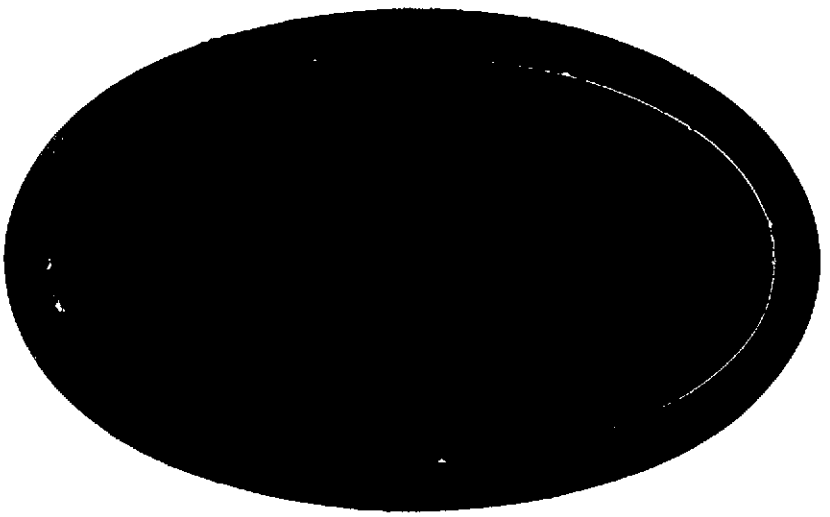
2 MMP



(NASA-CR-120182) SUMC/MPOS/HAL INTERFACE STUDY Final Report (Intermetrics, Inc.) 86 p HC $7.50 CSCL 09B

N74-20839

Unclas
G3/08 16079

Final Report

SUMC/MPOS/HAL

Interface Study

December 1973

Prepared by:

Joseph A. Saponaro
Alex L. Kosmala

Prepared for:

George C. Marshall Space Flight Center
Huntsville, Alabama 35812 under contract:
NAS 8-29607

by:

Intermetrics, Inc.
701 Concord Ave.
Cambridge, Mass. 02138

Foreword

This document is the final report of a study to analyse the feasibility of providing a HAL capability on the NASA/MSFC SUMC multiprocessor operating under the proposed MPOS. The study was sponsored by the NASA Marshall Space Flight Center, Huntsville, Alabama, under contract NAS 8-29607 entitled, Research Study on Memory Hierarchy. It was performed by Intermetrics, Inc., Cambridge, Mass., under the direction of Alex L. Kosmala. Technical monitors for MSFC were Mr. Bobby Hodges and Mr. Jim L. Lewis.

Publication of this report does not constitute approval by NASA of the findings or conclusions contained therein.

Table of Contents

1.0 INTRODUCTION

This report describes the implementation of the HAL/S language on the IBM-360, and in particular the mechanization of its real time, I/O, and error control statements within the OS-360 environment.

1.1 Scope and Objectives

This material is the result of an evaluation of the HAL/S Language/SUMC Computer Operating System (MPOS) interface under contract NAS-8-29607 for Marshall Space Flight Center. In accordance with the scope of this effort the objectives are twofold:

a) TASK 1 - An analysis and general description of HAL/S real time, I/O, and error control statements and the structure required to mechanize these statements. The emphasis is on describing the logical functions performed upon execution of each HAL statement rather than defining whether it is accomplished by the compiler or operating system.

b) TASK 2 - (a) An identification of the OS-360 facilities required during execution of HAL/S code as implemented for the current HAL/S-360 compiler.

(b) An evaluation of the aspects involved with interfacing HAL/S with the SUMC operating system utilizing either the HAL/S-360 compiler or by designing a new HAL/S-SUMC compiler.

1.2 Assumptions and Groundrules

Although HAL has been implemented on the 360 via a FORTRAN translation, the information presented in this memo is based on: (a) the Shuttle version of HAL, termed HAL/S language defined in "HAL Language Specification", Intermetrics, Inc., 15 April 1973. It has been modified and accepted by NASA/JSC for the Space Shuttle; and (b) the corresponding HAL/S-360 compiler implementation as defined by "HAL/S-360 Compiler System Functional Specification", Intermetrics, Inc., 13 July 1973. This document specifies a HAL/S-360 BAL code generator which compiles and executes on the 360; (c) Intermetrics is involved in an effort for Rockwell International Corp. to define the architecture of the Space Shuttle Flight Computer Operating System. This effort includes analyzing interface requirements between Shuttle Flight Operating System (FCOS) and HAL/S compiled code. A preliminary FCOS Requirements and Architecture document dated 19 July 1973 by Intermetrics is relevant to the MSFC effort and has been utilized in this report.

It is important to note the status of this HAL/S-360 compiler relative to the description presented herein. The preliminary version of the compiler, which was delivered to NASA/JSC on 1 August 1973, does not contain all HAL/S features. An updated version, containing real time and error control statements and other capabilities is scheduled for delivery on 1 October 1973. Accordingly, some aspects of the HAL/S implementation are still undergoing modifications. In addition, some aspects of the implementation are tailored for Shuttle software development, such as the expected flight processing environment and operating system characteristics. As a result, aspects of the detailed design and mechanization are not presented completely at this time. For example, the detailed mechanization of data sharing, process swapping, file I/O, are not finalized. This report does, however, attempt to present the current understanding and design intentions.

2.0 SUMMARY OF MAJOR POINTS

The emphasis of the MSFC effort is to focus on aspects and problems relative to HAL interaction with a target computer operating system and specifically MPOS. The following paragraphs summarize major points and highlight aspects which should be pertinent to the MSFC effort. These are: 1) HAL/S-360 real time implementation, 2) HAL/S interfaces with OS-360, and 3) Implications of HAL/S to interfaces with the SUMC/MPOS. Detailed information on these topics is presented in this report.

2.1 HAL/S-360 Real Time Implementation Summary

(a) The HAL/S-360 compiler is currently implemented as a "self-contained" system which executes as a single task/job step under OS-360. A load module is created by a "HAL Link Step" using the 360 linkage editor. The load module contains all HAL/S compiled programs/tasks, external procedures, and compool blocks which are pertinent to the run, together with a collection of run time routines. This load module or HAL/S system is then loaded and executed under OS as a single task.

(b) All HAL/S process management functions, that is control over the scheduling and dispatching of HAL/S program and task blocks, are implemented through HAL run time routines. The HAL/S real time control statements (i.e., SCHEDULE, TERMINATE, WAIT, CANCEL, SIGNAL) are interfaced from the compiler directly to HAL/S run time routines and not to OS-360. The HAL/S run time routines utilize internally defined process queues. The process states and state transitions are controlled by HAL/S compiler run time routines. The compiler generates "branch and link" commands to the appropriate HAL/S routine to implement execution of its real time statements. All HAL/S event tables, event queues and the processing of event expressions are performed by HAL/S run time routines. There is no interaction with OS-360 for servicing event variables.

A timer queue and HAL/S process interaction with timed events is controlled by HAL/S run time routines. The logical implementation of these routines is presented in Sections 4, 5, and 6 of this report.

(c) OS-360 control and OS task interaction is limited to supervision of the HAL/S system load module. It is unaware of the existence of multiplicity of HAL/S processes and queues.

In summary, HAL/S interacts with OS-360 only at the "HAL/S load module level" or system level as a single OS task and not at the statement level or HAL program/task block level (i.e., a HAL process).

(d) The HAL/S-360 implementation does not execute in "real-time" on the 360. HAL/S pseudo time is maintained in "machine units" by HAL/S run time routines. Internal pseudo clock registers are updated in machine units which are decremented by a "clock tick" HAL run time routine after the execution of each HAL/S statement. The intended objective is to model the estimated execution time of each HAL statement for a specific Shuttle flight computer on the 360, and to maintain this simulated clock as HAL statements are executed on the 360. This should aid in the testing of flight software by direct execution on the 360 without requiring simulation. Preliminary versions of the statement execution time model will utilize HAL/S statement 360 execution time. The HAL/S-360 system does not utilize the real time OS-360 clocks.

(e) In HAL/S-360, the compiler inserts "hooks" between the code generated for each HAL/S statement to enable recording of variables, implementation of diagnostics, clock updating, process control, and other functions. These HAL/S-360 hooks will ultimately be interfaced to the NASA/JSC/SDL simulation facility to enable Shuttle avionics environment updates and diagnostics.

(f) The HAL/S-360 memory management is controlled primarily by the compiler for an individual program, and by the link function for all programs. Each HAL/S compiled unit consists of a "code block" and "data block". The data block contains both a static part used for the storage requirements of all code block declared variables and a dynamic portion used by the compiler for linkage

and other temporary storage. The compiler and "linker" determine the total size of memory required for each compilable unit and system. On the current 360 system, memory requirements for the entire load module are determined by the linkage editor. At execution of the load module the user specifies to OS-360 a region size that is sufficient for both the load module and OS requirements for I/O. The design of a memory management scheme is, of course, a function of the target computer, its operating environment, etc., and not a function of the language. It is therefore determined as part of the target computer compiler design. It is not currently designed for a virtual memory environment such as has been contemplated for SUMC. However, it does not utilize overlays or other memory conservation techniques, hence it may be relatively simple to adapt to a paging system.

(g) The host operating system under which HAL/S-360 runs is OS-360 MVT release 21.6. It currently interfaces with the following limited set of OS-360 functions, which are primarily used for I/O (Sequential and File), controlling HAL/S error handling and performing diagnostics: FIND, OPEN, CLOSE, DCB, GETMAIN, DEVTYPE, GET, PUT, READ, WRITE, SPIE, STAE, SYNADAF, SYNADRLS, ABEND, TIME. See Section 9 of this report.

2.2 Evaluation of HAL/S-360 Mechanization and OS-360 Interactions

After evaluating the difficulties of implementing the HAL/S language real time statements with a more direct interaction with OS-360, it was considered more expedient and practical to implement HAL with minimum interface to OS-360. This approach was taken particularly because of the intended use of HAL/S on the 360 as a Shuttle test-bed functional simulation facility. The HAL/S 360 compiler implementation for the flight computer will necessitate a more direct interface between compiler and flight computer operating system to enable direct implementation of HAL real time and multitasking features. A preliminary definition of these interfaces is provided in the "Requirements and Architecture" document referenced in Section 1.2 .

Although it is possible to utilize OS-360 macros to implement statements, several difficulties would be encountered in attempting to match HAL/S language requirements with OS-360 capabilities. They are briefly discussed below.

(1) Under OS, you must be in a priveleged (System) mode to attach an independent task. In the standard user mode only dependent tasks may be attached. In HAL/S both dependent and independent processes are scheduled. An OS-360 user-mode-only implementation would necessitate a "work around" technique to implement HAL.

(2) The DETACH facility of OS is not exactly compatible with the HAL TERMINATE statement requirements.

(3) The full implementation of HAL schedule statement options (i.e., time, event, cyclic, etc.) would necessitate work around techniques utilizing "outer loop" OS tasks and/or events to monitor for conditions,and to initiate HAL processes in accordance with schedule statement options.

(4) The scope of HAL events is not compatible with OS events. In OS only one task can wait for an event. Not so in HAL - multiple tasks can wait for the occurrence of an event or event expression.

(5) OS events correspond only to HAL/S "latched" events, and do not include HAL/S unlatched events.

(6) OS does not allow for direct processing of an event expression with all the options as specified in HAL/S language. A work around would be necessary.

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

2.3 HAL/S Interaction and Implementation on SUMC

Although a detailed review of the SUMC Multi-Processing Operating System (MPOS) was not completed, several observations are made at this time relative to HAL/S implementation in SUMC and interaction with MPOS:

(1) The preliminary MPOS functional design document supplied to Intermetrics is not in sufficient detail for analysis and evaluation of HAL/S-360 and MPOS interfacing requirements as per items 2B and 2C of Intermetrics letter of 9 July 1973 to MSFC. However, it does not appear as though it were meant for this purpose and subsequent MPOS design documents may contain sufficient detail for successful completion of these items.

(2) The HAL/S-360 compiler as delivered to NASA/JSC can be implemented with similar capability in the SUMC/MPOS with relatively minimum modifications. Assuming that SUMC executes 360 compatible code and that MPOS contains those OS routines which the HAL/S system uses as specified in Sect. 9, it should be almost directly compatible. The HAL/S capability, however, would be similar to the MSC version, namely:

   (a) HAL/S operates as a separate self-contained system

   (b) It will function in the same machine with other PL/1 or FORTRAN jobs

   (c) It does not operate in real time

   (d) It contains no multi-processing or MPOS multi-programming features

   (e) It contains no specific virtual memory utilization features

   This would appear to be the easiest, most cost effective route to obtaining HAL/S for the SUMC computer.

(3) Depending on the modifications to HAL/S-360, the intended use of HAL by MSFC, and the dollars available, several modifications could be made to remove some or all of these restrictions.

(a) Through relatively simple modifications HAL/S-360 could be made to execute in near real time. Since HAL/S-360 timer management is centralized, it could be stimulated by the actual SUMC computer clock(s) and timers.

(b) The implementation of HAL/S-360 in the SUMC virtual memory may be relatively simple, depending on the paging technique and memory management system utilized. The structure of HAL/S-360 code/data blocks are contiguous blocks, not overlayed or shared for efficient memory use. The appropriate "hooks" for the SUMC virtual memory system should be fairly straightforward. It requires further analysis of SUMC to evaluate the cost of this feature.

(c) To implement direct interfaces to MPOS for HAL real time statements and process management would probably most effectively be implemented by a new compiler (HAL/S-SUMC). Significant changes appear to be necessary to the HAL/S-360 compiler/system to accomplish this task as discussed below.

2.4 HAL/S Interfacing with MPOS

It is possible to utilize some of the defined MPOS functions directly for implementing HAL/S real time. However, a thorough design analysis is necessary to determine all "mapping functions" for HAL/S compiler code generation for each HAL/S real time statement and the appropriate restrictions either in MPOS or in HAL/S.

There are several ways to approach this effort, all of which will depend on the intended use of HAL/S on SUMC; (i.e., flight software/development/interactive, etc.), the status of MPOS relative to the compiler, and the cost. One approach is

a new compiler design which would assume that MPOS functions and features are fixed and that HAL/S be implemented in this framework. The appropriate tradeoff in the complexity required to achieve HAL/S requirements within MPOS versus its cost and need should be made. HAL/S capabilities could be limited where appropriate to MSFC functions. It is fundamentally in the HAL real time and other areas where any problem would exist. This approach could be as costly as any new compiler - and could be a patch-work of code caused by the work around techniques required to implement HAL/S. Of concern specifically, is that HAL/S program and task blocks are required to co-exist in the SUMC environment with PL/1 and FORTRAN jobs as individual processes and not as a HAL/S system.

One wonders if the processing requirements imposed by all these languages can be simultaneously satisfied by a universal operating system both efficiently and correctly. If the particular SUMC system selected for HAL/S were flight oriented and contained all HAL/S programs, it would be more likely accomplished.

A second approach is to design a HAL/S-SUMC compiler assuming the MPOS is not fixed. Accordingly, the design effort would entail negotiation of HAL/S interfaces to the new compiler in order to maximize efficient execution and minimize contractor interference during operating system and compiler development. Such an effort will occur in the development of the HAL/S-Flight Computer code generation. It presumably will result in a well designed "matched" interface.

Another possible alternative is to presume that both HAL/S-360 and MPOS are fixed but to modify and re-build some of the compiler, particularly the supplied run time package. It could perhaps be done in a language like PL/1,attempting to maximize and exploit those PL/1/MPOS interfaces which would need to be resolved anyway. A more detailed analysis is required to examine the mapping of HAL/S constructs for processes and process management into the PL/1 constructs for the SUMC version of PL/1. Additionally, the HAL/S-360 compiled BAL code would need to be sub-structured such that it could be interfaced with these PL/1 drivers. This is a very preliminary suggestion but may be worth further investigation.

3.0 DESIGN APPROACH FOR HAL/S RUN TIME ENVIRONMENT ON THE 360

3.1 Background/Rationale

The HAL/S language is designed with a collection of problem oriented, machine independent, source language statements. These statements are fed into a HAL/S compiler whose function is to map the statements into machine instructions and output "blocks" of code capable of being executed on a particular "target computer system". Its source language contains real time and multi-tasking control statements to facilitate aero/manned space programming requirements. These real time statements were designed and incorporated into the language syntax specifically to express application programmer intent in a machine/system independent way. The statements are primarily defined for mechanization and use within a real time, flight computer type environment with a similar type operating system, and not a non-real time batch environment where the statements are normally not required.

The initial HAL/S compiler has been specified for implementation on the IBM-360 - a large non-real time ground computation facility. The HAL/S-360 code generator is expected to be used as a software development/analysis tool and will execute Shuttle flight software in the 360 environment primarily in a functional simulation mode. A HAL/S flight computer code generator will compile the "real" flight code which will be used to execute in actual flight hardware, and in an interpretive computer simulation (ICS) mode on the 360.

Since the characteristics of the operating system and the processing environment of both systems (Flight Computer/360) are different, the compiler mechanization of the HAL real time, I/O, and error control statements are different.

However, in both cases the HAL/S real time compiler mechanization generates code, interacts with the target machine operating system and implements the specified intent of each HAL statement (i.e., SCHEDULE A in 10 causes program A to be scheduled for execution in 10 machine units of time).

The flight computer is a HAL/S only, real time, constrained software environment. The flight computer compiler mechanization will necessitate more direct interaction with the flight operating system to insure all HAL generated code meets real time demands and operates within memory management and time management schemes designed for the Shuttle Software system. The HAL/S-360 System involves implementation within a multi-job/language environment and a general purpose non-real time operating system. The result is that the 360 HAL compiler mechanization has been designed to function as a "stand alone system" which can co-exist with other jobs on the 360 and interacts with the OS-360 minimally - only at the HAL system level. It also contains diagnostics and tracing capabilities which are useful in this environment but not required as part of a flight computer code generator.

3.2 Design Approach

Accordingly, the design approach for the 360 HAL/S compiler implementation of real time statements encompasses the following:

(a) Run time subroutines are written in BAL and supplied with the compiler, to implement each HAL/S real time statement (i.e., SCHEDULE, CANCEL, WAIT, SET, RESET, SIGNAL, etc.). These routines are specified in the HAL/S-360 compiler specification, Sect. 6.8. These routines in effect control all HAL process management functions and interface directly with compiler generated code for each statement.

(b) All process queues, services, and process swapping control is maintained under the control of HAL/S routines but not using OS-360 multiprogramming facilities. The HAL/S system is viewed by OS as a single task and provides services and controls to it.

(c) A simulated clock is maintained by a HAL/S time handler routine which is entered at the completion of each HAL statement. It decrements a timer by the amount of machine units the statement consumes.

(d) Since a flight computer memory management scheme is generally a static one, the HAL/S-360 implementation presumes code and data are available at execution.

(e) HAL I/O statements are implemented using a limited set of OS routines and 360 type peripherals/ channels.

(f) HAL error control statements ON & SEND are implemented by HAL run time routines. OS-360 is utilized only to trap some 360 error conditions. Process reactivation or termination is accomplished via HAL run time software.

