPDATE BLOCKS IN THIS TASK
RELEASE DYNAMIC MEMORY
REMOVE TASKS IORB's FROM QUEUE
DOES THIS TASK HAVE A PARENT TASK
YES
NO
RETURN ALL TCB's OF TERMINATED TASKS TO EXECUTIVE WORK AREA
ENABLE INTERRUPTS AND EXIT TO DISPATCHER

Figure 6.2 Flowchart of Program Check Interrupt Software

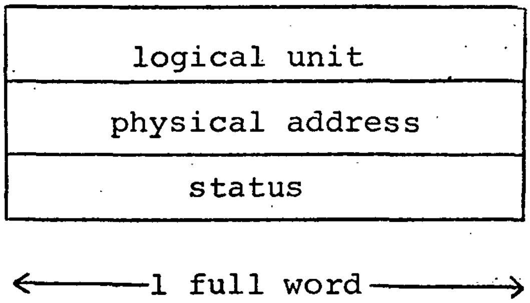


Figure 6.3 Format of Redundant Equipment Table Element

6.4 Failures in a Quad-Redundant System

Phase B efforts to date have indicated that a form of voting and/or comparison will be used for detecting failures in a quad-redundant computer. The following are significant questions in the design of error detection techniques and the software required to support them.

a) How are the computers synchronized: via software or hardware, and how often? This could be a very difficult task for the software alone.

b) What data is voted or compared to detect the error? If the bus outputs are compared for example, identical simultaneous input data must be presented to all computers to eliminate effects of small timing differences.

c) If a comparison mode is recommended, it may be impossible to maintain the software in the "active" computer identical to that in the redundant computers.

It must be pointed out that the techniques of voting and comparing will detect only hardware failures. Software is inherently non-redundant, and errors or inadequacies in its specifications cannot be detected in this way.

6.4.1 Error Recovery of Shuttle Computer Hardware

The problems or recovery, via software, after the detection of a computer failure can be severe. Error detection by voting on and/or comparing the outputs of two or more redundant operating computers is favored in the current Phase B avionics system approaches. Such techniques can be made less difficult to implement if the elements being compared are complete units, i.e., including a complete memory, CPU and I/O controller. A detected failure would result in the disability of a complete computer and its replacement by a standby. However, if redundancy, error detection and recovery are taken to the level of the memory unit, which is then considered as an element of the system independent of the processors, the complexity of the reconfiguration problem increases. The recovery from a memory module failure requires either the replacement of the failed module by an identically loaded copy, or the regeneration of its state prior to the hardware failure. This involves the continuous updating of spares, or an initial load with a consequent delay in system operation.

Failure detection by pure comparison imposes the problem of determining, in the event of a comparison failure, which of the processors is defective. An approach might be to terminate operations in both computers, run diagnostic routines in each, and then reconfigure once the failed computer is identified. However, reconfiguration does pose the following questions:

1) What happens to the time-critical processes that may have been active at the time?

2) If the active computer is the one that failed, how does it hand off control to its backup?

3) What is the next step if both computers indicate failure?

These discussions are not to imply that the problems are insoluble, but more to underline the impact of placing the recovery and error detection responsibilities, of redundant computer hardware, into the software. During the course of this work, careful hardware/software trades must be made to identify clearly the impact on software of these functions.

6.5 Mode Switching

During critical mission phases the MDC Phase B Study [2] calls for all four computers to be processing in a redundant mode of operation. In the event of a failure one of them can be powered down while the remaining three continue processing. In noncritical mission phases, however, only one active computer is necessary. Hence, in a transition from a noncritical to a critical mission phase, it is necessary to switch from a simplex to a redundant mode of operation.

In performing this transition the active computer must supervise the loading of main memory for the other three computers and synchronize their start up. The data to be loaded falls into three categories: phase independent, phase dependent and time critical, such as the mission clock. The first two categories can be loaded from the MMU. The third category of data must be loaded from the active computer, but this transfer can use the MMU as an intermediate device.

The transition from simplex to redundant mode should be done in the noncritical phase before the full redundant computing power is necessary, i.e., before the critical phase begins. This allows time for transfer of data and synchronization, while the computers are not in a critical mode of operation.

6.6 Synchronization

Several approaches have been taken to solve the problem of synchronizing the operations of several redundant computers executing the same software in parallel. MDC/TRW [2] rely on extra hardware (an external clock) sending minor cycle synchronization pulses to the four executing computers. On the other hand, IBM [5] relies on software communication between computers to synchronize the start of tasks. The particular method chosen depends heavily upon the architecture of the computing systems. However, some general principles do apply.

Although the computer operates in a highly involved and complex fashion, it is deterministic and exact: a given operation will always yield the same result if repeated with the same input data. The major problem for computer comparison in a real time environment such as the Shuttle is the synchronization of computations which involve time dependent functions and input data. Any detection of the computers not being synchronized must be treated as an error.

Synchronization can be achieved by:

a) central control of the computer clocks;

b) careful gating and distribution of input data;

c) strict identity of hardware and software operation.

A comparator/voter mechanism adds to the hardware and software complexity. It also incurs operational delays, because time is required:

a) to wait for synchronization of clock and data;

b) to perform the comparison;

c) to decide on the results of comparison;

d) to take corrective action.

To minimize overhead, the comparison should, therefore, take place at a fairly high level of operation, rather than instruction by instruction. Comparing the operation of the computers at the point where they influence their environment, i.e., at the computer/bus interface, is a logical choice, provided that outputs occur frequently enough.

Comparison and voting can be done in varying degrees, with varying hardware and software complexity:

a) Majority voting on the output data of three or more computers, reducing to comparison with diagnostics when less than three good computers remain. The bus receives only the data derived from the majority vote. Failure isolation and correction is automatic as part of the voting process. The complex voter that this requires must be sufficiently redundant and possess adequate error protection to meet the failure tolerance criterion, bacause it is an in-line element in the data bus.

b) Majority voting on the indications of health, but not on the output data. One computer is selected to be "active" and its outputs control the bus directly. The other computers are used as standards to provide independent checks on the operation of the active computer. A voting mechanism decides on the basis of a majority of comparator results whether the active computer is operating correctly. It may also determine which of the inactive computers has developed a failure (see Figure 6.4). In the event of a failure of the active computer one of the others is made active. The voter mechanism may be considerably simpler than the data voter of the previous paragraph, since it only operates on binary values; its response time need only match the reconfiguration dynamics, not the transmission frequency of the bus. Furthermore, since it is not an in-line element of the system, it may not have to meet the same stringent failure tolerance requirements. Each comparator can be considered a part of a computer's I/O section and is thus naturally redundant. In fact, the comparison could be performed, by software, internal to each computer.

As a consequence of voting binary, rather than many-valued byte or word data, the simplicity of the second method pays a penalty in the lower inherent certainty of correctly interpreting failure conditions. There is a greater possibility for split vote situations to arise with binary variables, and a greater likelihood of identical multiple failure. However, these conditions will only arise when failures in the comparison and voting logic itself produce erroneous indication of computer health; the lower complexity of this voter will aid the achievement of the necessary reliability.

For either voting approach once less than three good computers remain, reliance must be placed on self-diagnostic to determine the faulty computer. No self-diagnostic technique can be infallible; a disagreement between two computers could yield the following conditions:

a) One computer determines itself to be faulty, the other finds itself healthy. This is the expected result.

b) Neither computer detects a malfunction. This may be because the fault was transient, or because it was a border-line case beyond the capability of the disgnostic method.

c) Both computers detect malfunctions. This event is highly unlikely in the case of uncorrelated random errors, but may easily occur for common mode problems such as physical environmental transients (e.g., power supply and thermal variations).

One insidious possibility for a processing failure that may not be trapped by any of the techniques discussed so far is that of the software error. The software in each of the redundantly operating computers must, for the purpose of comparison and voting, be virtually identical. It is, therefore, inherently non-redundant. A software fault will produce data which, being identically erroneous, will appear to compare correctly.

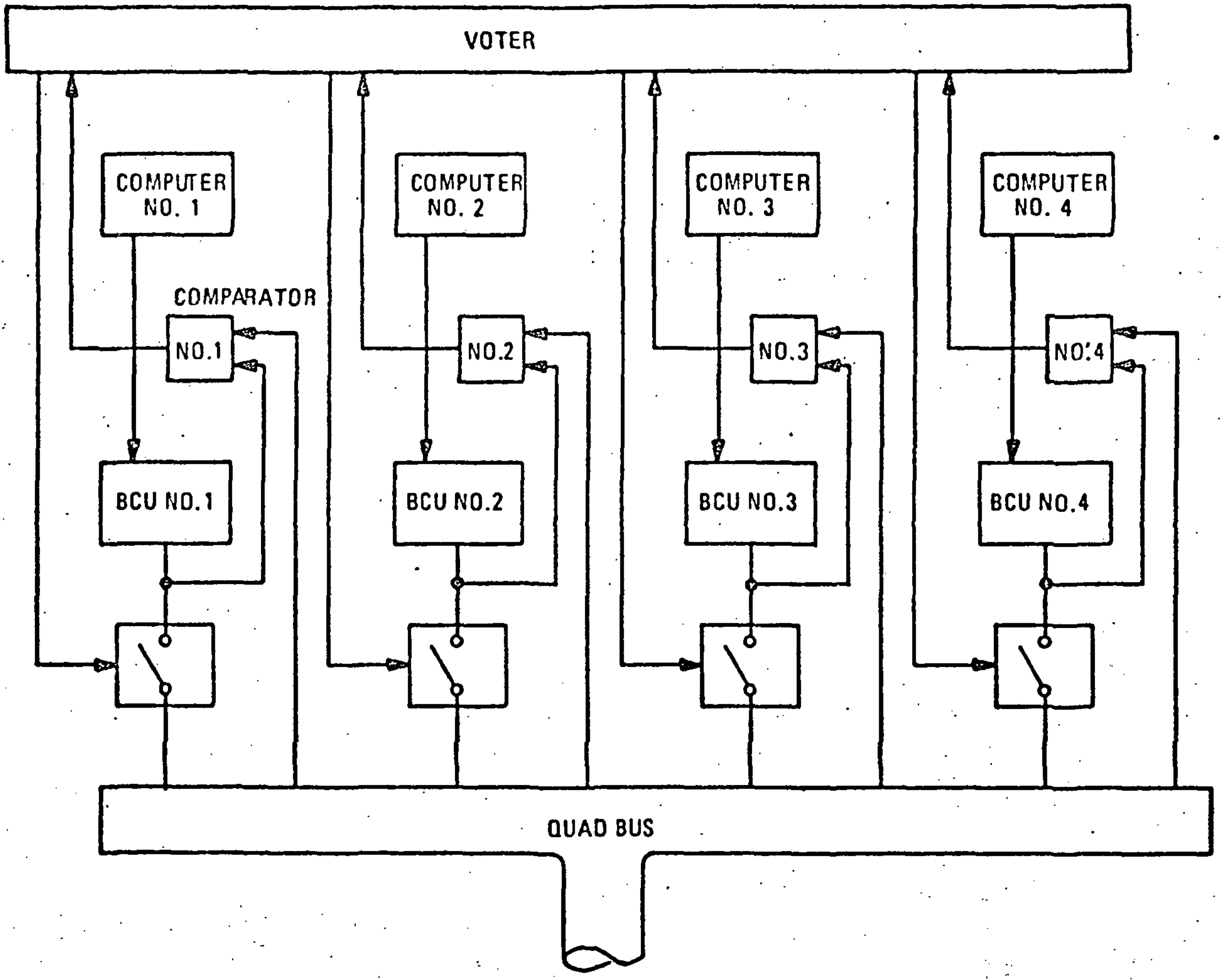


Figure 6.4 Computer configuration with external comparator and voter

Page Intentionally Left Blank

Chapter 7

Secondary Storage Management

7.1 Introduction

The primary purpose of the executive's secondary storage management functions is to supervise data transfer between the computer's main memory and the MMU. These routines must insure correct MMU accesses by tasks, so that the integrity of data transfers is preserved. This chapter will explore the algorithms for performing these functions.

7.2 Data Set Structure

A data set is a collection of records. Data sets may be, for example, program modules, flight data, or display skeletons for the shuttle's graphic display units. All data sets on the MMU are listed by name and address in the DSD. The length of a record is constant throughout a data set and is stored in the DSD.

When a task reads or writes upon a data set, it must operate on complete records. Each read or write operation is done for one entire record. Thus, all blocking and unblocking operations on data within a record are performed by the task and not by the executive.

