952-5497-2

SOFTWARE DATA

SDRL # 45

Second Submission
(Final)

H. H. Bloem
R. E. Eckstrom
D. G. Stark
R. B. Talmadge

IBM NO 66-M22-024A

ORIGINATING GROUP Department M22

CONTENT APPROVED BY

CONTRACT NO AF04(695)-904
DAC-A-066-608

DATE March 21, 1966

First Submission: March 5, 1966

IBM FEDERAL SYSTEMS DIVISION
ELECTRONICS SYSTEMS CENTER
OWEGO NEW YORK

TABLE OF CONTENTS

		Page
1.0	INTRODUCTION	1
1.1	General Considerations	1
1.2	Programming System	2
1.3	Implementation Plan	4
1.4	Application Programs	5
2.0	THE SPACEBORNE SYSTEM	6
2.1	Program Operation	7
	2.1.1 Program Structure	8
	2.1.2 Segments	9
	2.1.3 Repetition Rate	9
	2.1.4 Flow Control	9
	2.1.5 Links	10
2.2	Executive Control	10
	2.2.1 System Executive	10
	2.2.2 Basic Executive	13
2.3	System in Operation	15
	2.3.1 Scheduling	15
	2.3.2 Segment Flow	19
	2.3.3 Program Control Block	19
	2.3.4 System Flow	20
	2.3.5 Core Management	21
	2.3.6 Interrupt Supervision	22
	2.3.7 Asynchronous Input/Output	23
	2.3.8 Program Loading	25
	2.3.9 Core Storage Layout	26
2.4	Program Initiation	28
2.5	System Initiation	28
2.6	System Overhead	30
	2.6.1 Executive Times	31
	2.6.2 Core Storage Requirements	35
2.7	Flow Charts and Tables	35

Page

3.0 GROUND SUPPORT SYSTEM 49

3.1 General Requirements 49
3.2 Existing Operating Systems 50
3.3 Program Preparation 53
3.3.1 OS/360 Language Processor Output 53
3.3.2 The Linkage Editor 54
3.3.3 The Preparation Language 56
3.3.4 Operation of the Preparation Processor 59
3.4 Language and Language Processors 60
3.4.1 Assembly System 61
3.4.2 The Compiler Systems 62
3.4.3 Segment Definition 64
3.5 Simulation and Test 67
3.5.1 The Simulation Supervisor 68
3.5.2 Program Test 70
3.6 Integration of the Ground Support System 72

4.0 IMPLEMENTATION PLAN 74

4.1 Task Sequence 74
4.2 Milestones 84

5.0 APPLICATION PROGRAMS (Classified Attachment IBM CD-3-260-8793A)

LIST OF FIGURES

		Page
1.2	DCSG Software - Hardware - Vehicle Functional Interface	3
2.1.1	Program Structure	8
2.2.1	System Executive	11
2.2.2	Basic Executive	14
2.3.1	Scheduling Example	18
2.3.9	Core Map	27
2.6.1	Intersegment Execution Times	32
2.6.2	Typical Operating Overhead Times	33
2.6.3	Core Requirements	34
2.7.1	Program Scheduler	37
2.7.2	Segment Flow Analyzer	38
2.7.3	Overall System Flow	39
2.7.4	Core Manager	40
2.7.5	I/O Interrupt Processor	41
2.7.6	I/O Request Processor	42
2.7.7	Program Loader. Part I.	43
2.7.8	Program Loader. Part II.	44
2.7.9	Segment Flow Table	45
2.7.10	Program Request Queue	45
2.7.11	Program Execution List	46
2.7.12	Load Request Queue	46
2.7.13	Unit Request Queue	46
2.7.14	I/O Interrupt Queue	46
2.7.15	Unit Control Block	47
2.7.16	Core Usage List	47
2.7.17	Program Index	47
2.7.18	Core Residence List	47
2.7.19	Program Control Block	48
3.3.2	Linkage Editing	55
3.4.2	PL/1 Subset	65
4.1	DCSG Checkout Sequence. Part I	86
4.2	DCSG Checkout Sequence. Part II	87
4.3	Program Development Implementation Plan	88
4.4	Program Validation Implementation Plan	89

1.0 INTRODUCTION

This paper is submitted in response to the software data requirements requested in the Statement of Work for the Data Computation Subsystem Group (DAC-A-66-608/WS), Items 2.2.3.3, 2.2.3.7 and 2.2.3.8. Although total systems requirements are not known, the programming system design recommended here satisfies all requirements that can presently be foreseen. Furthermore, the design is flexible enough to accommodate a wide variation in these requirements without substantive change.

1.1 General Considerations

For the purposes of this discussion it is assumed that the hardware configuration of the DCSG Guidelines (Document Number 3-449-0296) is a basis for the system, that the two computers are identical, and that the software system applies to both. These assumptions have been made only as a means of supplying detail, as the system design is quite general and applies to any similar configuration. For the same reason, computers from the IBM 4 Pi line have been assumed for the AVE, with a compatible IBM System/360 computer for the AGE.

Analysis of the Guidelines has established the following requirements as being of fundamental importance in the systems design:

1.1.1 Each of the computers in the DCSG is capable of independent action and generally each is involved in separate modes of operation. However, both must be able to operate in any mode, or even in several modes simultaneously.

1.1.2 The action of both computers must be controlled externally, either by the operator or from the ground. Since flexibility of external interaction is of great

importance, information must be supplied from within the system in order to provide a basis for reasonable decisions.

1.1.3 A number of functions are common to all modes of computer operation. Provision must be made to implement these overlapping functions, both system and application, so as to optimize their utilization.

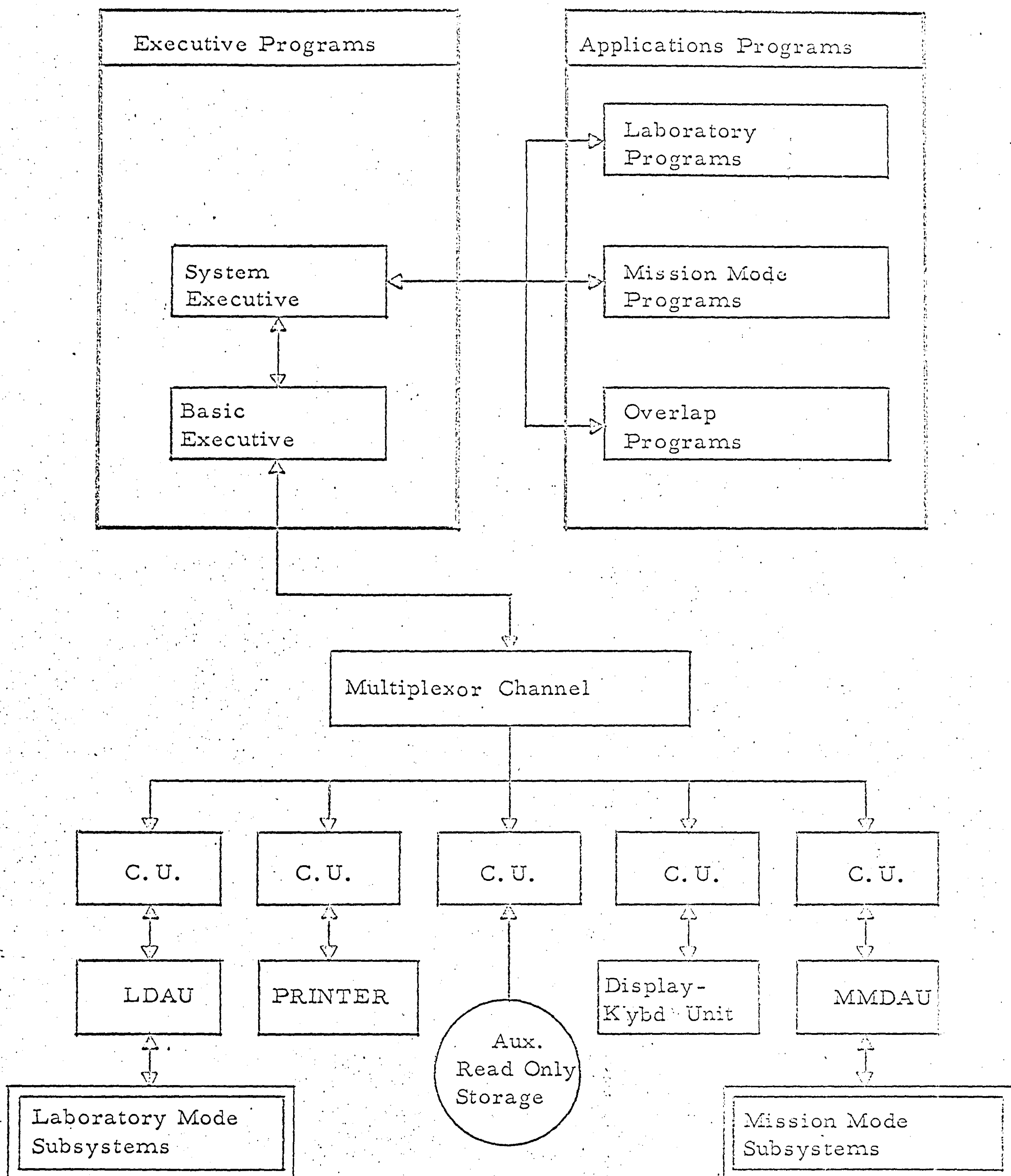
1.1.4 Program preparation (compilation, assembly, and checkout) should be done on a ground based computer which is widely available to the application programmers. An extensive software system is necessary in order to facilitate program preparation.

Consideration of these requirements has led to the design of an integrated programming system to support the operational programs.

1.2 The Programming System

The programming system is divided into two main areas: The Spaceborne System, which is concerned with the execution of operational programs on the DCSG, and the Ground Support System, which is concerned with the preparation and testing of programs on ground based computers.

Figure 1.2 illustrates the general organization of the Spaceborne System and the functional interface of software and hardware components of the DCSG with other vehicle subsystems. The hardware control units shown serve as interfaces between the I/O channel and the attached devices. To control their operation as well as the operation of the system and application programs, identical executive programs are resident in each computer. In order to minimize the complexity of the executive routines, and to reduce overhead required for control, programs



DCSG Software - Hardware - Vehicle Functional Interface

Figure 1.2

must submit to a certain amount of discipline in their structure and operation. This discipline is fully described in Section 2.0 of this document, which covers the Spaceborne System.

Analysis of the Ground Support System requirements has shown that many of these are met by the operating system for System/360. This operating system (OS/360) is a comprehensive system embracing a variety of programming languages and language processors as well as extensive library, editing and debugging facilities. Since OS/360 is a standard product of IBM, it is widely known and readily available to the applications programmer. Two functions must be added to the Standard System. First, there must be a means of preparing the final form of programs to be executed in the DCSG. Second, there must be a method of simulating the operation to these programs, in order to facilitate checkout. These additions provide a complete Ground Support System, which is discussed in detail in Section 3.0.

1.3 Implementation Plan

The development and production of validated programs for a system such as the DCSG requires a carefully developed and controlled plan. The plan must include procedures to span the time from the state of requirements definition to the final flight configuration, with milestones that provide a precise control over the distinct areas of software development and testing. Section 4.0 discusses such a plan in terms of the formal milestone documentation and the steps that should be followed to assure orderly development of the flight programs and the support software.

1.4 Application Programs

Application programs are those which are executed in the AVE to perform the data processing required for both the Laboratory and Mission Modes. These programs are executed under the supervision of the Spaceborne System executive program, which interleaves their execution where necessary to meet real time requirements.

Section 5.0, which is a classified attachment to this report, discusses computing requirements for individual application programs. For Laboratory Mode programs the requirements were generated as the result of analyses based upon two principle sources: the preliminary definition of the programs given in the DCSG Guidelines; and the Subcontractor Technical Directives for some of the individual functions. Estimates of computer storage requirements and program execution times are provided, as well as functional and math flows. For Mission Mode programs the requirements and estimates have been provided by General Electric.

2.0 THE SPACEBORNE SYSTEM

In contrast to previous spaceborne applications, which have involved computers with limited capability dedicated to a small number of functions, the DCSC application involves a complex environment with two computers supporting a number and variety of functions. Successful operation in such an environment requires efficient overall utilization of the hardware, considerable operational flexibility, and simplification of programming procedures. Experience has shown that these can be best obtained by centralizing system control in an executive program. Consequently, the Spaceborne System is based upon such an organization.

Some of the specific benefits which are realized by adopting executive control are the following.

2.0.1 It provides for the insulation of application programs from hardware configuration changes. Thus, hardware may be incorporated to improve overall system performance without affecting the code involved in the application programs. For example, a computer system with a selector channel in addition to a multiplexor channel may be necessary to meet peak I/O data rate requirements. Such an addition can be made without the necessity of reprogramming the applications.

2.0.2 It provides an internal environment in which functions can be carried out simultaneously, yet independently. Hence, Mission mode and Laboratory Vehicle programs can operate in the same computer when desired, and no special planning in program construction is required in order to do so.

2.0.3 The executive undertakes the loading and scheduling of programs for execution. Only those programs required at any one time need be resident in core storage. Hence, core storage requirements may be balanced against the factors of weight and power independently of the total number of programs required in the system.

2.0.4 Centralization of control functions eliminates a considerable amount of redundant code. In addition, adjustments which may be required in control functions are localized, and are not reflected in the application programs. Moreover, such universal functions as external communication, power off control, and inspect and change, can be carried out in an orderly, uniform manner.

2.0.5 Reliability of the overall operation is improved, since the executive can detect component failure, and, within limits, compensate for such failure.

2.1 Program Operation

Program operation is concerned with two major areas: the action of the executive in controlling operational programs; and the structure required of such programs to allow for efficient operation. In this application the most significant requirement of operational programs is that they be executed at a predetermined rate, based upon real-time constraints. This requirement has a substantial effect on the organization of both the executive and the operational programs.

To insure that the required rates are met, the executive program provides a multiplexing procedure to allocate CPU time to the requested programs. The executive itself is structured to operate in the multiplexing mode whenever possible. It should be emphasized that the executive is responsible for this multiplexing, and

the fact that several operational programs may be executing concurrently is of no concern to the programmer.

2.1.1 Program Structure

To simplify the multiplexing procedure and to reduce overhead while maintaining flexibility, the operational programs are structured as shown in figure 2.1.1.

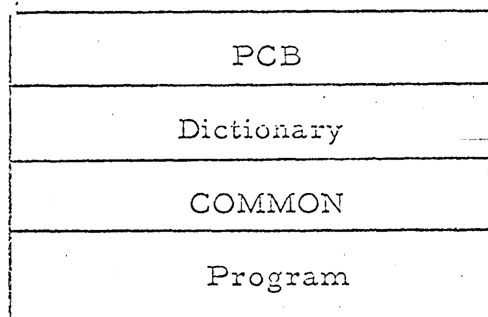


Figure 2.1.1

2.1.1.1 The Program Control Block (PCB) is generated by the system during program preparation and contains information required to properly execute the program. This information governs both the multiplexing procedure and the flow of control.

2.1.1.2 The Dictionary consists of pointers to words within the program which are location dependent and therefore must be adjusted when the program is brought into core storage for execution, or relocated during execution. The dictionary is also generated by the system during the preparation process.

2.1.1.3 The Program COMMON area is the data area which is used to communicate between segments of an executing program. The structure and size of this area is defined during program preparation.

2.1.1.4 The Program consists of the actual instructions to perform the required computations. These instructions are the result of compilation or assembly during the preparation process.

2.1.2 Segments

In addition to this physical structure, the program is logically structured so that the executive receives control at specified intervals. This logical structure involves division of the program into sections whose execution time does not exceed the interval. Such a section of the program is called a segment.

2.1.3 Repetition Rate

Each segment is also an entry point for the program to receive control from the system. In order to meet processing requirements, a certain number of program entries must be made each second. This number is called the Program Entry Repetition Rate. It is provided during the preparation phase and is stored in the PCB. The executive examines the repetition rates for all programs to be run concurrently in order to interleave their execution. In addition to this information, programs may request execution at particular time intervals, or specific times.

2.1.4 Flow Control

Flow of control during execution is governed by means of sequence control information contained in the Program Control Block. This information is provided in the form of Sequence Control Statements during program preparation. These statements direct the order in which segments are executed by providing the precedence relationships between segments and conditions dictating their entry.

The order can be dynamically altered during execution by means of triggers set by the executing segments and tested by the executive.

Use of the Sequence Control Statements permits a complicated program to be described in a straightforward manner, and allows a flexibility in program modification not otherwise obtainable.

