NASA CONTRACTOR REPORT

SPACEBORNE COMPUTER EXECUTIVE ROUTINE FUNCTIONAL DESIGN SPECIFICATION

Volume III. Executive Routine Primitives and Process Control

by James R. Kennedy, Sr.

Prepared by
COMPUTER SCIENCES CORPORATION
FIELD SERVICES DIVISION, AEROSPACE SYSTEMS CENTER
Huntsville, Ala. 35802
for George C. Marshall Space Flight Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • OCTOBER 1971

TECHNICAL R

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3.
NASA CR-1869		

4. TITLE AND SUBTITLE	5. REPORT DATE
Spaceborne Computer Executive Routine Functional Design Specification, Volume III. Executive Routine Primitives and Process Control	October 1971
	6. PERFORMING ORGANIZATION CODE
7. AUTHOR(S) James R. Kennedy, Sr.	8. PERFORMING ORGANIZATION REPORT #
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Corporation Field Services Division, Aerospace Systems Center 8300 South Whitesburg Drive Huntsville, Alabama 35802	10. WORK UNIT NO.
	11. CONTRACT OR GRANT NO. NAS8-24930
12. SPONSORING AGENCY NAME AND ADDRESS National Aeronautics and Space Administration Washington, D. C. 20546	13. TYPE OF REPORT & PERIOD COVERED Contractor Report
	14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

16. ABSTRACT

This report discusses the concept of a process, and formalizes the process state transitions activated by application program usage of system primitives. Approaches for implementing the required control capabilities in both software (the traditional approach) and digital hardware logic are detailed. Logic network and control sequencing requirements are derived, and the associated circuit diagrams are shown. Software procedures for performing a similar function are developed and depicted in an ALGOL-like source program form. A brief comparison of the two approaches is made.

This document is Volume III of a three-volume report entitled "Spaceborne Computer Executive Routine Functional Design Specification." The other two volumes are:

Volume I: Functional Design of a Flight Computer Executive Program for the Reusable Shuttle

Volume II: Executive Design for Space Station/Base

17. KEY WORDS		18. DISTRIBUTION STATEMENT
Executive Routine	Multiprocessor	Unclassified - Unlimited
Operating System	Multiprogramming	
Hardware Executive	Scheduling	
Real Time Monitor	Spaceborne Computer	

19. SECURITY CLASSIF. (of this report)	20. SECURITY CLASSIF. (of this page)	21. NO. OF PAGES	22. PRICE
Unclassified	Unclassified	95	$3.00

*For sale by the National Technical Information Service, Springfield, Virginia 22151

VOLUME III

EXECUTIVE ROUTINE PRIMITIVES AND PROCESS CONTROL

TABLE OF CONTENTS

VOLUME III

EXECUTIVE ROUTINE PRIMITIVES
AND
PROCESS CONTROL

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

DEFINITION OF SYMBOLS

AAIJ	Allow all interrupts and jump. The jump instruction which transfers processor control to a process return address or entry point.
Backlog	An amount of work which has been scheduled but has not been completely processed.
Compute Cycle	The complete transition cycle ("idle," "ready," "running").
III	Initiate Interprocessor Interrupt. A computer instruction used by one central processor, under executive control, to signal another central processor for the purpose of assigning tasks.
LPS	Load Processor State Register. The executive instruction which enables the executive to set the state of a processor to insure system protection.
LSR	Load Storage Register. The executive instruction which sets the memory access boundaries to those of a given process.
PCB	Process Control Block
Preempt Dispatcher Action	The act of seizing control of a processor from a process for assignment to a higher priority process.
Process	The sequence of actions performed in order to complete a task.
Process Construction	The act of executing a set of procedures that create a process.
State Diagram	A representation for a finite state machine that has inputs and outputs.
Task	A specific quantity of work to be accomplished.

TCL	Trap Control Line. Indicator denoting that a trap has occurred.
TDR	Trap Designator Register. Indicator denoting whether a processor is the trap processor or not.
TDS	Trap Designator Set
TPL	Trap Processing Line. Indicator to denote that trap processing is occurring.
Trap	A list-driven processor capability activated by the process control mechanism; a transfer of processor control to a specified location as a result of some event or condition requiring special attention.