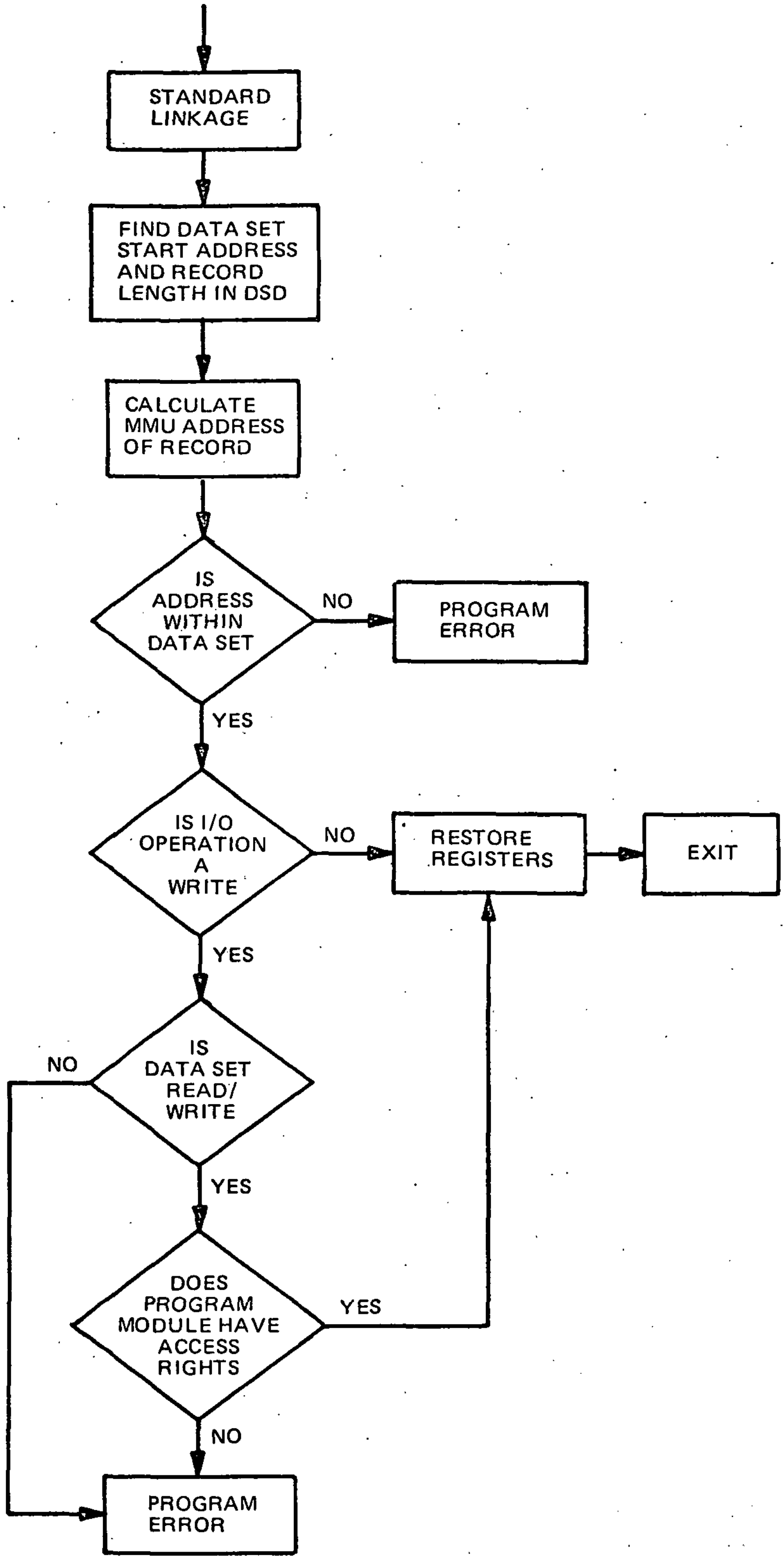
7.3 The Secondary Storage Supervisor

The secondary storage supervisor is called as a subroutine of the I/O management routines. One of the functions of the secondary storage supervisor is to calculate the MMU data addresses referred to in I/O commands. This calculation is based upon the data set start address, the logical record within the data set referred to, and the device geometry. Different types of MMUs, such as disks and drums, will each have a different geometry. Hence, a detailed description of calculating a physical data address is very dependent upon the MMU used.

The number of bytes to be read or written in an I/O command will correspond to the physical record length of the data set. This parameter will be dynamically supplied to channel programs by the secondary storage supervisor from DSD information.

If each data set is systematically organized so that its physical records are contiguous on the MMU and the addresses of these records are monotonically increasing as we proceed from the beginning to the end of the data set, an important error checking feature can easily be achieved. By similarly organizing the DSD entries, i.e., in terms of increasing MMU addresses, each physical record address calculated by the secondary storage supervisor can be checked to be sure it is indeed within the specified data set. This check is done by comparing the record address with the beginning address of the next data set in the DSD. If the former is greater, an error exists in the logical record number specified in the I/O command. A software error condition then results.

If, in addition, the data set specified is to be written upon, the secondary storage manager will check to see if the data set is indeed read/write, and if the requesting program module has access rights. If these two conditions are not true, a software error condition again will result. The flowchart for this algorithm is presented in Figure 7.1.



STANDARD LINKAGE
FIND DATA SET START ADDRESS AND RECORD LENGTH IN DSD
CALCULATE MMU ADDRESS OF RECORD
IS ADDRESS WITHIN DATA SET
NO
PROGRAM ERROR
YES
IS I/O OPERATION A WRITE
NO
RESTORE REGISTERS
EXIT
YES
IS DATA SET READ/ WRITE
NO
YES
DOES PROGRAM MODULE HAVE ACCESS RIGHTS
YES
NO
PROGRAM ERROR

Figure 7.1 Flowchart of Secondary Storage Supervisor

Page Intentionally Left Blank

Chapter 8

Executive Design Parameters

8.1 Introduction

In the course of developing this executive system several design parameters must be left unspecified, e.g., the maximum number of elements that the system queues should accommodate, or the amount of main memory reserved for dynamic allocation to tasks. The nature of these parameters makes assigning numerical values to them at this time very difficult because they are highly dependent upon the characteristics of the application software, the computer and system architecture, and the avionics subsystems eventually chosen for the Space Shuttle. In this chapter we will attempt to isolate these parameters and by doing so identify those parts of the executive implementation that should be parameterized. Parameterization allows for the easy regeneration of new versions of this executive as needed, each tailored to a specific shuttle mission.

8.2 Synchronous Versus Asynchronous Control

The executive software design can support a fully synchronous mode of operation in which all application software is run in the foreground, or fully asynchronous in which all application software is run in the background. Tasks that require careful synchronization with real time, that are highly repetitive, that are short, that are self-contained, are obvious candidates for the cyclic foreground.

Tasks that do not require first order timing specifications, that have wide variations in timing, that require large timing factors for safety, and that are interactive with outside events are candidates for the background. The percent of foreground versus background use of the system depends upon the nature of the application tasks to be executed.

Another parameter dependent upon foreground versus background use of the system is the length of the minor cycle time interval. NR/IBM [1] recommends 40 msec, while MDC/TRW [2] recommends 20 msec. The actual value, of course, depends upon the rate at which subsystems must be sampled in the command response data bus system recommended.

Within a minor cycle care must be exercised so that the foreground computations and I/O requests can be accomplished in this time interval. Any overlap into the next minor cycle is a system overload condition, which requires corrective action.

8.3 Executive Control Element Sizes

The table presented below is a list of each of the executive's directory and queue elements and their storage requirements.

Element	Main Memory Needed
Task control block	38 full words
I/O request block	11 full words
Event control block	2 full words
Event descriptor byte	1 byte
Program module directory element	3 full words
Data set directory element	3 full words
Dynamic core directory element	2 full words
Cyclic control table element	5 full words
Redundant equipment table element	3 full words

The maximum number of these elements that each table must accommodate should be parameterized.

8.4 Task Management Parameters

1) Task Priority Levels: 6 priority levels were chosen. Levels 0-2 serve very specific purposes as previously explained. However, levels 3-5 are merely reserved for executing background tasks. The number of these priority levels can be varied dependent upon background task requirements.

2) Size of Main Memory: while not an executive system parameter, the amount of main memory available influences the software design. For example, it determines the maximum number of tasks that can be concurrently scheduled, the amount of dynamic memory available, and the number of software events the system can support.

3) Software Events: these events are predefined; i.e., they are not dynamically created during flight. Within this category of events, some are exclusive, some latched and some unlatched. These characteristics should be parameterized.

4) Executive Resources: the size of the compool and the organization of dynamic core should also be parameterized. The characteristics of these areas of memory are very dependent upon the number of tasks that can be scheduled concurrently and the amount of main memory available.

5) Maximum Number of Tasks: a limit must be imposed upon the maximum number of tasks that can concurrently be scheduled. Exceeding this limit implies a system overload condition exists because more tasks exist than the system has resources to allocate. Among these resources are main memory to create TCBs, dynamic core, and CPU time. The limit imposed on the number of tasks, in turn, determines the maximum sizes of the system TCB queues.

6) Frequency of Servicing the Time Wait Queue: servicing this queue every minor cycle can impose a high executive overhead. However, if the tasks on this queue are serviced every N minor cycles, there would be a reduction in overhead depending upon the value of N chosen. N can be parameterized.

8.5 Supervisor Call Parameters

The particular parameters associated with each SVC are listed in the next chapter. However, it must be pointed out here that the number of SVC and the services provided by each are system parameters. Since the mechanism for using SVCs is included in the system design, which ones are implemented can be left to the disgression of the system designer based upon application software needs.

Chapter 9

Application Task Interfaces

9.1 Introduction

As we have already seen the interfaces between application tasks and the executive are the SVCs. These represent the only means application tasks have of using the services provided by the executive.

This chapter will list the parameters needed by each of the SVCs described in previous chapters. So far 16 SVCs have been defined, which meet all the needs of the application tasks to run within this system. However, should further executive services be necessary, more SVCs can later be defined and easily included in the framework of this executive system.

9.2 SVC Parameters

SVC Number	SVC Name	Parameters to be Supplied
1	FREEMAIN	None
2	SECURE	Compool data addresses; address of copy area if a copy is necessary; lock address of compool areas to be locked; type of locks to be established.
3	RELEASE	If update of compool is to be done, addresses of data to update compool.
4	COPY	Compool data addresses; address of copy area.

5	LINK	Program module ID Priority
6	END	None
7	SCHEDULE	Program module ID; priority; scheduling conditions: a) none, i.e. unconditional, b) at a specific time, c) after a time interval, d) for some software event or events.
8	WAIT	Conditions of wait: a) until some time, b) for some time interval, c) for some event or events.
9	SIGNAL	Event name; On, off.
10	TEST EVENT	Event name; pointer to flag
11	CHANGE CCT	Pointer to old CCT entry; point to replacement.
12	DISPCHECK	None

13	READ	ECB pointer;
		Core address;
		Logical device;
		Data set name;
		Logical record;
		Timer option; pointer to location in which timer value is to be stored.

14	WRITE	Same as READ except no timer option
15	CHECK	ECB pointer.
16	RECOVER	Address of recovery procedure.

Appendix A

Operation and Control of the Data Bus*

A.1 Data Bus Access and Control Philosophy

Since the Shuttle data bus constitutes a central communications resource shared among multiple terminals and a central controller, a fundamental feature of its design is the method by which it is allocated to a particular communication path. The data bus system is essentially a "party line" shared by all terminals: when access is granted, the bus is dedicated to a single communication path between a transmitting and receiving station.

Selection of the bus access method is a basic decision because it constrains the design of both the remote terminal and the bus control unit.

A.1.1 Command Response Addressing

In a command response addressing scheme access to the bus is centrally managed by the controller. Under this concept, the controller transmits an appropriate command to the terminal including: synchronization header, terminal address, function to be performed (transmit, receive), data, and parity coding. Upon recognition of its address, the terminal interprets the command and begins transmitting or receiving the appropriate data.

Using command response access, a terminal does not initiate any communication unless it is commanded to by the controller. Terminals only "speak" when "spoken to".

*The discussion in Appendix A and Appendix B is taken from an Intermetrics, Inc. study on a standard interface definition for avionics data bus systems [8].

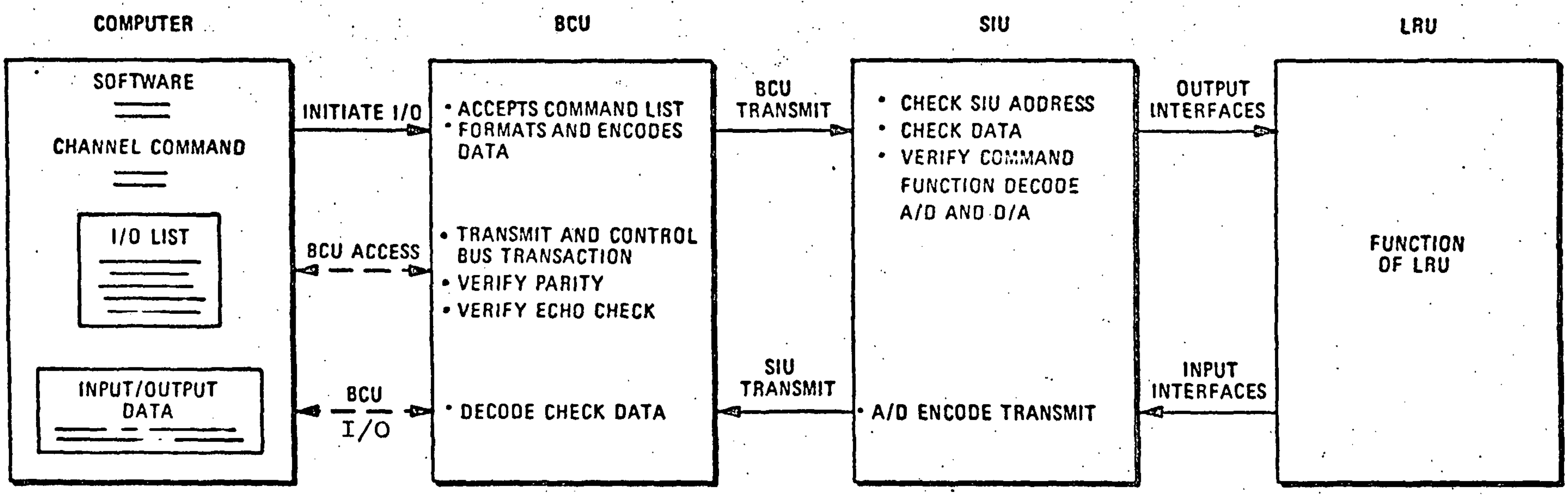


Figure A.1 Basic functions during a bus transaction

In contrast to the polling scheme a terminal is not "polled" as to whether it wants the bus or not but rather is "commanded" to send or receive a message. Command/response addressing is similar to a polled system in that a terminal responds only when addressed.