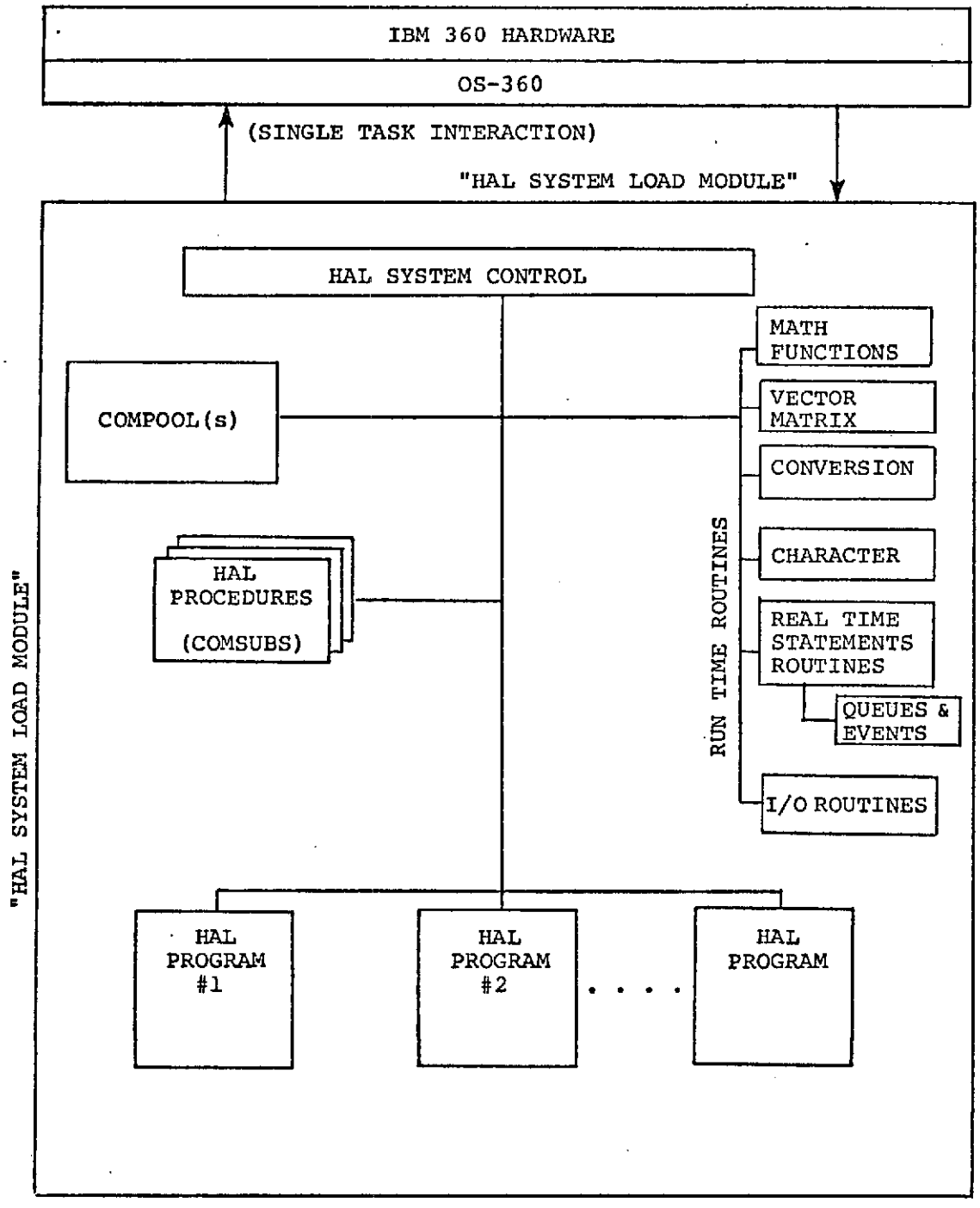
A general overview of the static organization of HAL/S on the 360 is illustrated in Figure 3-1. The HAL/S run time system for the IBM 360 is operated as a single task under OS-360 control. HAL/S source statements are compiled, the separately compilable units linked together into a single HAL/S system load module and executed as a single job step task. HAL/S "compilable units" are described in Section 4.

A "HAL/S system load module" is created by the linkage editor. Memory requirements for the load module are determined by the linkage editor as described in Section 4.1.

The HAL system load module consists of the code and data blocks for each compilable unit as output by the HAL compiler, together with a collection of HAL run time routines automatically brought in by the linkage editor. These run time routines consist of math routines, I/O routines, conversion routines, built-in functions, and routines to implement the HAL real time statements defined in Section 6 of the HAL compiler functional specification. On the 360 this is termed the "HAL run time executive" or "process manager". The functions and logic of these HAL run time routines (i.e., process management) is described in Sections 5 and 6 of this report.

Figure 3-1

HAL SYSTEM ORGANIZATION FOR THE IBM 360



O.S. FUNCTIONS

- HAL/S SYSTEM LOAD MODULE
  EXECUTION CONTROL (NO MULTI-PROG)
- I/O SERVICES
- TRAP FIELDING

HAL FUNCTIONS

- ALL HAL PROCESS MANAGEMENT (i.e., TASKING)
- HAL EVENTS/SERVICES
- HAL TIME/SERVICES
- HAL ERROR CONTROL
- HAL I/O

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

4.0 BASIC CONCEPTS AND FUNDAMENTALS FOR HAL/S REAL TIME

Prior to describing the logical functions performed to implement HAL real time statements, it is necessary to define basic terms and concepts pertinent to HAL/S. The following topics are presented in this section: HAL/S Compilable Units, HAL/S Dynamic Units (i.e., processes), Process State Transitions, Process Control, HAL/S Memory Organization and HAL/S Data Structures and Queues.

4.1 HAL/S Compilable Units

There are three types of compilable units defined in the HAL/S Language specification: (1) a program block, (2) external HAL procedure block (COMSUB(s)), or (3) common data COMPOOL block(s). Each of these is a collection of HAL/S source language statements which can be compiled separately to facilitate multiple programmer groups working on the development of a common system. In order to run, all the HAL/S compilable units which are scheduled or called and/or are pertinent to the collection of programs in the system must be linked together. The object code for each compilable unit is linked, and made ready for execution.

4.2 HAL/S Dynamic Units (e.g., a Process)

The term process is used to describe a dynamic entity, or unit of work, to the flight computer operating system. There is a relationship between the static block structure of HAL/S and its dynamic role in that only TASK and PROGRAM code blocks may be processes. A procedure block can not be a process.

A Process Control Block (PCB) defines a process (see Sect. 4.5). A process is associated with a block of executable instructions, its name, and an assigned work area for its data. A program or task becomes a process when it is SCHEDULED and is no longer a process when it completes execution. Process completion occurs when it is TERMINATED, when it executes a CLOSE or RETURN statement with all dependent processes completed, or when the end conditions of a cyclic SCHEDULE statement or CANCEL statement are satisfied.

A process must be in one and only one of three states: Running, Ready, or Waiting.

a) Running - The process is allocated the CPU and is being executed.

b) Ready - The process is ready to run and only requires the CPU. A process must enter the ready state prior to running.

c) Waiting - The process is awaiting a condition. These conditions include:

   1) a time to occur
   2) an event(s) or event expression to occur
   3) the completion of all dependent processes

d) A program or task block which is not a process can be considered as code in an "INACTIVE state".

HAL provides all services for state transition implicitly with the execution of specified HAL/S real time statements.

4.3 HAL/S Process Management & Control

Processing is controlled by the HAL/S Process Manager. It controls the execution of all processes in the process queues by giving control to processes which are ready for execution on the basis of priority. The highest priority ready process is given control.

Processes are scheduled for execution by other processes. They are inserted into the process queues by the execution of a HAL/S SCHEDULE statement. Processes may be scheduled for execution by several options:

a) Scheduled at a particular time

b) Scheduled at a particular event or combination of events

c) Scheduled immediately

The scheduler may also be requested through the options of the HAL/S SCHEDULE statement to continue execution of a process on a time iterative or cyclic basis and/or until a particular event or time condition is met.

A process is allocated the CPU on the basis of priority and remains running until: a) it is completed; b) it voluntarily releases the CPU by entering a wait state; or c) it reaches a breakpoint and a higher priority process is ready to execute.

A breakpoint is a point during execution of a process which is determined to be "convenient" to allow the process to be "swapped" if a higher priority process exists in the queues and is ready to run. It is inserted frequently enough in longer duration processes to allow acceptable timing response for other high priority processes demands.

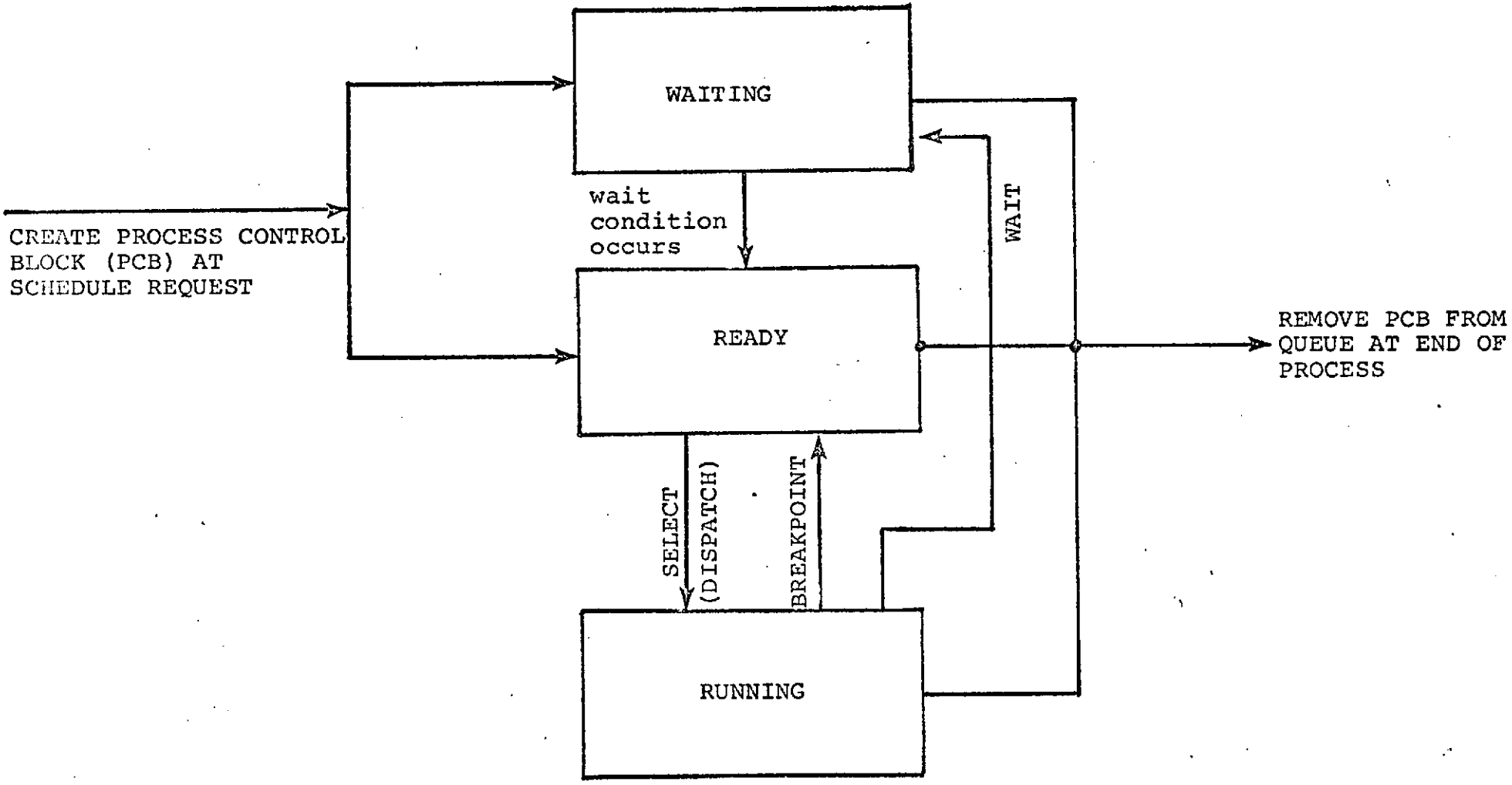
The breakpoint concept and the methods of implementation are discussed in Section 4.7.

4.4 Process State Transition

A simplified version of the transition of process states and their conditions is illustrated in Figure 4-1. Processes are scheduled into either the wait or ready state depending on the conditions supplied in the statement. A waiting process is placed into the ready state only after the condition it was waiting for occurs. Once a process is in the ready state it is allocated the CPU on the basis of priority by a "process selector" function. The selector is entered at the end of a process, at a breakpoint (if a higher priority process exists) or if a process voluntarily removes itself from the running state via a WAIT statement. On a Simplex CPU system, only one process can be in the running state, and it remains running until it ends,or issues a WAIT,or a higher priority ready process exists as determined at its next breakpoint.

Fig. 4-1

Simplified Process State Transition



Strictly speaking, a running process does not have the CPU dedicated to itself at all times. The CPU is automatically allocated to service all interrupts and traps which occur during a running process.

A process may be completed and its PCB removed from the queue from any of these states.

4.5 HAL Process Management Queues & Data Structures

This section presents a discussion of the fundamental queues and tables used by the HAL/S run time routines to implement real time statements. These are:

a) Process Control Block

b) Process Priority Queue

c) Time Queue

d) Event Queue

4.5.1 The Process Control Block (PCB)

A PCB is an element in the process priority queue. It is associated with a single process. It is inserted into the queue when a process enters an active state (i.e., when it is scheduled) and is removed from the queue when the process is terminated.

Each PCB will be fixed in size but the number of PCB's varies. The recommended approach to PCB allocation is to create and initialize the PCB from a fixed region reserved for this purpose.

The information required in a PCB is illustrated in Figure 4-2 and described functionally below.

Fig. 4-2

Process Control Block (PCB) Information

PRIORITY QUEUE LINKAGE
PRIORITY
PROCESS STATE INFORMATION
TASK/PROGRAM FLAG
ENTRY ADDRESS
PROCESS DEPENDENCY LINKAGE (FATHER, SON, BROTHER)
CYCLIC CONTROLS
SAVE AREA
EVENT QUEUE COUNT
TIME QUEUE COUNT

a) Priority Queue Linkage

This field contains a pointer to the next PCB in the priority queue.

b) Priority

Process priority assigned in SCHEDULE statement.

c) Process State Information

This field contains the following information:

- READY/WAIT - Is process ready for execution?
- WAIT ON DEPENDENT PROCESS - Is process waiting for dependents?
- INTER-CYCLE WAIT - Is process cyclic and between cycles?
- INITIATED - Has process begun execution (at least once if cyclic)?
- CANCELLED - Is process to be terminated at the end of its current cycle (if cyclic), i.e., has a CANCEL statement been issued for this process?

d) Task/Program

Is process a task or a program?

e) Entry Address

Pointer to the program entry for this process.

f) Process Dependency Linkage

This field contains:

- Pointer to PCB of father process (a null pointer indicates an independent program process).

- Pointer to PCB of one son process (a null pointer indicates a process with no dependent processes).

- Pointer to next PCB in a chain of "brother" PCB's.

g) Cyclic Controls

This field contains:

- CYCLIC - A flag indicating whether or not the process is cyclic.

- TYPE - This indicates whether the cyclic type is REPEAT AFTER, REPEAT EVERY, or immediate (from SCHEDULE statement).

- VALUE - A scalar indicating inter-cycle wait time if TYPE is AFTER or complete cycle time if TYPE is EVERY.

h) Save Area

This field is for saving key register(s) for the process. Its contents depend on the breakpoint implementation, the machine selection, and compiler conventions. However, its intent is to allow the re-establishment of the process' addressing and machine environment when control returns to the process after a process swap.

i) Event Queue Count

This integer indicates how many Event Queue elements refer to this PCB.

j) Time Queue Count

This integer indicates how many Time Queue elements refer to this PCB.

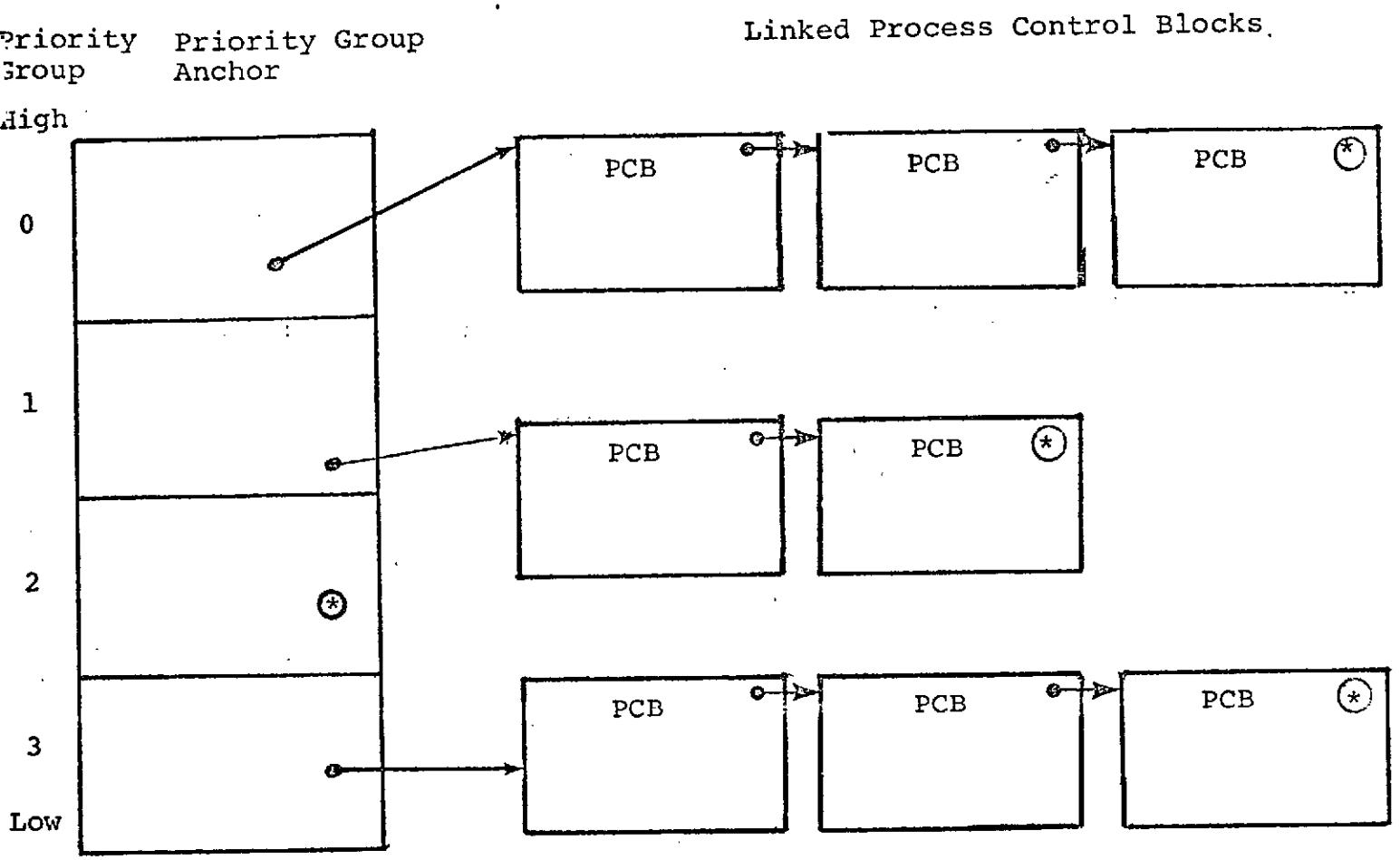
4.5.2 Queues

A queue is a threaded list or chain of elements. The concept involves each element pointing to another element in accordance with some rule (such as next lower priority) with the head of the chain pointed to by an anchor for the queue. An entry is inserted into a queue by searching the queue from the anchor until the element is found which satisfies some condition at which time pointers are exchanged. It is a simple and efficient data structure which is used in most operating systems. Several different queues are required in the Shuttle FCOS for handling processes, events, time, and I/O.