FOREWORD

The work reported herein was administered in the Systems Research Branch, Computer Systems Division, Computation Laboratory, MSFC, with Bobby C. Hodges assigned as Contracting Officer's Representative. In addition to his routine duties as Technical Monitor, Mr. Hodges has added significantly to our insight into and understanding of related NASA programs through careful planning, coordination with in-house effort, and encouragement.

VOLUME III

EXECUTIVE ROUTINE PRIMITIVES AND PROCESS CONTROL

SUMMARY

The feasibility of partitioning an executive routine into primitive controls, scheduling functions, and supporting supervisory software is shown. The report outlines an approach to a functional partitioning that lends itself well to automation.

Based upon general requirements for run-time support to arbitrary processes, a basic set of executive routine primitives is defined. Using an inductive approach, a state transition diagram is developed, thus forming the basis for the development of a finite state automata to control all processes. The state diagram is then used to derive a digital hardware logic device that provides the necessary sequential processing. A stacking mechanism is introduced to link the control hardware to certain software support procedures. The stack is processed through the use of a hardware trap scheme for executing procedures that have been stacked.

A software approach to mechanizing a comparable capability is developed. The method of depicting this approach is comprised of showing program flow diagrams and high-level source language statements. The architectural framework for specifying the software approach is taken to be the UNIVAC 1108 Multiprocessor System.

Finally a comparison of the two approaches shows that a hardware implementation is practical and displays significant advantages in terms of cost and system overhead. Although no explicit comparisons are given to contrast the two approaches with respect to weight, volume, reliability, and power consumption, the overwhelming simplicity of the hardware approach is felt to obviate the need for detailed comparisons. The results are clearly in support of a hardware logic design approach.

SECTION I. INTRODUCTION

The supervisory aspects of controlling program execution with a general purpose digital computer environment have become so complex that not only is it now difficult to implement scheduling procedures that have predictable effects, but it has also become difficult to establish design requirements and describe a control method that exhibits desirable features. This complexity is, of course, an inevitable result of a tendency to want more out of computing systems in terms of the total number of tasks completed in a given time interval. Since faster program development is also desirable, systems have been further complicated by the addition of functional responsibility in the area of development support through program debug, text edit, language translation, and file maintenance capabilities.

Relatively good success has resulted from recognizing that many functional responsibilities can be simplified through partitioning or segmentation into well defined and easily managed subfunctions. The purpose of this monography is to outline an approach to applying partitioning procedures to the supervisory function. The objective is to show simplicity in the methods involved, and to analyze and compare several possible methods for implementation.

The following discussion outlines the concept of a "process" as it relates to the computer executive function. Based on this concept, the important features of process "construction," "primitives," "control," "termination," and "monitoring" are discussed. Techniques for control implementation are examined, and examples of usage are cited. The discussion ends with concluding remarks and recommendations for further effort.

Implementation of the control concepts discussed in the report has historically been accomplished through the application of software engineering in order to design, fabricate, and test executive system programs that perform the required functions. Recent research and development has isolated program control principles that are reasonably general purpose, and necessary in all but the simplest sequential batch programming systems. The report expands on these principles, and emphasizes a digital logic approach to implementation; a software approach is also indicated.

To establish a frame of reference for the major points of interest that follow, the concept of a process and the meaning of construction of a process are outlined.

A. Concept of a Process

The usual quantity of work referred to in discussions regarding computer systems is the "task." "User tasks" and "system tasks" have been accepted as terms for describing the entities that an executive system deals with in its supervisory capacity. A task will be similarly regarded in this discussion as a specific quantity of work to be accomplished.

Most of the discussion, however, will be concerned with the sequence of actions performed in order to complete a task. This sequence is referred to as a "process" and is discussed by Lampson /1/ and others /2/3/4/. It is important to note that a process may execute code from either system or user (application) programs, or both, in a more-or-less arbitrary order. Since the control devices of process-oriented systems may include those for stopping process execution, code which is shared among several (possibly concurrent) processes must be structured to support unsynchronized, multiple (simultaneous) execution instances. Such a program is often referred to as reentrant in that one execution instance can be suspended and another begun, both at (virtually) any location in the code. With a multiprocessor system, literal simultaneity is possible.

The definition of a "process" can be extended recursively by also considering a set of sequential processes to be a process. Such a definition would allow for (and require) a nested control capability. However, this discussion is concerned only with the simpler definition since no generality is lost.