A fundamental characteristic of command response control is that the "intelligence" of when, what, and how often to communicate is in the controller (i.e., computer software). There are consequently no access conflicts to resolve or local decisions required.

A.2 Control and Operation of the Data Bus by the BCU

Once a particular access method is selected, the communications procedure established to perform a single I/O transaction impacts the design of the bus system elements. The following steps, illustrated in Figure A.1, must be taken in order for a single computer to send and receive data from a set of avionics equipment.

a) In a command response access concept, the computer directs all I/O requests in the system. It indicates along which bus line and to which remote terminal the message is routed, and if data is requested, where to put it when it has been obtained.

b) The BCU must encode the message and transmit it to the proper remote station over the selected bus line.

c) The remote terminal responds to the command, selects the appropriate channel to the LRU and executes the appropriate functions to obtain the data.

d) Signal conditioning and conversion take place at the terminal, which then encodes and transmits the data back to the control unit.

e) The established error-control scheme is maintained throughout the transaction.

f) The BCU transfers the data to the computer and informs it of the completed request or list.

The details of this transaction influence the bus message format, the functions of bus elements, and communication security. The message format and structure must satisfy the data acquisition and distribution requirements, without unduly complicating the bus hardware design. A level of transmission "security" must be established to minimize the probability of an undetected error, without significantly increasing the equipment complexity or message overhead. The following sections provide a general discussion of bus operation and the bus format and structure.

A.2.1 Bus Message Format

In general there are four basic parts to the structure of any communication message: the message header and terminator, the address and routing information, function code, and message content.

Message Header Sync	Address Routing	Function Code	Message Content	EOM Sync

The first three parts of the message are associated with the communication system.

A.2.1.1 Message Header and Terminator. Message synchronization is required to enable terminals to recognize the start of a message and is usually a unique control signal recognized by the terminal. It is essential that the synchronization signal be different and clearly distinguishable from data to avoid misinterpretation. The characteristics of the sync signal will depend on the modulation technique selected. It is usually assigned a pulse width or phase change different from the standard data bit.

There are four possible sync signals: at the beginning and end of the BCU to SIU message and at the beginning and end of the SIU to BCU message. However, from a communication point of view they are not all necessary. The end of the BCU to SIU message can be distinguished by the "idle bus" when the BCU stops transmitting; similarly for the end of the SIU to BCU message. However, detection of an "idle bus" may cause circuit difficulties in either the BCU or SIU. The use of different sync signals for BCU to SIU messages and SIU to BCU message rules out inadvertent SIU to SIU communications, since the SIU need only respond to a BCU sync.

In any case, the only positive requirement for any address system is that there by a sync signal, clearly distinguishable from data, so that each terminal can begin to look for its own address in synchronization with the message. The need for other sync signals for end of message, accept, knowledge, etc., is a function of the communication procedures and the details of the implementation.

A.2.1.2 Address and Routing. The address portion of the message identifies the sender and receiver by "to X" "from Y". In a centrally controlled system, where there is no terminal-to-terminal communication, there is no requirement for the "from" part of the address. All communications are initiated by the BCU with transmitting/receiving occurring only between BCU and one SIU.

The "to" part of the message identifies the path to the LRU via an SIU address and an EIU address. A separate EIU address is necessary when the bus terminal communicates with more than one EIU. If the SIU and EIU were combined into a single unit, then the address could be combined.

A.2.1.3 Group Addressing. A group addressing capability would be required to send a single message to more than one SIU or EIU, as might be required to enable a passive flight recorder on the line to receive data intended for other terminals. Group SIU addressing could be an advantage in transmitting the same data to every element of a distributed subsystem, such as the individual quads in the RCS system. Group addressing would be useful in the central management of a redundantly configured subsystem, particularly if identical commands are issued by the computer to every redundant unit.

Group addressing on the bus requires the SIU to recognize more than one address. However, there is the problem of coordinating the return transmissions of echo or data messages. Coordination could be implemented in several ways: by sequential time slotting of the SIU responses, by ignoring the echo in the passive device, or by a contention access method. The SIU, EIU address and function codes would need to be coded in a way which would have group meaning. The tradeoff here is between the added complexity of the SIU and BCU hardware, and the additional software and memory to store multiple commands instead of one. A modification to the computer/BCU message to provide a routing indicator and a list of SIU addresses, which would enable the BCU to send multiple messages, could alleviate the computer software burden.

In summary, however, it is felt that group addressing is probably not worth the additional complexity in bus system design if, as has been estimated, there is adequate capacity in speed to accommodate the inefficiences encountered.

A.2.1.4 Function Code. The function code field of the bus command specifies the action to be taken by the interface unit in acquiring or distributing data or signals to the LRU. The structure and format of this field is directly impacted by the requirements of the electronic interface portion of the remote terminal. In order to provide the capability of interfacing the majority of electronic equipment, the following types of interfaces would be required:

a) digital parallel,

b) digital serial,

c) analog data,

d) discrete.

The function code does not have to be in a standardized format for all terminals. More parallel digital signals may be required for a particular LRU, but less analog. The electronic interface itself need not be standardized. The function can be decoded and interpreted by specially tailored function controllers at the terminal. Alternatively the function code could represent the address of a location in a control memory which stores special control sequences within the interface unit. There are several ways of organizing the function code field, which are discussed in the following paragraphs.

a) Channel Addressing

Under this concept, each interface is assigned a channel address, and the function code becomes part of the address structure. Group addressing is possible only if channel addresses are in sequence (e.g., 2 through 6, not 1, 3, 5, etc.). Input or outputs may be implicit in the channel address number, or specified via a format. The interface unit is required to distinguish between input and output channel addresses, to determine if data is to be sent back.

Channel addressing is the simplest function code to implement and allows the greatest flexibility. However, it can be very inefficient if channel addresses are not assigned in a way which can be effectively utilized.

b) Functional Classification of Interfaces

In this method interfaces are functionally classified and a code for each class or subclass is defined. For example, all communications can be functionally organized into the following categories: commands, moding, functional input, functional output, and others. The functional categories are assigned a coded number and all interfaces are assigned to a category. A function code would then involve input or output of all data in the corresponding category. Obviously each major category can be further subdivided into subclasses by extension of the function code field. A significant advantage of this method is that the efficiency of information transfer can be much higher if information is generally transferred in a block. It can also be useful from the computer's point of view, since all data in the "functional group" may be desired at the same time (e.g., all status information).

c) Memory

The final approach involves a small memory, of a few hundred words. The function code specifies a location in the memory which contains instructions for data input and output. The memory could store channel addresses or sequences corresponding to an interface function. A memory with a read/write capability could be altered inflight to accommodate changes to a subsystem's operation demanded by different mission phases.

A small high speed memory of the read/write or read only type described above is well within the state of technology. This concept provides the most general and flexible capability, although it obviously increases the complexity of the EIU. Memory size could be expanded to accommodate increases in equipment requirements, or to extend the terminal capability to provide functions such as limit checking of data, or the monitoring of LRU status. Ultimately the terminal becomes a small computer capable of providing a local service to the equipment and thereby reducing bus traffic.

A.3 Operation and Control of the Data Bus by the Computer

Viewed from the computer the data bus is a single, relatively high speed, asynchronously operable, peripheral I/O device, capable of performing data gathering and data distribution. Under the command response access concept, the computer initiates and directs I/O operations on the data bus. It directs I/O by commanding the bus control unit with a set of I/O requests. The BCU then controls and synchronizes the data bus system to carry out these requests. Most likely, the bus system will be mechanized in a way which allows the bus to operate independently of the CPU once an I/O command is issued by the computer. This means that the data bus system and computer operate asynchronously.

A.3.1 Overview of Computer I/O Operations

There are two basic approaches to the design of the computer software for controlling the activities of the bus. The first is the synchronous, fixed I/O method, in which I/O control is based on a predetermined execution sequence and a fixed time cycle. The second schedules I/O operations on a demand basis. The characteristics of the two are summarized in the following sections. To a large extent the computer executive and I/O control structure can be considered independently of the control structure chosen for the bus.

A.3.1.1 Computer I/O Operation in a Synchronous Structure. Fixed sequence structured software requires I/O operations to be interleaved with processing tasks in the minor cycle. The inputs required by processing tasks in a minor cycle must be available prior to execution of the minor cycle.

The concept requires commanding the BCU (or dispatching I/O), each minor cycle to input data required for the "next minor cycle", and output data from the "last cycle". I/O software for controlling the data bus is operated in each minor cycle. For example:

Bus Activity	Inputs for processing during N Outputs from N-2	Inputs for processing during N+1 Outputs from N-1	Inputs for processing during N+2 Outputs from N
Computer Activity	Process inputs from N-2 for output during N	Process inputs from N-1 for output during N+1	Process inputs from N for output during N+2
Minor Cycle	N-1	N	N+1

The dispatching of an I/O command list to the BCU can occur at the beginning of each minor cycle. However, it is necessary that the list of I/O be completed by the bus system prior to the start of processing the next minor cycle. Thus, the bus will be operating for only a portion of the minor cycle at a percentage of its speed. For example, the BCU may be commanded for 16 ms of I/O every 20 ms. In this case there would be 4 ms idle bus time unless the BCU were commanded again to perform some additional I/O on checkout functions.

At the beginning of each cycle I/O commands are checked for errors. If no errors have occurred, the next I/O list is sent to the BCU and computer commences its processing sequence.

If I/O errors occurred, an error recovery and fault isolation routine must be operated and the sequence of processing tasks re-scheduled accordingly. Prior to the end of the minor cycle I/O scheduling is operated to set up the I/O command list for the next dispatch to the BCU.

Since much of the Shuttle data bus design conducted to date has postulated this philosophy of software operation, it will be assumed for the description of BCU activities in the following sections.

A.3.1.2 <u>Computer I/O Operations in a Demand Structure</u>. The alternative approach to fixed sequence I/O is scheduling I/O operations on a demand basis. Typically, this is accomplished in asynchronously controlled software structures as follows:

a) when an I/O request is made by the computer software, control is transferred to an I/O scheduler, and a command is inserted into an I/O queue.

b) The task requesting the transfer is placed into a "wait state".

c) Upon availability of the I/O device, the queued I/O requests are processed via the dispatcher which uses an algorithm, e.g., first in/first out (FIFO), to determine which I/O request to service next.

d) The I/O requests are sent to the BCU one at a time, or in a list for bus execution.

e) When the I/O request has been serviced, the issuing task is informed and allowed to continue.

This approach is used on large ground-based systems, particularly where I/O requirements are not known or impossible to predetermine. The demand I/O concept does not appear consistent with command response or fixed sequence scheduled processing tasks. However, if a distinction were made between computer input and output requests, output requests because of their independence of processing tasks may lend themselves to demand scheduling.

A.3.2 Computer to Bus Operations

An evaluation of the requirements of the interface between the computer software and BCU is directly dependent on the design of the BCU. There are obviously tradeoffs between complexity in the BCU hardware design and the computer software. The BCU in an extreme case could become a computer itself, dedicated to communications functions, supplying all communication of data in and out of the bus system. At the other extreme, it could simply perform time synchronization, transmitting and receiving control, and error coding. Somewhere in the middle, the basic BCU capabilities can be extended by providing the BCU with a limited set of registers and logic, and a direct memory access (DMA) interface to the computer's memory. By cycle stealing from the computer, the DMA can supply commands and data to the BCU directly from the memory. Commands and data are sent to the BCU either by incorporating a starting address and the number of commands into the channel command word, or by chaining commands and instructing the BCU via the operation code in each bus command. A limited capability will be assumed for purposes of this discussion, although comments are made on areas where an expanded BCU capability may lessen the software problems. The basic computer-to-BCU operations are the following:

a) I/O dispatching - involves commanding and controlling the BCU with I/O to be performed.

b) I/O scheduling - involves scheduling bus commands to be issued the next minor cycle.

c) I/O error processing - checking previous I/O commands issued for errors and taking appropriate action.