4.5.2.1 Process Priority Queues. Each element in this queue is a PCB which is active (i.e., in either a ready or wait state). PCB's are chained together in this queue by means of the priority queue linkage field of the PCB. Process priorities could be divided into a small number of major priority groups (say, 4), with each group containing a number of sub-priority levels. The Process Priority Queue is actually a collection of four distinct queues, or one for each priority group. The PCB's are ordered within each queue first by sub-priority level, and then within each level on a first come first served basis. The anchor of a given queue points to the first PCB in the queue. This is the PCB for the process which (a) is assigned to the priority group represented by this queue, (b) has a sub-priority level at least as high as every other process in the queue, and (c) was scheduled before any other process in the queue having the same sub-priority level. These queues are illustrated in Figure 4-3. In the HAL/S-360 only a single priority queue is used.

Fig. 4-3

PROCESS PRIORITY QUEUE



⊛ signifies a null pointer

4.5.2.2 Time Queue. Each element in the Time Queue is associated with a single process and represents a desired timer interrupt. A Time Queue element is inserted into the queue by the FCOS Timer Enqueue Routine upon execution of a SCHEDULE statement (AT, IN, or UNTIL options) or a WAIT statement (no option on UNTIL option). The Timer Enqueue Routine is also called by the Process Initiator just before initiating a process (REPEAT EVERY) or at the end of a cycle (REPEAT AFTER). The Time Queue is ordered by absolute time of expiration, with the anchor pointing to the queue element representing the next timer interrupt. The Time Queue is illustrated in Figure 4-4 and consists of the following elements:

Pointer to next ordered Time Queue element

Absolute time of desired interrupt

Pointer to the PCB affected by the interrupt

Type of queue element (this determines what action should be taken upon occurrence of this interrupt)

When the timer interrupt occurs, the process awaiting the top queue element (pointed to by the anchor) is made ready. The top queue element is removed from the queue, and the (programmable) timer is set to the time left until the time indicated by the next queue element.

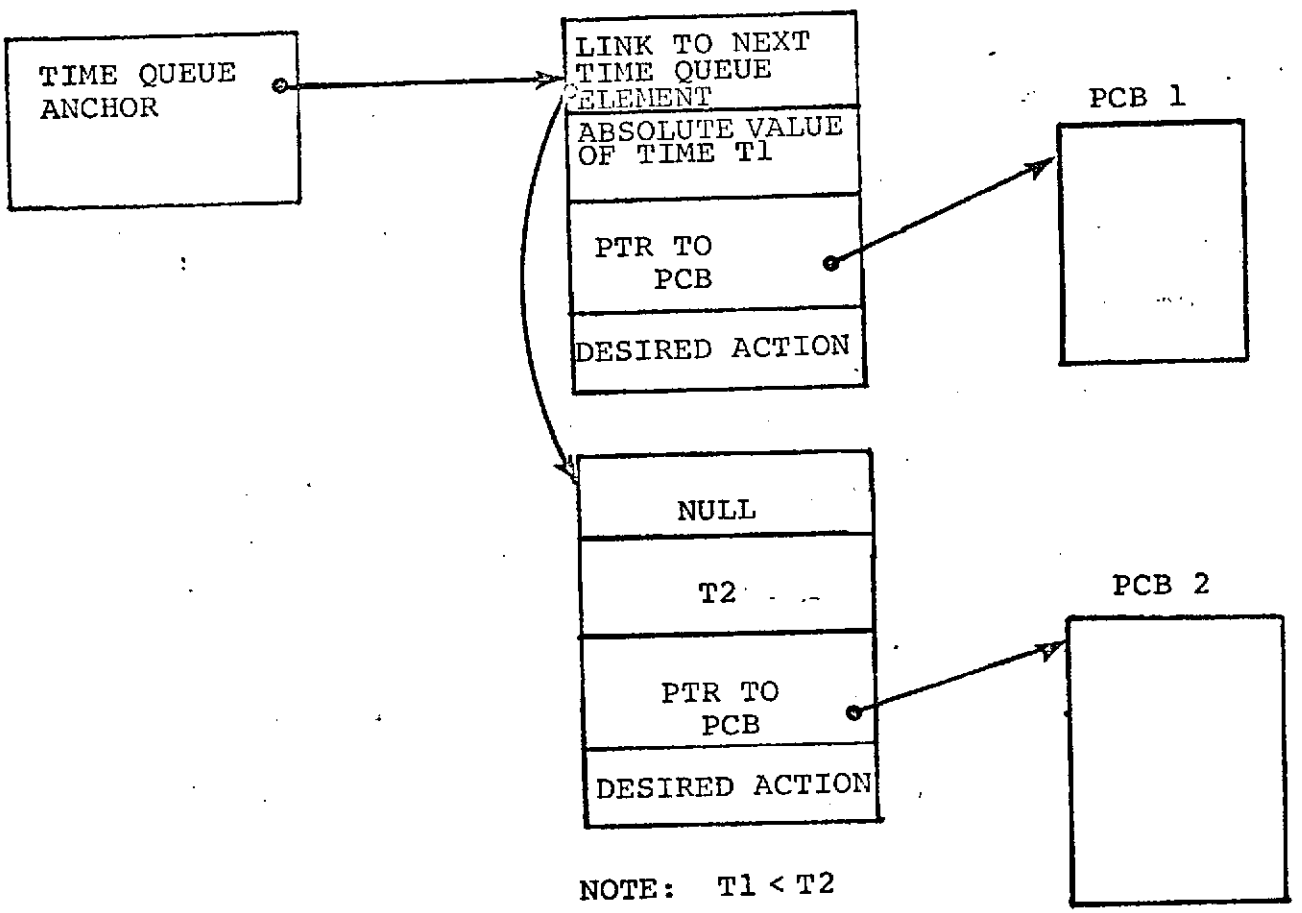
4.5.2.3 Event Queues. Events are declared HAL/S variables which have boolean on/off states. These are both "pulsed" or unlatched or non-pulsed latched events in HAL/S. Events are stimulated by the use of a SIGNAL statement. Event queues tables and expression evaluation are described in the process services section of this report.

4.6 HAL/S Memory Organization

In order to understand the memory scheme proposed, the concept of organizing a program into a code block and a data block is introduced. HAL/S programs generated by the compiler will be organized into a code block which consists of closed sets of sequential machine instructions, and a data block consisting of a continuous block of memory which contains all program declared data variables and temporary work areas required by the compiled routine.

Fig. 4-4

TIME QUEUE



The code block will be generated in a form which is capable of being executed in any part of the memory,assuming appropriate base registers and pointers are preset at entry. It shall require no internal modification to the code block. Also, the code block is constructed such that its data block may be located anywhere in the operating memory. The code produced can be "bound" to its data block by initializing several base registers and pointers to identify the beginning location of the program work area and beginning of its temporary area. The binding of code and data block does not require internal alteration of the code block.

4.6.1 HAL/S Organization of Program Data Memory Block

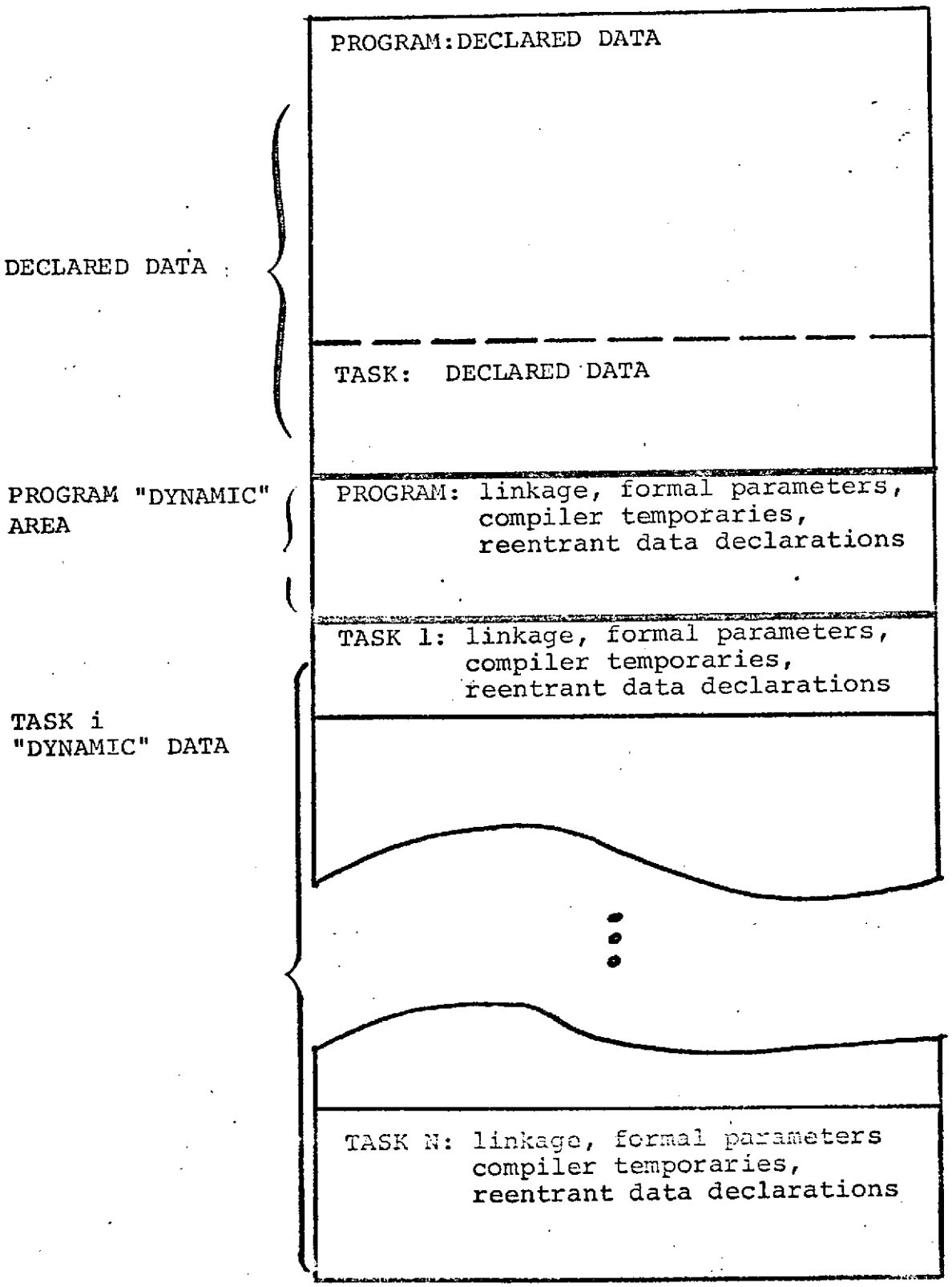
The allocation and management of a HAL/S program's data block area is primarily a compiler function. The compiler determines part of the size of the data block at compile time. A data block contains all internally declared data including data for all tasks and procedure blocks contained in the program. The block also contains a section dynamically controlled by compiler for procedure linkage, parameter passage and compiler temporaries which operate as a stack during program execution. Since a program and all its task blocks are processes, capable of being scheduled, the data block is structured with a static and dynamic area for each process as illustrated in Figure 4-5.

Given this structure of a contiguous data block for each HAL program, it is theoretically only necessary to set a single data pointer to the start of the contiguous data block area for either the initiation of a program or one of its tasks. On the 360, if the data block exceeds 4K more than 1 pointer is required. Also, ideally, the size of the program's data block is determined such that it includes all memory requirements for execution of the program and its tasks. Once a data block is assigned to a program it does not need more memory at any time during execution.

On the 360, however, the size of the dynamic "stack" area for a HAL/S compilable units data block cannot be determined until after all external reference to procedures are resolved during the link phase. Memory requirements must

LOGICAL CONSTITUENCY OF
HAL PROGRAM DATA

Fig. 4-5



be added into the stack for those procedures or routines invoked during execution of the program and the requirements are not known at compilation time. As a result, a "HAL/S link job step" is operated as part of generation of the HAL/S load module. This HAL/S link job step calls the 360 link function, performs a tree search of all calls to routines, looks for error conditions such as recursive calls, etc., and determines the stack size for each program or task block in the load module.

In summary, the memory requirements for each HAL program (& COMSUB) in the HAL system load module is determined prior to execution.

4.6.2 STATIC & AUTOMATIC

The use of the STATIC and AUTOMATIC attributes in a HAL/S data declaration are described in the Language Specification. AUTOMATIC variables are re-initialized at entry to the block, in which they are declared, STATIC are not. This applies to variables declared at the program, task and procedure block levels. Hence, the handling of STATIC and AUTOMATIC program and task blocks in a load module, which are also the HAL/S processes in the load module, are similar. This means that a program which becomes a process, completes execution (i.e., returns to an inactive state) and is subsequently rescheduled, gets assigned the same static storage area. Its automatic storage is, however, reinitialized.

The use of the HAL/S INITIAL attribute corresponds to these rules on the 360. AUTOMATIC variables are initialized with code at each entry to the block, whereas STATIC are not.

4.7 Process Swapping & Breakpoint Concept

The exchange of control between processes (i.e., allocation of the CPU), termed a "swap" can involve overhead. The question of when to swap is related to the arrival of a "higher" priority" process, generally resulting from an interrupt.

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

a) to provide rapid response to asynchronous events; and

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

In single-processor systems, particularly aerospace real-time systems where most or all of the computation is devoted to a single functional application with interrelated processes, the introduction of interrupt-mode computation can raise some standard multiprogramming problems, (i.e., re-entrancy and data sharing conflicts). Thus, methods for masking or inhibiting interrupts were added, and the nature of the functions allowed in the interrupt-mode were restricted.

The negative aspects of interrupts, i.e., timing response uncertainties and potential data/program conflicts can be minimized by causing interrupts to schedule processing whenever possible, as opposed to performing it. This provision reduces the multiplicity of possible timing situations, since process swapping can be made to occur only at specified intervals (i.e., breakpoints).

Accordingly it is considered desirable to utilize hardware interrupts. The primary consideration becomes when to swap a higher priority process resulting from an interrupt, such that response time requirements can be satisfied and conflicts avoided.

It is assumed that there will and should be timing requirements but with some tolerance. Because of this tolerance, it is possible to control the points at which processes may be swapped. As long as the swap point is within some definable limit, it can then be placed appropriately in order to minimize the overhead involved in swapping process control. The overhead saved includes for example: protection from conflict of resources (e.g., memory locks, exclusive procedures), and saving and restoration of resources (e.g., register sets, fast memory).

The effect is that the compiled code can then make full use of a flight computer's resources without having to guard against a potential process swap (except possibly the foreground). The implementation of this compiler controlled feature would be accomplished by a means of compiler "insertion" of a breakpoint at the appropriate swap points within the process. The

breakpoint logic would test to see if there is the need of a process swap. This code is itself overhead. However, this overhead is but a small percentage of the process execution time and memory space as compared with the overhead involved with the implementation of controlled protection features.

4.7.1 Breakpoints

There are several considerations and issues concerning the implementation of breakpoints as a means of controlling process swapping. Some of these are:

a) Why are breakpoints used?

b) How are breakpoints mechanized?

   1) in line test in software
   2) call
   3) via hardware or hardware support

c) Where are breakpoints inserted?

   1) time intervals
   2) statement levels

d) How is breakpoint implementation overhead minimized?

The major recommendations for the flight computer implementation of HAL are as follows:

a) Processes swapped at breakpoints inserted into the code by the compiler.

b) Breakpoints shall be inserted at sufficient frequency to satisfy response requirements.

c) The length of time between breakpoints shall be constrained.

d) Implementation and mechanization techniques of breakpoints is dependent on computer hardware features and shall be determined during the detailed design of compiler.

e) The compiler will implement all breakpoints. In the HAL/S-360 a breakpoint will occur at each statement.

f) The preference for mechanizing breakpoints is an in-line test inserted by compiler to determine if higher priority jobs exist.

g) Breakpoints are inserted by the compiler at points in program code commensurate with an estimated period of execution time.

h) Breakpoints shall not be inserted in update blocks to prevent sharing conflicts.

i) Compiler inserted breakpoints shall be visible in the listing.

4.7.2 Summary

In the 360 version of HAL, breakpoints will be implemented at each HAL statement. That is, the HAL/S-360 run time routines will look for a higher priority process ready for execution at the end of execution of each statement, and if one exists a process swap will occur. The breakpoint policy and implementation scheme for the flight computer has not been determined as yet.

5.0 MECHANIZATION AND STRUCTURE OF HAL/S-360 REAL TIME

The purpose of this section is to describe the overall structure and control of the HAL/S-360 run time system. Figure 5-1 illustrates the organization of the system. There are basically four major sections:

1) A HAL/S Start Routine which gains control from OS-360 and initializes the HAL/S run

2) A HAL/S Process Manager which performs the selection (dispatching) and initiation of all HAL/S processes in the process queues. It is the central control element

3) A HAL/S statement processor which is invoked after execution of each HAL/S statement. It performs a series of functions at each statement such as: updating simulated clocks, checks for higher priority processes , determines when a process swap is required and performs tracing and diagnostics when required.