---

[1]Lampson, B. W.: A Scheduling Philosophy for Multiprocessor Systems. Comm ACM, V 11, N5, pp. 347-360, May, 1968.

[2]Dijkstra, E. W.: The Structure of "THE" - Multiprogramming System. Comm ACM, V 11, N5, pp. 341-346, May, 1968.

[3]Wirth, N.: On Multiprogramming, Machine Coding, and Computer Organization. Comm ACM, V 12, N9, pp. 489-498, September, 1969.

[4]Hansen, P. B.: The Nucleus of a Multiprogramming System. Comm ACM, V 13, N4, pp. 238-250, April, 1970.

B. Construction

Process construction is the act of executing a set of procedures that create a process. In its broadest sense, process construction consists of program design and coding followed by translation to machine-executable code, collection into a module that is mapped onto main (instruction) memory, input to main memory, and creation and initialization of a block of main memory that constitutes a set of state variables for control of the process.

For purposes of this discussion, a truncated definition will be sufficient. It includes:

- Collection and mapping,
- Input to instruction memory, and
- Process control block formations.

Collection is a gathering together, from several sources, of the various uncollected routines constituting a set of process code. As the code is gathered, it is mapped one-for-one word-wise onto some contiguous set of instruction memory locations. Once collection and mapping has been accomplished, the code can be written into instruction memory and a process control block (PCB) constructed.

Figure 1 shows a possible structure for a typical PCB /5/ beginning at the ring pointer PCBRING. The contents of this block are for the most part self-explanatory; table 1 defines these entries. (In the event that multiple processors are allowed to concurrently execute common code from a given process, the entire process must be reentrant. The PCB structure shown will not support this requirement. Although it is certainly of academic interest, this possibility is not considered further here.)

---

[5] The process control block discussed here is similar to the "Exchange Package" discussed in /6/ and the "Job Area" of /7/.

[6] Reference Manual - Control Data 7600 "Preliminary Computer System," Control Data Corporation Publication Number 60258200, Revision 02, 1969.

[7] Huberman, B. J.: Principles of Operation of the Venus Microprogram. Mitre Technical Report MTR-1843, The Mitre Corporation, Bedford, Mass., 1 May 1970.