A.3.2.1 Dispatching I/O: Computer/Bus Interface. The BCU is provided with a list of I/O commands by loading an I/O channel with a command word from the computer (see Figure A.2). The channel command word must contain sufficient information to enable the BCU to execute all the appropriate I/O commands in the list. Once this channel is loaded, the computer and BCU may operate independently. The channel command word contains an address of the first BCU command, and the number of BCU commands to be processed. (BCU commands may also be linked by address chaining.) The BCU commands can be stored in sequential memory locations, and the list operated on in sequential order by the BCU. Upon completion the BCU can be instructed to interrupt the processor with an I/O complete signal. (Alternatives, more in line with a "no interrupt" policy, can be devised, such as a "BCU busy" signal accessible to the computer enabling it to determine status of the BCU.) In either case, it is necessary to coordinate the asynchronous operation of the computer and BCU so that the computer is aware of the status of the BCU.

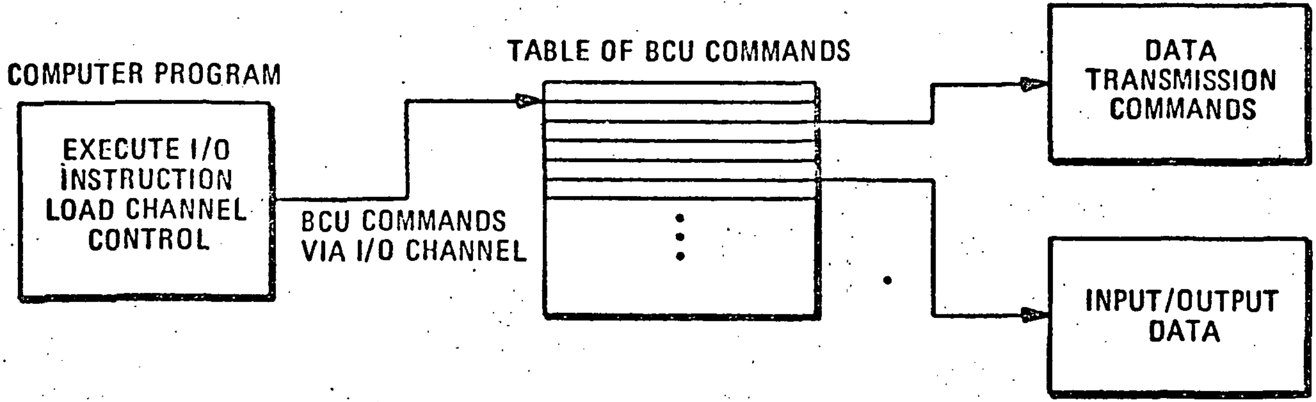

COMPUTER PROGRAM
EXECUTE I/O INSTRUCTION LOAD CHANNEL CONTROL
BCU COMMANDS VIA I/O CHANNEL
TABLE OF BCU COMMANDS
DATA TRANSMISSION COMMANDS
INPUT/OUTPUT DATA

Figure A.2 Computer to BCU I/O command operation

A.3.2.2 BCU Command Format. The BCU command format must contain instructions for the BCU to execute the computer's I/O request. A single command will contain four parts: control information for the message, status information, skeleton bus message format, data linkage addressing information.

BCU control op code	I/O status	Bus command			Linkage to data
		SIU #	Function code		

a) Control

The control part of the BCU command contains information pertaining to the type of operation requested of the BCU. Examples of individual BCU operation codes are Read, Write, Skip, Linkage. With fixed I/O tables in the computer's memory, a "no-operation" code may be desirable to skip commands at certain times such as unrequired jet on commands in a fixed I/O schedule. If the BCU contained memory, and was more of a communication processor, this part of the BCU command may contain a pre-programmed BCU memory address for execution.

b) Status Bits

Status bit(s) are required to enable the computer to determine if the bus command was completed successfully. The computer must be informed of bus errors so that it can reconfigure and reschedule accordingly. An incomplete I/O transaction will result in rescheduling the processing tasks. An "incomplete I/O" status indication may also be desirable.

c) Skeleton Data Bus Message

The skeleton bus message contains the actual bus command associated with the I/O transaction. The contents of the bus message format were discussed in Section A.2.1.. It contain contains information which is both fixed and variable during the course of the mission. Specifically, the terminal addressing will vary with the status of the avionics configuration; a specific communication path must be chosen prior to execution of the command. For example, a request for data from a redundant subsystem (e.g., radar) requires information as to which LRU is active, and which data path to use. It is reasonable to assume that configuration management is a computer software function, and therefore this information must be supplied to the BCU in some form.

The degree to which the computer will need to modify the bus message format at run time will depend on the extent and capability of the BCU.

In order to establish fixed I/O command tables required by the synchronous I/O method it may be useful to define a symbolic and "physical" relationship similar to that used with tapes, disks, etc., in a conventional facility. In this case a symbolic assignment, such as ISS_A or ISS_S for inertial subsystem active and standby respectively, will be associated with the subsystem. The symbolic identification is then associated via configuration tables to a physical unit such as ISS#1, ISS#2, etc. Predetermined I/O bus commands would be generated using symbolic identification and their physical identification determined at run time by the computer or by the BCU via the transfer tables of the computer. Path identification for a specific physical unit (i.e., which SIU/EIU address) must also be determined dynamically.

If each physical unit had a single path, i.e., a unique address (BUS#, SIU#, EIU#) the problem is solved. However, there is more than 1 path to each unit; the address must be determined from the status of buses and SIU's. The complexity of this problem will, of course, depend on the redundancy interfacing and cross-connections established in the system. For example, consider a system configuration of a quad-redundant bus, 4 SIU's, and up to 4 EIU's per SIU. There could be up to 64 possible paths depending on the cross-strapping.

Physical Unit	Bus	SIU	EIU
LRU #1	1	A	X
	2	B	Y
	3	C	Z
	4	D	W

If the SIU is an extension of the bus such that SIU_A cannot be addressed via bus #2, then there are 16 possible paths to a specific LRU. If the SIU were cross-strapped to the bus and interfaced to a single LRU, then there are only 4 paths to it.

The function of inserting addresses could be allocated to the BCU, assuming it had memory, by sending it a table of physical equipment codes, and the current path. The current path would be updated by the configuration management task as configuration switching occurred.

d) Data Linkage Addressing

This part of the bus command identifies the computer memory location of the data to be output, or the destination of the data input from the bus. If the bus format allows block transmission, then the number of words is variable, and must be obtained from the bus message itself.

A.3.2.3 Computer I/O Error Processing. An unsuccessful I/O transaction detected by the BCU during bus operations is eventually communicated to the computer, using the error control bits in the bus command table. If the BCU is commanded with a list of I/O requests, an I/O error will not be detected until the start of the next minor cycle. At the beginning of each minor cycle, the error status of all messages is checked. If errors occur, the minor cycle task schedule is modified accordingly, and the I/O error recovery procedures are initiated. Some of the alternatives are:

a) the I/O request could be rescheduled via an alternate path. A reconfiguration of equipment may be required.

b) Fault isolation tasks could be initiated to determine what to reconfigure (the BCU, SIU, or subsystem may have failed).

c) The sequence of tasks contained in the following minor cycle must be altered, delayed entirely, or allowed to continue with "old" data.

A.3.3 I/O - Processing Memory Conflicts (Buffering and Interlocking)

Independent operation of the bus and computer can result in a conflict over the access to common data. This problem occurs when a processing task is using data while the bus control unit is at the same time attempting to input or output the same data for the same memory locations. The problem is more likely to occur for data that is sampled at a high frequency, when use of the data cannot be easily synchronized. It is also more likely to occur in a block of data rather than a single word because of the inherent interlock of a single word access. For example, attitude angle information from the inertial unit may be in use by the digital autopilot task when the BCU inputs new values via the DMA. In this case the autopilot is operating on partly new and partly old values. This problem can be avoided by several approaches:

a) the I/O input and output in this category can be buffered into different memory locations. It may be transferred to other locations, or a pointer can be switched between two sets of registers for the data item, one set for I/O, one for processing. Input data may in any event require to be smoothed or compensated prior to use. This is the general concept of "double buffering" of input or output.

b) The data could be interlocked via a control indicator or busy bit, during the time either the BCU or the computer is using it. However, this would require the BCU to access, test, set and release the indicator with a consequent increase in its complexity.

c) I/O can be planned by predetermining and adjusting the sequence of I/O commands to avoid the conflict. I/O commands can be designed to occur at the opposite end of the cycle from the conflicting processing task. This approach, although consistent with synchronous bus control and I/O philosophies, appears risky due to the inaccurate estimates of timing. It is, in fact, similar to the approach used to solve the memory conflict problem in Apollo. This was only partially successful, and it could only be verified by extensive testing.

A.4 Description and Analysis of I/O Transactions

A.4.1 Definition of an "I/O Transaction"

An "I/O transaction" is defined as the complete sequence of operations performed by the BCU in carrying out a single I/O request from the computer. Once the BCU has received and interpreted a command from the computer, it synchronizes the terminals on the line, transmits a message to the specified terminal and receives the appropriate response. A transaction occurs between the BCU and a single terminal. It is the basic bus communication activity. It is independent of any other transaction over the data bus system. There are two types of I/O transactions that are performed by the data bus: read and write transactions.

a) A read transaction is the sequence of steps performed by the bus system in acquiring data from the avionics equipment. It can be termed a "get" command, to sample a specified LRU equipment interface.

b) A write transaction is a sequence of steps to send data to an LRU interface. It can be described as either a "receive" command, or a "do" command. The SIU receives the data or command and delivers it to the specified equipment interface.

A third type of transaction may be required, termed an "SIU Event Status Command", in which the BCU transmits a command message to an SIU, requesting it to return its event status register.

This transaction enables the computer to determine if random events (interrupts) have occurred at LRU's connected to a particular SIU station. A rescheduling of processor tasks and read/write transactions may be necessary as a consequence of the event.

A.4.2 Functional Description of Bus Transactions

A discussion of how the bus system performs a transaction provides another step towards a specification of the bus/SIU/EIU hardware design. In order to describe the operation of the bus during a transaction an assumption must be made with regard to a specific bus to SIU to EIU configuration, and an error control approach. It is important to emphasize that this section is intended to describe the functions required at each bus element, and not to select a final design. Several configurations of a standard bus terminal were considered, but a detailed bus command format was only designed for one.

The example configuration assumes a physical separation of SIU and EIU. Each SIU is connected to only 1 bus line and may service up to 8 EIU's. Each EIU provides analog and digital interfaces to equipments. The other terminal configurations assume no logical separation of the SIU and EIU, and are cross-strapped to all four buses.

The error control method selected for analyzing the transaction is transmission error detection through vertical and horizontal parity, and path verification by address echo.

A variable number of 8-bit data bytes was selected as the basic transmission format. A 3-byte command format is selected since 16 bits are considered inadequate to provide the range of addressing and function codes. A minimum of 18 bits are required for the command word in this configuration (7 for SIU address, 3 for EIU address, and an 8 bit function code).

Figure A.3 illustrates a representative format designed around the 3 byte command message with a variable data message. The asterisked fields are mandatory. Representative use for the other bits in the 3 byte command are discussed below:

*a) SIU address (up to 128 since only one terminal address per station is required. See Section 3.)

b) SIU transaction bit. This bit may be used to command an SIU station to send an event status message. This is a two byte response from an SIU containing the status of 16 events or conditions that are assigned among EIU's at a terminal. Each is set in an EIU by the occurrence of a local random event such as a hand controller movement, display input, or fault occurrence.

*c) EIU address (up to 8 EIU's per SIU)

d) Error control bits. These are sent in an echo message from SIU to BCU when an error occurs associated with the LRU. Typical of the possible error response conditions are:

1) parity failure at EIU

2) EIU/LRU busy

3) no response by EIU

4) improper channel

This information could be provided by a special request to the SIU. Making it part of the command format simplifies SIU/EIU logic. If the information were not provided to the BCU, a "no echo" response for all the above conditions will be treated in the same way.

e) I/O control. This control bit determines whether the specified channel address is an input or output operation.

f) Block. This field of the command message identifies a single or multiple channel address group. It is used in conjunction with "block size" to specify the size of the message block.

*g) Channel Address. This specifies the EIU interface by one of the methods listed in Section A.2.1.4.

h) Block Size. The block size identifies the number of channels to be sampled.