2.1.5 Links

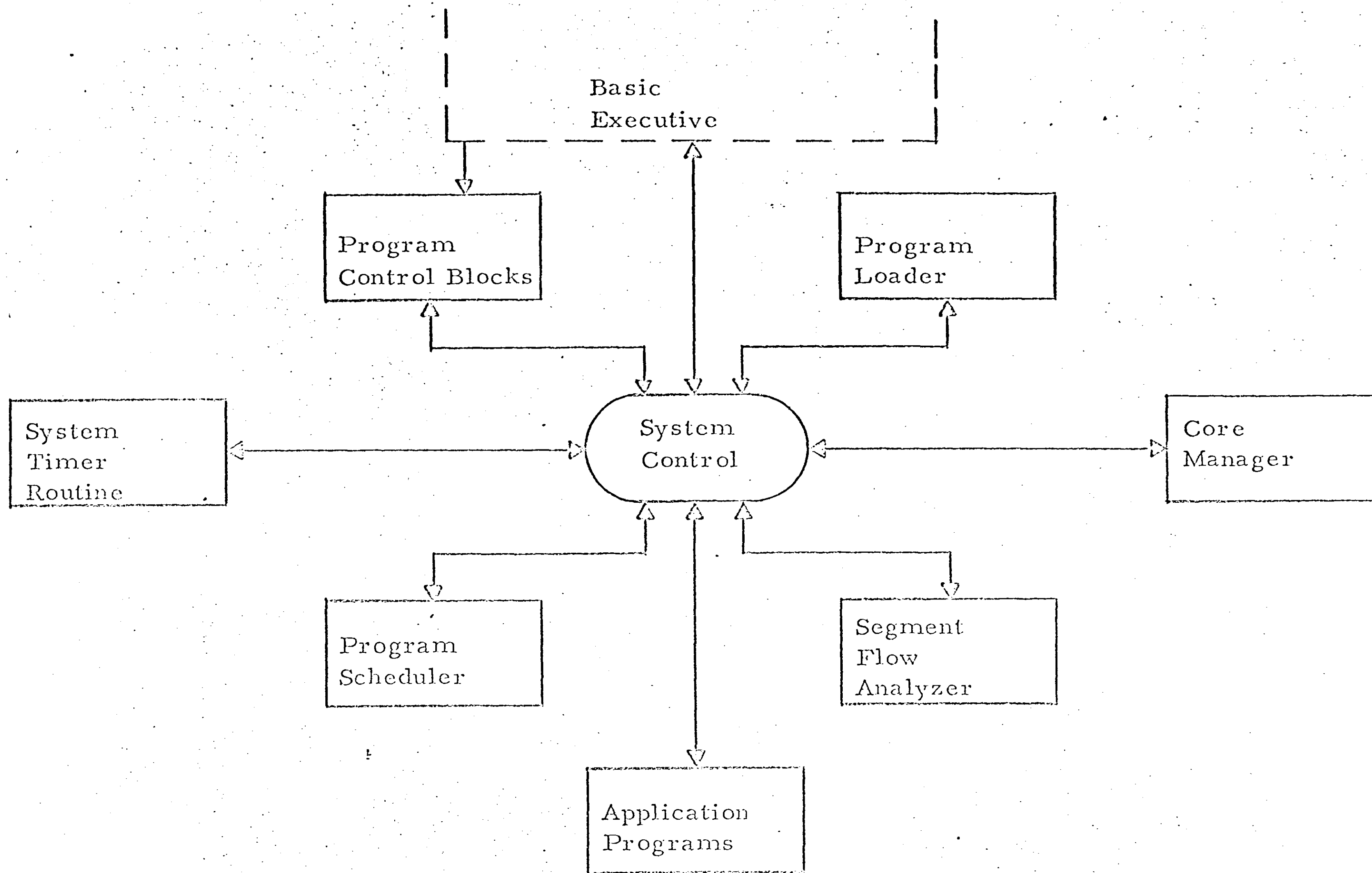
The repetition rate requirement for executing programs applies to all segments of the program which are resident in core storage at the same time. Those programs which are too large to be contained entirely in core storage may be divided into links, each of which can have a different repetition rate. In operation, a single link for the program is active at a time. Communcation from link-to-link is by means of the program COMMON area. Repetition rates are not maintained across link boundaries, since a variable period of time is required to load each link.

2.2 Executive Control

Executive Control is divided into two parts, called the System Executive and the Basic Executive.

2.2.1 The System Executive

The System Executive provides overall supervision of program execution and the logical interface between the internal and external environments. Requests to initiate or terminate a program may come from ground control, from the operator, from the system timer, or from a currently active program. The System



System Executive

Figure 2.2.1

Executive honors these requests by itself whenever possible. However, if they would lead to time or space conflicts with active programs external control is asked to resolve the difficulty.

Figure 2.2.1 shows the overall structure of the System Executive. The functions of the components are as follows.

2.2.1.1 Program Scheduler. The Program Scheduler determines which of the active programs in the multiplexing loop is to be executed during the next time increment. It also determines if requests to add programs can be honored.

2.2.1.2 Segment Flow Analyzer. The Segment Flow Analyzer is activated at the completion of execution of any program segment. It determines, by analysis of the program Segment Flow Table, the next segment to be executed in that program, and updates the corresponding Program Control Block with the address of the entry point.

2.2.1.3 System Timer. The System Timer monitors the list used to initiate those functions whose execution is based upon specified time intervals.

2.2.1.4 Core Manager. The Core Manager maintains information specifying the location and amount of unused core storage. Also, if core storage becomes fragmented so that there is not enough contiguous free core to load a given program, the Core Manager will relocate enough programs to provide the required space.

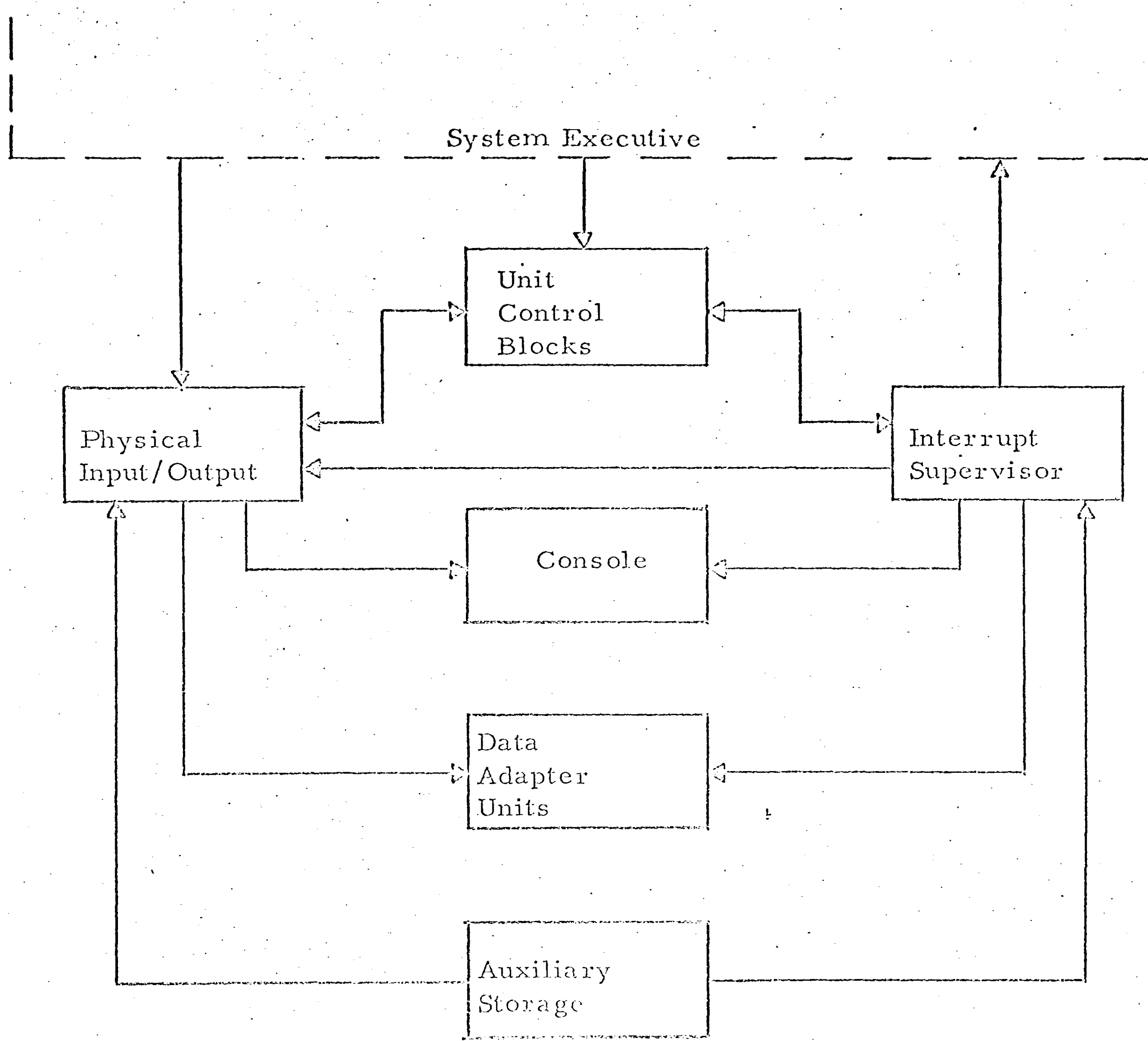
2.2.1.5 Program Loader. The Program Loader services the queue of load requests. It accesses the auxiliary storage, initializes the Program Control Block, adjusts the location dependent quantities in the program text to their resident location, and indicates to the System Executive that the program has been loaded.

2.2.2 The Basic Executive

The Basic Executive provides all services which are associated with access to physical devices and the synchronization of the system. Its major functions include interrupt supervision, configuration control, and all asynchronous I/O.

Figure 2.2.2 shows the overall structure of the Basic Executive. The functions of the components are as follows.

2.2.2.1 Physical Input/Output. The Physical Input/Output routines are responsible for transmission of information between devices and core which is asynchronous with program execution. Included in this list are the console, Data Adapter Units, and the auxiliary storage unit. Requests for I/O, whether from the control system or from the application programs, are queued and serviced on a first-in-first-out basis for each device, with provision for a single level of priority, It is also the function of the I/O section to set flags in the Program Control Block to indicate that I/O is in operation so that the Scheduler can bypass the program. The flags are reset at the conclusion of the I/O operation and the entry address for the program is supplied.



Basic Executive

Figure 2.2.2

2.2.2.2 Interrupt Supervisor. The Interrupt Supervisor handles all interrupts which the system is required to service. These include input/output, external, program, machine check, and clock interrupts.

2.2.2.3 Unit Control Blocks. The tables supply information connecting physical and logical devices, device status, and synchronization control.

2.3 System in Operation

2.3.1 Scheduling

Every program in the system has associated with it one of four possible states. These states are:

2.3.1.1 Active. The program is currently in the execution list, and its segments receive control according to frequency requirements.

2.3.1.2 Waiting. The program is waiting for some specified event. It will not receive control again until the event occurs.

2.3.1.3 Dormant. The program is in core storage but is not in the execution list. It may be overlaid when loading another program.

2.3.1.4 Inactive. The program is within the system but is not in core storage. When its execution is required it must be loaded from auxiliary storage.

The Program Scheduler maintains a list of pointers to the PCB's of active and waiting programs. This list is used in conjunction with a scheduling

algorithm in order to determine which program shall next receive control. The algorithm is designed to distribute CPU time to programs in the multiplexing loop in proportion to their repetition rates, and to intersperse the entries as evenly as possible.

The selection process is based upon a distribution factor, Δ_i, calculated for each program at loop initialization. If there are n programs, this factor is given by

$$\Delta_i = \frac{R}{r_i}$$

Where r_i is the Program Entry Repetition Rate of program P_i. The number R is a normalizing factor which can be chosen arbitrarily in order to simplify computations with the Δ_i. In this discussion, it will be assumed that

$$R = r_1 r_2 \ldots r_n$$

so that the Δ_i are all integers.

Two counts are kept in the PCB of each program in the loop.

2.3.1.5 A selection count, c_i, which is interrogated to determine program sequence.

2.3.1.6 An entry count, k_i, which is used to verify that the program has received at least r_i entries in any one second.

The loop is initialized whenever a program is added or deleted. Initialization consists of recording the current clock time in each PCB, calculating Δ_i, and setting

$$c_i = \Delta_i$$

$$k_i = 0$$

for each program.

Then at each stage, the sequence of program execution is determined by selecting the program P_i whose c_i is a minimum. If there is more than one such program, the first one in the list is selected. For that program, and that program only,

$$c_i + \Delta_i \rightarrow c_i$$

If the program is active, it is given CPU time. If it is waiting, another selection is made. The purpose of treating waiting programs this way is to ensure that they fit properly into the loop when they again become active.

If the program is given CPU time, the entry count k_i is increased by one. It is compared against r_i, and if equal, it is reset to zero and a test made to confirm a correct repetition rate. It the current time minus the time previously recorded in the PCB is less than one second, the program received its requested number of executions. The new clock time is recorded in the PCB, and the selection process continues. This provides an active check on the operation, and permits adjustment of the load to unusual conditions.

Figure 2.3.1 exemplifies results obtained with the scheduling algorithm. In this example, three programs, with repetition rates 4, 2, and 1 respectively, are initialized into the loop at time zero. After one cycle of selection, and part of another, a fourth program, with repetition rate 3, is added to the loop. The example shows the pattern of selection, and the effect of the reinitialization process in smoothing this pattern.

Time	c_1	c_2	c_3	c_4	k_1	k_2	k_3	k_4	Program Selected
0(1)	2	4	8		0	0	0		1
1	4	4	8		1	0	0		1
2	6	4	8		2	0	0		2
3	6	8	8		2	1	0		1
4	8	8	8		3	1	0		1
5	10	8	8		0(2)	1	0		2
6	10	12	8		0	0(2)	0		3(3)
7	10	12	16		0	0	0(2)		1
8	12	12	16		1	0	0		1
9	14	12	16		2	0	0		2
10(4)	6	12	24	8	0	0	0	0	1
11	12	12	24	8	1	0	0	0	4
12	12	12	24	16	1	0	0	1	1
13	18	12	24	16	2	0	0	1	2
14	18	24	24	16	2	1	0	1	4
15	18	24	24	24	2	1	0	2	1
16	24	24	24	24	3	1	0	2	1
17	30	24	24	24	0(2)	1	0	2	2
18	30	36	24	24	0	0(2)	0	2	3
19	30	36	48	24	0	0	0(2)	2	4(5)
20	30	36	48	32	0	0	0	0(2)	1

(1). Initialize with programs P_1, P_2, P_3, of rates 4, 2, 1.

(2). Count reset and new clock marker recorded.

(3). Pattern of selection for the three programs is 1, 1, 2, 1, 1, 2, 3.

(4). Program P_4 introduced, rate 3. Initialization reoccurs.

(5). Pattern of selection for the four programs is 1, 4, 1, 2, 4, 1, 1, 2, 3, 4.

Figure 2.3.1

Scheduling Example

Figure 2.7.1 is a flow chart of this Scheduler which illustrates the operation in some detail.

2.3.2 Segment Flow

The entry to a program is determined by the Segment Flow Analyzer which interrogates an internal form of the flow information supplied by the programmer. A set of 32 triggers, unique to each program,indicates conditions which are currently active in the program. These triggers are set by the program, not the system, as the result of calculations which may affect the flow. Their use is optional, and is intended to supply a convenient means of directing alternate program activity.

An example of the use of these statements is given in Section 3.3.3. Suppose for this example, that the segment flow analyzer has received control following the execution of Segment S_2. The internal flow text specifies that it then test trigger T_1 to determine the next segment. Having done so, it places the location of the entry point of the chosen segment in a cell in the PCB. This cell always contains the location of the program entry to be taken when the program next gains control.

Figure 2.7.2 is a flow chart outlining operation of the Segment Flow Analyzer, while Figure 2.7.9 shows details of the flow text which appears in the PCB.

2.3.3 Program Control Block

The Program Control Block contains the following information.

2.3.3.1 Link Control Information, which includes the link priority, the repetition rate, and a list of all links in the program. The latter is used by the System Executive to determine which link to load and execute.

2.3.3.2 The Segment Flow Table. This table is generated during preparation from information in the flow control statements.

2.3.3.3 The Program Status Indicators.

2.3.3.4 Pointers which indicate: the location to which control should proceed when the program is next entered; and the entry in the Segment Flow Table which specifies the flow after the completion of the current segment.

2.3.3.5 The Program Triggers.

More detail on the Program Control Block content will be found in Figure 2.7.19.

2.3.4 System Flow

The flow of control within the system is geared to the beginning and ending of execution of program segments. Each segment returns control to the system either when its execution has been completed or when waiting for some event. The following description of the flow of control assumes that control has just been returned to the system from a program segment (refer to flow chart of Figure 2.7.3).

2.3.4.1 If return is from segment completion, the Segment Flow Analyzer is entered to determine the next segment to execute for this

program. If return is not from segment end, the return location within the segment has already been saved by the routine which processed the event request.

2.3.4.2 The Interrupt Supervisor is entered to enable and service any pending interrupts.

2.3.4.3 Control is given to the I/O routine to acknowledge and post completion of previous I/O requests.

2.3.4.4 The System Timer requests are entered into the scheduling queue.

2.3.4.5 Loading completions are recognized.

2.3.4.6 Any program termination request is processed.

2.3.4.7 Program initiation requests are processed.

2.3.4.8 The Program Scheduler determines the next program to be executed.

2.3.4.9 Control is given to the previously determined segment of that program.

2.3.5 Core Management

The Core Manager is activated when there is a need to find space for a program, or to free space when a program is completed.

The current status of core utilization is maintained in the Core Usage List (see Figure 2.7.16 for details). Core 'cleanup' occurs only when necessary, at which time enough programs are relocated (moved toward low order core) to acquire the necessary space. Programs most used will then gravitate toward low order core, minimizing the need for further relocation.

Action of the Core Manager is illustrated in the flow chart of Figure 2.7.4.

2.3.6 Interrupt Supervision

Several different types of interrupts must be handled within the system; input/output, external, program, machine check, and interval timer.