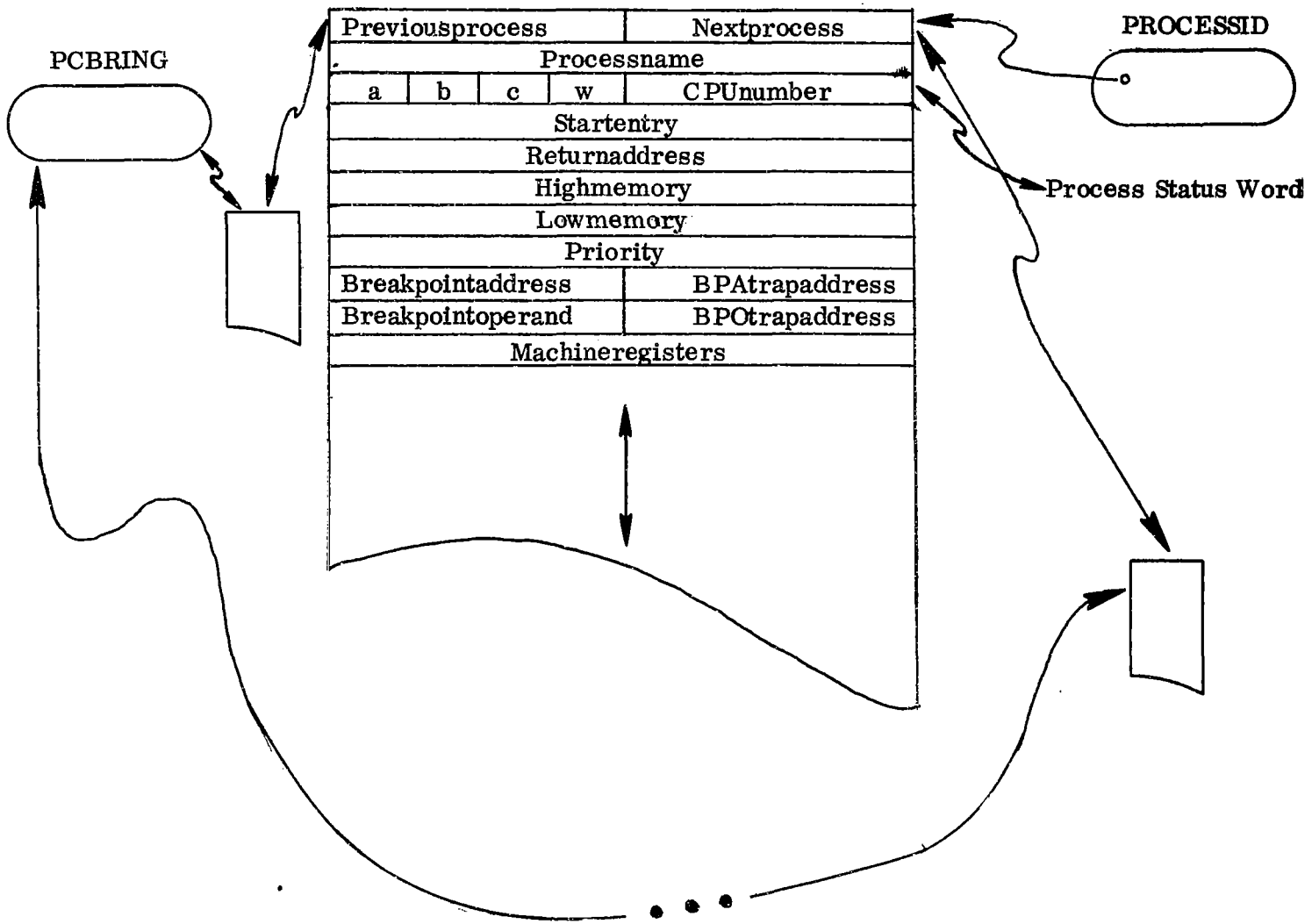


FIGURE 1. PROCESS CONTROL BLOCK

TABLE 1. PROCESS CONTROL BLOCK ENTRY DESCRIPTIONS

ENTRY	DESCRIPTION
Previousprocess	Pointer to the predecessor PCB on the ring.
Nextprocess	Pointer to the successor PCB on the ring.
Processname	Unique name for this process.
a b c	Three bit process state indicator.
Priority	Relative process priority.
Startentry	Instruction memory address of the first instruction.*
Returnaddress	Instruction memory address of next instruction in case process activity is stopped; execution will be resumed at this location. Initially has the value of startentry.
Highmemory	Largest instruction memory address associated, for protection and access purposes, with this process.
Lowmemory	Smallest instruction memory address associated, for protection and access purposes, with this process.
CPUnumber	Hardware address of the processor unit associated, during execution, with this process.
Breakpointaddress	Instruction memory address which, if it becomes the argument of an instruction fetch cycle, will cause an internal processor trap to a predetermined instruction memory address specified by BPAtrapaddress.**
Breakpointoperand	Instruction memory address which, if it becomes the argument of a datum fetch cycle, will cause an internal processor trap to a predetermined instruction memory address specified by BPOtrapaddress.**
Machineregisters*	A block of words reserved for saving all programmable processor registers when process activity is stopped. Must include all registers depicting process state information.
w	Counter showing the number of unserviced WAKE primitives invoked for this process.

* The exact meaning of all main memory addresses is dependent on the details of hardware addressing. The preliminary organization shown here is merely representative. For instance, if data and instructions are separated a high and low data memory address would be required; if a paged memory is used, the page file map would be saved.

**These values would have meaning only when the associated processor is operating in a debug mode.

At the time of construction, the values of a, b, and c, discussed in detail below, are all set to zero, as is the case with Returnaddress, CPUnumber, Breakpointaddress, and Machineregisters, while Startentry is stored in its designated space for the life of the process. It should be pointed out that no specific memory word size or processor register set is assumed; the organization of the PCB is, therefore, subject to optimization in a specific architectural case. Also, the storage area reserved for PCB's is assumed to be protected through definition of a set of privileged instructions (at least a "store-PCB" instruction).