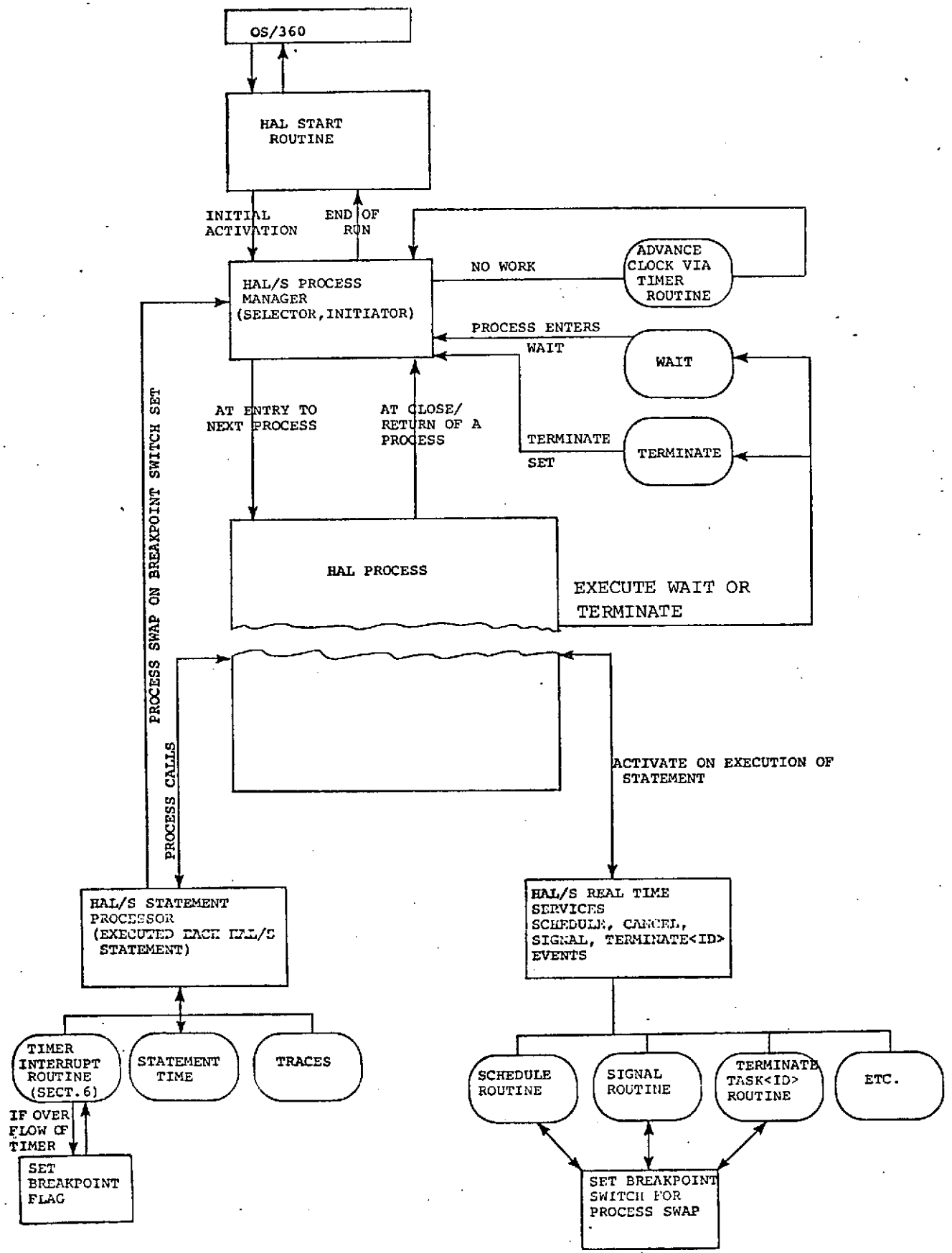
4) A set of HAL/S process management service routines which are called by the process on the execution of a SET, RESET, SCHEDULE, CANCEL TERMINATE<ID>, SIGNAL event statements. The logic and functions performed by these routines are described in Section 6 of this report.

As an overview, a process is given control by the process manager when it is the highest priority ready process. During execution it calls the HAL/S statement processor after each statement. It keeps track of time and diagnostic requests. A process may schedule, cancel, or terminate other processes during execution. This is done by the compiler inserting code to call the appropriate HAL/S process service routine.

It is important to note that process service routines such as the scheduler do not result in a process swap. These routines set a "breakpoint switch" when the action taken results in a higher priority ready process than the process currently running. The breakpoint switch is independently tested (currently by the statement processor each statement).

Fig. 5-1

OVERVIEW OF CONTROL AND DYNAMIC STRUCTURE
HAL/S-360 REAL TIME



When a process executes a wait or terminate (self) statement it results in a process swap and the appropriate action is taken for updating the PCB entry.

A process continues to run until it either ends normally or executes a CLOSE or RETURN statement. At this point, the process manager selects the next ready process.

The process manager completes the run when all queues are empty. If an abnormal error condition occurs, it causes the run to be aborted.

These run time routines have been implemented in BAL on the 360 and are estimated at a total size of 3-4K bytes of code.

5.1 HAL/S Start Routine

The HAL/S system load module is given control by the operating system with an ATTACH macro (it may be also CALLed). Once the "HAL load module" gets control from OS, the HAL start routine performs various initialization functions. It prints out a HAL/S header, and sets up run time parameters input through JCL PARM field such as lines/page, channel # for system messages, # of errors before abort, debugging options etc. It also issues SPIE and STAE macros to trap program interrupts and abnormal abort (ABEND) conditions.

The SPIE macro specifies an exit routine address which is used in the HAL system to signal the appropriate HAL error conditions for recoverable errors, performs fix up if required and continues execution. The STAE macro is used to specify an exit routine address which prints as much HAL unique diagnostic information before OS-360 terminates the run.

HAL start must initiate the run. It does this by scheduling the "initial HAL process" to establish the first entry in the queues. Although this function can be expanded it currently schedules the first HAL program in the load. HAL/S start then calls the Process Manager.

5.2 HAL/S-360 Process Manager

The Process Manager is the function which controls the state of execution of all processes in the priority queue. It consists of a process selector which chooses a process ready for execution, and a process initiator which controls the starting, cycling, and normal end of process execution. The scheduler and terminator which create and remove processes from the system are part of the application process control services and are described in Section 6.

5.2.1 The Process Selector (Dispatcher)

The process selector chooses a process, then gives it control, so that it may proceed with execution. The choice is limited to those processes in the ready state. If there are no ready processes, the system would normally (in a flight computer environment) enter an idle state, and would remain idle until a process is brought to a ready state - normally through the occurrence of a time or event interrupt. In the HAL/S-360, however, the system is advanced through this time interval by decrementing the simulated clock to zero - forcing an interrupt. This should cause a process to enter a ready state and if not, the HAL/S-360 run is ended.

In general, there may be more than one ready process, so the choice is based on priority; i.e., the relative importance of the various ready processes. Any number of schemes may be used to represent this ordering, but the most effective and efficient way is the priority queue.

From the point of view of the Process Selector, only ready processes need be part of the queue. In such a case however, the enqueuing and dequeuing of PCB's would be necessary every time a process changed state (which occurs more frequently than process selection). On the other hand, if all scheduled processes (running, ready, and waiting) are on the queue, the priority queue need be altered only when scheduling or terminating a process. This latter design is the one chosen for the 360. The process selector must skip over waiting processes in the queue to find the first ready process, but this is still less time consuming than altering the queue frequently.

After the selector picks a process, it either uses the resume information (save area) in the PCB to restart the process at its suspended or swapped point, or it initiates the process at its beginning if it has not yet executed.

Figure 5-2 indicates that the selector starts at the top of the queue when looking for the first ready process. If the selector was entered because a process entered the wait state, search time is considerably reduced if the selector first checks the breakpoint flag. If it is not set, the search may start with the next process on the queue instead of at the top. The breakpoint flag is set whenever a process having a higher priority than the running process is readied (i.e., whenever a process swap should occur at the next breakpoint).

5.2.2 Process Initiator (Fig. 5-3)

The process initiator is a routine which gets control from the process selector the first time a process starts executing. The program or task which was scheduled as a process is called as a subroutine of the process initiator. When the program or task executes a RETURN or CLOSE at its highest level, control comes back to the process initiator, which performs the following functions:

1) Causes the process to wait until all dependents have terminated

2) If the process is not cyclic or is a cancelled cyclic process, it is terminated by calling the terminate subroutine, and control is passed to the process selector

3) If another cycle of a cyclic process is indicated, the program or task is called again, after possibly placing the process into an inter-cycle wait state (EVERY or AFTER, options from SCHEDULE statement). If the cycle type is AFTER, the timer enqueue routine is called to start the AFTER interval. The EVERY interval is set up once when the process initiator is entered, and is automatically repeated by the timer interrupt routine.

Fig. 5-2

PROCESS SELECTOR

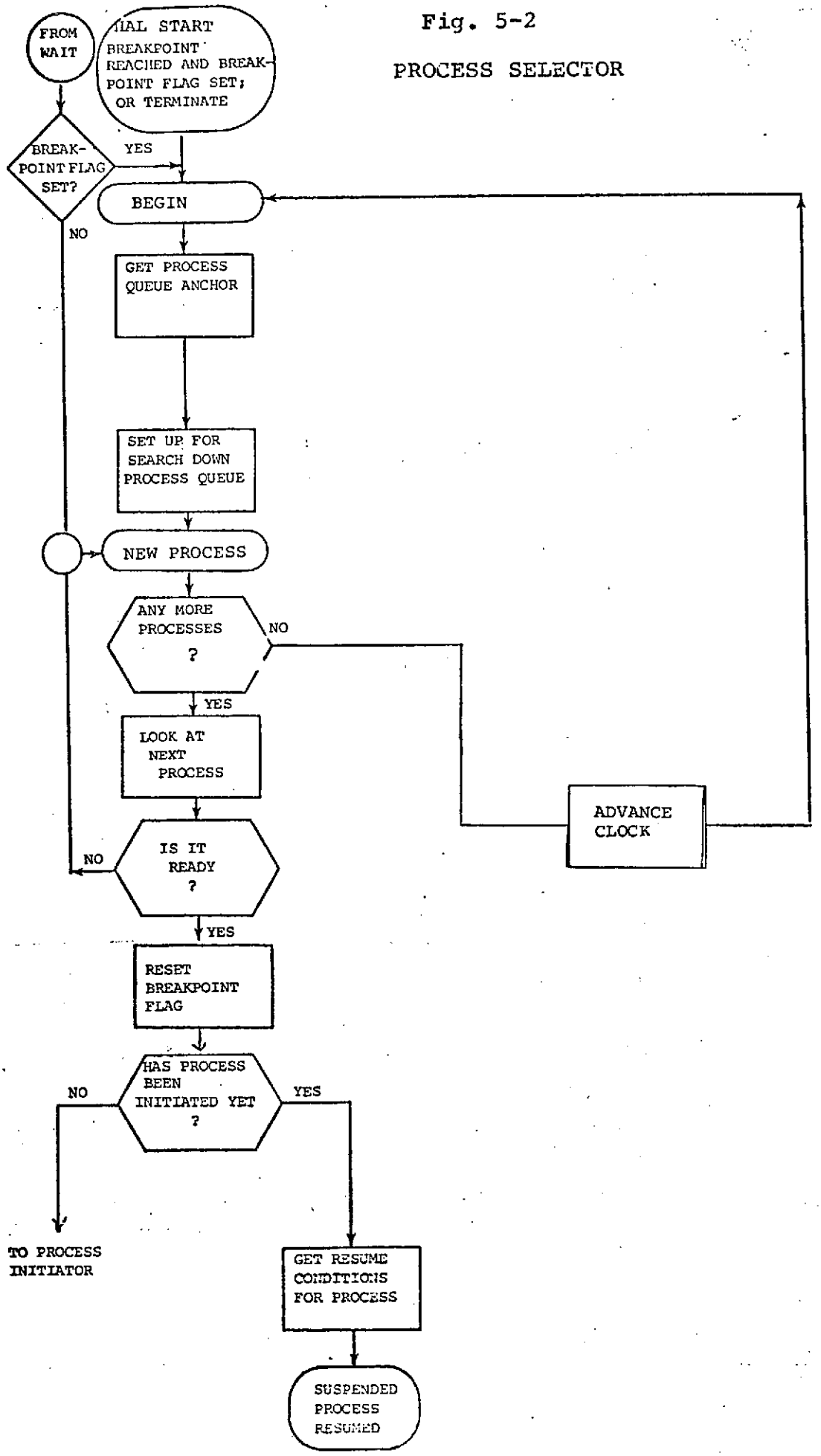
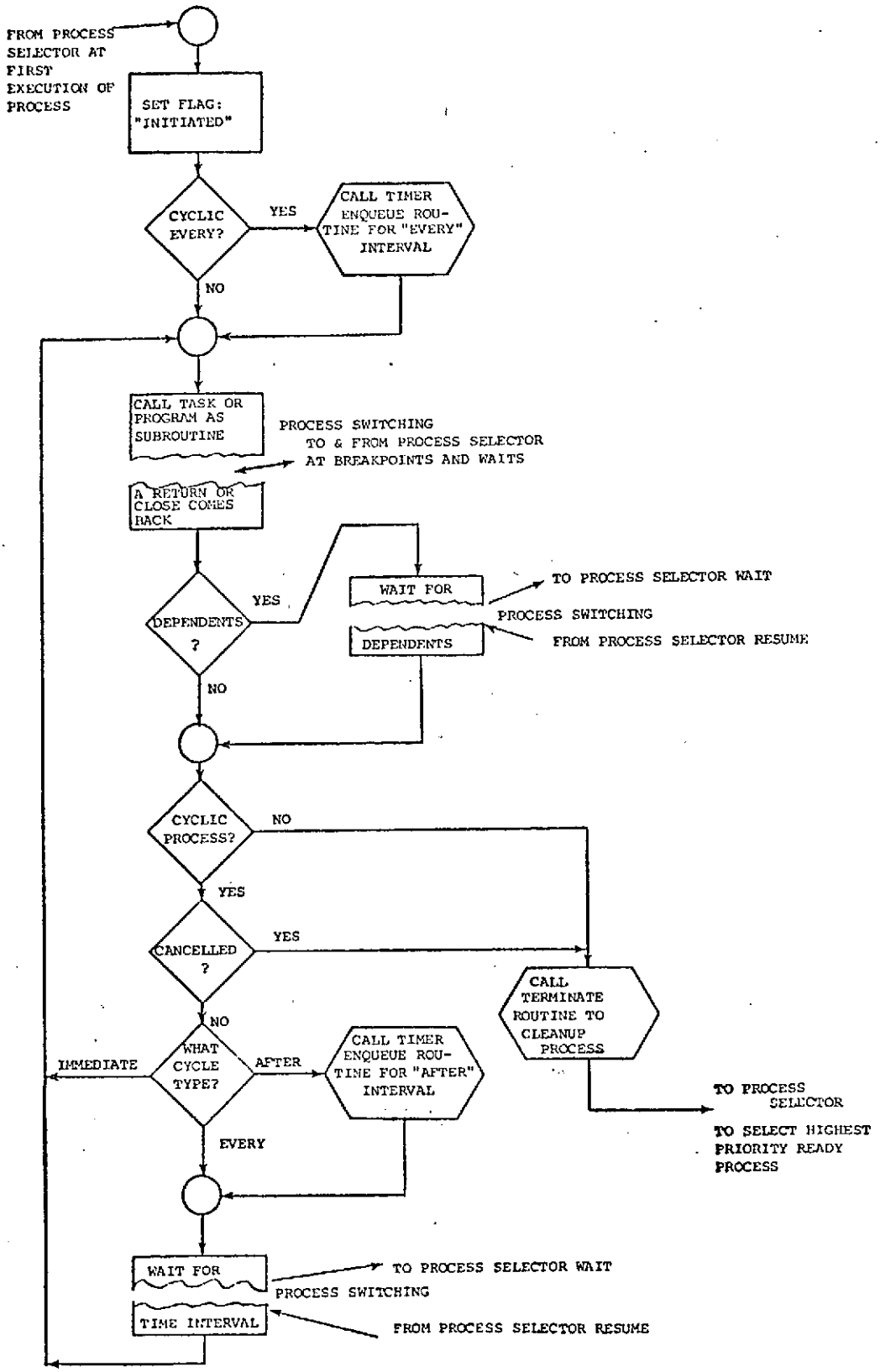


Fig. 5-3
Process Initiator

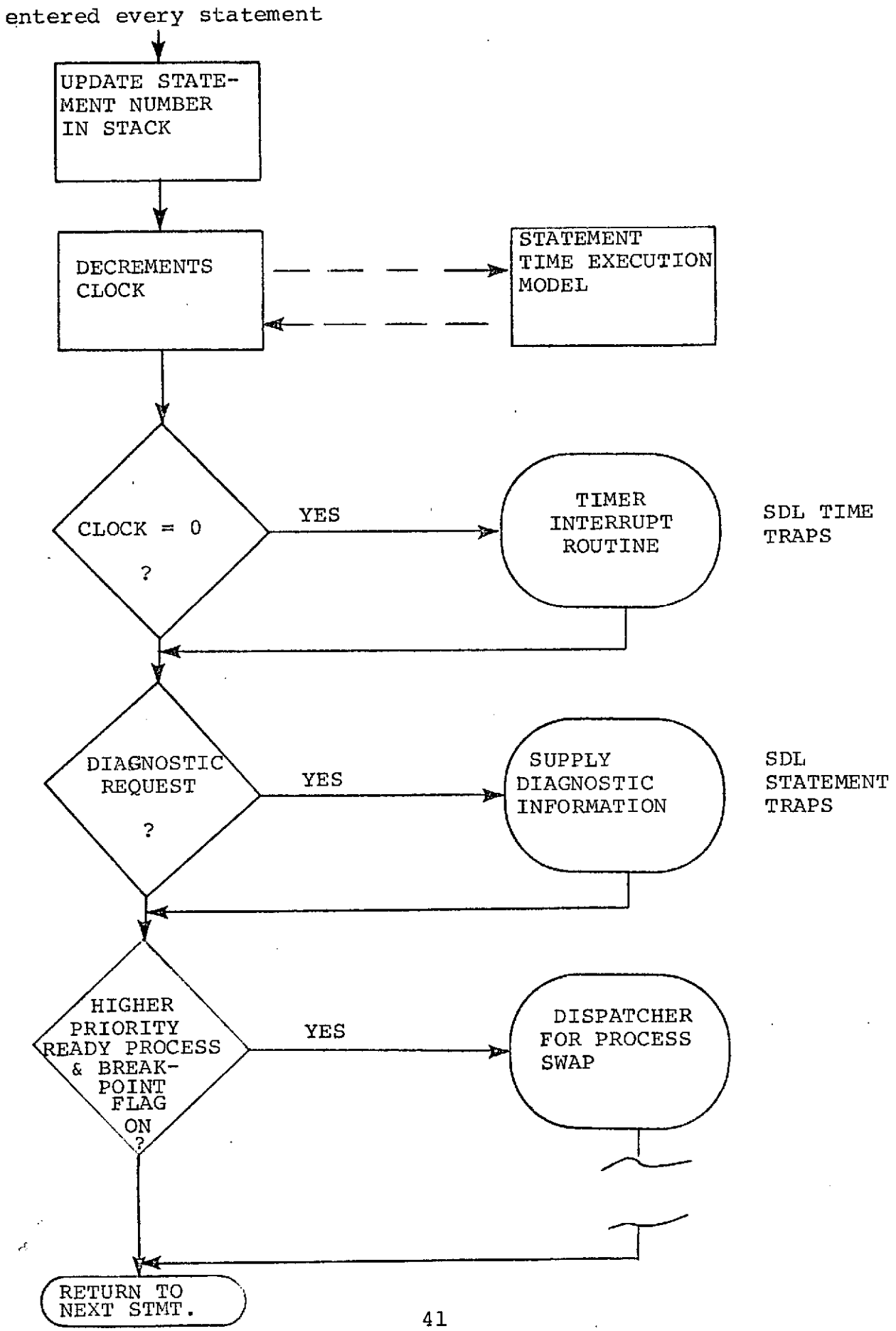


5.3 HAL/S Statement Processor

The HAL statement processor logic is executed at the end of each HAL/S statement. Its logic is illustrated in Figure 5-4.

Fig. 5-4

HAL/S-360 Statement Processor



6.0 FUNCTIONAL DESCRIPTION OF HAL/S-360 REAL TIME STATEMENT SERVICE ROUTINES

This section presents a description of the structure and mechanization of all process control services. It includes the HAL/S real time statements of:

SCHEDULE

CANCEL

TERMINATE <ID>

SET, RESET, SIGNAL <event>

WAIT <event>

Time and Time Management

The description of the statements does not include changes to HAL/S which may have occurred subsequent to the April HAL/S specification. It also does not include the UPDATE PRIORITY statement.