The I/O interrupts are disabled during the processing of a segment. When control is returned to the system at the end of any segment, the I/O interrupts are enabled for one instruction time. All pending interrupts which occurred during the previous segment are then serviced.

Unsolicited I/O interrupts are included in the group of external interrupts. When such an interrupt occurs, it is acknowledged at once, and an entry is made in the program queue according to the priority for the type of interrupt. Control is then returned to the program segment which was being processed. The program associated with each interrupt is activated and scheduled between segments.

Program interrupts, with the exception of overflow, are transferred directly to the System Executive with an appropriate flag set. If the interrupt

indicated an error, external control is notified and can then elect to continue processing, or can terminate processing in order to initiate equipment diagnostics.

Machine check interrupt indicates an equipment failure and control is transferred to the System Executive with the proper flag set.

The interval timer interrupt is serviced immediately, in order to update all system clocks.

Interrupt processing details are to be found in the flow chart of Figure 2.7.5.

2.3.7 Asynchronous Input/Output

These requests are made within application programs. They are also made by the system in order to load programs from the auxiliary storage. The basic macros provided for Input/Output are:

READ

WRITE

WAIT

The operands of the READ and WRITE instructions provide for the specification of device, word count, and location of data. The WAIT macro is a synchronization device, indicating to the system that the program is to be placed in WAIT status until the I/O action is complete. In this case, control is transferred to the System Executive. In the absence of a WAIT indication, control is returned to the segment as soon as the I/O request is queued.

The I/O queue entry contains the following information (see Figure 2.7.14):

2.3.7.1 The location and length of the I/O area;

2.3.7.2 The location of the Program Control Block;

2.3.7.3 The pointer to the next queue entry (this chain extends from the Unit Control Block for a device).

The procedure followed for the I/O request is shown in Figure 2.7.6. Basically, the following actions occur:

2.3.7.4 Structure the queue entry;

2.3.7.5 Set flags in the Program Control Block to indicate that I/O is current, and that the storage area cannot be relocated;

2.3.7.6 If the device is not active and a data path is available, initiate I/O. If WAIT is not specified in the request, return is made to the program. Otherwise set flags in the Program Control Block to indicate that the program is waiting, set the return entry, and return to the System Executive.

The procedure followed between the processing of segments and upon the occurrence of an I/O interrupt is as follows (for details, refer to the flow chart, Figure 2.7.5):

2.3.7.7 Remove queue entry;

2.3.7.8 Remove the appropriate I/O and wait flags from the Program Control Block;

2.3.7.9 Return control to the System Executive.

2.3.8 Program Loading

Program loading is accomplished by a group of routines which communicate through a program request queue. Some of these routines operate as system subroutines, while others operate within the multiplexing loop. The steps used to effect program loading are as follows:

2.3.8.1 Requests to load a program are placed into the load request queue by means of a system subroutine.

2.3.8.2 A subroutine examines the Program Index (Figure 2.7.17) and Core Residence List (Figure 2.7.18) to determine if the program is in core. If so, loading is complete, and the PCB location is returned to the System Executive.

2.3.8.3 If not, a set of multiplexed routines access the auxiliary storage, obtain core for the PCB, and bring it in. The PCB location is given to the System Executive for analysis.

2.3.8.4 If the Scheduler determines that the program can fit into the multiplexing loop, the Core Manager is invoked to acquire the necessary space, and the program is brought into core.

2.3.8.5 A multiplexed relocation routine makes the necessary address adjustments and notifies the System Executive that the program is ready for execution.

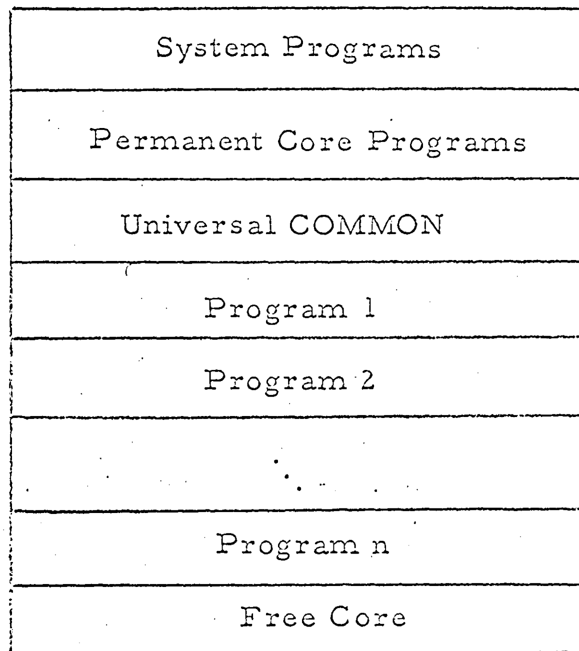
Flow charts illustrating details of the loader operation are to be found in Figures 2.7.7 and 2.7.8. The first of these covers the initiation process, which obtains the PCB for examination by the executive. The second covers action which occurs to complete the loading process.

2.3.9 Core Storage Layout

Figure 2.3.9 indicates the overall layout of core storage during execution. Two areas not previously mentioned are used as follows.

2.3.9.1 Permanent Core Programs. In addition to the executive program, two types of programs reside permanently in core. First, there are programs which because of their associated response times must always be immediately available for placement in the multiplexing loop. Included in this category are some application programs, and such overlap functions as those associated with external communication. Second, there are common subroutines, such as sine and cosine, which are provided as system services in order to economize on overall core requirements.

2.3.9.2 Universal COMMON. Program COMMON provides for communication between the segments and links of a given program. Universal COMMON, on the other hand, provides for communcation between programs. It also provides residence for data, such as attitude and ephemeris information, which is updated periodically by resident programs for utilization by any program.



Core Map

Figure 2.3.9

In contrast to program COMMON the data structure of Universal COMMON is determined by the requirements of the overall system. Once this is specified, items may be accessed by the program as from any other data area. The actual core location of Universal COMMON and program COMMON is of no concern to the programmer, since the system loads these locations into pre-determined registers prior to entry to any segment.

2.4 Program Initiation

Requests to initiate execution of a program may come from ground control, the operator, the System Timer, or from an already active program. The request is processed by the System Executive, which calls upon the Program Loader to determine the size of the program, its repetition rate, and its location.

If the program cannot be scheduled without conflict, information is presented to external control which specifies the type of conflict, and a list of active programs including their size and their repetition rates. It is expected that this information will be used either to discontinue some program or specify that the request be delayed. If no conflict exists, the loading process is carried out as previously described.

2.5 System Initiation

The structure of information on the auxiliary storage device is dictated in part by the requirements of system initialization and in part by the structure of the resident programs. Initial records contain not only the resident system, but also the Unit Control Blocks which are associated with the I/O devices, the initial

location of free core, the definition of Universal COMMON, the list of initial programs to be processed by the Scheduler, and the Program Index. The proposed structure is as follows.

2.5.1 Resident System.

2.5.2 Initialization Record.

2.5.3 Unit Control Blocks (see Figure 2.7.15).

2.5.4 Definitions of free core and Universal COMMON.

2.5.5 List of initial programs to insert into the multiplexing loop.

2.5.6 Program Index, one entry per program, which contains the position of the Program Control Block, and its size (see Figure 2.7.17).

2.5.7 Program Files. Each program file consists of the PCB, and the Relocation Dictionaries and texts of each link.

System initiation, either as a cold start or to effect a restart, begins with an Initial Program Load Sequence activated by the operator or by ground control. The Resident System is loaded from the auxiliary storage device together with the initialization record and the Unit Control Blocks.

The Unit Control Blocks are examined to determine the equipment configuration. External control may be requested at this point to enter any deviations from the standard configuration due to equipment malfunction or other operational constraints. Such information is used to modify the Unit Control Blocks to reflect

the current configuration. In addition, external control may supply restart information. Next, the definitions of free core and Universal COMMON and the list of initial programs to be entered into the multiplexing loop are then loaded. The Program Index is examined to locate the required programs. Once located, they are loaded from the auxiliary storage device and entered into the multiplexing loop. External control is then informed of the current activity of the DCSG and is then free to enter requests to load and execute additional programs.

The termination procedure, including program requested power off, is localized within the System Executive. When a request for termination is recieved an orderly shutdown procedure is initiated. In this procedure, information is saved in order to effect a possible restart. Such information is retained within the system if the configuration permits; otherwise, it is presented to external control to be entered at some future time.

2.6 System Overhead

Estimates for the processing times and core storage requirements of various system functions were obtained by analysis of the flow charts and tables presented in paragraph 2.7, together with trial programming of critical functions. Time estimates are confined to non-multiplexed functions, as only these are critical with respect to overall operation. Space estimates, however, include the entire resident system.

2.6.1 Execution Times

The most significant system time is that expended in controlling the overall flow. This intersegment execution time, as explained in paragraph 2.3.4, involves some routines which are always invoked, and some which are invoked only upon the occurrence of certain conditions. Furthermore, execution times vary with certain parameters, such as the number of entires in a queue, or the number of programs in the multiplexing loop.

In order to obtain a clear picture of execution time, then, the charts are in two parts. The first part, Figure 2.6.1, shows the execution times for all routines which may be required. These times are given in terms of the number of instructions executed. The second part, Figure 2.6.2, summarizes the total execution time per second for several operational conditions which might be considered typical. Total execution times are based on an average instruction execution time of 5 microseconds.

INTERSEGMENT EXECUTION TIMES

ROUTINE	INSTRUCTIONS EXECUTED
Flow Control Driver	20
Program Scheduler	
Basic	15
For each additional comparison	6
Segment Flow Analyzer	
Minimum	8
Each operation performed in Segment Flow Table	22
Interrupt and I/O Processing	
Interrupt Action	12
Normal I/O Action	40
Channel End	24
Device End	15
Request End	14
System Timer	
Update	8
Each program requested	10
Program Initiation	
Normal	94
If time is not available	175
If space is not available	175
Program Termination	
Basic	10
Return Core	34
Return Time	20
Program Scheduler Initialization	
Basic	10
Each program in Execution List	32

Figure 2.6.1

TYPICAL OPERATING OVERHEAD TIMES

Example 1. Assumptions: Three programs executing concurrently with repetition rates of 60, 30, and 10. One I/O request per program, per second.

The system components used, and the system time for one second is:

	Executed instrs.	Time (ms)
System Executive Driver	2000	10.0
Program Scheduler	2480	12.4
Segment Flow Analyzer	2500	12.5
Interrupt and I/O Processor	1400	7.0
	8380	41.9

Example 2. Assumptions: Three programs executing concurrently with repetition rates of 60, 30, and 10. Three I/O requests per program, per second.

The system time for one second is:

System Executive Driver	2000	10.0
Program Scheduler	2840	14.2
Segment Flow Analyzer	2500	12.5
Interrupt and I/O Processor	1980	9.9
	9320	46.6

Example 3. Assumptions: Five programs executing concurrently with repetition rates of 10, 20, 40, 20, 10. Four I/O requests per program, per second.

The system time for one second is:

System Executive Driver	2000	10.0
Program Scheduler	3040	15.2
Segment Flow Analyzer	2500	12.5
Interrupt and I/O Processor	2520	12.6
	10060	50.3

Example 4. Assumptions: Five programs are run concurrently for five minutes, with one program termination and one program initiation during this interval. Four I/O requests per program, per second.

The system components used and the total system execution times are as follows:

System Executive Driver	600000	3,000
Program Scheduler	912000	4,560
Segment Flow Analyzer	750000	3,750
Interrupt and I/O Processor	756000	3,780
Program Termination	64000	320
Program Scheduler Initialization	340000	1,700
Program Initiation	96000	470
	3,516,000	17,580
Average/second	11,720	58.6

Figure 2.6.2

CORE REQUIREMENTS

	NUMBER OF INSTRUCTIONS	NUMBER OF WORDS
System Executive		
Routines		
Driver	20	
Program Scheduler	120	
Segment Flow Analyzer	82	
System Timer	28	
Program Loader	270	
Program Relocator	40	
Queue Management	47	
Core Management	110	
Operator Communication	75	
Program Initiation	240	
	1032	1239
Tables		
Program Request Queue		6
Program Execution List		10
Program Index		50
Load Request Queue		4
System Timer Queue		5
Program Control Blocks		200
Core Residence List		30
Core Usage List		20
		325
Basic Executive		
Routines		
I/O Request Processor	76	
Interrupt Action	26	
Interrupt Processor	110	
I/O Select Routines	60	
Error Recovery Routines	75	
Supervisor Call Processor	20	
	367	441
Tables		
Unit Control Blocks		176
I/O Request Queue		200
Interrupt Queue		36
Interrupt Cells and Diagnose Area		50
		462
	TOTAL	2467

Figure 2.6.3

Examination of these charts shows that flow control overhead averages about 5% of the available CPU time.

Other non-multiplexed times which affect program operation are those for I/O requests and Supervisor Calls for system services. The average number of executed instructions for these are:

I/O Request	60
Supervisor Call	10

2.6.2 Core Storage Requirements

Figure 2.6.3 lists the core storage requirements for the executive routines. The word count has been generated by assuming a conversion factor of 1.2 words per instruction.

2.7 Flow Charts and Tables

Collected in this paragraph are a number of flow charts detailing operation of the system, and descriptions of the information content of the more important tables utilized by the system.

The flow charts are:

2.7.1 Program Scheduler

2.7.2 Segment Flow Analyzer

2.7.3 Overall System Flow

2.7.4 Core Manager

2.7.5 I/O Interrupt Processor

2.7.6 I/O Request Processor

2.7.7 Program Loader. Part I.

2.7.8 Program Loader. Part II.

Table information is as follows:

2.7.9 Segment Flow Tables

2.7.10 Program Request Queue

2.7.11 Program Execution List

2.7.12 Load Request Queue

2.7.13 Unit Request Queue

2.7.14 I/O Interrupt Queue

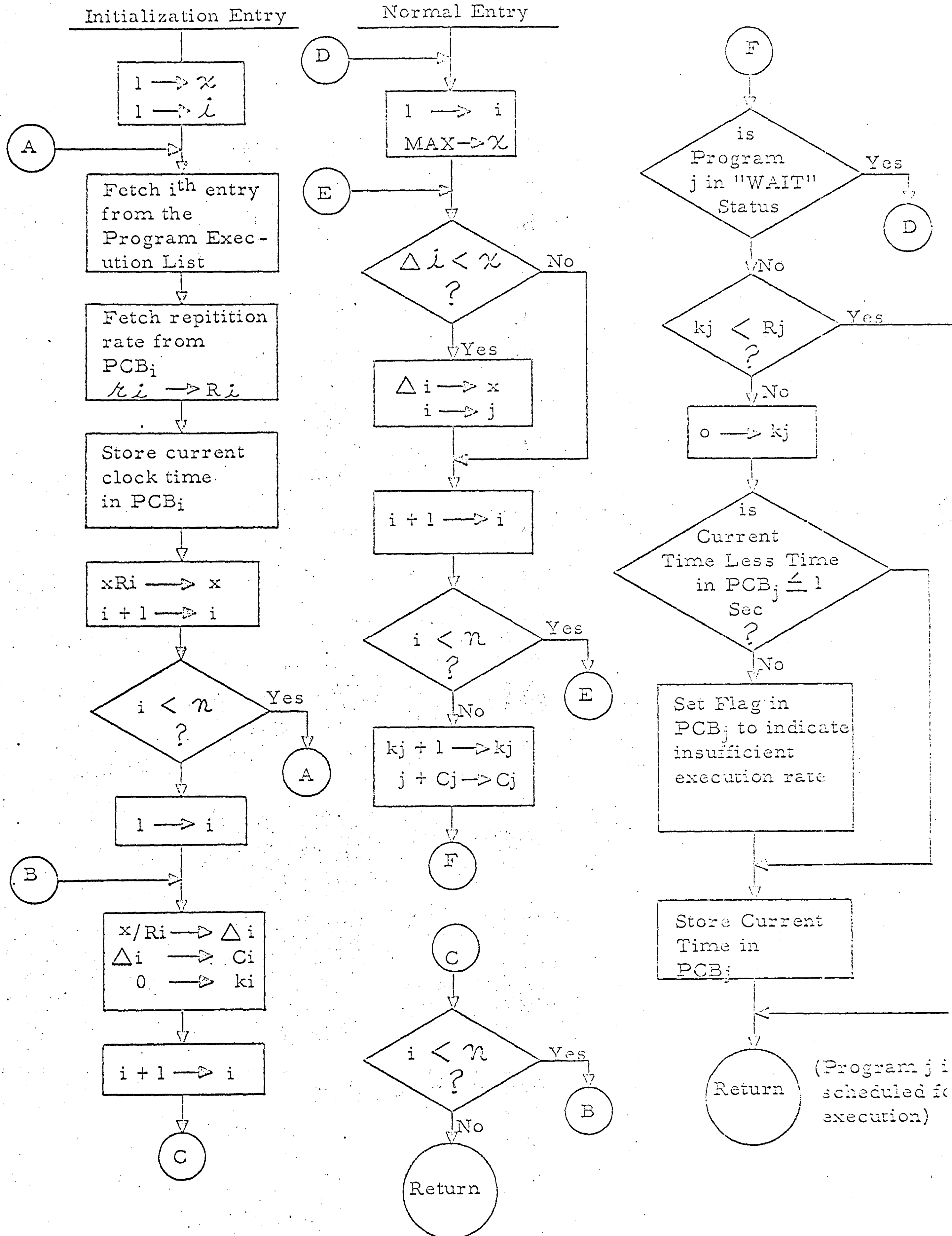
2.7.15 Unit Control Block

2.7.16 Core Usage List

2.7.17 Program Index

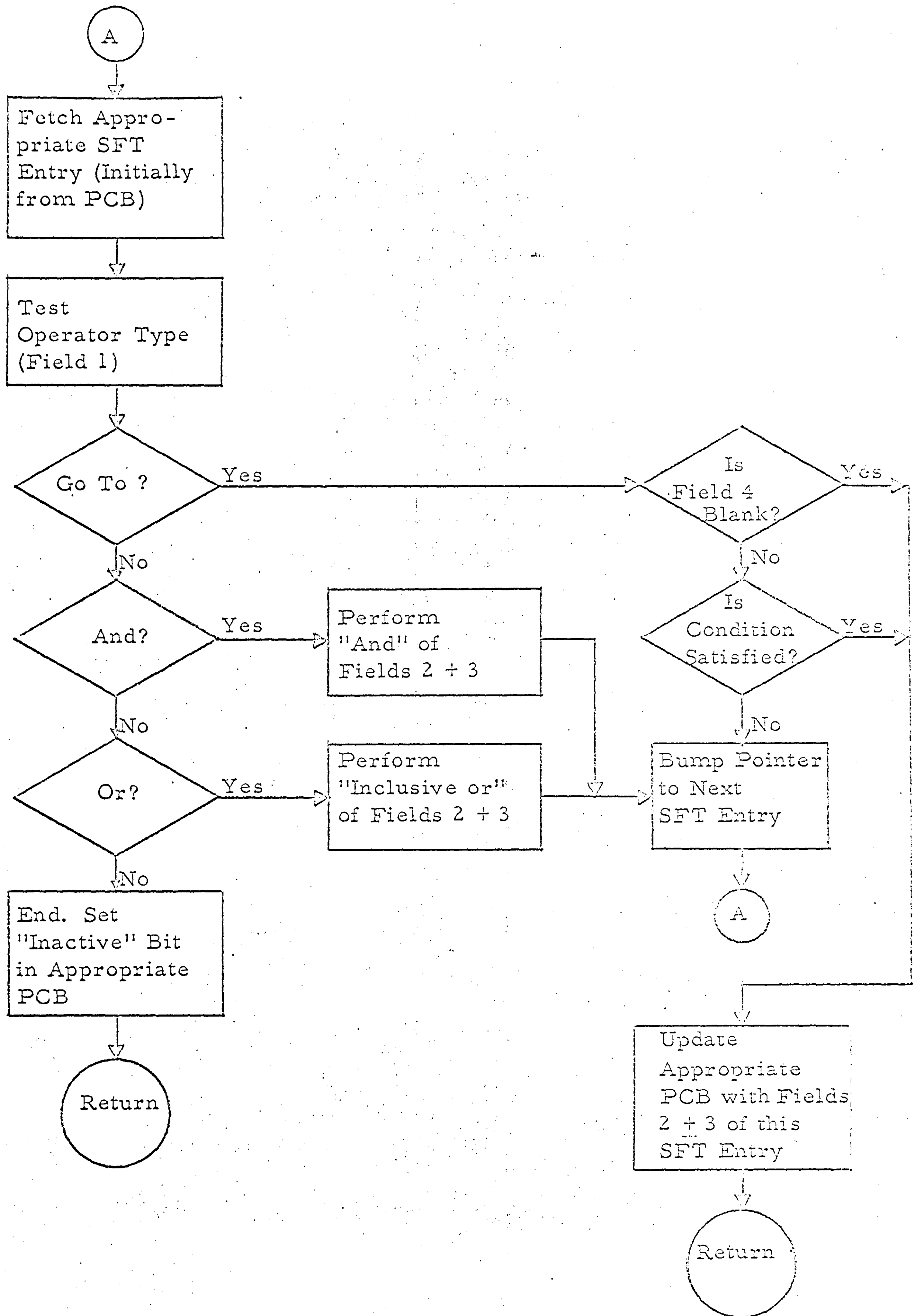
2.7.18 Core Residence List

2.7.19 Program Control Block



Program Scheduler

Figure 2.7.1



Segment Flow Analyzer

Figure 2.7.2

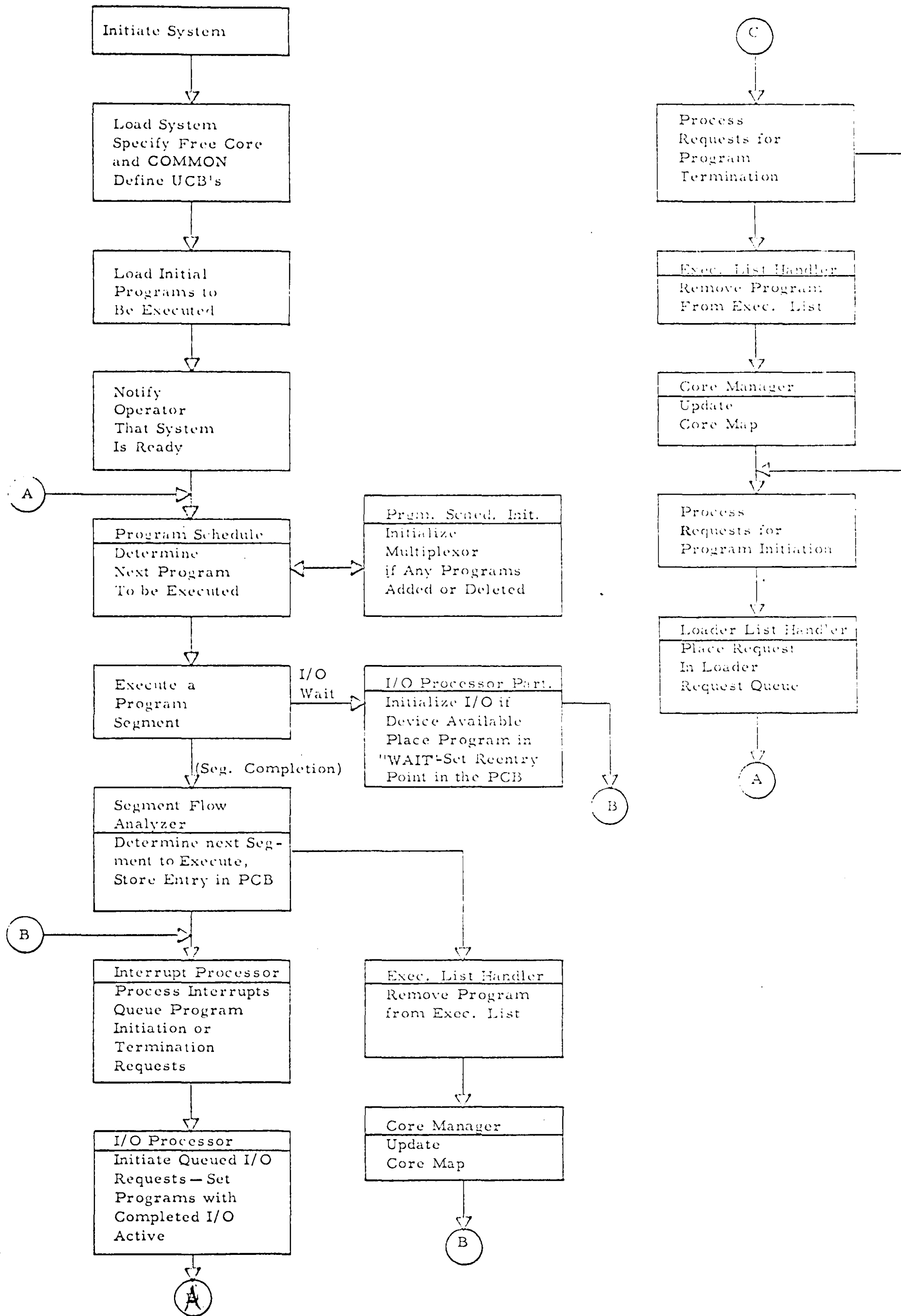
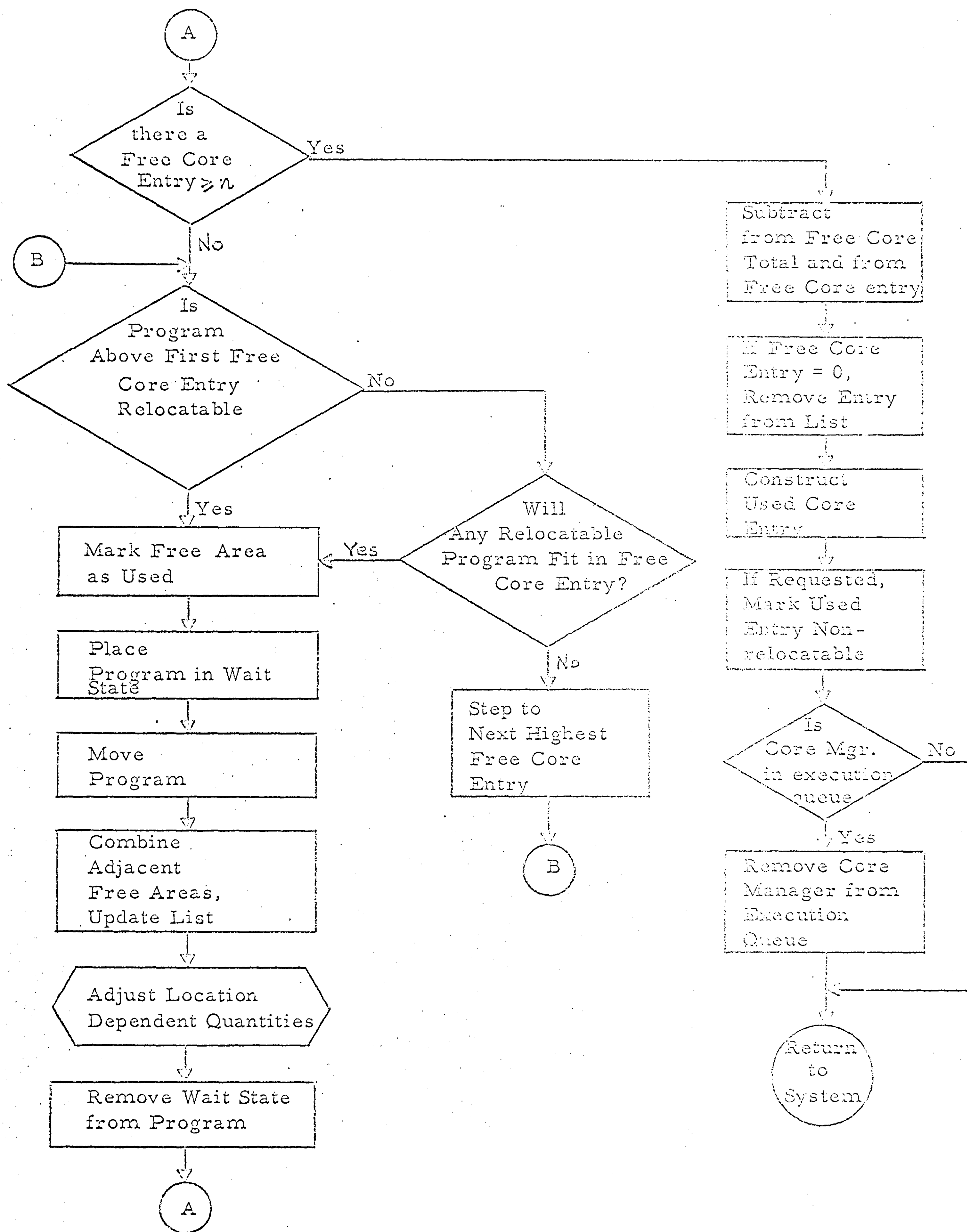
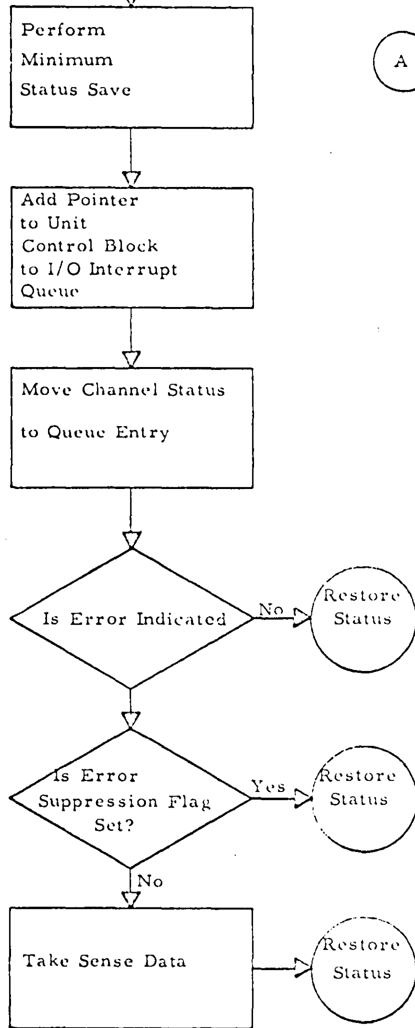


Figure 2.7.3
Overall System Flow



Core Manager

Figure 2.7.4



Interrupt Processor

A

Is Interrupt Void?

Yes

Return to System

Error

Yes

Enter Appropriate Error Recovery Routine

No

Clear (Possible) Appropriate WAIT-Set Sync Call

Does Program still have pending I/O

Yes

No

Remove Non-relocable Flag From Program Area

Remove Queue Entry

Was Interrupt Channel End?

Yes

Mark in UCB

No (Device End)

Clear Channel End from UCB

Is Pending Request on Subchannel?

Yes

B

B

Activate Device

Yes

Does Device Have Pending Req?