SECTION II. PRIMITIVES

In order to accomplish control of processes, a set of primitive operators, or instructions, is defined. The exact effect of these primitives, when invoked by a process, is discussed in the next section; the general effect, however, is to alter the state of a process through specific executive system action. The states of all processes known to the system are kept current by recording these values as three bit binary numbers in the appropriate PCB's. The names used here for the primitives were chosen because of the intuitive thoughts evoked by them.

A. Wait

This primitive has no explicit arguments (parameters) associated with it. It is a command executed by a process when the process is executing and cannot proceed until some arbitrary, requested event has occurred. The effect of this primitive is to stop process action and thereby make the associated relinquished processor available for assignment to another process that is ready to proceed. The implied argument of this primitive is the "Processname" of the invoking process. Because no other arguments are recognized, it is not possible for process "A" to keep another process, "B," from proceeding by direct use of the WAIT primitive.

When execution of a process that has invoked a WAIT primitive is resumed, it will continue at the instruction immediately following the instruction sequence that invoked the WAIT. The address of this instruction is saved as Returnaddress in the appropriate PCB by the WAIT mechanism. The WAIT mechanism also saves all pertinent processor registers in the PCB and sets the values of a, b, and c to indicate that the process is in a "waiting" state. Other WAIT mechanism functions are discussed in the section on "Control."

B. Continue

When a process has placed itself in an inactive state through a WAIT, it can be resumed only by another (cooperative) process through use of a CONTINUE primitive. This primitive specifies the name of the waiting process as a parameter. The PCB for the specified process is located by use of its unique name (for instance, by searching a hash-coded table that associates a pointer, such as PROCESSID in figure 1, with the name); the process is then placed in a "ready" state wherein it may compete, on the basis of its relative priority, for assignment of a processor.

It is important to note that, prior to use of WAIT, every process must be assured that some cooperative process will invoke a CONTINUE in behalf of the waiting process. Normally, certain system support routines in the form of subroutines or supervisor calls would be provided to allow a process to request that it be continued for specific reasons. Examples might include completion of an input operation, expiration of a specified time interval, granting of a request for device assignment or storage allocation, etc.

C. Wake

After a process has been constructed, it is in the "idle" state. In this state, the process is prepared for execution but is not yet activated. The WAKE primitive is the mechanism for causing a specified idle process to be placed in the "ready" state where it will have processor time allocated for execution.

When a process is "idle" and a WAKE for it is invoked, the WAKE primitive will cause the process "startentry," stored in the PCB, to be copied into the returnaddress space in the PCB. The WAKE primitive specifies the object process name as an argument.

D. Stop

This primitive is invoked by a process that is in the "running" state in order to indicate to the system that it has completed its execution and wishes to return to the idle state. · Once a STOP has been invoked, subsequent execution instances occur only as a result of WAKE primitives invoked for this process. Each such execution instance will begin at the process "startentry" saved in the PCB.

E. Suspend

This primitive and its converse, RELEASE, are defined to enable a process, "A," to stop and restart another process "B." This capability is provided primarily for the purpose of stopping a specified process to enable it to be examined intimately by some other process. From the point of view of the suspended process, there is no discernible effect; it therefore has no knowledge of having been suspended. While suspended, the process's data, instructions, and machine register contents can be examined or modified dynamically.

While the uses of SUSPEND are many, only debugging and synchronization are mentioned here. Several processes can be synchronized to the nearest instruction by breakpointing and suspending until all processes have been suspended. Then, upon release, they will be closely synchronized. In order to debug process code or analyze algorithm failures, it is necessary to suspend execution nondestructively to permit observation.

F. Release

The act of invoking a RELEASE primitive for a specified process will cause the process to revert back to the state it was in at the time of the most recent SUSPEND for the process (an exception is discussed below). For instance, if the process was executing code at the time of a SUSPEND, a cancelling RELEASE will cause it to continue execution where it was suspended.

G. Termination

The discussion has considered process construction and control. For completeness, primitive controls are specified to allow a process to terminate itself or for another process to cause its termination.

Self-termination can be accomplished by a process through an EXIT primitive, while termination of a process by an external mechanism is accomplished through an ABORT primitive. Processes that are aware of an internal anomaly may ABORT themselves also.

In effect, an EXIT will cause a process to be placed in the "idle" state, followed by release of main memory assigned for appropriate process code and PCB residence. An ABORT has all of the effects of an EXIT with additional capability for "post-mortem" main memory dumps and other terminal actions to aid in debugging.