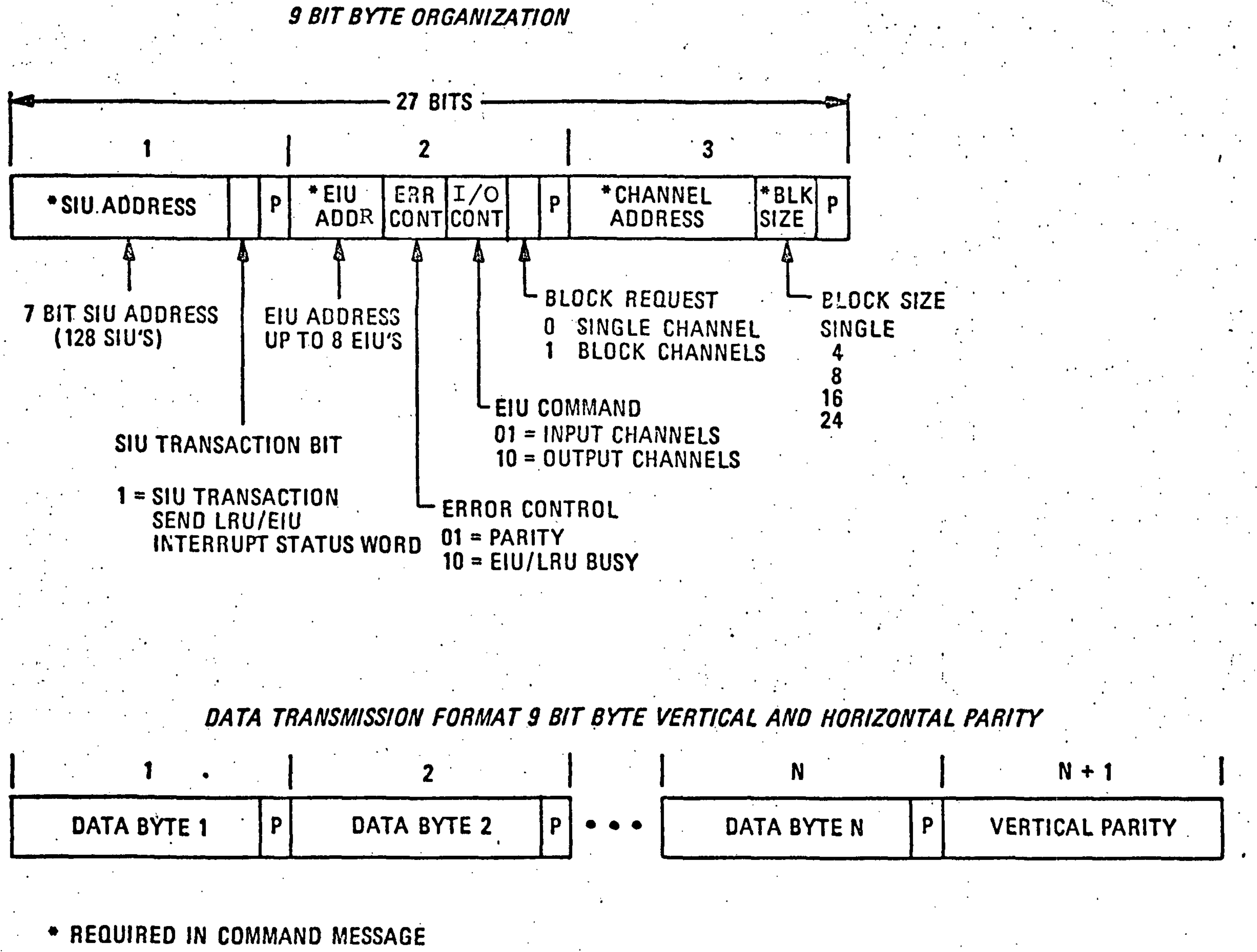


Figure A.3 Representative bus command message organization

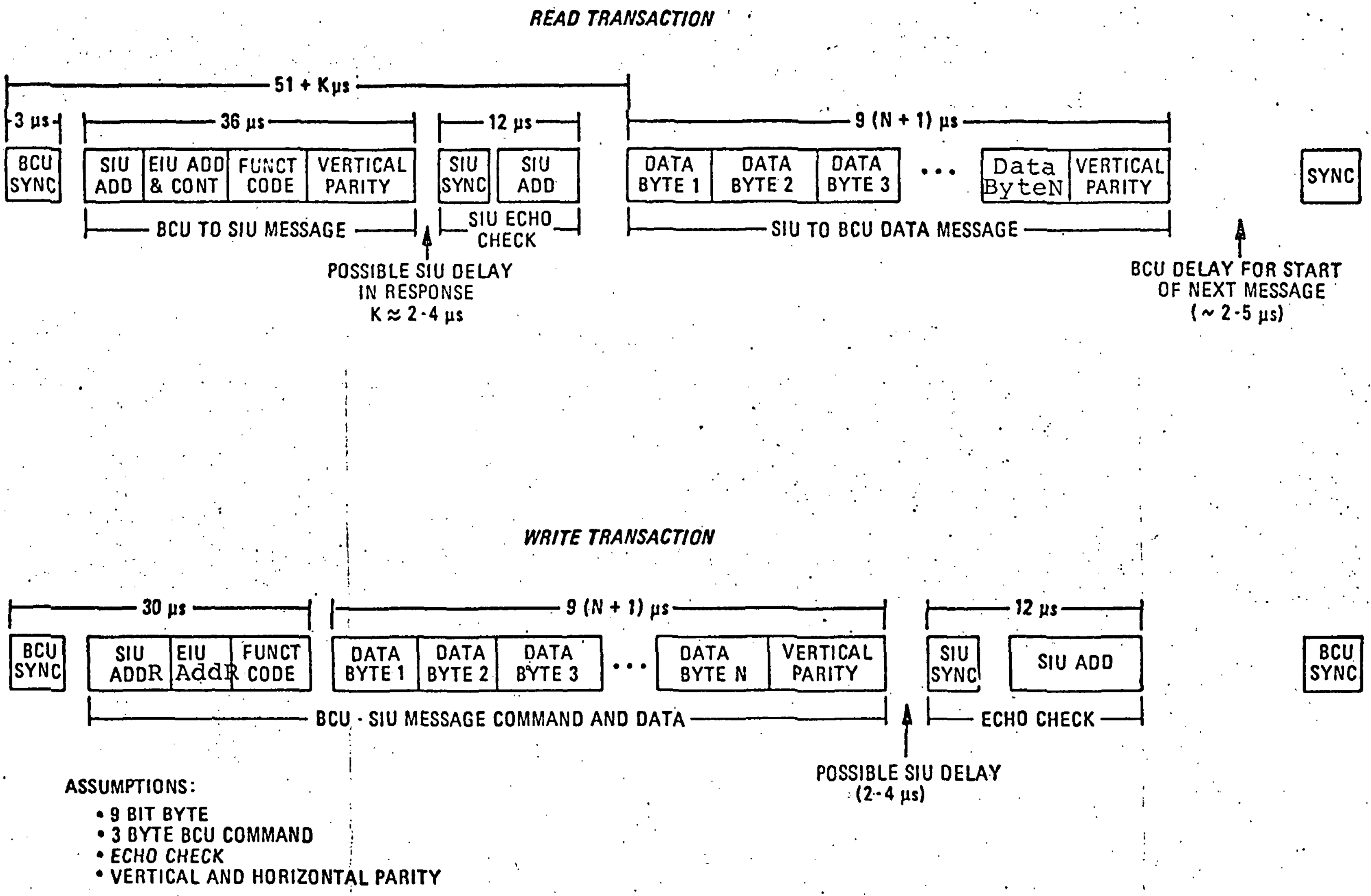


Figure A.4 Sample read/write transactions

A.4.3 Description of the Transaction Sequence

The steps involved in read and write transactions using this format are illustrated in Figure A.4. A brief description of the transaction is as follows.

a) A read transaction begins when the BCU initiates a sync signal on the bus, followed by transmission of the bus command word. The BCU then waits the response.

b) All "up" receivers on the line receive the sync signal. Each compares the SIU address in the message with its own prewired address. If no match occurs the rest of message is ignored, and then each SIU monitors the line for the next BCU sync.

c) If the address check shows agreement, the SIU decodes the EIU address and then routes the message to the specified EIU over a serial channel*, while checking for horizontal parity in each byte.

d) The SIU awaits the parity check signal from the EIU to insure that the message was received properly, and upon its receipt, transmits an echo message to the BCU. If the EIU does not accept the message, the SIU transmits its address echo with the appropriate error control bits set in the second byte of the command word.

e) During the time the SIU is transmitting the return echo, the EIU decodes the function code (channel address or memory), multiplexes the requested input channels, performs A/D conversion if required, and sends the requested data to the SIU. A time lag is incurred by this process, termed the LRU latency. It is discussed below.

f) The SIU verifies parity and continues transmitting the data message to the BCU.

The BCU, after transmitting the initial command, monitors the line for the return echo. If no echo is received within a fixed time interval, a transmission error is deemed to have occurred, and the computer is informed via the I/O error control.

When the BCU receives the echo check, it accepts the requested number of data bytes, verifies parity, and transfers the data to the requested locations in computer memory, after which the read transaction is completed.

* Serial transfer is considered advantageous in minimizing the number of interconnections.