6.1 The Process Scheduler

The Process Scheduler is the process service routine which gets control when a HAL SCHEDULE statement is executed. It creates a process by putting a new Process Control Block (PCB) containing the proper information on the priority queue. When the scheduler returns to its caller, the new process is either in the ready state or in the wait state (if the AT, IN, or ON option was specified). It is then the dispatcher's (i.e., process selector's) responsibility to give it control at a process swap point.

The options to the SCHEDULE statement are handled by separately testing for the occurrence of each one. If an option is specified, the appropriate processing is performed. Sometimes this is accomplished by a call to a system routine such as the event enqueue routine to set up an event expression, or to the timer enqueue routine to enter an interval in the timer queue. A parameter is passed to these routines specifying what action to perform (ready or cancel the process) when the requested condition (time interval expires or event expression becomes true) occurs. The Event Processor is called if a process event was declared for the program or task.

Other SCHEDULE option processing is done local to the scheduler. A specified priority is assigned by setting the priority field in the PCB (used to determine the position on the priority queue). If the option DEPENDENT was specified, the scheduler places the new PCB on the dependence queue of the running process.

Parameters to the scheduler routine are listed below:

A) OPTIONS:

DEPENDENT

initial conditions (none, IN, AT, ON)

PRIORITY

REPEAT options (none, EVERY, AFTER, REPEAT with no delay)

cancel condition (none, UNTIL <event exp>, UNTIL<time>, WHILE<event exp>)

B) LABEL or RUN-TIME REFERENCE - program or task i.d. used in the Program Directory Index.

C) TASK/PROGRAM - is process a task or a program?

D) PROCESS EVENT - (optional) pointer to process event, if declared

E) WAIT TIME - (optional) time specified in AT or IN phrase

F) CANCEL TIME - (optional) time specified in EVERY or AFTER phrase

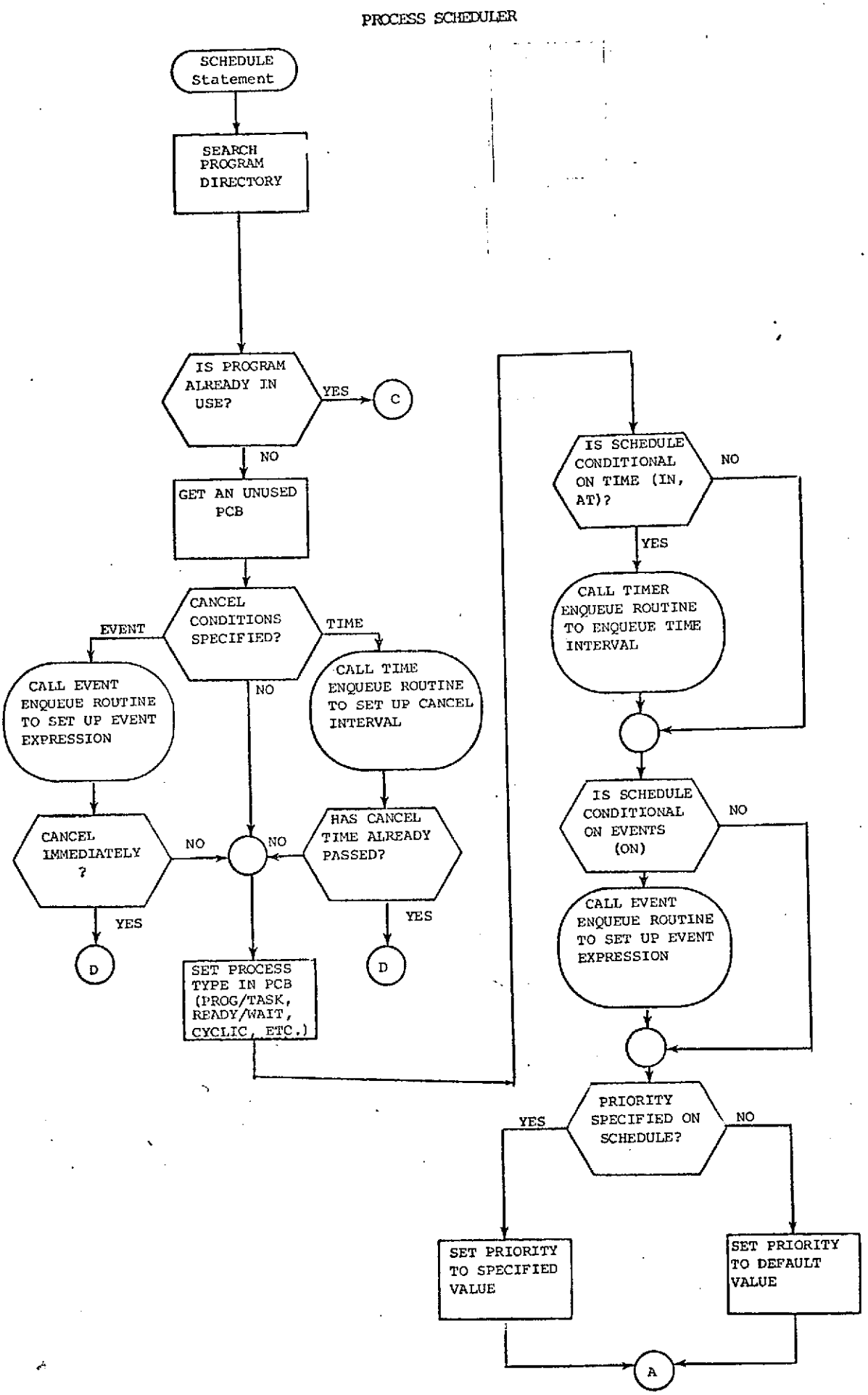
G) PERIOD - (optional) time specified in EVERY or AFTER phrase

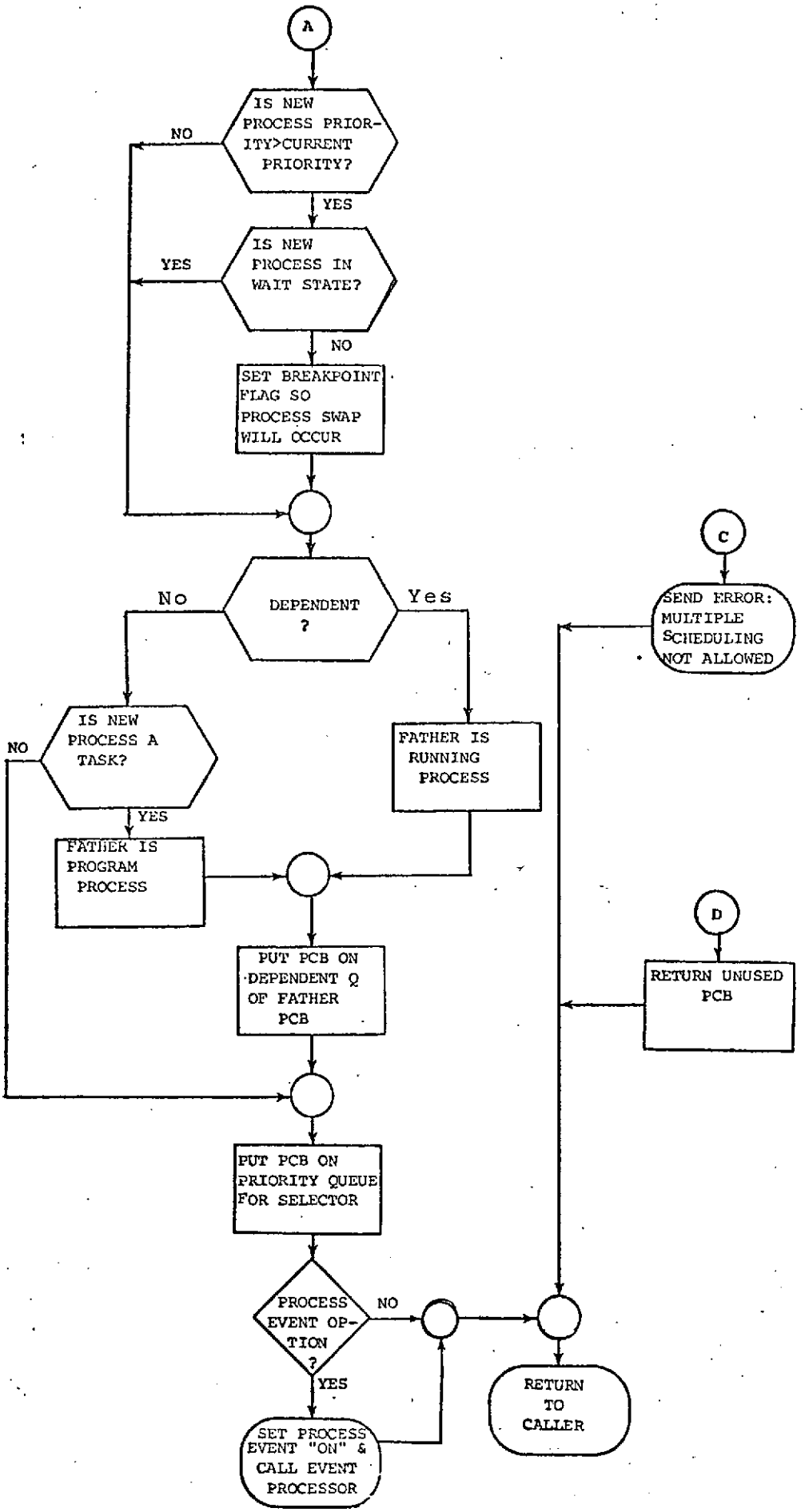
H) WAIT EVENT EXPRESSION - (optional) pointer to event expression structure used in ON phrase

I) CANCEL EVENT EXPRESSION - (optional) pointer to event expression structure used in UNTIL or WHILE phrase

Functional flow of the scheduler is illustrated in Figure 6-1.

Figure 6-1


PROCESS SCHEDULER
SCHEDULE Statement
SEARCH PROGRAM DIRECTORY
IS PROGRAM ALREADY IN USE?
YES
C
NO
GET AN UNUSED PCB
CANCEL CONDITIONS SPECIFIED?
EVENT
TIME
NO
CALL EVENT ENQUEUE ROUTINE TO SET UP EVENT EXPRESSION
CALL TIME ENQUEUE ROUTINE TO SET UP CANCEL INTERVAL
CANCEL IMMEDIATELY ?
NO
NO
HAS CANCEL TIME ALREADY PASSED?
YES
YES
D
D
SET PROCESS TYPE IN PCB (PROG/TASK, READY/WAIT, CYCLIC, ETC.)
IS SCHEDULE CONDITIONAL ON TIME (IN, AT)?
NO
YES
CALL TIMER ENQUEUE ROUTINE TO ENQUEUE TIME INTERVAL
IS SCHEDULE CONDITIONAL ON EVENTS (ON)
NO
CALL EVENT ENQUEUE ROUTINE TO SET UP EVENT EXPRESSION
PRIORITY SPECIFIED ON SCHEDULE?
YES
NO
SET PRIORITY TO SPECIFIED VALUE
SET PRIORITY TO DEFAULT VALUE
A


A
IS NEW PROCESS PRIOR-ITY>CURRENT PRIORITY?
NO
YES
IS NEW PROCESS IN WAIT STATE?
YES
NO
SET BREAKPOINT FLAG SO PROCESS SWAP WILL OCCUR
DEPENDENT ?
No
Yes
C
SEND ERROR: MULTIPLE SCHEDULING NOT ALLOWED
IS NEW PROCESS A TASK?
NO
YES
FATHER IS RUNNING PROCESS
FATHER IS PROGRAM PROCESS
D
RETURN UNUSED PCB
PUT PCB ON DEPENDENT Q OF FATHER PCB
PUT PCB ON PRIORITY QUEUE FOR SELECTOR
PROCESS EVENT OP-TION ?
NO
YES
SET PROCESS EVENT "ON" & CALL EVENT PROCESSOR
RETURN TO CALLER

6.2 CANCEL Process Service Routine

The CANCEL statement provides a safe way to terminate a process, avoiding the danger of half-results. If the process has not yet begun execution or is in between cycles of execution, it can be safely terminated by immediately calling the terminate subroutine. In any other state, however, the process is allowed to run to the end of its current cycle. A non-cyclic process in this case would be unaffected. The cancel flag in the PCB is set by the CANCEL routine, and tested by the process initiator before starting another cycle. If it is set, the processor initiator calls the terminate subroutine. See Figure 6-2 for a flowchart of the CANCEL Service Routine.

6.3 TERMINATE

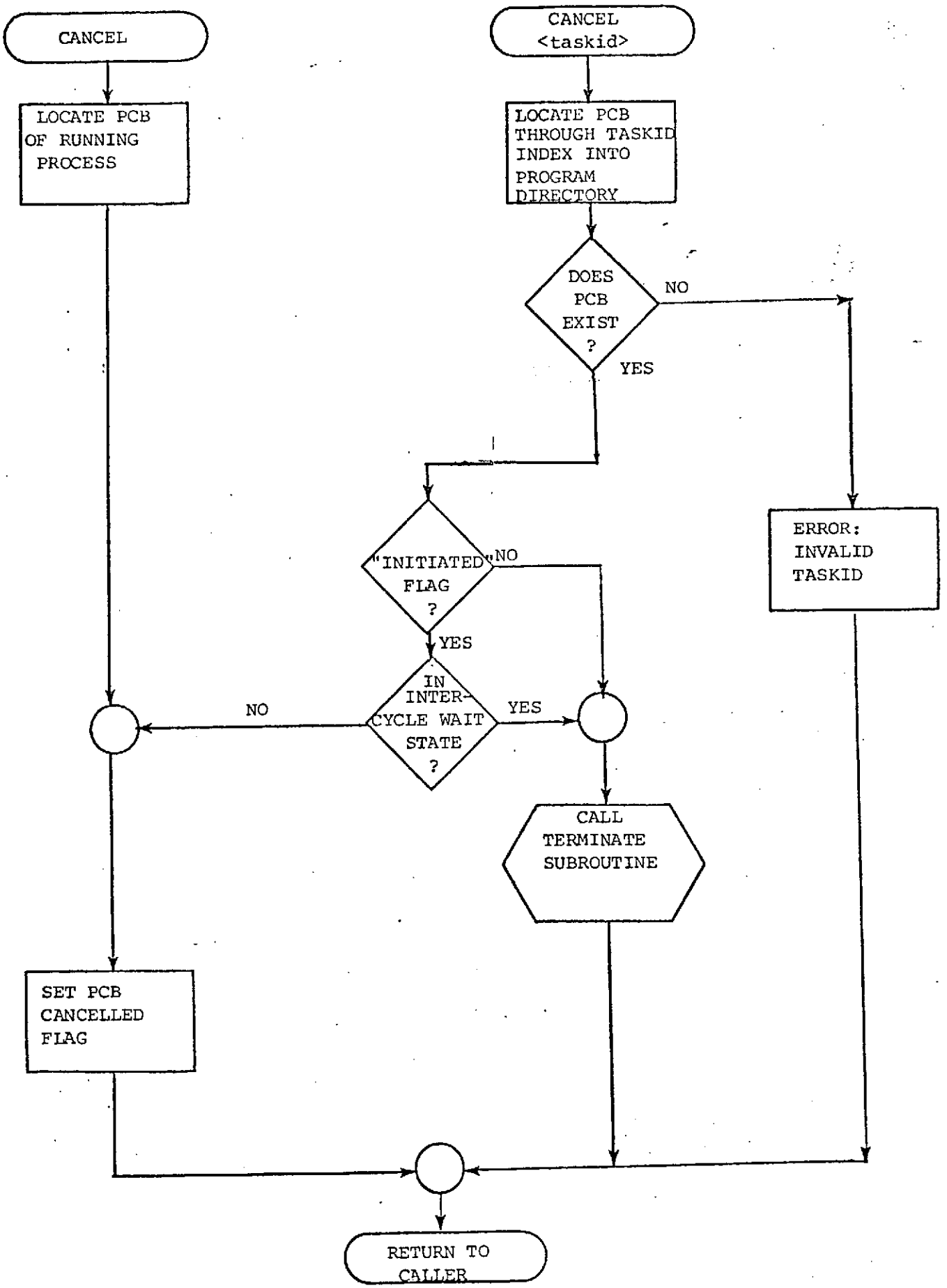
The TERMINATE statement allows for immediate and unconditional termination of a process and all its dependents. Termination involves cleanup of pending conditions (time, event) and allocated resources, and removal of the PCB from the priority queue. Since these actions must be taken for all kinds of termination (TERMINATE, CANCEL, RETURN, CLOSE), a terminate subroutine is used to carry out the cleanup work. The TERMINATE statement service routine merely locates the PCB address, checks if the active process is allowed to terminate the specified process, then calls the terminate subroutine. A flowchart of the TERMINATE Service Routine appears in Figure 6-3.