SECTION III. PROCESS CONTROL STATES

In this section, the use of system primitives is discussed through considerations with respect to state diagrams. Also, an alternate form for the state diagram is introduced in order to support the development of logic expressions for process control by an executive system.

Many of the details related to logic techniques are considered to be routine in the field of digital systems engineering. Similarly, many of the historically-"software" concepts are routine to the systems programming field. The (sometimes) tutorial nature of the discussions that follow has the major purpose of introducing readers in each field to the concepts and techniques of the other in order that more integrated systems will be appreciated.

The conventions used in the state diagrams developed here are standard in that arrows are used to show transitions; the name or symbol adjacent to the tail of each arrow is the identifier for the primitive, or input, which will cause the associated transition. Given a particular state, certain primitives may cause no state change. Also, certain primitives may not be valid. For instance, in the "ready" state, a WAIT primitive cannot occur. Such transitions as are nonvalid, or cause no state change or action, are not shown to avoid clutter.

A. Compute Cycle

A process enters the domain of the state diagram to be developed through process construction when the values of a, b, and c are set to zero in its PCB. For this reason, the discussion begins with state zero. The normal sequence of state transitions for a process is "idle"-"ready"-"running." This is shown in figure 2. A process in an "idle" state (state 000) will go to the "ready" state as a result of some other process having invoked a WAKE in behalf of the idle process. In the "ready" state, a process competes for processor time. An executive procedure, known as the "dispatcher," examines a queue of entries representing all processes in the ready state. Once the dispatcher determines a match between some process and processor, it will "dispatch" the process through a special system primitive which causes the processor to start execution of the appropriate process. The process is thereby placed in the "running" state. While a process is "running," it can invoke a STOP, thereby placing itself in the "idle" state again.

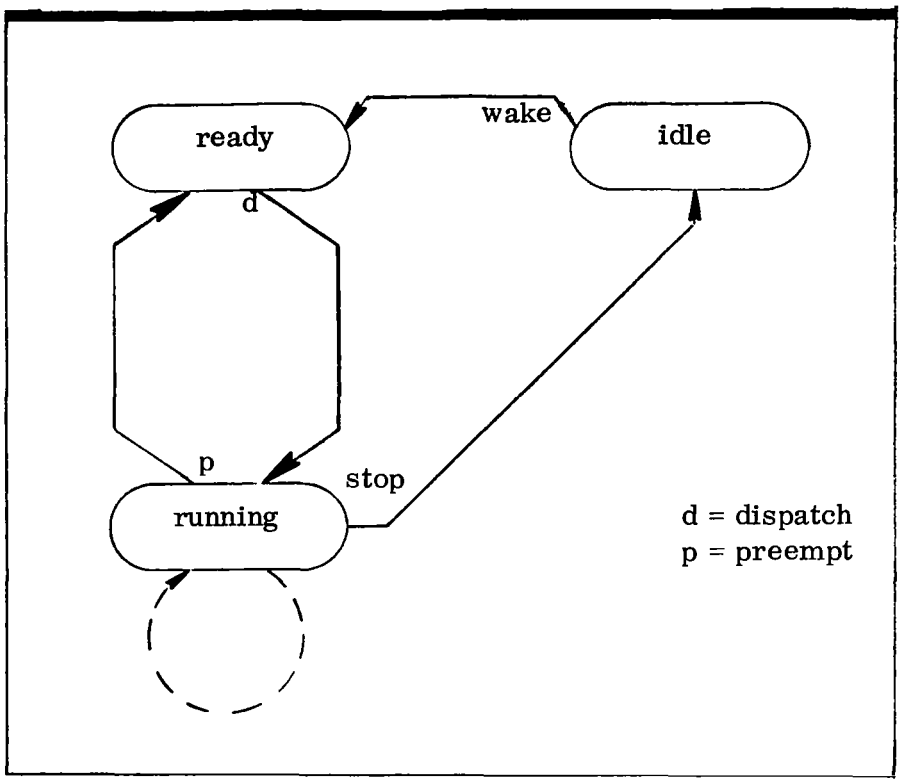


FIGURE 2. INITIAL STATE TRANSITIONS

The complete transition cycle ("idle," "ready," "running") will be referred to as a "compute" cycle. Many processes can be satisfied by simple, repeated loops through the compute cycle. Each time through the cycle,

- an idle process would be waked up by another process or the system,
- processor time would be allocated,
- the process would execute, and
- a STOP would be invoked to delay progress until the next compute cycle occurred or was needed.