No

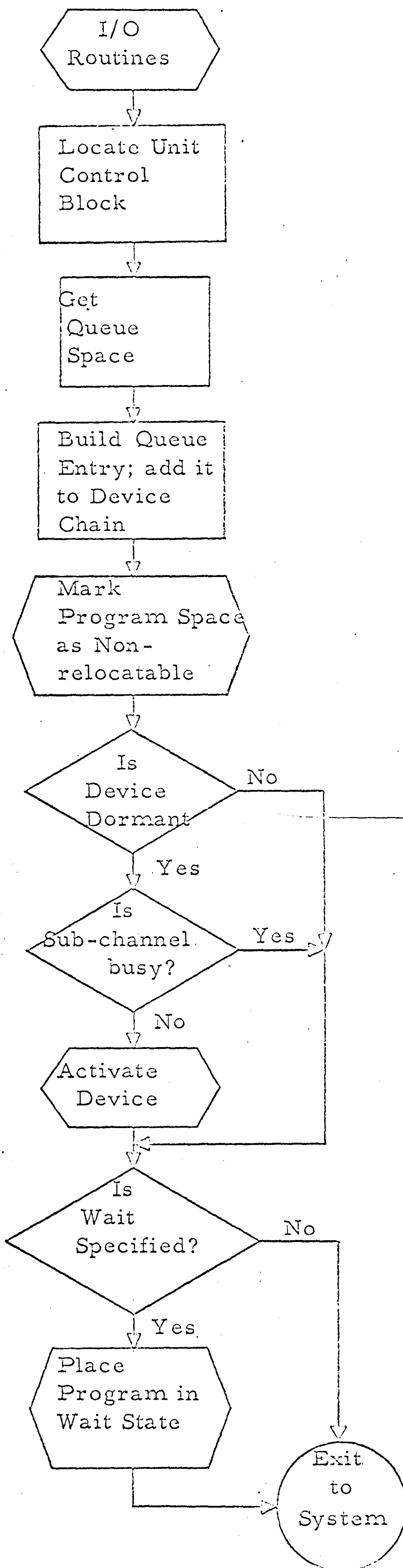
Clear Sub-channel Activity Cell

A

A

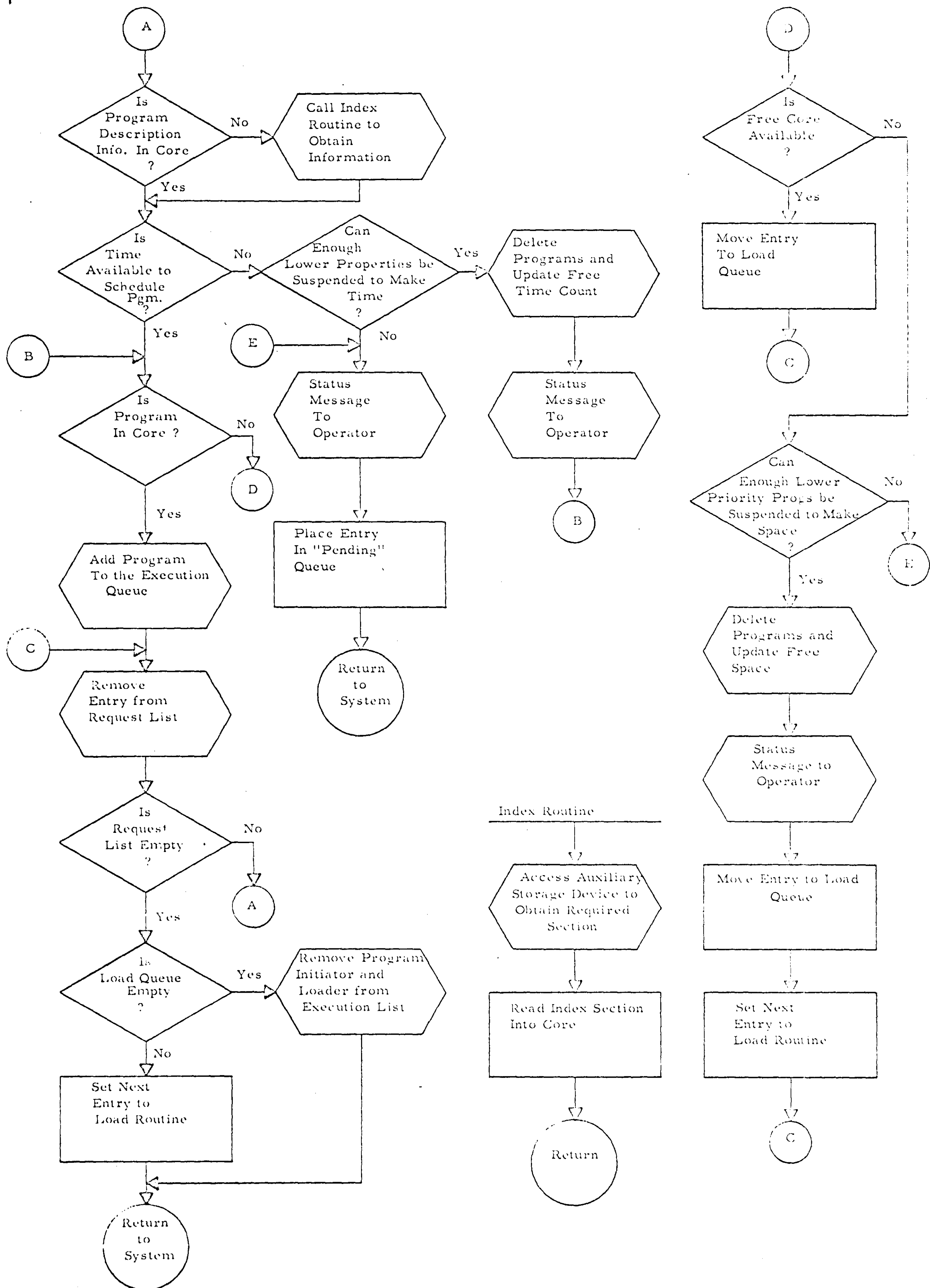
A

I/O Interrupt Processor
Figure 2.7.5



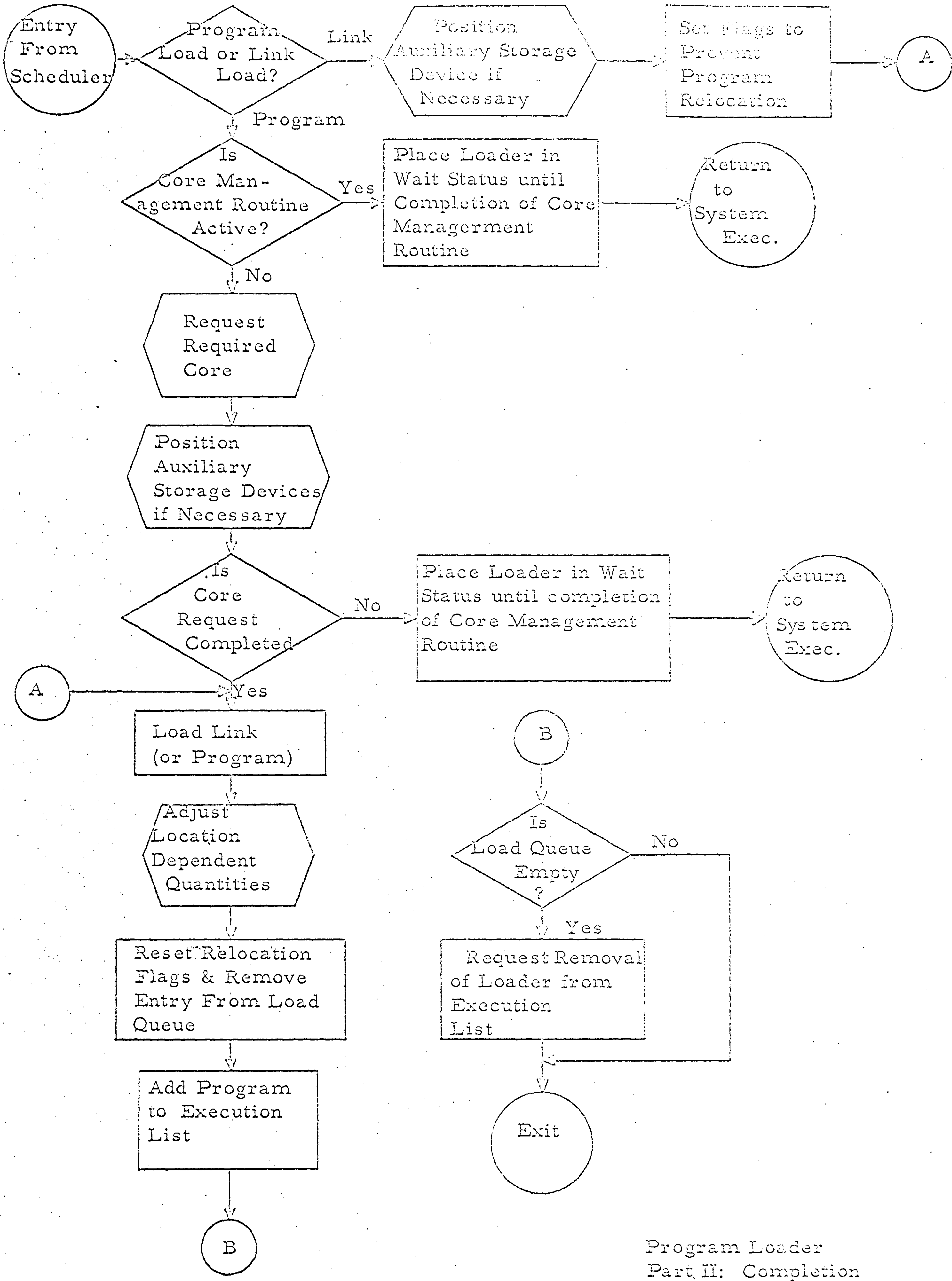
I/O Request Processor

Figure 2.7.6



Program Loader. Part I Initiation

Figure 2.7.7



Program Loader
Part II: Completion

SEGMENT FLOW TABLE FORMAT				
FIELD 1	FIELD 2	FIELD 3	FIELD 4	Remarks
GO TO	S_n	FS_n	Blank R_i T_i	Blank indicates unconditional GO TO
AND	R_i $\overline{T_i}$ T_i	R_i T_i $\overline{T_i}$	R_i	2 wds/segment
OR	R_i T_i $\overline{T_i}$	R_i T_i $\overline{T_i}$	R_i	
END	Blank	Blank	Blank	

S_n — Entry Point to Segment n

FS_n — Segment Flow Table entry of flow description from segment n

R_i — Internal boolean trigger i to be used by the segment flow analyzer

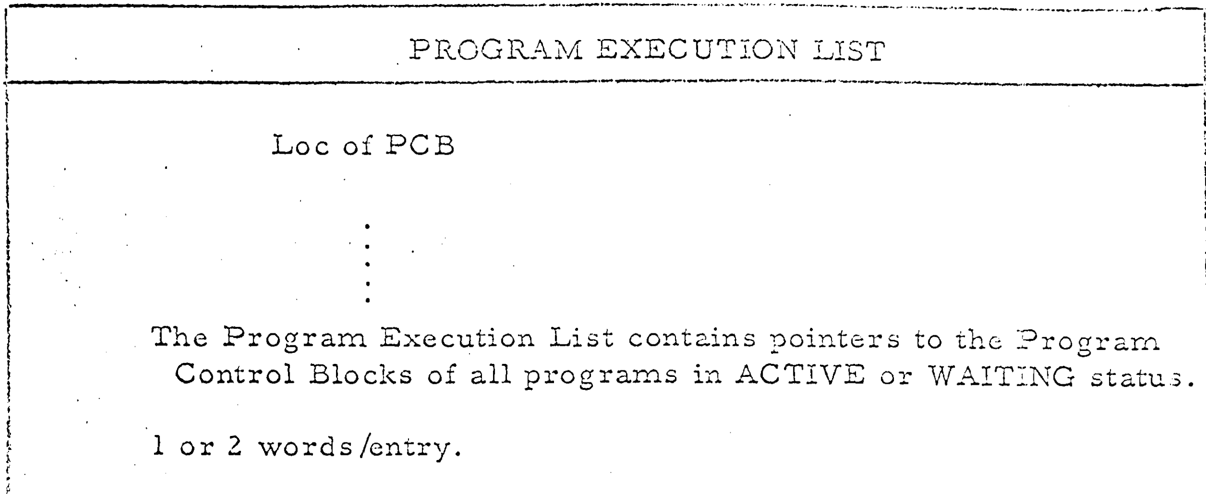
T_i — Program trigger i set by the application program

$\overline{T_i}$ — 'NOT' or complement of trigger i.

Figure 2.7.9

PROGRAM REQUEST QUEUE
Program I.D., Link No. ⋮ ⋮ The Queue discipline is FIFO within priority, If Link No. is not specified, the first link is assumed. 1 wd/entry

Figure 2.7.10


PROGRAM EXECUTION LIST

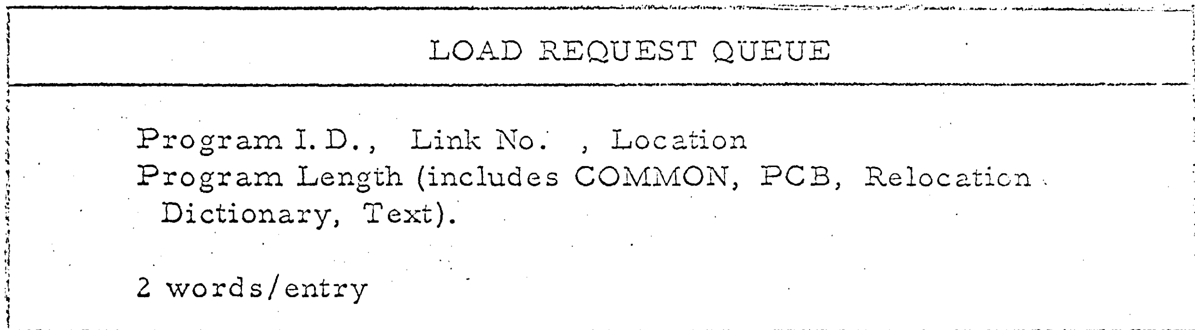
Loc of PCB

⋮

The Program Execution List contains pointers to the Program Control Blocks of all programs in ACTIVE or WAITING status.

1 or 2 words/entry.

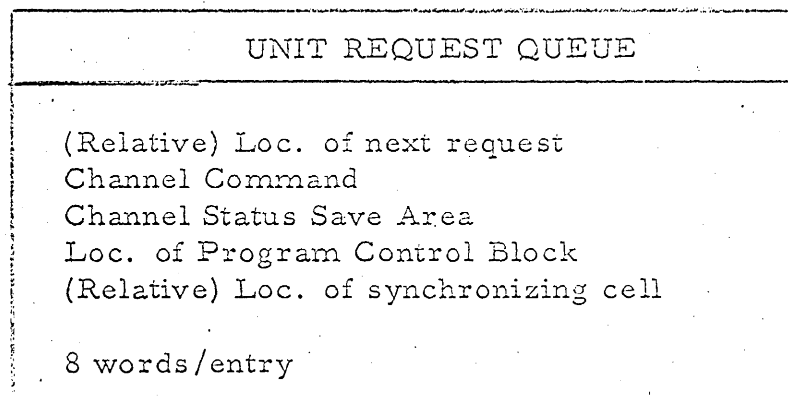
Figure 2.7.11


LOAD REQUEST QUEUE

Program I.D., Link No. , Location
Program Length (includes COMMON, PCB, Relocation Dictionary, Text).

2 words/entry

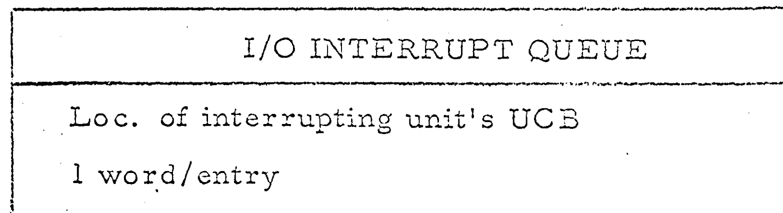
Figure 2.7.12


UNIT REQUEST QUEUE

(Relative) Loc. of next request
Channel Command
Channel Status Save Area
Loc. of Program Control Block
(Relative) Loc. of synchronizing cell

8 words/entry

Figure 2.7.13


I/O INTERRUPT QUEUE

Loc. of interrupting unit's UCB

1 word/entry

Figure 2.7.14

UNIT CONTROL BLOCK

Device type, Device Address, (Relative) Loc. of first request in Unit Request Queue

(Relative) Loc. of last request in Unit Request Queue, (Relative) Loc. of subchannel activity cell

Subchannel unit chain, Device Status (Physical/Logical)

Device Dependent (Positioning) Information

One/Device — 4 words/Unit Control Block

Figure 2.7.15

CORE USAGE LIST

Used/free indicator
Non-Relocatable indicator
Area Length
Origin of Area

1 word/entry

Figure 2.7.16

PROGRAM INDEX

Program I.D., Location
Priority, Repetition Rate
Program length (includes COMMON, PCB, Relocation Dictionary, Text)

2 words/entry

The Program Index can be divided into n equal sections only one of which would reside in core at any time.

25 entries always in core.

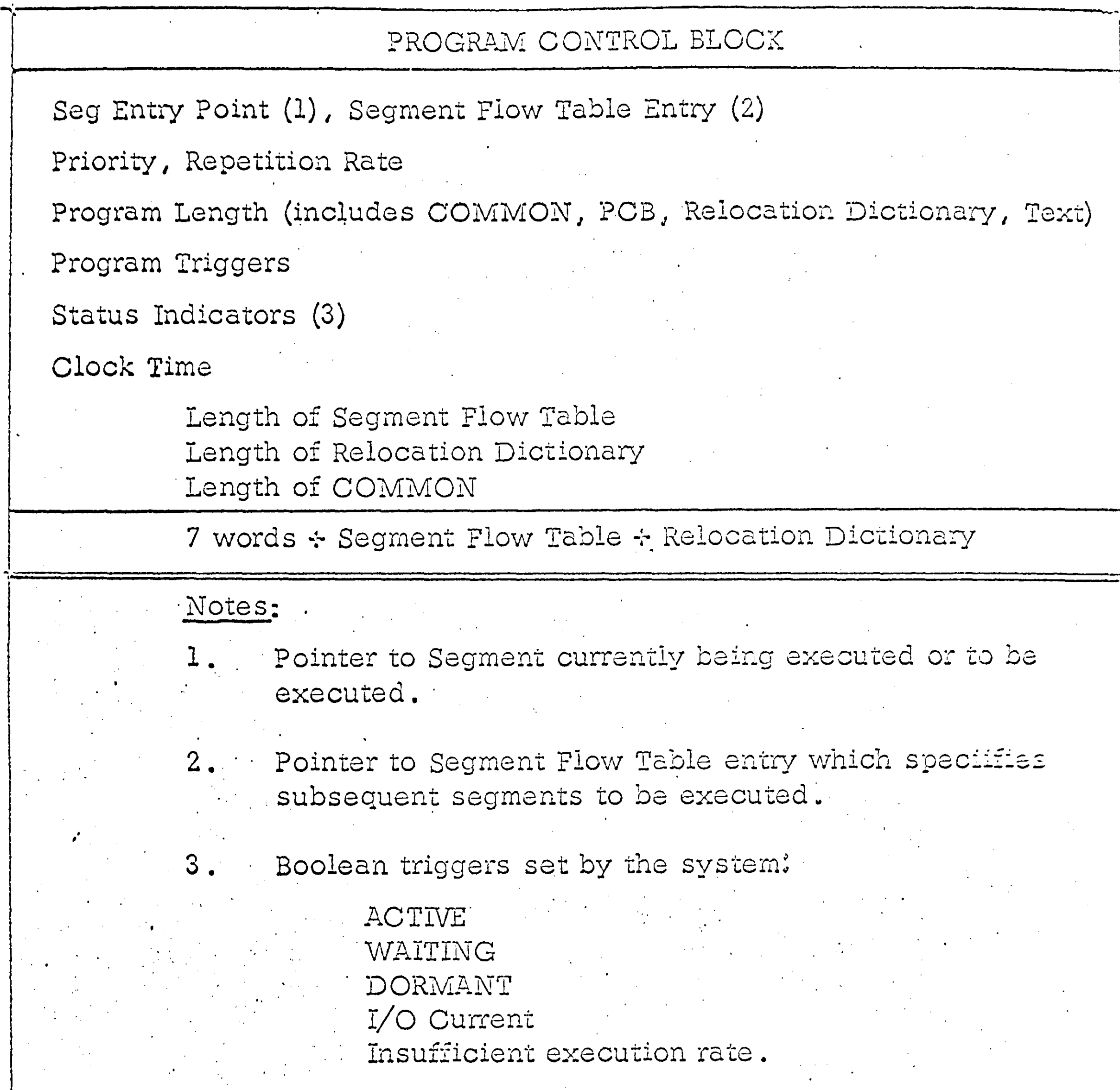
Figure 2.7.17

CORE RESIDENCE LIST

Program I.D.	Loc. of PCB
⋮	

30 entries, 1 word/entry.