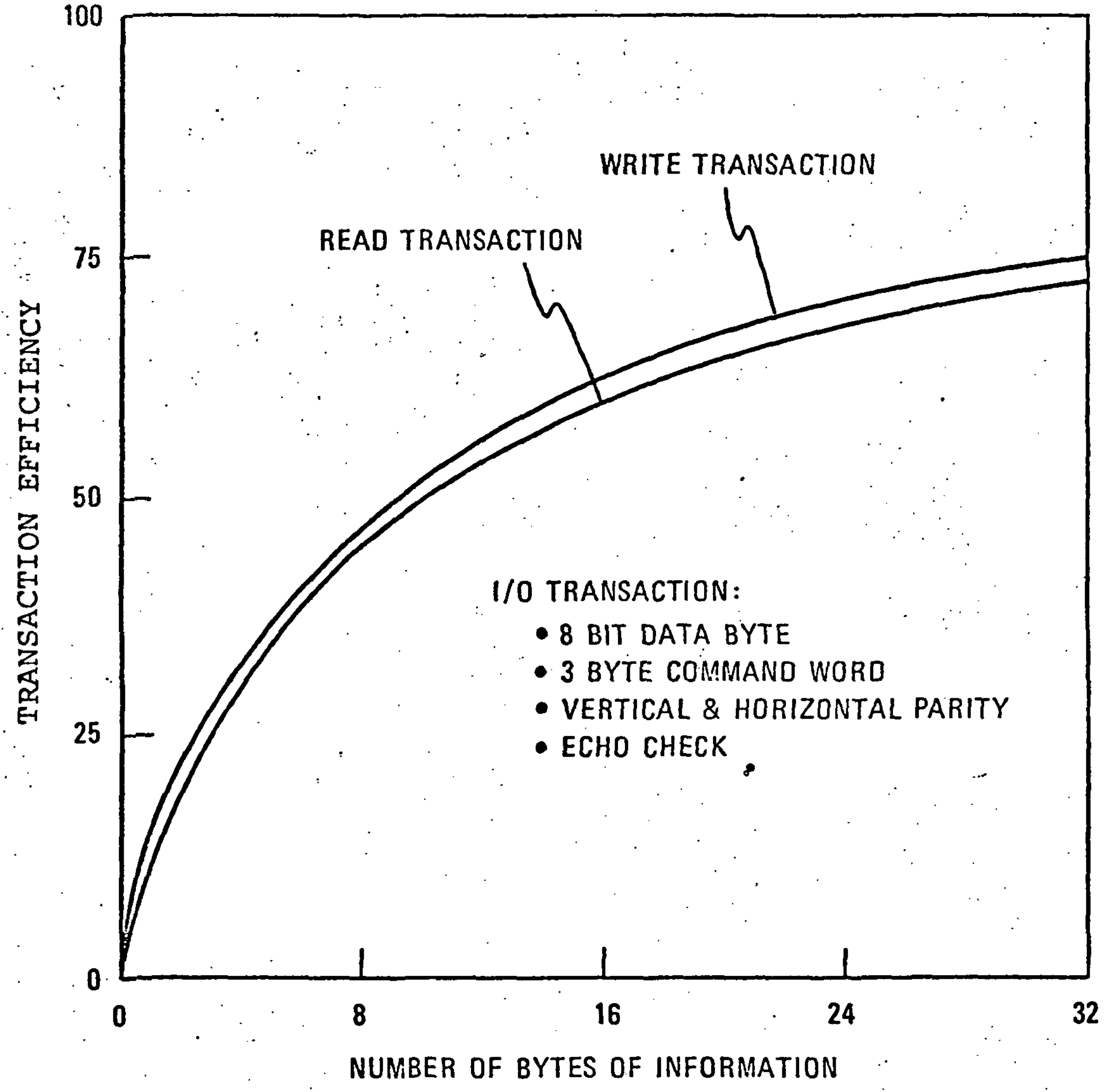


Figure A.5 Bus I/O transaction efficiency

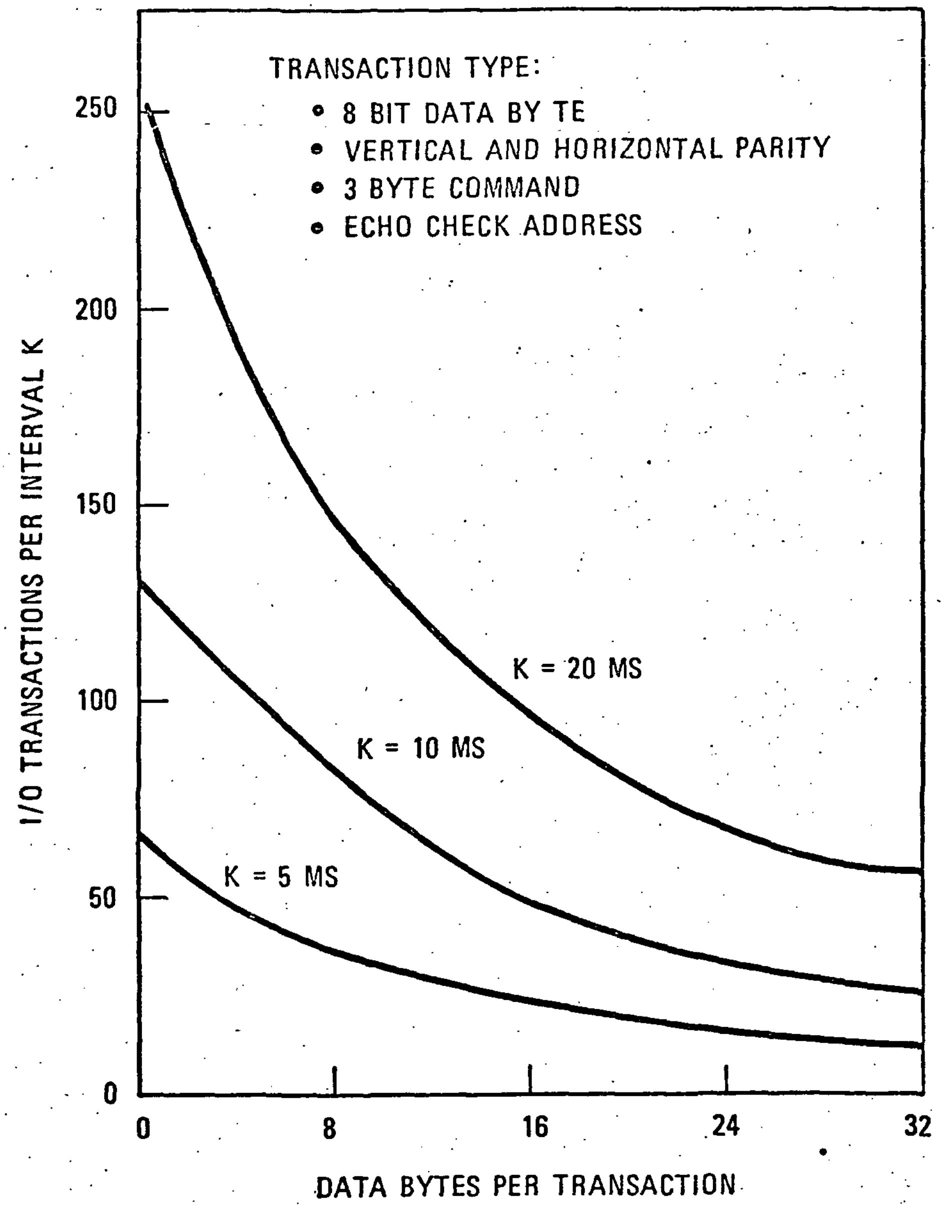


Figure A.6 Frequency of I/O transactions versus number of data bytes

Write transactions are performed using similar procedures as illustrated in Figure A.4. A total time to complete an I/O transaction using this command structure and error control procedures has been estimated for a block of size N bytes to be approximately:

WRITE transaction = (59 + 9N) μs

READ transaction = (69 + 8N) μs

A.4.4 Bus Efficiency and Latency

A.4.4.1 Efficiency. The bus utilization efficiency can be computed by the ratio of information bits in a transaction to the total number of bits in the transaction. If we consider the total number of bits in a transaction to be the total transaction time (including delays, etc.) times the bus speed (assumed to be 1 MBPS) we obtain a worst case estimate of bus efficiency. Information transfer efficiency estimates for a 3-byte command format are illustrated in Figure A.5.

The bus system will operate at about 50% efficiency for transfers of 10 or more bytes. This illustrates the obvious fact that to maintain efficiency the software should be structured to obtain information from LRU's in blocks. For example, status data should be obtained in functionally related groups, such as all temperature readings.

A significant factor is the number of I/O transactions that the bus can complete in a minor bus control cycle. Figure A.6 contains a plot of the I/O transactions, consisting of a given number of data bytes, which can be completed during a fixed interval of time. Based on an average block of length 8 data bytes, approximately 70 transactions can be completed during a 10 ms interval. It is apparent that even though the efficiency of information transfer may be less than 50% in most cases, the actual number of transactions completed during an interval of time should be adequate to service the expected Shuttle I/O requirements. Figure A.6 illustrates that careful scheduling of the bus during any minor cycle will be required, particularly if the size of blocks vary.

A.4.4.2 Subsystem Latency. When a read transaction command is received by the EIU, an interval of time is required, called the latency time, for the EIU to interpret it, to carry out the command, and return the data. A delay can be causes by analog-to-digital conversion, serial/parallel conversions, inherent equipment dynamics, etc. If an I/O request from the computer has a latency time exceeding a certain fixed interval, it must be organized into two or more transactions. An example is the computer request for DME transponder range. The inherent characteristic of the DME is that to obtain range to a specific point, the DME measures the time a signal takes to traverse the distance to that point and back again. The latency time required for this operation is intolerable in the I/O transaction structure described above. This type of transaction must be divided into two transactions: one to command the range to be read, and the other for reading the range. Coordinating these interdependent transactions so that they occur at the right time, presents problems to the I/O scheduling software design.

A form of latency occurs for certain types of block data transfer from computer to subsystem. Error control that depends on horizontal and vertical parity cannot provide verification of the correct receipt of a data block until the last byte has been received (the last byte is, in fact, the vertical parity byte). To prevent erroneous data from being transmitted to a subsystem, the complete block must be buffered at the terminal until it is verified. It is subsequently transmitted to the subsystem for which it is intended. However, this second transmission may take a considerable time, by bus standards: a 32 byte block will take over 0.25 milliseconds at 10^6 bits per second. This is enough time for several other transactions to take place.

For both kinds of latency, it is essential to allow no inadvertent interference with the terminal from other transactions. For this reason it is desirable to provide for the indication of an EIU/LRU "busy" condition via the status bit(s) associated with the SIU echo return. This bit can be interrogated by the BCU to provide an I/O error indication to the computer whenever another command is addressed to the busy terminal.

A.5 I/O Timing Difficulties

A class of system problems exists in the operation of a time shared bus which is associated with the correlation of data and commands with "time". For example:

a) Correlation of data and absolute time. Several system computations demand the acquisition of data from separate subsystems at the same time. For example, a navigation measurement combines sensor data with attitude information, correlates both to the same absolute time, and updates the navigation data. With a synchronously controlled data bus, in which samplin is performed only at fixed minor cycle intervals, time may only be established with a granularity of the sampling period. That is, all samples taken during one minor cycle are associated with the same time tag. If a finer time reference is required it must be provided by a local clock. In an asynchronously driven bus system a finer reference time quantization may be obtained because a specific I/O command may be serviced within approximately 100 μs (depending on the I/O queue backlog).

A related processing problem arises in the derivation of a rate of change by differencing two measurements. In this case a difference in time must be either assumed or computed for two measurement samples. For high frequency samples obtained with a synchronously driven bus, the order of the I/O command in the list may be important, particularly if a fixed delta time is assumed in the calculation.

b) Local precision timing. Another problem that may arise concerns the precision timing of events at geographically separate and remote subsystems, for example, the timing and coordination of firing commands to the RCS jet thrusters. From a system point of view, it is desirable to design such subsystems to receive a message which contains not only the command but also the firing interval. The impact on I/O complexity, bus traffic and response, of separate transmissions to command the thruster on and then off could be considerable if this type of bus activity predominates. The capability for local precision timing may be incorporated into the subsystem or terminal.

Page Intentionally Left Blank

Appendix B

Data Bus Error Control

B.1 Introduction

Since the Shuttle data bus provides the sole communications for onboard avionics equipment, an important design requirement is that it provide a reliable transfer of information in the presence of both permanent and transient failures. Permanent failures are caused by equipment failures and are a direct function of the simplicity and reliability of the data bus system elements (i.e. BCU, bus, SIU, EIU, and LRU). Transient failures are caused by such effects as electromagnetic interference, which must be anticipated in the Shuttle environment. The characteristics of the interference are anticipated to be predominantly impulsive, and primarily caused by coupling to the line of transients and noise from switches, motors, relays or other sources. "Burst errors" involving multiple errors close together are to be expected in this environment. A major task of the data bus design will be to incorporate an error control approach which provides "security" of communication in the presence of noise of largely unknown characteristics.

Several error control techniques have been applied in communication systems to reduce the probability of undetected errors. The techniques generally attempt to satisfy a probability goal within the system design constraints of cost, weight, power, or bandwidth.

There are two basic objectives of the shuttle data bus error control scheme to be satisfied in the presence of potential permanent and transient errors:

a) To maximize the probability that a transmitted message is correctly received by the correct terminal;

b) To minimize the probability that an incorrect message is received.

Most commonly a particular error detection scheme has been coupled with retransmission or forward error correction. Various forms of information coding to obtain an error detection and/or correction capability have been used. Numerous codes have been devised to satisfy a particular communication channel error probability. Prior to discussing the specific error control approach appropriate to the shuttle data bus, a review of information coding schemes is presented with a discussion of their advantages and disadvantages.

B.2 Information Coding Review Discussion

B.2.1 Coding Theory

Coding modifies the message to be transmitted by adding redundant bits to the transmitted message. These extra bits are examined at the receiving terminal to determine whether an error has been introduced and in some cases to locate the error bit within the message so that it can be corrected.

The methods of detecting and correcting errors can most easily be explained with the aid of the concept of Hamming distance. Briefly, the Hamming distance between two strings of binary symbols (of equal length) is the number of positions in which the symbols in the string are different. Thus, the symbol strings 1100 and 1000 are separated by a Hamming distance of 1, while 1100 and 0011 are separated by a distance of 4.

In the study of codes, one of the parameters of interest is the minimum Hamming distance between any two valid code words in the set (for codes in which all the code words contain the same number of bits). Thus, if a code has a minimum Hamming distance of two between any code words, at least two symbols must be changed in order to change one valid code word into another valid code word. With such a code it would be possible to detect any single symbol error, and also many but not all, possible errors affecting more than one symbol.

B.2.2 Single Parity

A common example of such a code is the single parity check, in which the code word is generated from the binary message string to be transmitted by adding a single bit such that the total number of "1's" in the code word is even (or odd). The choice of even or odd parity has no effect on the random error correcting properties of the code, and is usually made to facilitate the detection of certain equipment failures which can produce all "1's" or all "0's" in the received message.

In particular, errors affecting an odd number of bits will be detected but errors affecting an even number of bits will not. The single parity bit is extensively used for error control, principally because of its simplicity in terms of hardware. It is effective against random independent noise.

B.2.3 Error Correcting Codes

For some applications, the mere detection of an error is not sufficient. It is necessary to determine from the received symbol string the nature of the error, or, to be more precise, to determine the message that should have been received in the absence of noise. This can be achieved by error correction codes.

B.2.3.1 Hamming Single Error Correcting Code. The well-known Hamming single error correcting code is an example. This is a code having words of length 2^m-1 where m is any integer. There are m parity bits and 2^m-1-m information bits. The construction of the code word from the message bits will be illustrated for m=3.

Bit Position	B_1	B_2	B_3	B_4	B_5	B_6	B_7
Parity-Message	P_1	P_2	M_1	P_3	M_2	M_3	M_4

The parity bits are determined from the equations:

$$P_1 + M_1 + M_2 + M_4 = 0 \text{ (or 1)} \quad \text{(modulo 2 additions)}$$

$$P_2 + M_1 + M_3 + M_4 = 0 \text{ (or 1)} \quad \text{"}$$

$$P_3 + M_2 + M_3 + M_4 = 0 \text{ (or 1)} \quad \text{"}$$

At the receiver, the three parity equations are checked to give three error states E_3, E_2, and E_1. (A "1" denotes that the equation did not check, and a "0" indicates that it did.) These three error bits are ordered as a binary number $E_3E_2E_1$, called the syndrom, which equals number of the message bit that should be changed.

If two or more errors occur in the transmission, then either the received word passes the parity tests and is incorrectly accepted by the decoder, or the decoder recognizes that an error has occurred but incorrectly identifies the nature of the error and incorrectly "corrects" the received message.

The Hamming codes that are discussed here have the interesting property that every possible received word is within the error correcting distance (in this case a "sphere" with a "radius" of a Hamming distance 1) of some valid code word. A code having this property is called a perfect code or a close packed code [1]. In general, most codes do not have this property. In fact, for codes capable of correcting more than one error, only a few such codes are known.

B.2.3.2 Augmented Hamming Codes. In the case of non-perfect codes, several strategies can be used when the received message is not within the specified correcting range of any valid code word. On one hand, the distance to each valid code word can be determined and the nearest valid code word selected for the decoder output. If two valid code words are equidistant, outside knowledge of the message probabilities could be used to resolve the tie. At the other extreme, any received message not within the assured error correcting range of the code could be labelled as a detected but uncorrectable error.