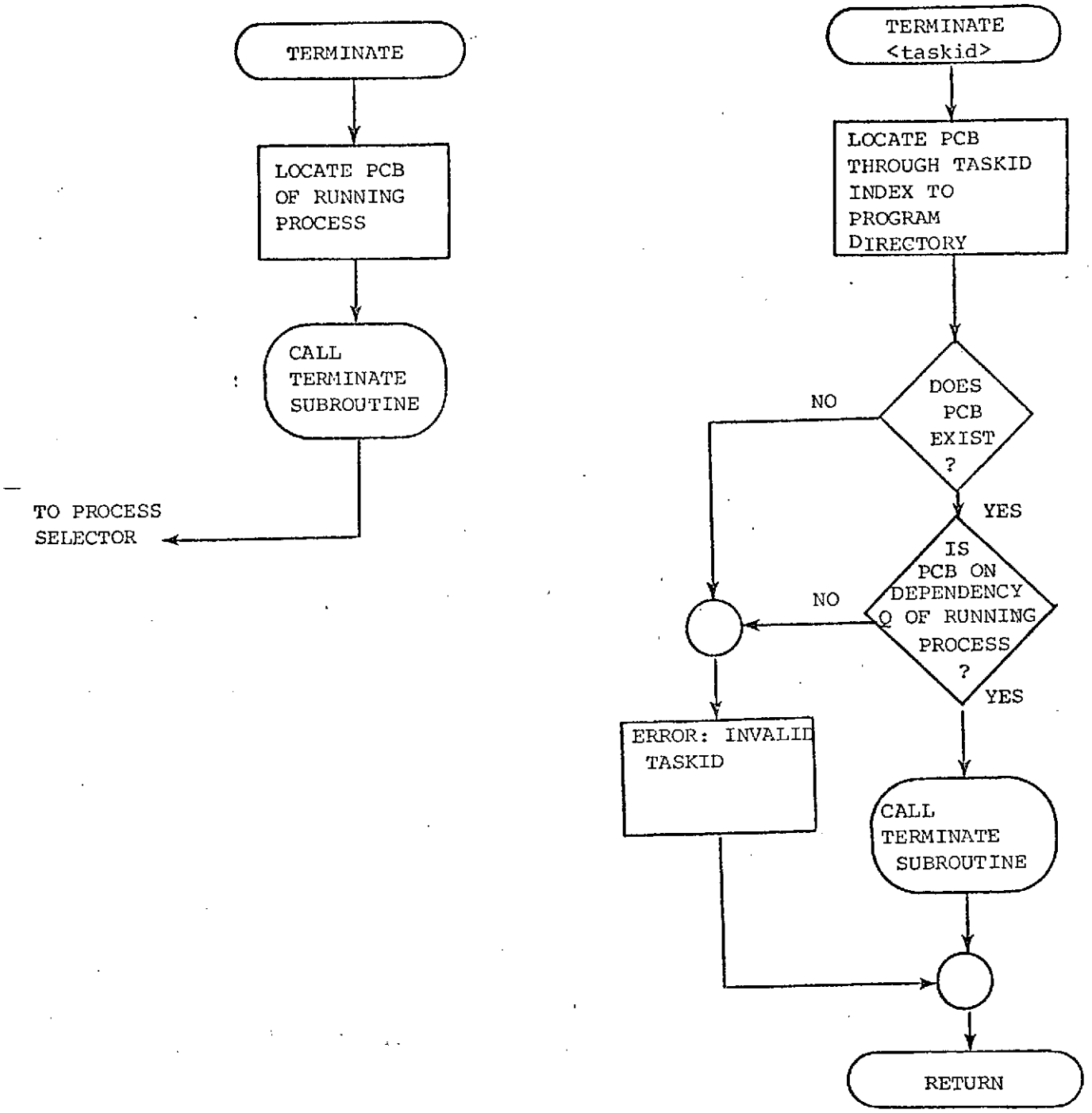
6.3.1 Terminate Subroutine

It is called by the TERMINATE statement service routine, by the process initiator, and by the following routines when a cancel condition occurs and the process can be immediately terminated: CANCEL Statement service routine, event processor, and timer interrupt routine. It performs the following functions on the process to be terminated:

a) cancels its active event expressions (found by searching the event queue until n blocks are found which point to the PCB, where n is the event queue count from the PCB)

Fig. 6-2 CANCEL STATEMENT SERVICE ROUTINE





TERMINATE STATEMENT SERVICE ROUTINE

Fig. 6-3

b) cancels its active timer intervals (found like event expressions - using the timer queue count from the PCB)

c) frees its allocated memory

d) frees EXCLUSIVE code it may have entered

e) frees any other shared resource it may have acquired

f) turns its associated process event off

g) removes and frees its PCB from the priority and dependency queues

h) terminates all its dependents in an identical manner

i) readies the father process if it is waiting for dependents and the terminating process is its last dependent

j) if any process events were turned off in f), the event processor is called

The terminate subroutine may cause other processes to become ready because 1) termination may satisfy the father's dependency wait, 2) turning the process event off may satisfy a WAIT FOR or SCHEDULE ON event expression, 3) freeing a shared resource (e.g., UPDATE lock) may wake up a process waiting for it. In any case, a process is made ready in the PCB, and, if it has a higher priority than the running process, the breakpoint flag is set. The process swap then occurs at the next breakpoint.

In addition to causing a process to be made ready, the terminate subroutine, in turning off the process event, may cause another process to terminate if a cancelling event expression is satisfied. The terminate subroutine and event processor must be coded to avoid recursive calls in such a situation.

6.4 Event Handling

The event handling system of process management carries out the signalling of events and performs specific actions when logical combinations of events, called event expressions, become true. Events are declared HAL language variables which have a boolean true/false or on/off state. These software events may be signalled (caused to change state) by a program statement or by an associated hardware occurrence. If a real time statement with an event expression is executed, the expression is immediately evaluated. If its value is not true, it becomes an "activated" event expression. An "activated" event expression is monitored until it becomes true or until the associated process is terminated. When an event changes state, "activated" event expressions are re-evaluated to determine if they have become true. If they have, the requested action is taken (ready or cancel a process). Thus event expressions have a life time beyond the execution of the containing statement.

The following statements can signal (change the state of) an event:

SET, RESET, SIGNAL - explicitly sets or pulses the state of the event (see Fig. 6-4)

SCHEDULE - implicitly sets the process event state to true, if the program or task was declared with a process event

RETURN, CLOSE, (at program or task level), CANCEL, TERMINATE - implicitly sets the process event state to false, if the program or task was declared with a process event.

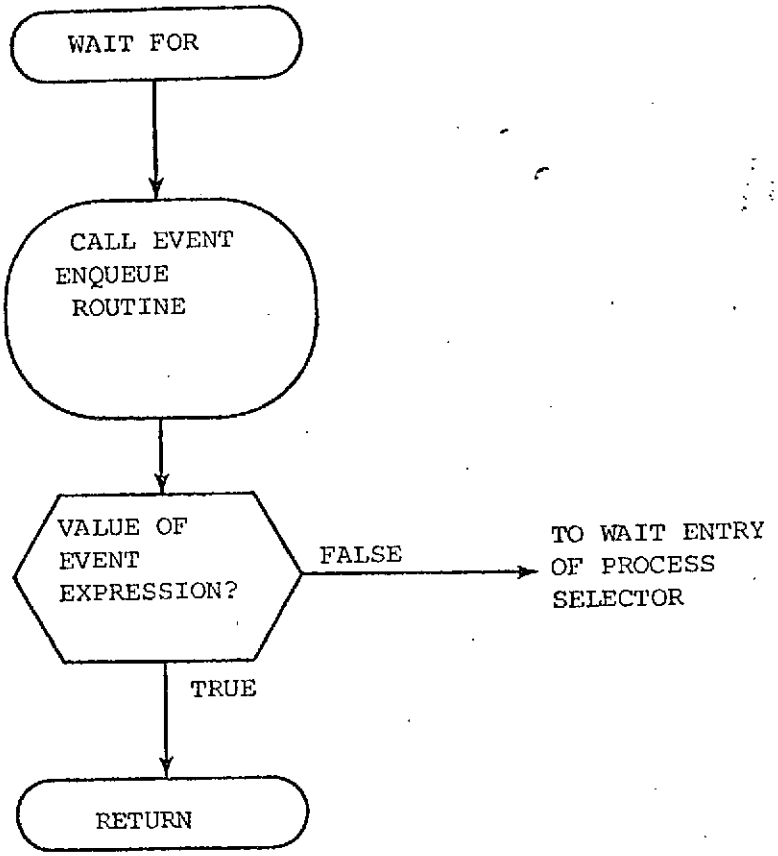
The following statements may explicitly specify an event expression:

WAIT FOR - causes the executing process to wait until the event expression is true (see Fig. 6-5)

SCHEDULE (with ON option) - causes the newly created process to wait until the event expression is true

SET, RESET, SIGNAL PROCESSING

Fig. 6-4



WAIT FOR ROUTINE

Fig. 6-5

SCHEDULE (with the WHILE option) - causes cancellation of the newly created process if the event expression is false (an implicit "NOT" is applied to the event expression)

SCHEDULE (with the UNTIL option) - causes cancellation of the newly created process if the event expression is true, with the stipulation that at least one cycle will be allowed to execute

Note: In addition, event expressions may be used in any context where a boolean or bit expression is allowed. However, in these contexts, the HAL/S does not monitor the event expressions. They are evaluated only once at the time the containing statement is executed, and unlatched events always appear in the false state.

The routines associated with these HAL statements are called by the HAL compiled code and in turn call system event and event expression handling routines. There are four types of event expressions; two specify wait conditions (WAIT FOR, SCHEDULE ON), and two specify cancel conditions (SCHEDULE UNTIL, SCHEDULE WHILE). Since the UNTIL and WHILE phrases are mutually exclusive, the SCHEDULE statement can potentially specify two event expressions. Compiled in-line code cannot always be used to evaluate event expressions since event expressions can remain "activated" asynchronously with respect to execution of compiled code. An event expression must therefore be communicated to the routine through an event expression structure, created by the compiler and passed by a pointer in the parameter list of the WAIT or SCHEDULE routine. See Figure 6-6. The WAIT or SCHEDULE routine then calls the enqueue routine, described below.

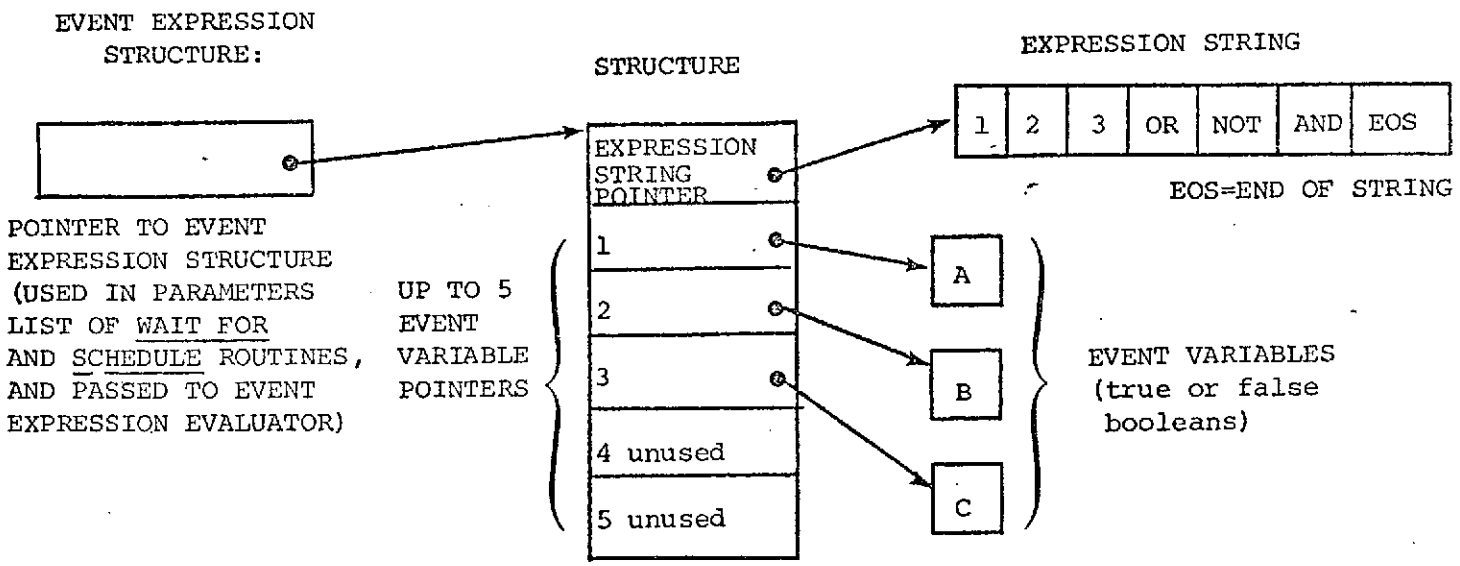
6.4.1 Event Expression Enqueue Routine

This routine is called by the WAIT FOR routine and by the scheduler to:

1) Test if the event expression is immediately true by calling the Event Expression Evaluator.

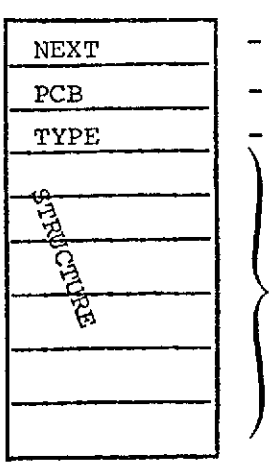
EVENT EXPRESSION STRUCTURE, EVENT BLOCK, EVENT BLOCK QUEUE

EVENT EXPRESSION: A AND NOT (B OR C)



The expression string is an encoded reverse Polish form of the event expression suitable for stack evaluation. Events A, B, and C are represented by 1, 2, and 3 respectively, indicating the relative positions in the event expression structure. The operators AND, OR, NOT, and EOS (End of string) are coded in a way which distinguishes them from event variable representations.

EVENT BLOCK:



- pointer to next event block or null
- pointer to PCB of associated process
- type of event expression (SCHEDULE ON, UNTIL, or WHILE, or WAIT FOR)

event expression structure as above

EVENT BLOCK QUEUE: representing 3 "activated" event expressions

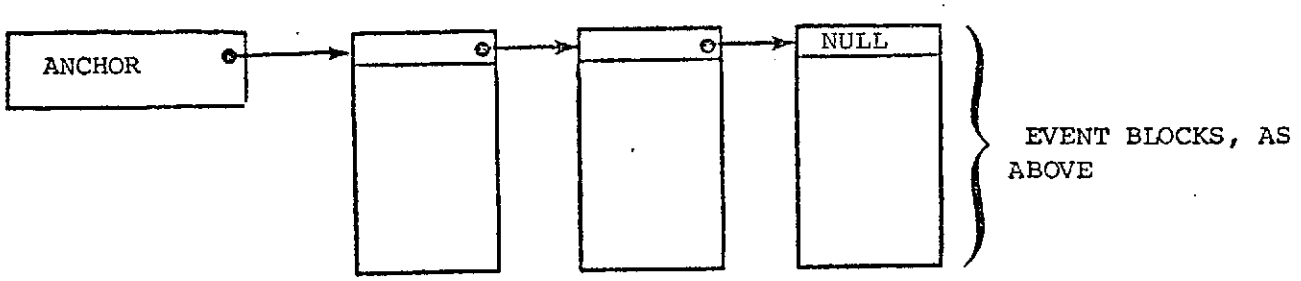


Fig. 6-6

2) If it is not, copy the event expression information to an event block and enqueue the block on the event block queue, thereby activating the event expression condition. (Event blocks are diagrammed in Figure 6-6.) If the expression is the wait type, the appropriate wait state is set in the PCB.

This routine has the following parameters:

1) TYPE of event expression (SCHEDULE ON, UNTIL, or WHILE, or WAIT FOR)

2) PCB POINTER

3) EVENT EXPRESSION STRUCTURE POINTER

If the expression is immediately true, an event block is not queued, and the routine returns with an indicator that the expression was not activated. In this case, the WAIT FOR routine does not pass control to the process selector, but returns control to the executing process.

The event expression structure must be copied to the event block because it is created by the compiled code in temporary storage, and does not remain beyond the execution of the statement. See Fig. 6-7 for a flowchart of this routine.

6.4.2 Event Expression Evaluator Routine

This routine is called by 1) the enqueue routine described above, and 2) by the event processor (described next) when an event has changed state. It takes a pointer to an event expression structure as input and returns a boolean result which is the value of the represented event expression. Using the polish string form of the expression and a simple push-down stack, it actually carries out the logical operations on the event variables. Since the condition is satisfied when the expression value is false for the SCHEDULE WHILE type and true for the other types, the routine inverts (applies the NOT operation to) the result of a WHILE expression. Thus, the Evaluator always returns true if an event expression condition is satisfied. See Fig. 6-8 for a flowchart of the Event Expression Evaluator.

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

EVENT EXPRESSION ENQUEUE ROUTINE

Fig. 6-7

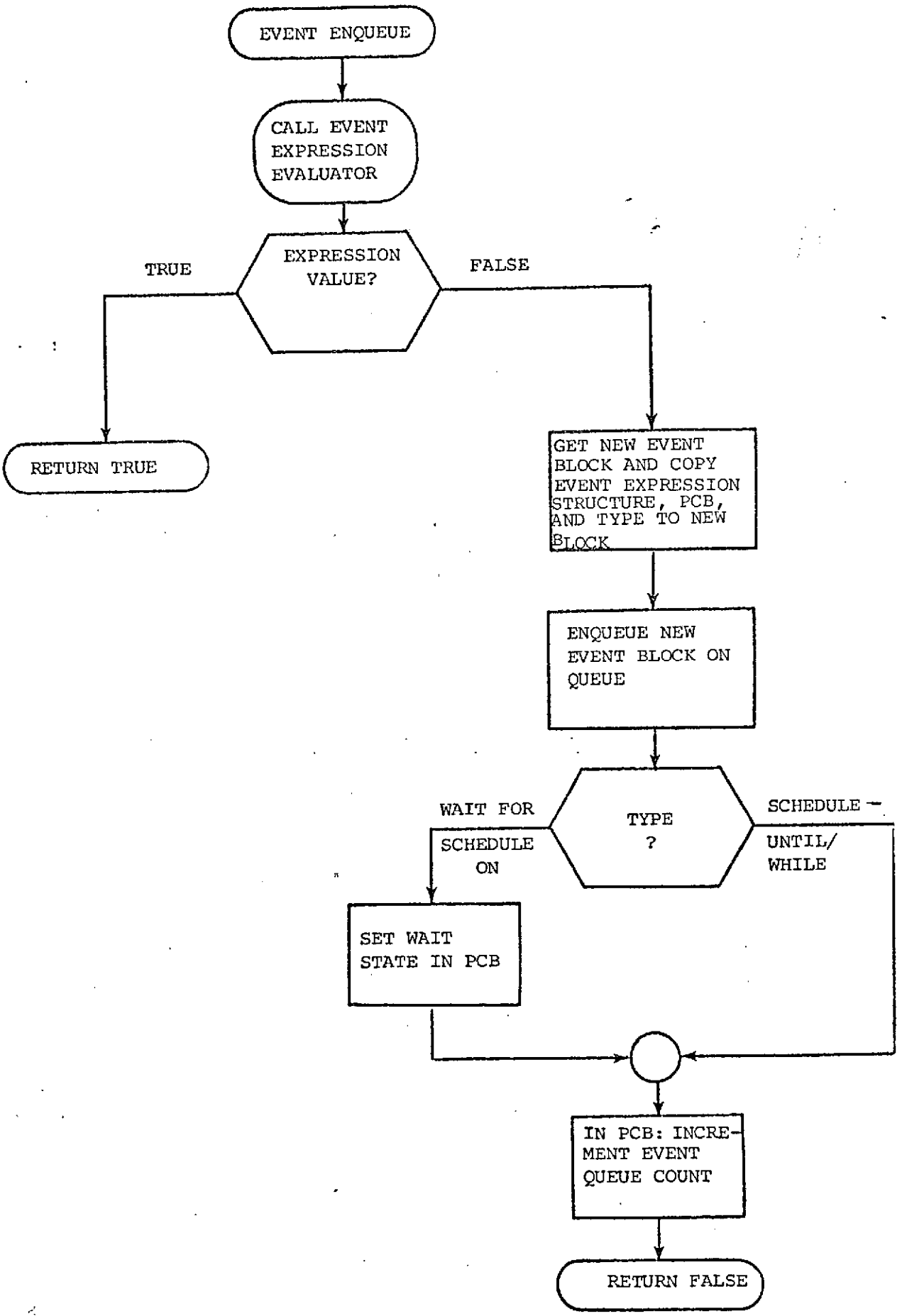
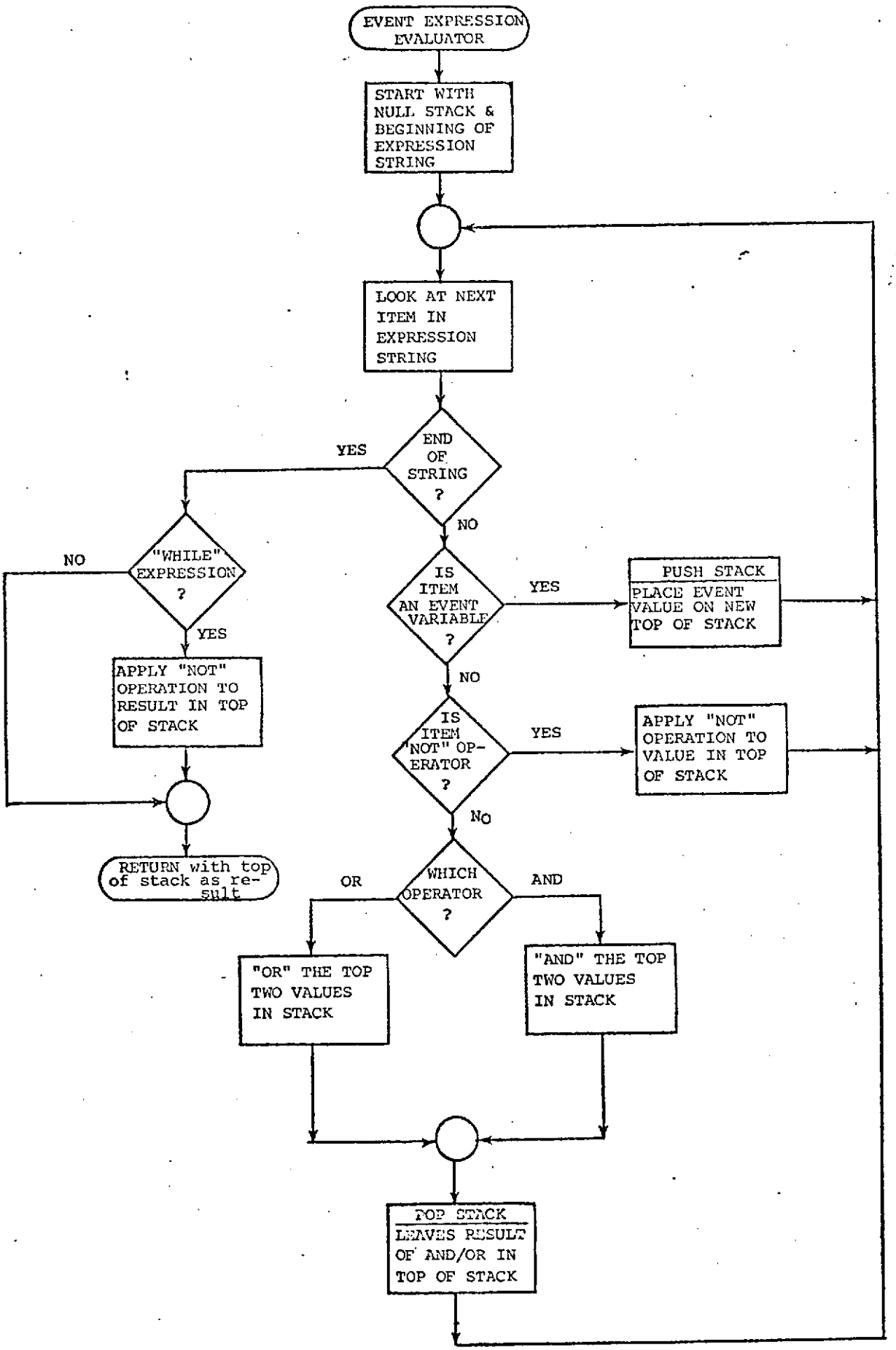