Figure 2.7.18


PROGRAM CONTROL BLOCK

Seg Entry Point (1), Segment Flow Table Entry (2)

Priority, Repetition Rate

Program Length (includes COMMON, PCB, Relocation Dictionary, Text)

Program Triggers

Status Indicators (3)

Clock Time

Length of Segment Flow Table
Length of Relocation Dictionary
Length of COMMON

7 words ÷ Segment Flow Table ÷ Relocation Dictionary

Notes:

1. Pointer to Segment currently being executed or to be executed.

2. Pointer to Segment Flow Table entry which specifies subsequent segments to be executed.

3. Boolean triggers set by the system:

   ACTIVE
   WAITING
   DORMANT
   I/O Current
   Insufficient execution rate.

Figure 2.7.19

3.0 THE GROUND SUPPORT SYSTEM

3.1 General Requirements

The purpose of the Ground Support System is to provide a means whereby system and applications programs for the DCSG can be produced, tested, and maintained on standard ground computer equipment. In order to develop such a system, and procedures for its use, which will result in the highest attainable level of confidence in the operation of the DCSG, it is necessary to provide a comprehensive, integrated set of services which relieve the programmer of clerical demands, while permitting freedom and flexibility in the organization and description of a problem. The services required can be grouped into the following functional categories.

3.1.1. Languages with which the programmer expresses computational algorithms, program structure, operational requirements, and preparation control.

3.1.2 Language processors which translate source statements into a form suitable for execution.

3.1.3 Libraries of source statements, which are available for modification and translation; of object programs, which may be combined in various ways; and of production programs ready for execution.

3.1.4 Editing routines which provide access to the libraries and implement the functions of updating, modifying, combining, reordering, and documenting the information.

3.1.5 A simulation environment in which to execute the programs under conditions as nearly as possible identical to their actual use.

3.1.6 Check-out facilities to test and debug programs.

3.1.7 An executive supervisor which exercises control over the whole system, and provides the mechanism by which activity is coordinated between the programmer, the services, and the operator. Such an assemblage of software and hardware is called an operating system.

3.2 Existing Operating Systems

Operating systems of varying degrees of sophistication are available for almost all standard ground equipment. For several reasons, it is desirable to use as much of an existing standard system as possible. First, the existing software represents a considerable investment in time and manpower which can effect a substantial saving in the overall development schedule. Second, additional important benefits result from the basic standardization obtained from a widely used

system: ease of coordination between remote geographical locations; recovery of programs developed for other applications; and simplification of user training.

However, no ground system can be used exactly as it stands, because in all such systems the designers have ignored the fact that the simulation environment (item 3.1.5) may not be identical to the environment of the ground operational system. Hence, the system processors produce code for operation within the ground computer, with linkages designed for operation under its executive supervisor. Since ground-based executive routines are not geared to meet the stringent real-time demands of the airborne environment, recovery of an existing system dictates two possible courses of action: modify the executive and the data management facilities for spaceborne use; or, insert routines under the ground executive to provide the proper simulation environment and the editing features necessary to convert programs to the form required by a spaceborne system.

With AVE equipment which is instruction set compatible with the AGE equipment the first course is possible. It is, however, neither practical nor palatable. It is not practical because the environments are so different that the resulting executive must be a compromise suitable to neither. It is not palatable because the resulting system is non-standard and hence requires extraordinary maintenance procedures of its own.

The second course of action is therefore proposed as the method of developing the Ground Support System. It is particularly attractive here because:

3.2.1 The AVE equipment proposed is instruction set compatible with the IBM System/360 family of computers. Thus, the code generated by existing language processors needs virtually no modification.

3.2.2 Operating System/360 is a comprehensive operating system embracing a variety of programming languages and language processors as well as extensive library, editing, and debugging facilities. It is maintained by IBM as a standard product.

3.2.3 Except for the Program Control Block, the program structure used in the Spaceborne System is identical to that obtained with the standard facilities of OS/360. Hence, the additional editing capability required is minimal.

3.2.4 The structure chosen for the Spaceborne Executive Program facilitates the development of the simulation environment operating under OS/360.

For these reasons, it is natural to build the Ground Support System to function as a job within OS/360. The remainder of this section discusses how this can be done: the use of the existing system; the modifications required; the additional programs needed; and the operation of the resulting system.

3.3 Program Preparation

A most important function in the Ground Support System is the production of programs for execution in both the Spaceborne and simulation environments. To do this, it is necessary to introduce a processor which accepts as input both the output of OS/360 language processors and statements by the programmer specifying the program construction, and produces as output the desired program. This program preparation processor is an editing routine which utilizes existing OS/360 facilities quite extensively.

3.3.1 OS/360 Language Processor Output

The output of all language processors in OS/360 is a relocatable module consisting of three parts:

3.3.1.1. The object code (relocatable);

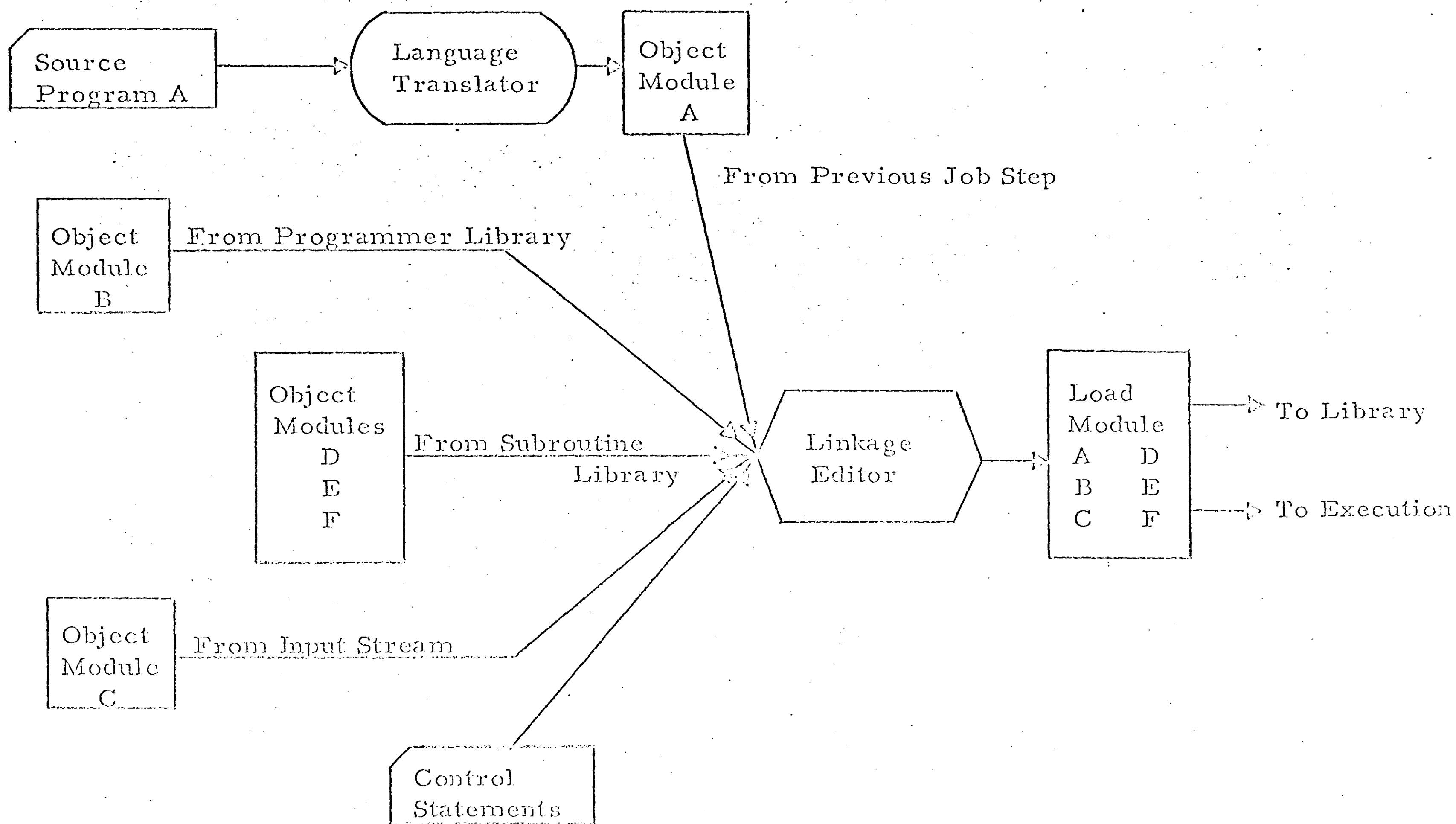
3.3.1.2. A Relocation Dictionary which specifies those words in the object code containing addresses which may need modification;

3.3.1.3 An External Symbol Dictionary which specifies those symbols not defined in the module itself, and those symbols which are entry points to be utilized by other modules.

It should be noted that the object code and Relocation Dictionary are the same as those used in spaceborne programs. Also, since the output of all processors has this format, modules resulting from different source languages can be edited into a single program without special effort. This permits the programmer to utilize more than one language within the same program, choosing the one most suitable for each function.

3.3.2 The Linkage Editor

Traditionally, loaders have been responsible both for resolving symbol definition between relocatable modules, and for binding the resulting program to core locations. In OS/360 these functions have been separated, and the first function assigned as the responsibility of the Linkage Editor. This editor will combine modules from a variety of sources into a single module of the same form. If the resulting module has an External Symbol Dictionary which contains only system symbols and entry points, it is a (relocatable) load module, and can function independently in the system.



Linkage Editing
Figure 3. 3. 2

The flexibility obtainable by this scheme is illustrated in figure 3.3.2, which shows the possible sources of Linkage Editor input, all of which can be utilized at the same time to produce a load module.

3.3.3. The Preparation Language

To produce a program in the Spaceborne format, the preparation processor need only take a load module, or a set of such modules if the program is to be composed of links, examine the External Symbol Dictionary and the preparation statements, and produce a Program Control Block to replace the External Symbol Dictionary.

Load modules which are input to the preparation processor contain only entry points and spaceborne system symbols, that is, the Program COMMON, Universal COMMON, and file symbols. The entry points are present for one of two reasons: they were used in building the load modules from other modules; or they are to serve as segment entry points.

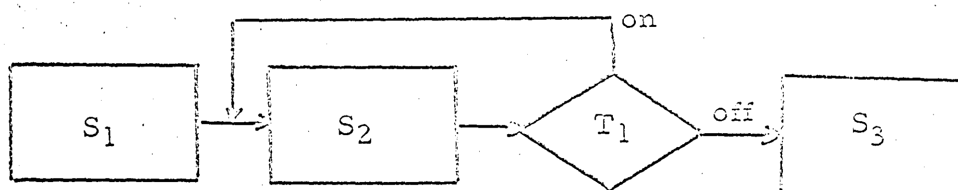
Th^ information that must be supplied by the programmer is the following: the names of the segment entry points; the Program Entry Repetition Rate; the composition of a link; the segment flow information.

Preliminary studies indicate that the language in which this information is supplied should be non-procedural, and requires only a minimum number of statement types. These are:

3.3.3.1. A statement whose function is to identify the program by name and define the correspondence between modules and links.

3.3.3.2. A statement whose function is to identify the segment entry points and to specify the repetition rate for a link.

3.3.3.3. A statement which specifies how flow control is to proceed upon completion of a given segment. A set of such statements describes the entire flow. For example, if the flow of control is symbolized by the diagram



where S_1 are the segment entry point names, and T_1 represents a condition associated with trigger one, the flow information content of the statements would be

$$\text{BEGIN} \longrightarrow S_1$$

$$S_1 \longrightarrow S_2$$

$$S_2 \longrightarrow S_2\ (T_1\ \text{ON}),\ S_3\ (T_1\ \text{OFF})$$

$$S_3 \longrightarrow \text{END}$$

There is no requirement that these statements be in any particular order, because the ordering is implied in the set as a whole.

3.3.3.4 A statement to signify to the preparation processor the end of information.

Default conventions are also established in order to simplify the programmer's usage of these statements. For example, the mechanism for flow control is for the programmer's aid and at his convenience. Since a simple sequential flow is common, there should be no need to specify it. This, and similar procedures, can be arranged as the default conventions assumed by the processor.

3.3.4 Operation of the Preparation Processor

The operation of the preparation processor portion of the Ground Support System then involves the following actions on the part of the programmer and the system.

3.3.4.1. The programmer writes, in one or more source languages, sets of statements which represent the computatations desired in a program.

3.3.4.2 These sets are processed by system language processors to produce relocatable modules. Standard OS/360 procedures allow both the source language statements and the relocatable modules to be placed in a library for later retrieval and usage.

3.3.4.3. Having decided upon the structure of a program, the programmer calls upon the Linkage Editor to produce load modules. The basic relocatable modules may come from a number of sources, for instance, the input stream, module libraries, subroutine libraries, and output of language processors. The programmer's responsibility is to combine the modules in such a way that there are no unresolved symbol references unknown to the Spaceborne System, and that at least one entry point name exists for every segment.

3.3.4.4 The Program Preparation Processor is then called. It reads the preparation language statements, collects the appropriate load modules, and produces the final form of the program. Again, the load module input may be from a resident library or may have been just prepared. Furthermore, the preparation statements may themselves have come from a source statement file in the system. The output may be retained in system files, and may also be used immediately in a simulation run.

Finally, it should be emphasized that because the Preparation Processor runs within OS/360, all these processes may be carried out in a single machine run.

3.4 Languages and Language Processors

The previous discussion assumed that the relocatable modules produced by the language processors were suitable for use in the Spaceborne System without change, and that all languages supported in OS/360 were equally suitable for Spaceborne application. In principle this is true, but in practice difficulties arise from several sources.

First, the languages may lack means of expressing some function required by the system. Second, the languages may be too rich,

allowing the expression of functions which require too much overhead for the spaceborne system. Third, the processors may produce modules which depend upon system facilities not available in the spaceborne system, or upon communication and linkage conventions foreign to it. Fourth, the code produced by a processor may not be efficient enough for the applications.

Consideration of these problems has led to the selection of three OS/360 Language/processor combinations as the basis for writing programs: the assembly system; FORTRAN IV; and PL/1. The following paragraphs examine the suitability of these combinations in terms of the previously mentioned sources of difficulty.

3.4.1. The Assembly System

Because of the instruction set compatibility between the AVE and S/360, the macro-assembly system of OS/360 is useable almost as it stands. Three areas of conflict which exist are readily handled.

3.4.1.1 The possibility of AVE instructions not in S/360, such as direct I/O, is implementable by adding to the macro-instruction library, and hence does not affect the system.

3.4.1.2 Machine instructions in S/360 not in AVE, such as the decimal set, are handled by creating a spaceborne assembly

mode in which usage of the offending instructions is flagged. Since the assembler operates in several such modes already, addition of a new mode is a simple task.

3.4.1.3 Macro-instructions exist in the library which are strictly oriented to OS/360. Some, such as GETMAIN, implement functions not available in the Spaceborne System. Others, such as READ, expand according to linkage and communication conventions which conflict with the Spaceborne System. Fortunately, the macro-instruction library is independent of the assembler. Hence, switching to a new library when in spaceborne assembly mode provides an easy solution to the problem.

The programmer need not be aware that the spaceborne assembly mode exists. If his interface is entirely with the Ground Support Supervisor (Section 3.6), the assembler will be called through it, and the mode change and macro library switch will occur automatically.

3.4.2 The Compiler Systems

The OS/360 FORTRAN IV language encompasses ASA FORTRAN, including its mathematical subroutine provisions. This is a well-known, widely used, thoroughly tested, high-level scientific programming

language. Its use is very desirable because experience has shown that compilers for this language can produce very efficient code, because it requires minimum training for the programmer, and because of the existence of a large body of checked-out programs.

FORTRAN is, however, not adequate by itself for the spaceborne applications. Its deficiencies stem from its scientific orientation, which limits computation almost exclusively to floating point. Apart from integers, there are no language facilities to describe fixed point data, or to obtain fixed point computation. For this reason, PL/1, which incorporates extensive facilities for fixed point computation, is suggested as a high-level language complement to FORTRAN.

However, PL/1 in its entirety is an extremely rich language, encompassing the functional capability of FORTRAN, COBOL, and ALGOL. Some of its storage allocation techniques, such as those for automatic variables, are simply not suitable for the spaceborne system; its data description capability is too elaborate for the applications; some of its conventions conflict with FORTRAN conventions. Clearly, then, what should be used is some 'reasonable' subset of PL/1 rather than the full language. Preliminary studies indicate that such a subset may be readily selected from the several classes of statements available within PL/1, as defined in the IBM publication 'PL/1 Language Specifications', form C28-6571-2.

Figure 3.4.2 specifies a subset which is deemed desirable for the application as presently defined. In choosing this subset, only those features of the language were retained which were felt to be useful in the spaceborne applications. Thus, for example, most fixed point capability has been included, while all multitasking control has been eliminated.

As for the processors themselves, both the PL/1 and the FORTRAN processors generally produce code which interfaces with the system by means of calls upon library routines. Since these library routines are incorporated into the program only upon its final assembly by the Preparation Processor, substitution can be accomplished by the system at that stage. Changes to the processors are therefore quite minimal.

3.4.3 Segment Definition

The one feature of the spaceborne system not directly translatable into source statements in any of the languages, is the segment. However, the segment is basically a structural and operational concept, not a procedural one, and is handled as follows.