B. The Work Variable

The diagram of figure 2, while satisfying many of the process control requirements, is nevertheless inadequate. For instance, it is possible that some event may occur causing a WAKE to be invoked while the affected process is in the running state. This is shown by the dotted transition in figure 2. In order to avoid the loss of such a WAKE, the state diagram is augmented to provide it with the ability to remember WAKEs that occur while a process is not in the "idle" state. This is accomplished through a state variable, "w," as shown in figure 3.

The scheme operates as follows: Each WAKE increases the value of "w" by 1 and each STOP decreases the value by 1. When a process invokes a STOP, a "testing" state is entered wherein the value of "w" is compared to zero. If "w" is greater than zero, it means that some event requiring processing has occurred, and the process is returned to the running state at the start address to execute another pass through the compute cycle. If "w" is zero, the process is placed in the "idle" state as shown /8/.

The state variable, "w," is located in the PCB and can be initialized during process construction to any value. If it is initialized to zero, the normal compute cycle will occur as discussed above. If it is given a positive initial value of, say, n, the first n STOPs will loop through the testing state and back to the running state thereby effectively ignoring the first n STOPs. This is a useful capability that aids the programming of initialization for certain cyclic processes for the first (n) time(s) the process enters the running state. No consideration has been given in this report to possible uses of initializing "w" to a negative value; only non-negative values are treated properly in the state diagrams shown.

---

[8] The interpretation of "w" as a semaphore /2/7/ is valid where STOP is similar to Dijkstra's "P" operator and WAKE is similar to "V."