An example of a code for the latter strategy is the augmented Hamming code generated from the Hamming code described earlier by adding one additional overall parity bit. This code has a minimum distance of four, and, while it is not a perfect code, every possible received sequence is within a Hamming distance of two of one or more valid words. This code can be used as a single error correcting, double error detecting code.

It is worth noting that a particular code can be used in a number of different ways, depending on how the decoder is mechanized. The extended Hamming code will detect some but not all higher order errors (and will "correct" some other high order errors to produce a wrong message). The same code could also be used as a triple error detecting code. In this case, the code will also detect many more of the higher order errors. In fact, it will detect any error pattern that does not convert the transmitted code word to another valid code word.

It has also been shown that this same code can correct all single errors and also all double errors in adjacent bits, provided the parity bit is not in error [19]. Using this decoding procedure very few if any higher order errors will be detected.

B.2.4 Higher Order Error Correcting Codes

Codes are known which have sufficient Hamming distance between valid words so that they can correct two or more errors in a block. In general, these codes are either trivial (repetition of each message bit an odd number of times with majority voting, called a binary repetition code), or are too complicated to describe in detail here.

Among the better known of the constructive (non-random) codes are the Reed-Muller codes [20], and the Bose, Chandhuri and Hocqueughem (BCH Codes). BCH codes are a generalization of Hamming codes for multiple error correction. The correction procedures are, however, fairly complicated. The technique for BCH error correction consists of solving the roots of a N degree polynomial and a set of N equations, where N is the number of correctable errors. The complexity of the correction process forces BCH codes to be considered only for error detection. Correction becomes feasible if a processing capability is available, and a delay in the receipt of the message is acceptable. BCH codes are cyclic codes and have the disadvantage of being sensitive to loss of synchronism since shifted cyclic code words are also valid code words.

B.2.5 Burst Errors and Burst Codes

In many instances where coding has been employed to detect or correct random errors in a data transmission system, the improvement in system performance has not been as great as expected. The reason is often that the assumption of additive white gaussian noise, or other mechanisms which generate independent bit errors, is not valid. Generally, in a real environment the errors occur in groups or bursts. Electromagnetic interference of duration longer than one bit transmission time would be an error source with this characteristic.

A simple example is provided below to illustrate such a problem. Consider the case of a system operating at one million bits per second, and using coherently detected amplitude modulation at 15 db signal to noise ratio. We will assume that the system is perturbed by gaussian noise so that errors are random and independent. The probability of a bit error for this condition can be calculated to be one in 1.26 x 10^8 bits. The code is a three error correcting code having 23 bits, with 12 of them information. The example is a special case known as the

Golay code. This code is close packed, and we can, therefore, neglect all of the possibilities of detecting higher order errors as they always result in a word error. The following observations are made:

a) a single bit error in a word is expected with probability $23 \times 7.9 \times 10^{-9} = 1.8 \times 10^{-7}$ per word, or once every 126 sec.

b) a double bit error will occur with probability 1.6×10^{-17} or once every 47.5 years.

c) the probability of three or more errors and consequently the probability of an undetected error in a word is vanishingly small.

If, however, the mechanism of the disturbance is such that for 10 consecutive bits the probability of error is 0.5, there will be an average of 5 errors in the burst of ten bits, so error bursts will occur every 630 seconds. Since .17 of these bursts will have three or less errors, and neglecting the fact that in some cases a burst laps over the division between two blocks, a decoding error will occur approximately every 25 minutes.

The description of the burst error channel given above is obviously a very simple case. Yet it illustrates the significant difference in conclusions which can be drawn about the expected performance of a control approach.

Some general observations can be made on the performance of error control codes in the presence of burst noise. If a code with a minimum Hamming distance of h is used as an error detecting code, any burst causing up to (h-1) errors will be detected. For bursts causing more than (h-1) errors, most, but not all, will be detected. The exact percentage of errors of various lengths that will be passed depends on the code used.

At the other extreme, if the burst is sufficiently long and severe, so that the received bits have no correlation with the transmitted message but are instead received with a probability of error of 1/2 for each bit, then an estimate of the probability of passing an error is again possible. If the coded word has n bits, k of which are information, the remaining (n-k) bits are redundant. The k information positions in the word can be filled by the random process with any bits, and there will then be one and only one set of values for the redundant bits that will result in a coded word. The probability of this particular set of values being chosen is $(1/2)^{n-k}$.

The assumption that a noise burst will result in bits being received as "1" or "0" with probability 1/2 is, however, not always valid. Sometimes a noise burst (or hardware failure) is more likely to cause errors in one direction, such as turning "1's" to "0's", than the other direction. Such situations arise from the details of the modulation scheme used and the design of the hardware, and are very difficult to evaluate in a general way. When possible, it is usually good design practice to design the code so that the most likely types of equipment failures will not result in a valid code word. Examples of this would be elimination of all "1's" and/or all "0's" as valid code words.

B.2.6 Fire Codes and Other Burst Codes

Some special error correcting codes have been developed which are especially applicable to error correction in channels which are subject to burst errors. For a given level of redundancy, these codes are able to correct more errors in a burst than would be possible if the errors were assumed to be random. These codes require long blocks and complicated decoding procedures. Two examples of these codes are cited:

a) Fire Codes

Fire codes are oriented towards a single burst of errors per message. They are inefficient for short blocks, however, and are not particularly good for multiple bursts on a single block.

b) Reed-Solomon Codes

The Reed-Solomon codes are a special case of the generalized BCH codes, oriented toward multiple burst error correction. They are moderately efficient, and for the same block length are similar to BCH codes in decoding complexity.

B.2.7 Horizontal and Vertical Parity Coding

A coding technique which has been proposed for the Shuttle baseline data bus systems is vertical and horizontal parity coding. This coding scheme assigns a single parity bit to each byte or word of the message (horizontal parity), and an extra byte or word for vertical parity on the preceding bytes. This approach detects all odd numbers of errors. An undetected error can only occur when each byte and every bit position contains an even number of errors. The scheme fails to detect errors only when an even number of errors, equal to or greater than four, occurs with the errors paired in rows and columns. The efficiency of this approach is moderately high for messages of several bytes, but is poor if the number of bytes of data in a message is small. For example, the effective information rate of an 8 bit byte of data would be computed by

$$E_{IR} = \frac{8N}{9(N+1)} \qquad \text{where N is the number of bytes}$$

It can be seen that for a small number of data bytes the efficiency is low (i.e. 44% for 1 byte, 59% for 2 bytes). When the block size increases, however, the coding scheme becomes more efficient (i.e. 79% for 8 bytes, 91% for 32 bytes). Although there are more efficient coding techniques, this scheme has a major advantage in that its implementation in terms of the encoding, decoding and detection logic required in the SIU, EIU, and BCU data bus equipment is probably the simplest.

B.2.8 Repeated Transmission

The repeated transmission of a data message over a single path is a well-known method for error detection. Detection is accomplished by requiring all messages received to be identical. The time diversity, or spacing of transmissions provides independence.

Implementation of this approach as the prime error control approach in the Shuttle data bus would require the BCU to transmit the (uncoded) data to the remote station, and vice versa, two or more times. The remote terminal would require a comparator or voter to determine an "acceptable" transmission. Retransmission for error correction is still required for ambiguous voting results.

The method is relatively simple to implement, but is very inefficient, particularly for block transmission. In order to get a Hamming distance four code for three error detection, the message must be repeated four times. The same error detecting capability can be obtained with many fewer bits using other coding schemes.

B.2.9 Transmission Over Multiple Paths

The transmission of the message over multiple separate paths between a single BCU and single LRU is similar to the redundant transmission over a single path. It is true that the message is received and verified at the output with less delay than is associated with the sequential transmission scheme, but on an overall basis, there is no improvement in the utilization rate of the available channel capacity. The necessity of providing parallel channels to allow continued operation in the event of a permanent hardware failure would directly affect the Shuttle data bus if it were the prime error control method used. It would require independent paths to be maintained for the FS mode of operation, increasing the number of buses required for FO/FO/FS.

The approach would increase the complexity of the BCU and SIU units, since it requires transmissions over multiple paths to be synchronized, so that comparison or voting could be performed at the receiver, or storage for delayed receipt.

B.2.10 Data Feedback/Echo Check

In this method, uncoded data is saved in buffer storage at the transmitting element and sent to the receiver. The receiving element transmits back the entire message. The transmitting element then performs a bit-by-bit verification of the entire message. Upon verification by the transmitter, the receiving element is instructed to use the information on receipt of a "verify" message from the transmitter.

If an error is detected the transmitting unit can retransmit the entire message. If the error was caused by an external noise transient, the second transmission should be valid. This method is referred to as an echo. One of the problems with this approach is the probability of transmitter's verification being in error. An endless chain of echoes may result in requiring the receiver to echo the echo, etc. Complete feedback of all data requires twice the time to transmit a message. Its main advantage is the high degree of error detection it provides.

B.3 Detection and Retransmission Vs. Forward Error Correction

In the analysis of data transmission systems, two distinct cases have been studied. The first case is Forward Error Correction, in which the decoder at the receiver studies the received message and, if an error is discovered, attempts to deduce the correct message from what was actually received. The second case is retransmission, in which the decoder checks the received message for signs of error, and if an error is detected the decoder informs the transmitter. The transmitter can then retransmit the message or take whatever other action is indicated.

A forward error correction scheme is considered undesirable for the Shuttle data bus since it would require too much complexity at the terminal and BCU, particularly for correcting more than 1 error in a message. The method preferred is to combine an error detection scheme with retransmission for recovery.

The advantages of the retransmission approach to error recovery are reduced complexity of the decoder and the reduction in the probability of an undetected error for a given level of coding.

The classic studies of retransmission systems were reported in two papers by Benice & Frey in 1964 [21]. In these papers, three cases were considered:

1. Idle RQ - in which the transmitter sends a message and then sits idle until the decoder indicates whether a retransmission is requested. Presumably, this includes a "no response" from the terminal.

2. Simple RQ - in which messages are sent continuously. When an error is detected and a retransmission requested, the source repeats the requested message.

3. Dual RQ - in which messages are transmitted as in Simple RQ, except that the requested message and all subsequent messages are repeated.

The Idle-RQ system appears to be most appropriate to the Shuttle data bus, since the bus traffic is expected to consist of a large number of relatively short communications between the bus controller and the many terminals along the bus. The advantages of the other schemes are achieved when full duplex transmission systems (simultaneous continuous transmission in both directions) is used. The Shuttle data bus is not expected to be used in this manner.

The conditions for which the Idle-RQ scheme becomes a poor candidate are not applicable to the Shuttle data bus. In many data transmission systems, the transit time of the channel is long compared to the length of a message. Thus, the transmitter wastes a lot of time sitting in the idle state waiting for the message OK or retransmit signal. In the Shuttle data bus, the round-trip time to the farthest subsystem will only be a few microseconds, or bits.

In the data presented by Benice & Frey, the computed probability of an undetected error for the Idle RQ system drops rapidly until a certain minimum probability is reached, and then no further improvement is possible. This behavior is traced to the failure of the retransmission request to be recognized at the transmitter. The minimum error probability is the probability that some kind of error will be detected in the forward message, and then the retransmission request is changed to a confirmation that the message was OK.

In the other two retransmission schemes, the retransmission request was encoded as a part of a message moving in the opposite direction and was, therefore, protected by the same level of coding as the original message. The occurrence of any error in a returned message was construed to be a retransmission request for the forward message. This attitude results in a small decrease in throughput rate, and a large decrease in probability of an undetected error.

In the Idle RQ scheme, Benice and Frey postulated a one bit confirmation message for most of the work, and this results in a minimum probability of undetected word error of about 5×10^{-8} for a bit error probability of 10^{-5} and a 511 word message. By changing the returned accept retransmit request message to a 7 bit format, the minimum probability of an undetected error was reduced to 5×10^{-38}. The point to be made here is that the retransmit request must be suitably protected if it is not to turn out to be the limiting factor in the probability of error in the transmission system. The penalty for this is a slight reduction in the throughput rate of the system, which does not appear to be a prime consideration in the Shuttle data bus system.

References for Appendix B

1. Berlekamp, E.R., Algebraic Coding Theory, McGraw Hill Book Co., New York, 1968.

2. Abramson, N.M., "A Class of Systematic Codes for Non-Independent Errors", IRE Transactions on Information Theory. PGIT5, No. 4. December 1969, pp. 150-157.

3. Peterson, W.W., Error Correcting Codes, The M.I.T. Press, Cambridge, Mass., 1961.

4. Benice, R.J. and Frey, A.H., Jr., "An Analysis of Retransmission Systems", IEEE Transactions on Communication Technology. PGCOM-12, No. 6. December 1964, pp. 135-145; and "Comparisons of Error Control Techniques", Ibid, pp. 146-154.

Appendix C

Literature Review of Avionics Executive Systems

The purpose of this appendix is to review several articles whose content relates to the Space Shuttle executive design. The executive features presented in the articles are outlined, and those having direct bearing on the Space Shuttle executive design are emphasized.