3.4.3.1 All languages provide for an entry point definition, and hence for a segment entry (see Section 3.3).

PL/1 SUBSET

FEATURES INCLUDED	FEATURES EXCLUDED
Assignment Statements	
Option 1, Scalar Assignment with Pseudo-variables	Pseudo-variables in Option 1
ONSOURCE	COMPLEX
ONCHAR	REAL
SUBSTR	IMAG
UNSPEC	PRIORITY
Option 3, Structure Assignment	Option 2, Array Assignment
Option 4, Statement Label Assignment	Option 5, Pointer Assignment
Control Statements	
CALL	CALL options
DO	TASK
GO TO	EVENT
IF	PRIORITY
RETURN	DELAY
STOP	EXIT
Error Control Statements	
ON	The ON option SNAP
	SIGNAL
	REVERT
Input/Output Statements	
GET	PUT options
PUT	PAGE
DISPLAY	SKIP
OPEN	LINE
CLOSE	READ
List-directed transmission	WRITE
Edit-directed transmission	DISPLAY options
Data lists with repetitive specification	REPLY
	EVENT
Format	DELETE
	Data-directed transmission, complex transmission, and data Picture format in edit-directed transmission
	Control
	Standard files

Figure 3.4.2 Part I

Data Declaration Statements	
DECLARE	
Attributes - BINARY/DECIMAL FIXED/FLOAT Precision String with BIT CHARACTER	Attributes - REAL/COMPLEX PICTURE String with VARYING PICTURE
LABEL	Asterisk length specifications
Dimension with signed decimal integer bounds	TASK EVENT
ENTRY GENERIC BUILTIN RETURNS INTERNAL/EXTERNAL ALIGNED/PACKED DEFINED INITIAL FILE STREAM INPUT OUTPUT PRINT ENVIRONMENT STRUCTURE	Dimension with bounds that are expressions or asterisks SECONDARY ABNORMAL/NORMAL USES/SETS STATIC/AUTOMATIC/CONTROLLED/ CONTROLLED(pointer-variable) LIKE RECORD UPDATE SEQUENTIAL/DIRECT BUFFERED/UNBUFFERED BACKWARDS EXCLUSIVE KEYED AREA POINTER
Storage Allocation Statements	ALLOCATE FREE
Program Structure Statements	
PROCEDURE BEGIN END ENTRY DO	PROCEDURE option RECURSIVE BEGIN reflecting storage allocation

Figure 3.4.2 Part II

3.4.3.2 All languages provide for a system return, with options. Selection of one of these as segment end involves no language changes. Also, its significance is interpreted by the spaceborne executive, not the OS/360 executive, and so involves no system change.

3.4.3.3 Trigger setting and resetting is done by subroutine call or system return. All languages provide for these.

3.4.3.4 No language features exist for monitoring segment execution time. However, such monitoring is operational and is performed by the simulator.

Thus no changes in the languages or processors are required because of the segmenting conventions of the spaceborne programs.

3.5 Simulation and Test

Another important function of the Ground Support System is the simulation of the spaceborne programs, both system and application, in order to ensure valid operation. For this purpose, it is essential to execute the spaceborne form of these programs in the AGE equipment without substantive changes. Because of the instruction set compatibility the programs can execute directly, but because of the environmental differences it is necessary to introduce a set of routines

which will mediate between the OS/360 ground environment and the internally created spaceborne environment.

This set of routines, called the Simulation Processor, also operates under OS/360. It divides naturally into two groups of functions. First, it exercises control over the set-up for simulation, including initiation of the simulation environment, interpretation of statements which specify data generation and debugging, and conversion of such information into a form suitable for use during simulation. Second, it monitors the execution, acting to supply data, requests from the (simulated) outside, and interrupts. It also collects the information about program execution specified by the debugging requests, or required by the spaceborne environment.

3.5.1 The Simulation Supervisor

Even though the spaceborne programs do not execute interpretively in the simulation environment, a Simulation Supervisor is always in control. The key to this control lies in the segment structure of the programs, and the treatment of privilieged instructions in S/360. The former is used to gain entry to monitor operational programs, collect debugging information, and effect interrupts. The latter is used to obtain control from the Basic Executive of the Spaceborne System in order to handle data insertion.

Consider, first, programs running in the simulation environment under the spaceborne executive programs. Since the simulator is itself an OS/360 program, it operates in the problem program state. Hence, when a program returns to the System Executive via a supervisor call, the interrupt is actually handled first by the OS/360 interrupt supervisor. This supervisor, recognizing the call as a type belonging to the Simulation Supervisor, transfers control to it. The Simulation Supervisor then

3.5.1.1 Clocks the execution time of the segment.

3.5.1.2 Examines the list of requests for debugging. If debugging information is desired at this point, it is collected and output through standard OS/360 routines.

3.5.1.3 Examines information which may have come in via the S/360 operator which is intended to simulate some external process. Such information for example, may be a request for the execution of some program, which, if the system were in the air, would have been made by the astronaut. If such information is waiting, it will exit to the System Executive as if an interrupt were pending. If not, it will return control without such indication.

Next, during the course of simulation execution, the Basic Executive will get control to handle I/O. Since the I/O instructions are privileged, they will not be executed. Again, an interrupt to the OS/360 interrupt supervisor will occur, this time to indicate privileged instruction execution. In order for the Simulation Supervisor to get control at this point it is only necessary to supply a second level interrupt handler in OS/360 which enlarges the function of the standard routine. No conflict with standard operation will a rise from this.

In effect, then, there exists a simple mechanism for supervising the spaceborne execution during simulation without the necessity for altering instructions in the programs. Also, since the Simulation Supervisor operates under OS/360, it is able to utilize the full capability of the system.

3.5.2 Program Test

In general, programming errors can be put into three categories: syntax errors, which are mistakes in the grammar of the programming language being used; mechanization errors, which are mistakes in translating an algorithm or solution from its original statement into a programming language; and logical errors, which are mistakes in the generation of the algorithm or solution to a problem, or possibly in the problem statement itself.

Errors in syntax are detected by language processors, and hence there are no special requirements on the simulator for these. Certain kinds of mechanization errors may also be detected by language processors, particularly those concerned with flow and naming conventions. Unreachable portions of a program, and undefined or ambiguously defined symbols are of this class. Here again there is no requirement upon the simulator for such diagnostic information.

For logic errors, however, and for errors which may arise because of the dynamic environment, the simulator must make provision for the user to insert test information, and to receive information about program flow and data transformation at various stages. The internal processes for handling this were described in Section 3.5.1. Externally, however, the Simulation Supervisor must be able to interpret statements from the programmer in three languages.

3.5.2.1 A debugging language by which the user can request pertinent information about his program. For example, such items as areas to be displayed at the end of segment execution, the flow of control actually followed by the program, and trigger settings at various times.

3.5.2.2 A data generation language by which the user can conveniently describe the data to be 'read' during the course of program execution.

3.5.2.3 A simulation control language by which the user can specify the interconnection between various programs to be used in a simulation, and ways in which programs are to respond dynamically, in closed loop fashion, to the execution.

Specification of these languages requires further study.

3.6 Integration of the Ground Support System

The previous discussions have covered the parts of the Ground Support System in a somewhat independent way. However, it is clear that in practice utilization of these parts will tend to be interdependent. One will want, for example, to be able to prepare a spaceborne program and cause a simulation of its execution in a single machine run.

No special support effort is necessary in order to do this, since each activity could be treated as a job step within the OS/360 definition. However, it is proposed that the entire Ground Support System be integrated under a single Ground Support Supervisor, which, as far as OS/360 is concerned, is just another operational program. From a programmer point of view, this will enhance efficiency because of the

necessity for fewer control cards. From an overall installation point of view, ground support operation, and the standardization of the maintenance and documentation of the spaceborne programs, is greatly simplified.

4.0 IMPLEMENTATION PLAN

The development and production of validated programs for the DCSG requires a carefully developed and controlled implementation plan. The plan must include procedures to span the time from inception to delivery of the operational programs, with milestones that provide a precise control over the distinct areas of software development. Such a plan is presented here.

4.1 Task Sequence

The sequence of tasks required for the implementation procedure is presented in the form of flow charts, figures 4.3 and 4.4, which cover the areas of program development and program validation respectively. These charts begin with definition of individual program requirements, and carry through to complete integration of all programs.

The following is a block-by-block description of the charts. For ease of reference, each block is identified in two ways. In the upper right-hand corner is a number which correlates the block with a descriptive paragraph. In the lower right-hand corner are abbreviations designating the general area of responsibility. These are as follows:

LV - Laboratory Vehicle

MM - Mission Module

OV - Orbiting Vehicle

DCSG - Data Computation Subsystem Group

4.1.1 Define Computer Program Requirements (Blocks 1.0 and 2.0)

Basically, three things must be done:

4.1.1.1 Analysis of problem requirements. That is, the identification of functions to be performed, and the allotment of specific tasks to the DCSG.

4.1.1.2 The development of system equations. The equations required to implement the DCSG assigned tasks are defined and preliminary math/logic flows are determined.

4.1.1.3 The definition of accuracy requirements and solution rates.

4.1.2 Ground Support System Specifications (Block 4.0)

The complete system of support software is designed and specified. This includes the internal design of the programs and generation of complete external specifications. Preliminary operating instructions are published to aid in the development of spaceborne programs. (The Ground Support System is outlined in section 3.0 of this paper.)

4.1.3 DCSG Spaceborne System Specifications (Block 4.1)

Preliminary functional design of the Spaceborne System is completed and documented. (The basic outline of this design is described in section 2.0 of this paper).

4.1.4 Coordinate Computer Requirements and Support Programs (Block 3.0)

This activity is mainly a function of coordination, control, and dissemination of information. Careful examination of the results of blocks 1.0, 2.0, 4.0, and 4.1 is made in order to assure DCSG and support

system compatability, eliminate duplication of effort, and identify specific subroutines. Support system write-ups are released to user groups, and approval of all activity must be given before work continues.

4.1.5 Definition of Program Interfaces (Blocks 1.1 and 2.1)

The interfaces between computer routines and external signals are identified and clearly defined. Also identified are the formats to be used, the methods of access to common data, and any restrictions the interfaces impose upon the programs.

4.1.6 Math Flow Generation and Simulation (Blocks 1.2 and 2.2)

For each program, a math and logic flow is generated, simulated, and regenerated until a functionally optimum definition is obtained. Closed loop simulation is performed on a general purpose computer to insure satisfaction of system requirements. From this point the effort may proceed through block 1.7 as an integrated program development, or as several semi-independent efforts.

4.1.7 Test Program Definition (Blocks 1.2.1 and 2.2.1)

This activity involves planning and definition of test routines and data to fully exercise the DCSG program during soft simulation. The test programs also serve as inputs to the functional simulation described in Blocks 1.6.1 and 2.6.1.

4.1.8 Math Flow Release (Blocks 1.3 and 2.3)

The math and logic flow is formally released and accepted, and becomes the basis for coding the operational programs, generating the test programs, and creating the functional simulation.

4.1.9 Generate Loop Closure Techniques (Blocks 1.3.1 and 2.3.1)

This activity is an extension of the closed loop simulation done in Blocks 1.2 and 2.2. It provides two outputs for use in later validation. The first output, indicated by the letters A and C on the chart, provides environmental information to the System Integration Laboratory (SIL) checkout (Block 5.0) and the Hot Mock-Up (HMU) Checkout (Block 6.0), which allows a dynamic hardware checkout on an individual basis for each routine. The second output, indicated on the chart by the letters B and D, provides dynamic environmental information for use in the soft checkout of the Laboratory Vehicle integrated programs and the Mission Module integrated programs (Blocks 5.4 and 6.4).

4.1.10 Detail Flow (Blocks 1.4 and 2.4)

The math and logic flow is adapted to the nature of the DCSG and programming artifices peculiar to the DCSG are introduced. All coding will be done from this detail flow.

4.1.11 DCSG Program Development (Blocks 1.5 and 2.5)

DCSG programs are coded and prepared by the Preparation Processor (see 3.3). Specialized hand checks are used to isolate as many errors as possible before beginning simulation.

4.1.12 Test Program Development (Blocks 1.5.1 and 2.5.1)

Test routines and data are developed from the plans generated in Blocks 1.2.1 and 2.2.1. Test programs are designed to exercise the operational routine as completely as possible.

4.1.13 DCSG Program Simulation (Blocks 1.6 and 2.6)

Using the DCSG Spaceborne System and the test programs, the operational program is exercised in all its modes to test for correct performance. These tests are soft simulations, performed first on an open loop basis and subsequently, where applicable, on a closed loop basis.

4.1.14 Functional Simulation (Blocks 1.6.1 and 2.6.1)

In order to develop an independent check of the transition from the released math and logic flow to the detailed flow, and to verify the programming artifices used in Blocks 1.5 and 2.5, the released math and logic flow is functionally simulated on a general purpose computer using the same test programs as the DCSG program simulation.

4.1.15 Comparison (Blocks 1.7 and 2.7)

The results of the DCSG program simulations and the functional simulations are compared to verify that results are within acceptable tolerances. This step completes the development phase.

4.1.16 Develop DCSG Ground Support Software (Blocks 4.2 and 4.2.1)

All Ground Support System programs are developed and checked out. These are then provided to the computer program users for their operational programming activities.

4.1.17 Spaceborne System Development and Checkout (Blocks 4.2.2 & 4.3)

The Spaceborne System implementation is completed and checked out on the DCSG simulator. The Spaceborne System is required during open loop DCSG program simulation (Blocks 1.6 and 2.6) as well as during hardware validation of the programs.

4.1.18 DCSG System Checkout Using Test and Checkout Programs (Blocks 4.4 and 4.5)

The DCSG system to be used during hard checkout of the operational routines is developed and validated with test programs, for use at point G in Figure 4.4. This is accomplished as described in figures 4.1 and 4.2, which outline the five basic levels of DCSG computer assembly and checkout and the main purpose of testing at each level. The abbreviation key for these figures is as follows:

CPU	Central Processing Unit
MGE	Maintenance Ground Equipment
KDU	Keyboard Display Unit
HCP	Hard Copy Printer
ASU	Auxiliary Storage Unit
CSC	Computer Subsystem Controls
LVDAU	Laboratory Vehicle Data Adapter Unit

4.1.18.1 At Level A, all communications with the central processing unit are through the MGE which provides controls and displays to govern and monitor the progress of testing. All program words (instructions and data) must be loaded into DCSG Main Storage from the MGE by means of card decks or program tapes.

4.1.18.2 Level B, testing extends to the inclusion into the subsystem of a previously verified Auxiliary Storage Unit. The ASU stores the kernel of the DCSG executive routine, which allows check programs to be transferred from ASU to main storage at

startup. The MGE provides power, start-stop control, and monitoring functions.

4.1.18.3 Manual interfaces are introduced to the DCSG at <u>Level C.</u> At this point, the MGE is required for primary verification of keyboard, display and printer operations. Following this verification, this portion of the subsystem (which is equivalent to a normal EDPM installation) can be completely checked out by use of the keyboard, display and printer. This checkout program is the first which will be available for inclusion in the operational ASU.

4.1.18.4 <u>Level D</u> testing brings in the computation subsystem controller. MGE involvement is necessary to check the CSC-CPU combined operations and to simulate the command system interface to CSC. The ASU checkout programs are essentially the same as at Level C.

4.1.18.5 <u>Level E</u> introduces checkout of the LVDAU, and hence represents complete DCSG checkout capability. The ASU programs for this level are more complex than those which will be used for on-orbit testing, and are of a somewhat different nature. The on-orbit test programs will be checked at this level as a series of subtests. The MGE also simulates telemetry and command interfaces, in so far as is necessary to simulate auxiliary ground checkout of the orbiting DCSG. Checkout programs at this level are

directly analogous to those required for the all-system test equipment group (ASTEG) at DAC.

Each of the software items identified above is required to meet Milestones 1 through 8. (On-orbit check programs are also required for Milestone 8).

4.1.19 SIL Individual Program Checkout (Block 5.0)

Individual programs which have completed the soft simulation and checkout are exercised in the SIL environment and verified in the DCSG. Verification takes place on the actual DCSG computer with as complete a closure as possible through actual physical interfaces. Simulated environment will be provided through the general purpose computer tied into the SIL complex, using programs generated in Block 1.3.1.

4.1.20 SIL Test Programs (Block 5.1)

Programs are developed for the general purpose SIL computer to provide stimuli to the individual DCSG programs during SIL checkout.

4.1.21 Interface Simulator (Block 5.2)

In the SIL complex, many of the interfaces to external devices must be simulated, requiring hardware simulators and computer programs for the SIL general purpose computer. Simulators must insure electrical and physical compatability between the DCSG, the device being simulated, and the environmental data generation.

4.1.22 Mission Module Hot Mock-Up (Block 6.0)

The effort involved in this area is similar to the SIL program checkout block (5.0). It consists of individual program checkout of mission

programs on a DCSG hardware setup.

4.1.23 HMU Test Programs (Block 6.2)

Again, this activity is similar to the SIL test program (Block 5.1). It provides the same type of input to the Hot Mock-up complex as is provided to the SIL complex.

4.1.24 LV Program Integration and Soft Checkout (Block 5.4)

At this point all LV programs are integrated with each other and with the Spaceborne System. A complete closed-loop soft simulation is performed to insure compatability and proper operation. Loop closure is provided as a result of the information generated in Block 1.3.1.