Fig. 6-8

EVENT EXPRESSION EVALUATOR



This routine has the following parameters:

1) TYPE

2) POINTER to event expression structure

6.4.3 Event Processor

This routine is called by the SET, RESET, and SIGNAL Service Routines for normal events and by the Scheduler and the Terminate Subroutine for process events. It re-evaluates activated event expressions by calling the Event Expression Evaluator for each event expression on the event block queue. If the Evaluator returns with a true expression, the Event Processor performs the appropriate action for that condition (readying or cancelling a process), and the event block is removed from the queue and freed. If an event block is encountered with the "terminated" flag set, it is removed and freed. The Terminate Subroutine need only set this flag to de-activate an event expression. See Fig. 6-9 for a flowchart of the event processor.

6.5 Timer Management

There are three programmable clocks specified in the Flight Computer. These clocks can be set via software and are continuously decremented by hardware in real time. They cause an interrupt when the count drops through zero and becomes negative. One of these clocks will be used to provide the timing services requested by the following HAL/S real time statements, and are simulated on the 360.

WAIT - causes the active process to wait for a specified time interval or until a specified time

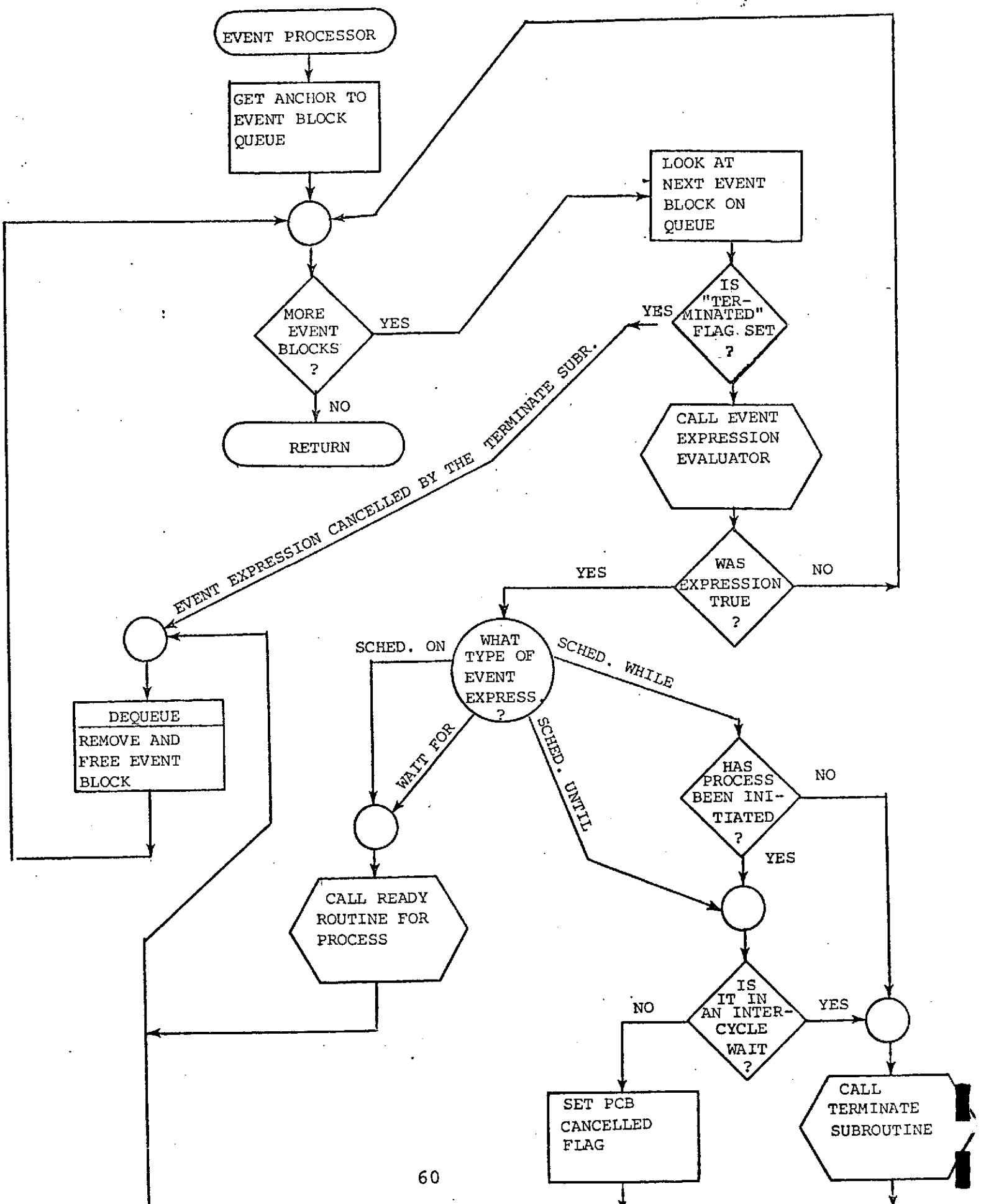
SCHEDULE (IN or AT option) - causes the newly created process to wait before initial execution

SCHEDULE (REPEAT EVERY or AFTER option) - causes the newly created process to execute cyclicly with a specified period between either the beginning (EVERY) or the end (AFTER) of one cycle to the beginning of the next

SCHEDULE (UNTIL option) - causes the newly created process to be cancelled at a specified time.

EVENT PROCESSOR

Fig. 6-9



These timing services are provided by two routines which control the use of the interval timer. The timer enqueue routine is called by any routine requesting a time interval. A type code indicates what action is to be performed when the specified time arrives. The timer interrupt routine is called by the interrupt fielder when a timer interrupt occurs. These two routines operate on a timer queue, each element of which represents a separate timer request. The queue is ordered by time of expiration, so that the first element on the queue is the next to expire. The value in the timer is such that it will cause an interrupt at the time specified in the top queue element.

6.5.1 Timer Enqueue

The timer enqueue routine takes the following actions:

1) if the time value (time of expiration) was supplied in relative form (as determined by the type), it is converted to absolute form.

2) if the time of expiration is already past, the routine returns with a "not enqueued" indication.

3) otherwise, a new queue element is acquired, the input parameters are copied to it, and the element is placed on the queue by order of time of expiration.

4) if the new element was placed on top of the queue in 3), the value in the hardware timer is altered to reflect the new top element.

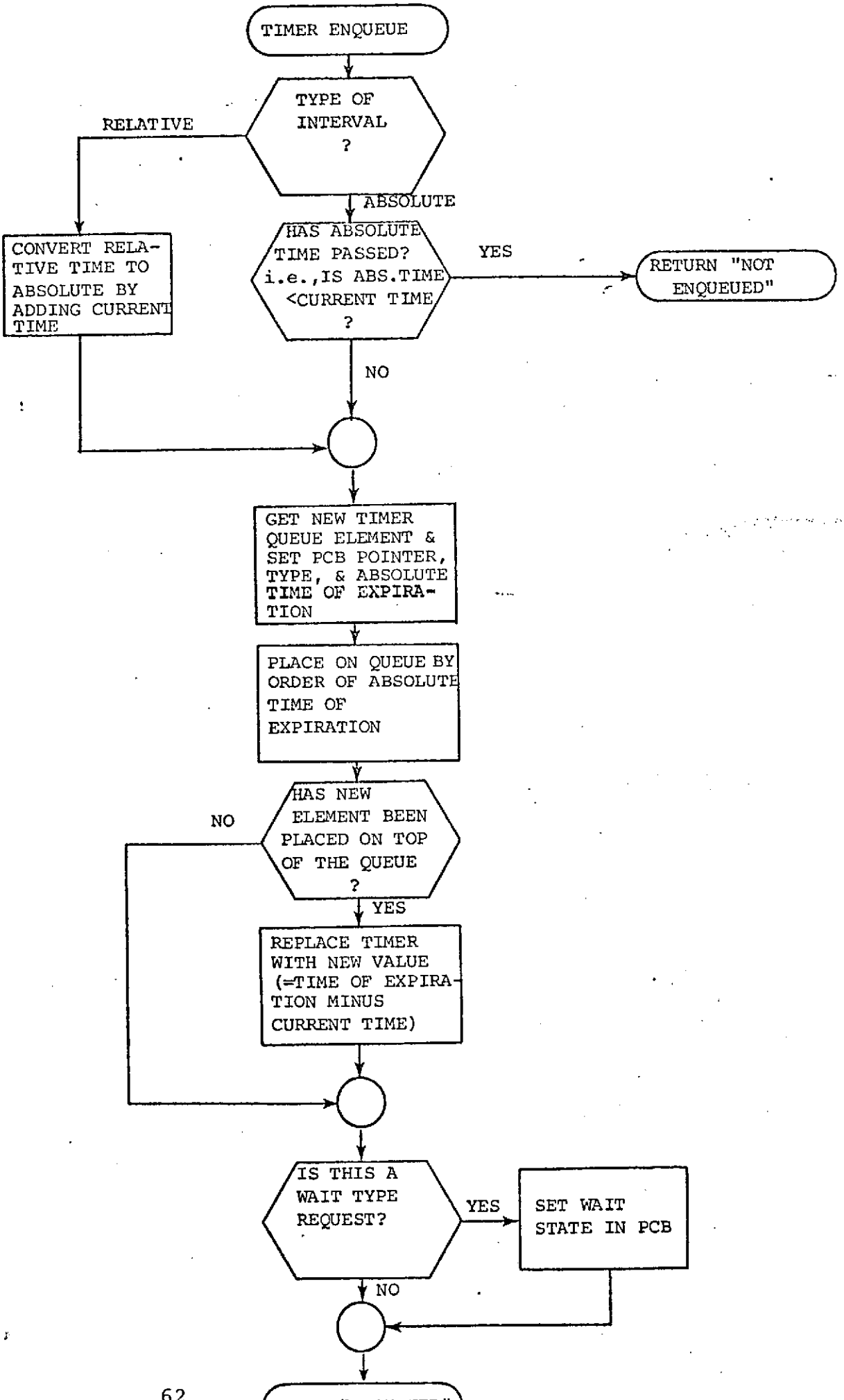
5) the routine returns with the "enqueued" indication.

A flowchart appears in Fig. 6-10.

This routine has the following parameters:

1) PCB pointer
2) TIME VALUE (relative or absolute)
3) INTERVAL TYPE

Fig. 6-10 TIMER ENQUEUE ROUTINE



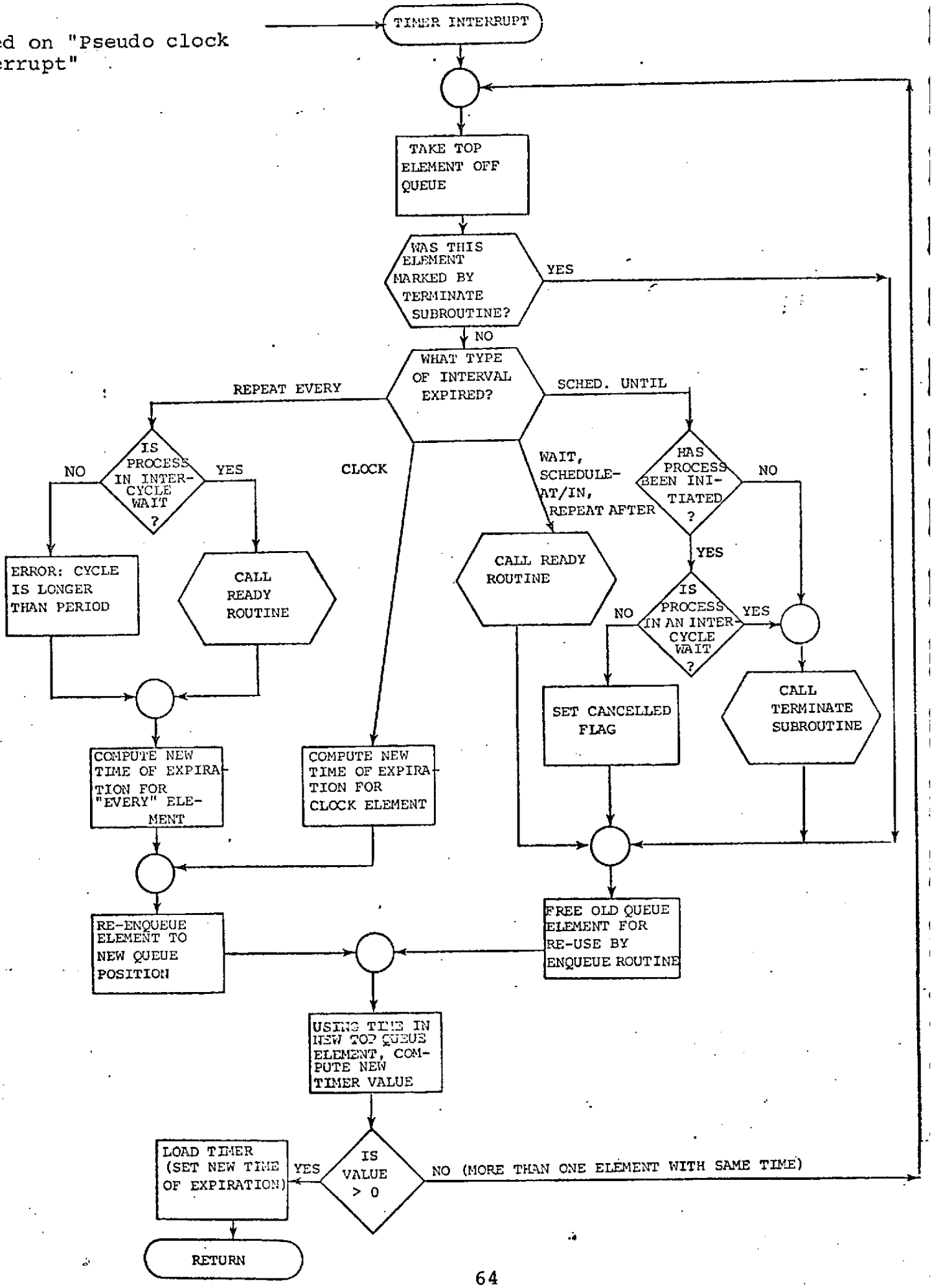
6.5.2 Timer Interrupt Routine

This routine gets control when the timer causes a pseudo interrupt. It takes the top element (the one representing the expired interval) off the queue, carries out the specified action, frees the old top queue elements, and loads the timer with the appropriate value for the new top element. The actions for the expired elements are to ready or cancel a process. A special test is made for an interval representing a SCHEDULE statement REPEAT EVERY option, since there is the possibility that the last cycle ran longer than the specified period between beginnings of cycles. If the process is not in an inter-cycle wait state, an error is indicated, and the process is not made ready. This causes the cyclic process to skip a cycle.

There is also a special element on the queue (called the clock element) which is used to keep the timer running in the absence of any timer requests. Both the clock element and any REPEAT EVERY elements are re-enqueued instead of freed, since they represent self-perpetuating intervals. The most appropriate value for the clock interval is the maximum value that can be placed in the timer. A flowchart appears in Fig. 6-11.

TIMER INTERRUPT ROUTINE Fig. 6-11

called on "Pseudo clock interrupt"



6.6 Detailed Interfaces for HAL/S-360

The detailed interfaces between the HAL/S-360 compiler code generator and HAL/S-360 run time services routines are presented below. These interfaces are subject to change during implementation and are only presented in preliminary form:

HAL/S-360 Compiler Real Time Package Interface

A) SCHEDULE statement. Routine name: SCHEDULE

Note: Of the following parameters, only the first two are required. The rest are optional, depending on the options flag.