I. "Improved Centaur Computer Operating System", by S.W. Matthews, AIAA Aerospace Computer Systems Conference, 1969[22].

The Centaur executive control allows for a system driven entirely by hardware interrupts, or entirely by a programmed task scheduler, or a combination of both. Matthews feels it apparent that a software system having hardware interrupts for asynchronous nonperiodic demands of peripheral hardware and a programmed task scheduler for semiperiodic tasks, would result in the most flexible hardware/software system. Such a structure is a desirable feature for an aerospace executive system as explained in Chapter 2.

The task scheduler is entered when a task ends or when the real-time interrupt occurs. It operates off a task table which is an ordered list containing the status of the functional tasks to be executed. The order of the list determines task priority since the table entries are processed in sequential order. Each entry contains a task start time, frequency for cyclic tasks, location of task, task interrupt bit, and a special action indicator.

The interrupt bit indicates whether a task has been interrupted by the executive task scheduler; that is, whether a higher priority task received the processor before the former task finished execution. The special action indicator is used as a flag to indicate the requirement of executing a communication or control subroutine. These routines can vary with the particular application and may be added to or deleted from the system as requirements demand. Thus, the system can adapt to its environment through special action routines.

Tasks are executed as a function of list position, interrupt status, and start time. Any task can interrupt any task following it on the list. Thus, the most frequently cycled task must occur first on the list. The control algorithm is shown in Figure C.1. This method of interrupting a task is undesirable on the Space Shuttle because it raises data integrity problems. Whenever a task is interrupted, a copy of all the data which this task shares with other tasks and which can be modified by other tasks, must be saved. The Compool approach is an effective solution to this problem.

II. "A Simple Real-Time Executive for an Aerospace Data Management System", by Peter Adler, MIT Draper Laboratory, E-2579, May 1971 [23].

The basic functions that this executive performs (as indeed) most executives do) are job dispatching, resource allocation, and I/O control.

The dispatcher works off a priority queue of jobs. It is entered when an application program ends and selects the highest priority job for execution. Three priorities are recommended, each having a queue organized on a FIFO basis. Both time and event scheduling are possible in the system. A wait queue for jobs awaiting I/O is suggested but no dynamics are presented.

Adler recommends dynamic storage allocation for job temporary work areas. Thus, reentrant programming and data sharing are possible. To avoid fragmentation of memory, all available storage is organized into equal size blocks with a threaded list structure. Although dynamic storage allocation is a desirable shuttle executive feature, it is unclear whether all allocated blocks should be of equal size. For example, if a task requires several contiguous blocks of storage, and if memory is already fragmented, contiguity will not be possible. However, by having a large sized single block of core ready for allocation, the task's request can be granted.

In Adler's system, jobs are segmented into 10 msec blocks. Every 10 msec a breakpoint allows the job to be suspended if a higher priority job is pending. A programmer must be sure all vital data are entered in temporary storage before a breakpoint occurs. This mechanism also aids in program verification and is a desirable executive feature.

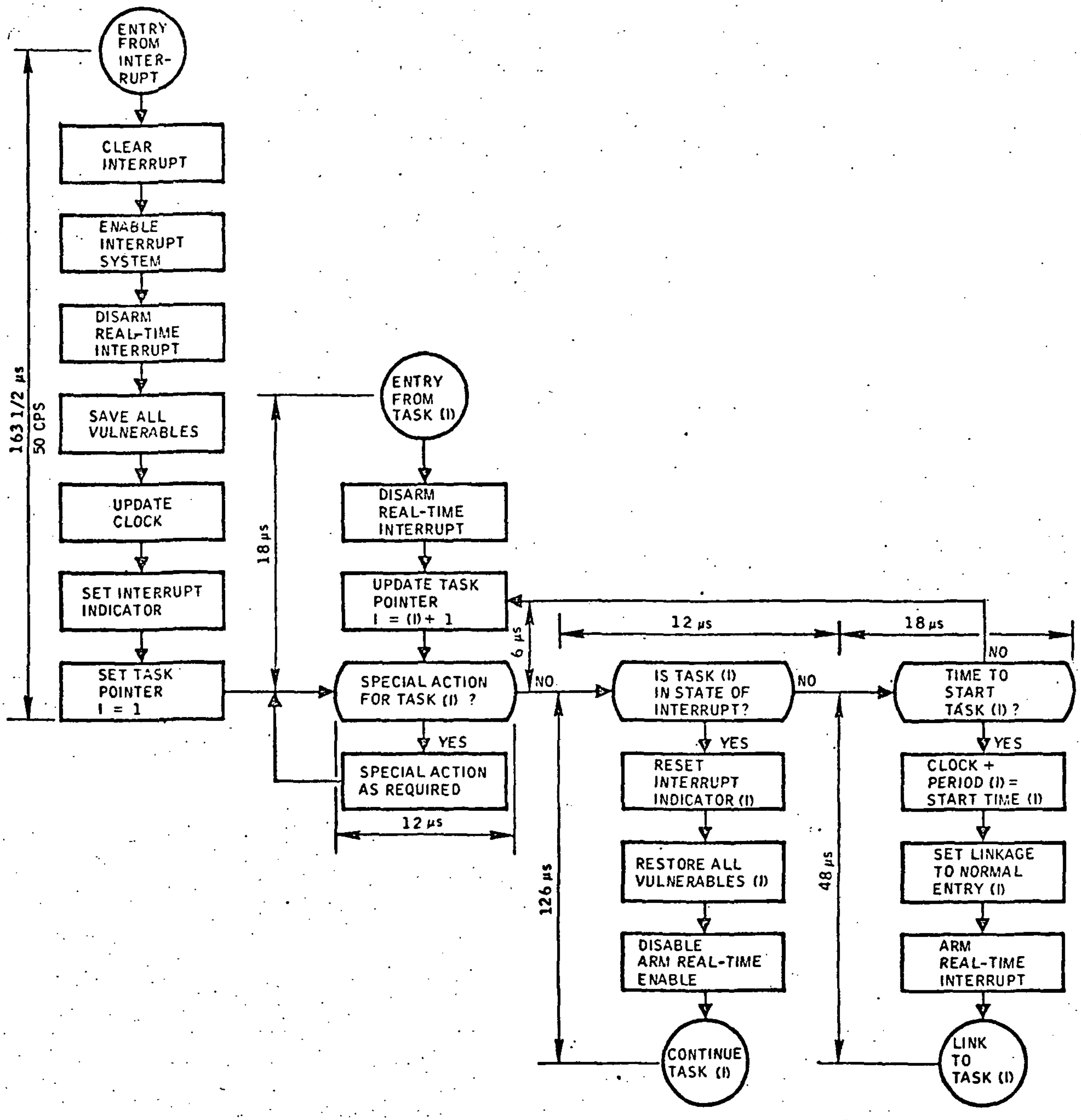

ENTRY FROM INTER-RUPT
CLEAR INTERRUPT
ENABLE INTERRUPT SYSTEM
DISARM REAL-TIME INTERRUPT
SAVE ALL VULNERABLES
UPDATE CLOCK
SET INTERRUPT INDICATOR
SET TASK POINTER I = 1
163 1/2 μs
50 CPS
ENTRY FROM TASK (I)
18 μs
DISARM REAL-TIME INTERRUPT
UPDATE TASK POINTER I = (I) + 1
6 μs
SPECIAL ACTION FOR TASK (I) ?
YES
NO
SPECIAL ACTION AS REQUIRED
12 μs
12 μs
18 μs
IS TASK (I) IN STATE OF INTERRUPT?
NO
YES
RESET INTERRUPT INDICATOR (I)
RESTORE ALL VULNERABLES (I)
DISABLE ARM REAL-TIME ENABLE
CONTINUE TASK (I)
126 μs
48 μs
TIME TO START TASK (I) ?
NO
YES
CLOCK + PERIOD (I) = START TIME (I)
SET LINKAGE TO NORMAL ENTRY (I)
ARM REAL-TIME INTERRUPT
LINK TO TASK (I)

Figure C.1 Control Module Executive Algorithm [22]

To avoid two jobs updating common data, an interlock bit mechanism is proposed. This mechanism could be avoided by judicious program segmentation such that the use of shared data is completed within one program segment.

As an alternative to I/O interrupts, Adler suggests I/O device polling which eliminates the need for hardware buffer queues. All I/O is done by one job to avoid conflicts. There is then one interrupt to allow initiation of the I/O monitor at a fixed frequency. This method of I/O handling is advantageous for data acquisition. However, when data is outputted, there is no way to know when the transmission is done since there is only one fixed frequency interrupt.

Adler avoids mention of synchronous vs. asynchronous structure. The executive he proposed allows time and event scheduling, so it is not fully synchronous. However, jobs can also be scheduled cyclically so it is not fully asynchronous either. This blend of the two structures is a desirable Shuttle feature.

III. "STS Software Development (Study Task 5)", MIT Draper Laboratory, E-2519, July 1970 [24].

MIT lists four criteria for the Space Shuttle executive system:

1. Efficient resource allocation

2. Sufficient features incorporated to permit efficient programming and running of mission oriented programs. These include:

    a. priority execution queue

    b. time execution queue

    c. event execution queue

    d. temporary storage allocation

    e. I/O scheduling

    f. I/O execution

    g. interlocking of shared data

    h. modification of protected data

3. Fast and simple executive execution, e.g., by avoiding looping, indexing, and indirect addressing.

4. Uncomplicated interfaces between executive and application programs.

In addition, application programs must conform to certain criteria.

1. Modularity: there must be rigid and well-defined rules for programs interfacing with each other.

2. Use of executive routines to minimize program overhead.

3. Program segmentation to allow long tasks to be safely interrupted.

4. Temporary storage requests must be done through the executive.

Dynamic storage allocation is also recommended to minimize conflicts over dedicated locations and to allow for reentrant subroutines. As mentioned above, this is a desirable executive feature on the Shuttle.

These criteria for both the executive and application programs support Intermetrics' views on Shuttle programming as evidenced in the features of our executive system design.

Page Intentionally Left Blank

BIBLIOGRAPHY

1. IBM Corporation, "Space Shuttle Phase B Software Specification (Preliminary)", IBM No. 70-D33-0020, December 21, 1970.

2. McDonnell Douglas Corporation, "Space Shuttle Data: Avionics", MDC E0395, June 30, 1971

3. IBM Corporation, "Programming Manual for System 4 Pi Model EP", in Aerospace Digital Computer Data for Mission Module Contractor, IBM No. 66-M22-020A.

4. IBM Corporation, "OS/360 Supervisor and Data Management Services", IBM No. GC28-6646.

5. IBM Corporation, "Space Shuttle Executive Control Program (preliminary)", Huntsville, Aug. 16, 1971.

6. Intermetrics, Inc., Development of an MSC Language and Compiler, Cambridge, Mass., June 1971, prepared under Contract NAS 9-10542.

7. Intermetrics, Inc., The Programming Language HAL - A Specification, Cambridge, Mass., June, 1971, prepared under Contract NAS 9-10542, MSC Document # MSC-01846.

8. Intermetrics, Inc., Standard Interface Definition for Avionics Data Bus Systems, Cambridge, Mass., May, 1971, prepared under Contract NAS 9-11477.

9. Coffman, E., et al, "Deadlock Problems in Computer Systems", Proc. Conf. sponsored by Software World, U. Sheffield, April 1970, pp.41-48.

10. Coffman, E., et al, "System Deadlocks", Comp. Surveys, 3(2), June 1971, pp. 67-78.

11. Denning, P., "Resources Allocation in Multiprocess Computer Systems", PH.D. Thesis, MIT, May 1968.

12. Dijkstra, E., "Structure of The Multiprogramming System", CACM, May 1968, pp. 341-346.

13. Habermann, A.N., "Prevention of System Deadlocks", CACM, 12(7), July 1969, pp. 373-377.

14. Holt, R.C., "Comments on Prevention of System Deadlocks", CACM, 14(1), January 1971, pp. 36-38.

15. Murphy, J.E., "Resource Allocation with Interlock Detection in a Multi-task System", Proc. FJCC, 1968, pp. 1169-1176.

16. Pepe, J., "Protection Strategies in a Multiprocessor Computer", Intermetrics, Inc., Multiprocessor Memo #03-71, July 1971.

17. Vyssotsky, V., et al, "Structure of the MULTICS Supervisor", Proc. FJCC, 1965, pp. 203-212.

18. Berlekamp, E.R., Algebraic Coding Theory, McGraw Hill Book Co., New York, 1968.

19. Abramson, N.M., "A Class of Systematic Codes for Non-Independent Errors", IRE Transactions on Information Theory. PGIT5, No. 4. December 1969, pp. 159-157.

20. Peterson, W.W., Error Correcting Codes, The M.I.T. Press, Cambridge, Mass., 1961.

21. Benice, R.J. and Frey, A.H., Fr., "An Analysis of Retransmission Systems", IEEE Transactions on Communication Technology. PGCOM-12, No. 6. December 1964, pp. 135-145; and "Comparisons of Error Control Techniques", Ibid, pp. 146-154.

22. Matthews, S.W., "Improved Centaur Computer Operating System", AIAA Aerospace Computer Systems Conference, 1969.

23. Adler, P., "A Simple Real-Time Executive for an Aerospace Data Management System", MIT Draper Laboratory, Report E-2579, May 1971.

24. MIT Draper Laboratory, "STS Software Development (Study Task 5)", Report E-2519, July 1970.