4.1.25 MM Program Integration and Soft Checkout (Block 6.4)

This activity involves the same operation for Mission Module programs as was accomplished for Laboratory Vehicle programs in Block 5.4.

4.1.26 LV Program Integration and SIL Checkout (Block 5.3)

This activity is the hardware checkout counterpart of Block 5.4. It involves integrating all LV programs into the SIL checkout facility with the Spaceborne System, and verifying proper operation on a closed loop hardware basis.

4.1.27 MM Program Integration and HMU Checkout (Block 6.3)

This is the counterpart of Block 5.3 for the Mission Module programs. Complete hardware checkout is on the Hot Mockup facility.

4.1.28 Computer Program LV and MM Integration (Block 7.0)

All LV and MM programs and the Spaceborne System are integrated into a complete mission program package to form the complete software package for a particular mission. Strict control is maintained to insure the integrity of the package in the final launch configuration.

4.1.29 Overall Soft Checkout of Integrated System (Block 7.1)

Final digital simulation of the complete package is performed at this point. Primary concern is to assure compatability of the LV, MM, and Spaceborne System in the operational configuration, independent of the hardware. This is a closed loop simulation which closely duplicates the operational environment.

4.1.30 Overall SIL Checkout of Integrated System (Block 7.2)

This activity provides a complete checkout of the integrated program package on the hardware. Tests similar to those in Block 7.1 are executed. Special emphasis is placed on running programs under real time conditions (such as random interrupts, dynamic relocation, and device dependent I/O timings) and the system is operated in a manner which duplicates the actual spaceborne configuration as closely as possible.

4.1.31 Release and Implementation of Complete Mission Package (Block 8.0)

This is the final control point for the package. At this point, validation is considered to be complete and the program package ready for flight, so that all program material and documentation is released to the launch site.

4.2 Milestones

Milestones have been placed at strategic points, so that a clear picture of program progress may be obtained. Eight milestones are defined for the Implementation Plan. These are as follows (again refer to Figures 4.3 and 4.4):

4.2.1 Milestone 1 - Computer Program Requirements

All requirements for the DCSG System, the Ground Support System, and the Spaceborne System have been defined. Documents are generated detailing these requirements.

4.2.2 Milestone 2 - Computer Program Concept and Test Plan

This document defines general math flow and plans for implementation and test. It is based upon Milestone 1 and serves as a foundation for all future effort.

4.2.3 Milestone 3 - Computer Program Interface Specifications

This document defines the hardware and software interfaces for the functions of the computer subsystem.

4.2.4 Milestone 4 - Computer Program Design and Acceptance Specifications

This document provides a detailed definition of the computer program math flow, and specifies details of implementation. DCSG program development will be initiated upon completion of this document.

4.2.5 Milestone 5 - Computer Program Coding Completion

Coding for the DCSG program has been completed, the test programs have been developed, and the system is ready for open loop simulation.

4.2.6 Milestone 6 - Computer Program and Associated Support Documentation

Individually checked-out and documented computer programs are delivered. At this point, program development is complete and the programs are presented for subsystems validation.

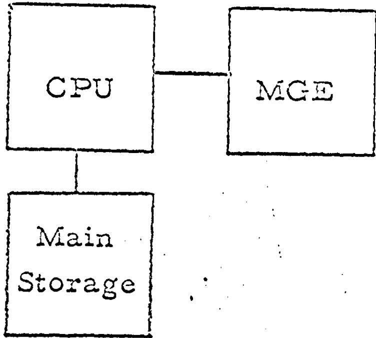
4.2.7 Milestone 7 - Subsystems Checkout Complete

All programs and hardware have been integrated and validated through the subsystem level. The computer programs are now ready for LV and MM integration.

4.2.8 Milestone 8 - Release and Implementation of Complete Mission Package

This is the final milestone and constitutes the completion of all program integration and testing. At this point, a completely validated set of programs is ready for flight.

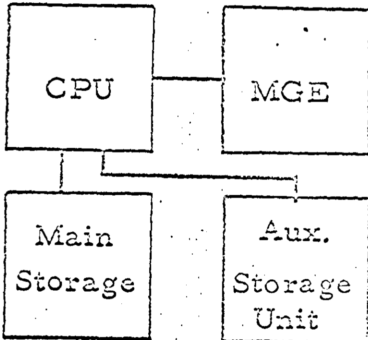
Level A



Requires: MGE Program
DCSG Check Programs

Checks: All CPU Operations
Main Storage
I/O Channel

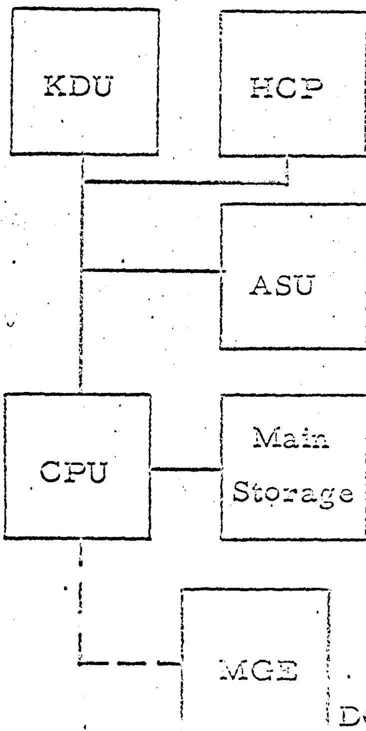
Level B



Requires: DCSG Executive Program
DCSG Check Programs
MGE Programs

Checks: Program Load
Program Execution
Full CPU-Main Memory-ASU Operations
I/O Control Unit #1

Level C



Requires: DCSG Executive Program
DCSG Check Programs
MGE Program

Checks: KDU and HCP Operations
Simultaneous DKU, HCP, ASU

FIGURE 4.1
DCSG CHECKOUT SEQUENCE

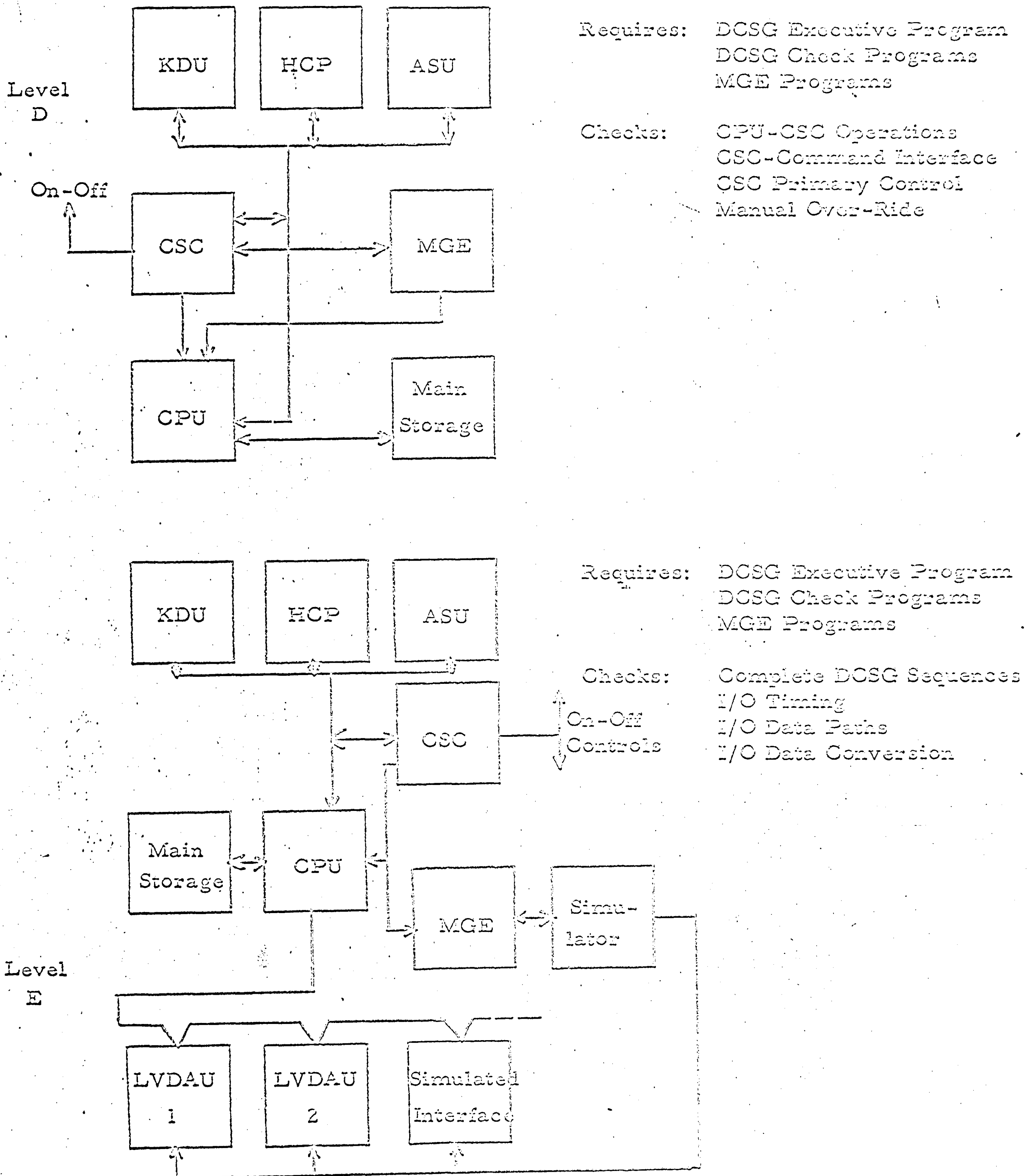


FIGURE 4.2
DCSG CHECKOUT SE[illegible]CE

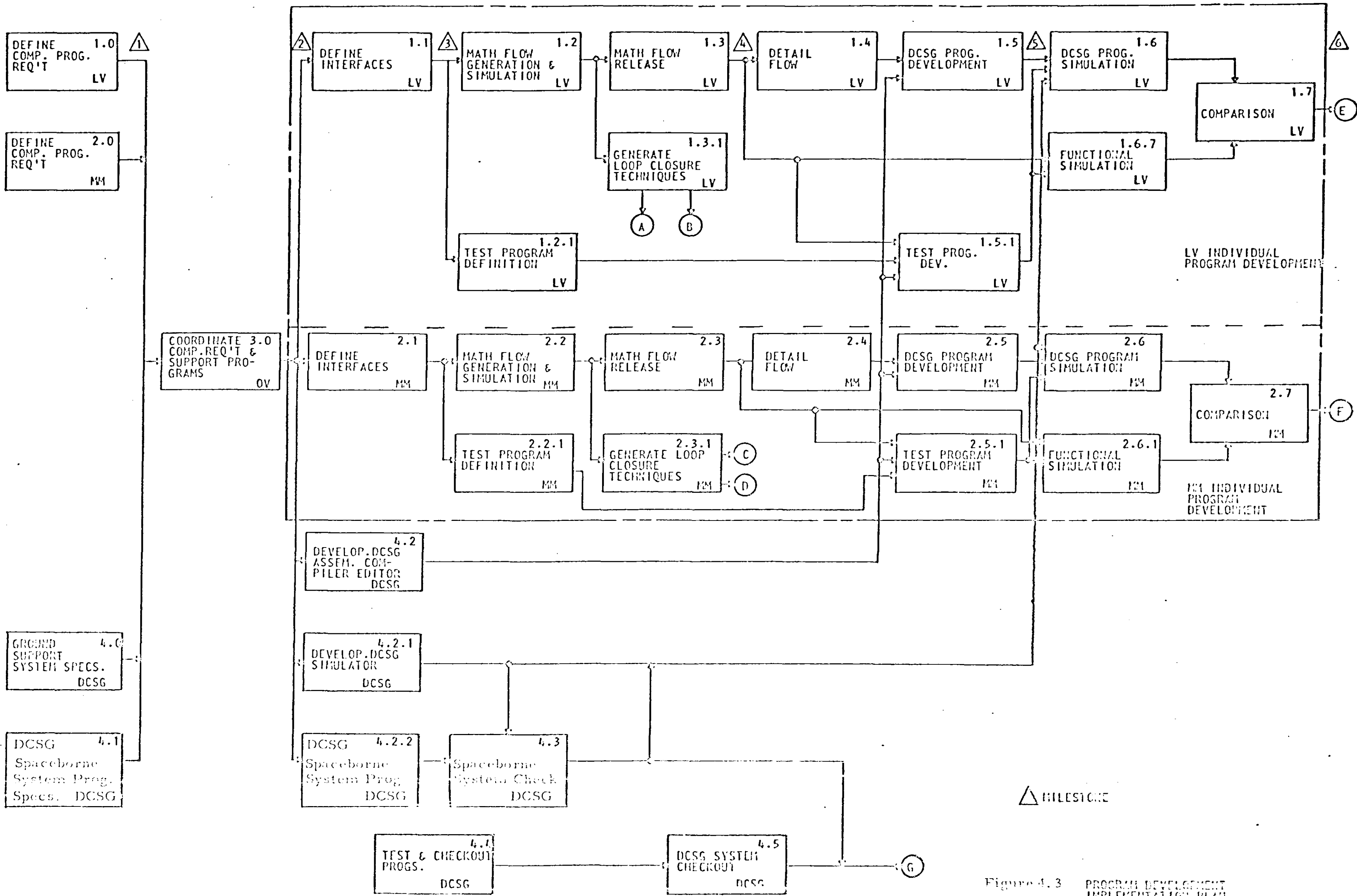
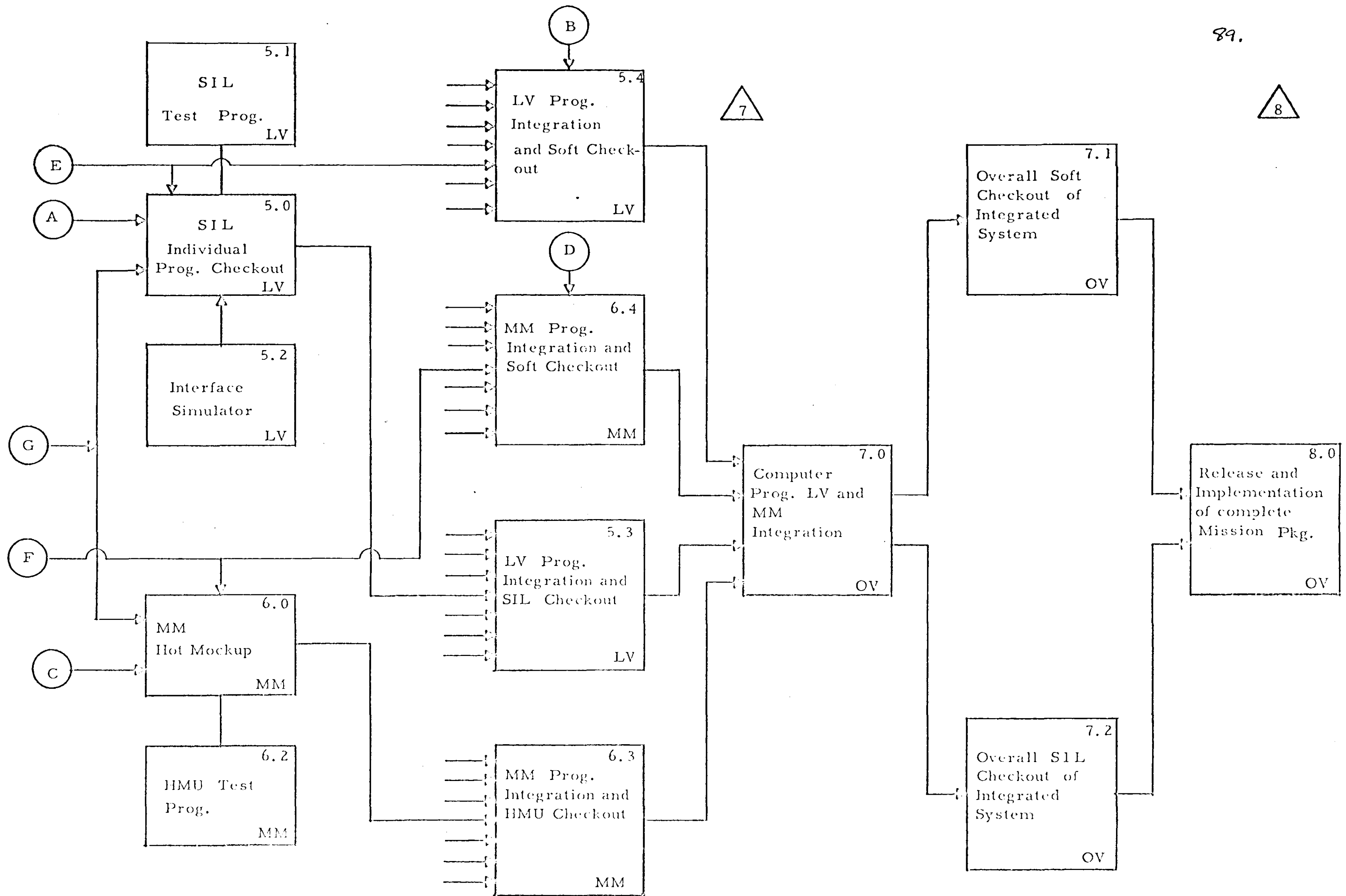


Figure 4.3 PROGRAM DEVELOPMENT IMPLEMENTATION PLAN



PROGRAM VALIDATION IMPLEMENTATION PLAN

Figure 4.4