Register	Meaning	Bit Position (low order 10 bits)		
R0	options flag:	XX	XXXX	XXXX
	TASK	..	....	...1
	process event	..	....	..1.
	initial conditions	..	....	XX..
	-none-	..	....	00..
	AT	..	....	01..
	IN	..	....	10..
	ON	..	....	11..
	PRIORITY	..	...1	....
	DEPENDENT	..	..1.	....
	REPEAT options	..	XX..	....
	-none-	..	00..	....
	REPEAT	..	01..	....
	REPEAT EVERY	..	10..	....
	REPEAT AFTER	..	11..	....
	cancel options	XX	....	....
	-none-	00	....	....
	UNTIL<time>	01	....	....
	WHILE<event>	10	....	....
	UNTIL<event>	11	....	....
R1	Entry address of program or task			
R2	ON <event expression> pointer			
R3	PRIORITY value			

R4 UNTIL or WHILE <event expression> pointer

F0 AT or IN time value

F2 REPEAT EVERY or AFTER time value

F4 UNTIL time value

B) TERMINATE STATEMENT

I. TERMINATE; EXTERNAL NAME: TERMIN no parameter

II. TERMINATE <taskid>; EXTERNAL NAME: TERMINT

R0: Parameter: entry point of program or task

C) CANCEL STATEMENT

I. CANCEL; EXTERNAL NAME: CANCEL no parameter

II. CANCEL <taskid>; EXTERNAL NAME: CANCELT

R0: Parameter: entry point of program or task

D) WAIT <arith exp>; EXTERNAL NAME: WAIT

F0: parameter: # seconds, double precision

E) WAIT UNTIL <arith exp> ; EXTERNAL NAME: WAITUNTL

F0: parameter: # seconds, double precision

F) WAIT FOR <event exp>; EXTERNAL NAME: WAITFOR

R0: parameter: point to event expression (see event expression)

G) WAIT FOR DEPENDENT; EXTERNAL NAME WAITDEP

no parameters

H) SIGNAL <event var>; EXTERNAL NAME: SIGNAL

R0: ptr to event variable (latched or unlatched)

I) SET <event variable>; EXTERNAL NAME: SET

R0: pointer to event variable (must be latched)

J) RESET <event variable>; EXTERNAL NAME: RESET

R0: pointer to event variable (must be latched)

K) UPDATE PRIORITY TO <arith exp> ; EXTERNAL NAME UPPRIO

R0: Priority

L) UPDATE PRIORITY <taskid> to <arith exp>; EXTERNAL NAME UPPRIOT

R0: Priority

R1: entry point of program or task

7.0 HAL/S-360 INPUT/OUTPUT

A detailed description of the mechanization of HAL/S-360 I/O is not presented in this report. However, some comments can be made pertinent to this effort: (1) There are two types of I/O in HAL/S: Sequential I/O and Random Access I/O. HAL/S-360 has been implemented to utilize 360 peripheral devices only - printer, tape cards, discs, etc. (2) It does not perform avionics data bus I/O. The I/O system design for the Shuttle Flight Computer is still under analysis by NASA/ Rockwell and Intermetrics. Subsequent to this I/O effort the HAL/S Shuttle computer compiler design will reflect any avionics I/O decisions.

7.1 360/I/O Mechanizations

It is relevant to describe which routines and features of OS-360 are utilized in performing the HAL/S READ, READALL and WRITE statements on the 360. A general description of the I/O system is as follows: Using 360 GET and PUT routines I/O buffers are loaded. An I/O initialization routine is called to set the statement mode and channel number etc. Subsequent conversion calls to the library routine are made for each data element in an I/O list. The data items are interpreted using the string, converted and loaded into the appropriate HAL/S declared variable. These HAL/S -360 I/O routines are:

A) IOINT - initialization of I/O operations

B) INPUT, OUTPUT - numerical input and output

C) CIN, COUT - character input and output
CINP, COUTP - character input and output partitioned

D) Conversion routines

HIN	single	integer	input
IIN	double	integer	input
EIN	single	scalar	input
DIN	double	scalar	input
IOUT	integer		output
EOUT	single	scalar	output
DOUT	double	scalar	output

The FILE statement is not implemented in HAL/S-360 as yet.

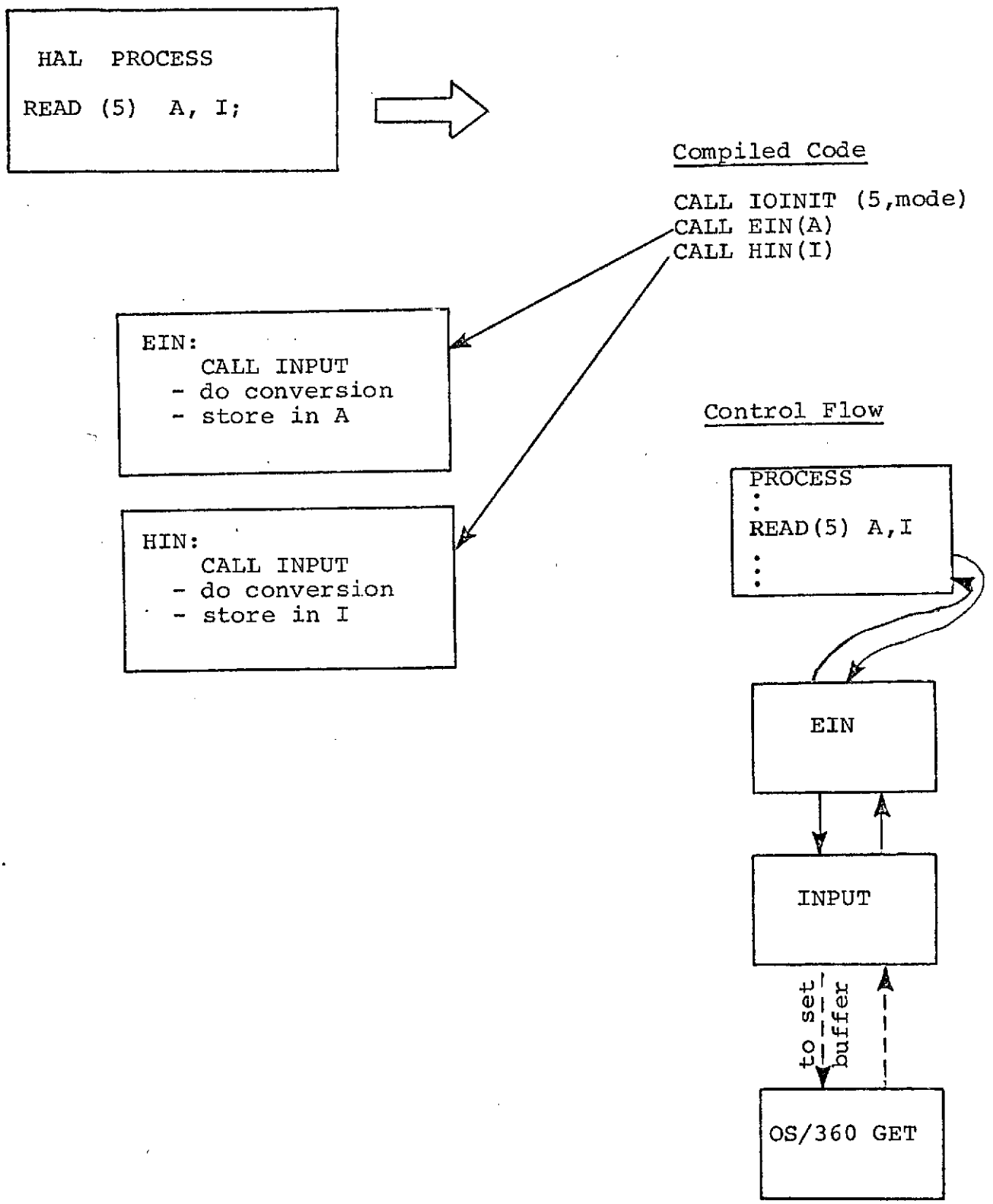
The HAL/S process executing the I/O statement is not placed into a HAL/S wait state. Control remains with the process while the I/O is completed by OS. If OS requires a wait to perform the I/O, the entire HAL/S system load module is placed into a wait condition under OS.

The exact OS routines used to perform I/O are presented in Section 9.

An illustration of the HAL/S I/O control flow and logic is presented in Figure 7-1.

FIG. 7-1

HAL/S I/O SYSTEM EXAMPLE



8.0 HAL/S-360 ERROR CONTROL

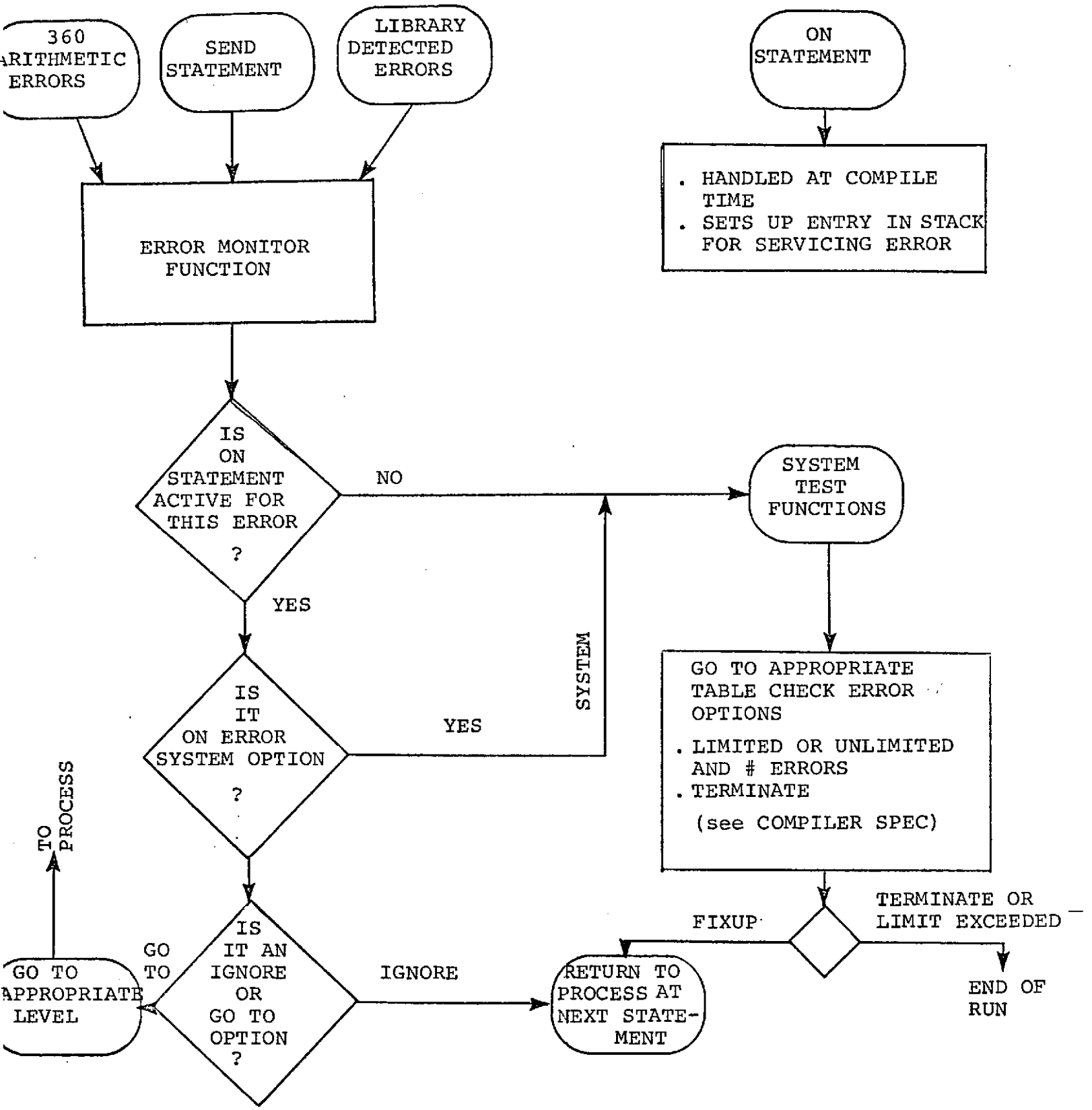
The HAL/S-360 ON and SEND statement are the error control statements. Their mechanization is not presented in detail at this time since it is still undergoing some design changes.

A general organization of the HAL/S-360 error control system is presented in Figure 8-1.

The compiler generates an entry in the STACK for each ON statement with a unique error number. Upon activation of an error condition via a SEND, a library error or 360 arithmetic error, the HAL/S error monitor routine is called. It determines if an ON statement is active for this error condition. If it is,the various HAL/S error options are checked (i.e., IGNORE, GOTO or SYSTEM) and the appropriate action taken. If an ON statement is not active or the SYSTEM option is requested, then the routine must do appropriate checks to determine if the process should continue or the run should end. The significant point is that the error conditions are handled by HAL/S-360 run time routines in in conjunction with the compiler. The only OS-360 routines utilized are SPIE and STAE as identified in Section 9.

FIG. 8-1

HAL/S ERROR CONTROL LOGIC



9.0 HAL/S-360 AND OS-360 INTERFACES

The following lists the active OS-360 routines used.

9.1 Sequential I/O - READ, WRITE

A) OPEN options - INPUT, OUTPUT, MF=L and E

B) DCB options - DDNAME=CHANNELX,
DSORG=PS,
MACRF=(GL,PL)
EXLST, EODAD, SYNAD
EROPT=ACC

Note: remaining fields are filled in via the DCB open exit specified by the EXLST operand. The DCBD macro is used to define the DCB fields

C) DEVTYPE - used to determine blocksize in DCB open exit if user did not supply it

D) GET, PUT - locate mode

E) CLOSE

F) SYNADAF, SYNADRLS - to analyze I/O errors

G) GETMAIN - for DCB area

9.2 Error Control, diagnostic capabilities

A) SPIE - traps the following 360 program interrupts :

9 - fixed point divide
12 - exponent overflow
13 - exponent underflow
15 - floating point divide

B) STAE - traps ABENDS (system and user) to provide abnormal termination information for backtrace and HAL variable dump (no retry routine is used).

C) ABEND - issues user abends on abnormal conditions

D) GETMAIN - for backtrace buffer

9.3 Miscellaneous

A) TIME - provide the time of day and date

B) For HAL variable DUMP:

OPEN - to open HAL symbol table and DUMP file

FIND, READ } to read HAL symbol tables

WRITE - to write dump file

CLOSE

GETMAIN, FREEMAIN } for symbol table buffers

10.0 SUMMARY AND CONCLUSIONS

As stated in Section 2.0, no specific recommendations for providing a HAL capability on SUMC under MPOS can be made at this time because a sufficiently detailed definition of MPOS is not yet available. In this section the OS interface characteristics of the current implementation of HAL/S-360 are summarized, and the various options for modifying it for compatibility with an eventual MPOS are reviewed. Most of these points have already been made in greater detail in Section 2.0.

10.1 Summary of Current HAL/S-360 Interface

The HAL/S-360 compiler is currently implemented as a "self-contained" system which executes as a single task/job step under OS-360. A load module is created by a "HAL Link Step" using the 360 linkage editor. The load module contains all HAL/S compiled programs/tasks, external procedures, and compool blocks which are pertinent to the run, together with a collection of run time routines. This load module or HAL/S system is then loaded and executed under OS as a single task. All HAL/S process management functions, i.e., control over the scheduling and dispatching of HAL/S program and task blocks, are implemented through HAL run time routines which utilize internally defined process queues. OS-360 control and OS task interaction is limited to supervision of the HAL/S system load module. It is unaware of the existence of multiplicity of HAL/S processes and queues.

The HAL/S-360 implementation does not execute in "real-time" on the 360. HAL/S pseudo time is maintained in "machine units" by HAL/S run time routines. The HAL/S-360 system does not utilize the real time OS-360 clocks.

In HAL/S-360, the compiler inserts "hooks" between the code generated for each HAL/S statement to enable recording of variables, implementation of diagnostics, clock updating, process control, and other functions.

The HAL/S-360 memory management is controlled primarily by the compiler for an individual program, and by the link function for all programs.

The host operating system under which HAL/S-360 runs is OS-360 MVT release 21.6. It currently interfaces with the following limited set of OS-360 functions, which are primarily used for I/O (Sequential and File), controlling HAL/S error handling and performing diagnostics: FIND, OPEN, CLOSE, DCB, GETMAIN, DEVTYPE, GET, PUT, READ, WRITE, SPIE, STAE, SYNADAF, SYNADRLS, ABEND, TIME.

10.2 HAL/S-MPOS Interface Implementation Options

The options available to interfacing HAL/S to the SUMC/MPOS fall into two categories:

a) Approaches that leave the current HAL/S-360 implementation either unmodified or only "slightly" adapted to SUMC.

b) Approaches involving major re-design of the compiler and/or SUMC/MPOS to yield a more optimal solution.

10.2.1 Minimal Modifications Approach

a) The HAL/S compiler as delivered to NASA/JSC can be implemented in the SUMC/MPOS with relatively minimum modifications. Assuming that SUMC executes 360 compatible code and that MPOS contains those OS routines which the HAL/S system uses as specified in Section 9, it should be almost directly compatible. The HAL/S capability, however, would be similar to the MSC version, namely:

   1) HAL/S operates as a separate self-contained system

   2) It will function in the same machine with other PL/1 or FORTRAN jobs

   3) It does not operate in real time

   4) It contains no multi-processing or MPOS multi-programming features

   5) It contains no specific virtual memory utilization features.

   This would appear to be the easiest, most cost effective route to obtaining HAL/S for the SUMC computer.

b) Depending on the modifications to HAL/S-360, the intended use of HAL by MSFC, and the dollars available, several modifications could be made to remove two of the above restrictions.

1) Through relatively simple modifications HAL/S-360 could be made to execute in near real time. Since HAL/S-360 timer management is centralized, it could be stimulated by the actual SUMC computer clock(s) and timers.

2) The implementation of HAL/S-360 in the SUMC virtual memory may be relatively simple, depending on the paging technique and memory management system utilized. The structure of HAL/S-360 code/data blocks are contiguous blocks, not overlayed or shared for efficient memory use. The appropriate "hooks" for the SUMC virtual memory system should be fairly straightforward. It requires further analysis of SUMC to evaluate the cost of this feature.

10.2.2 Major Re-Design Approach

It is possible to utilize some of the defined MPOS functions directly for implementing HAL/S real time. However, a thorough design analysis is necessary to determine all "mapping functions" for HAL/S compiler code generation for each HAL/S real time statement and the appropriate restrictions either in MPOS or in HAL/S.

a) One approach is a new compiler design which would assume that MPOS functions and features are fixed, and that HAL/S be implemented in this framework. HAL/S capabilities could be limited where appropriate to MSFC functions. It is fundamentally in the HAL real time and other areas where any problem would exist. This approach could be as costly as any new compiler - and could be a patch-work of code caused by the work around techniques required to implement HAL/S. Of concern specifically is that HAL/S program and task blocks are required to co-exist in the SUMC environment with PL/1 and FORTRAN jobs as individual processes and not as a HAL/S system.

b) A second approach is to design a HAL/S-SUMC compiler assuming the MPOS is not fixed. Accordingly, the design effort would entail negotiation of HAL/S interfaces to the new compiler in order to maximize efficient execution and minimize contractor interference during operating system and compiler development.

c) A third possible alternative is to presume that both HAL/S-360 and MPOS are fixed but to modify and re-build some of the compiler, particularly the supplied run time package. It could perhaps be done in a language like PL/1, attempting to maximize and exploit those PL/1/MPOS interfaces which would need to be resolved anyway. A more detailed analysis is required to examine the mapping of HAL/S constructs for processes and process management into the PL/1 constructs for the SUMC version of PL/1. Additionally, the HAL/S-360 compiled BAL code would need to be sub-structured such that it could be interfaced with these PL/1 drivers.