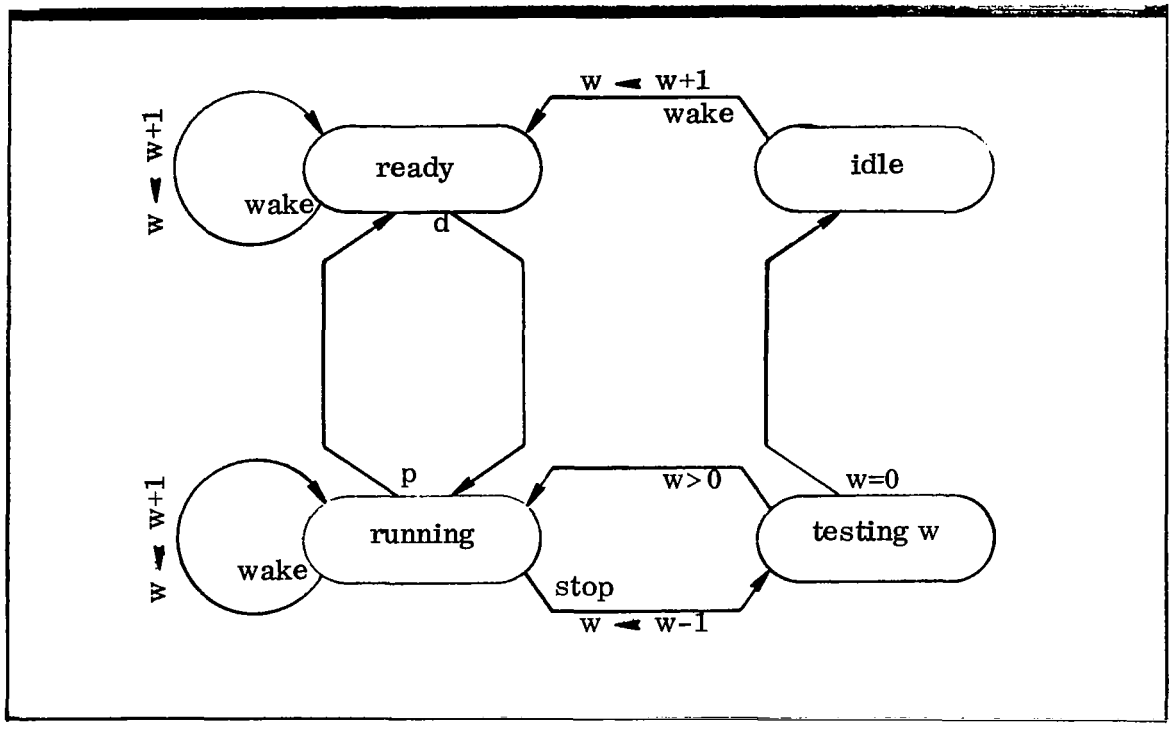


FIGURE 3. REVISED STATE TRANSITIONS

The state variable, "w," has several useful side attractions that result from its role in the control diagram. First, the instantaneous value of "w" is a count of the number of WAKE primitives that have not been "serviced" through dispatching and subsequent processing. In the special case where, as a result of cooperative process action, a process is "waked up" periodically, say every 10 milliseconds, to perform some calculation, the value of "w" can be measured by the system to determine whether the calculations are lagging behind. If, because of a low relative priority, the waked-up process is not dispatched before the next WAKE, the "w" can be used as a direct measure of how far behind the process is. "w" will be referred to as the process "work variable" because of the apparently close relation to system workload.

C. Dispatching

Returning to the state diagram of figure 3, it is seen that a process in the "running" state can have its processor seized by the dispatcher for assignment to a higher priority process. This is known as a "preempt" dispatcher action. It causes the preempted process to be returned to the "ready" state to await a future "dispatch" primitive that will allow it to proceed. "Preempt" and "dispatch" are special system primitives that can be invoked only by the dispatcher which, as part of the process control mechanism, has privileged access to the necessary control devices to assign processor time to a process.

Extending the discussion farther, it seems clear that the dispatcher might be designed to dispatch processes with higher "w" values first, all other factors (such as priority) being the same. If "w" is interpreted as a measure of work scheduled for a process, then the sum of all "w" values over all processes known to the control system can be thought of as an indication of the total scheduled work for the system at any given time. With this interpretation, the "system workload" can be measured as a function of "w."

In a "closed" system - such as certain industrial process control, message switching, avionics, ballistic missile defense, air traffic control, airline reservation, and other similar real-time systems - it would seem to be useful to sample "w" dynamically. Analytical techniques applied to probability distributions of sampled "w" values might then be used to develop feedback scheduling and dispatching algorithms to dynamically optimize system performance under varying workloads.

D. Cooperative Processes

We have thus far developed a scheme that shows the relation between WAKE and STOP through the introduction of the concept of a compute cycle and the process work variable, "w." This scheme is particularly applicable in the case of cyclic or repetitive processes such as those found in all computer controlled real-time systems. It is necessary to incorporate an additional pair of primitives to support control of a process that is "waiting" for some requested event to occur before it can proceed.

Consider the case of a request for data input. Unless the data are already buffered in main memory at the time of the request, it would be necessary in most cases to queue the request and place the requestor in a waiting state until the data have been input and converted for use. A similar situation arises in the case of requests for main memory space, peripheral devices, timed delays, etc.

Figure 4 depicts a "waiting" state which is entered by a process that invokes the WAIT primitive. When the condition necessitating a WAIT has been eliminated, the process is placed in the ready state through the use of the CONTINUE primitive. When the dispatcher assigns a processor, control resumes at the "returnaddress" saved in the PCB by the WAIT mechanism.

It is interesting to consider what would happen if an "unauthorized" CONTINUE were invoked as a result of either a hardware or software error. The waiting process would, upon entering the running state, assume erroneously that the request had been granted and proceed to cause a further promulgation of the original failure - possibly beyond the point of recovery if this point had not already been reached.

One way of preventing such a CONTINUE would be to build into the WAIT mechanism a procedure for generating a unique "key." This key would be made available to the procedure authorized to invoke the corresponding valid CONTINUE. Then, whenever a CONTINUE is invoked, the key supplied by the invoker would be matched to the unique key generated at the time of the WAIT. Mismatches would determine program errors or system failures. The details of this form of validity checking will not be considered further in this report, but they certainly form the basis for further examination and definition.

E. Suspended States

The final alteration to the state diagram is an extension of figure 4 to include optional "suspended" states. The suspended states are companions to corresponding nonsuspended ("waiting," "ready," "idle," and "running") states and are provided as a mechanism for stopping the progress of a process. The

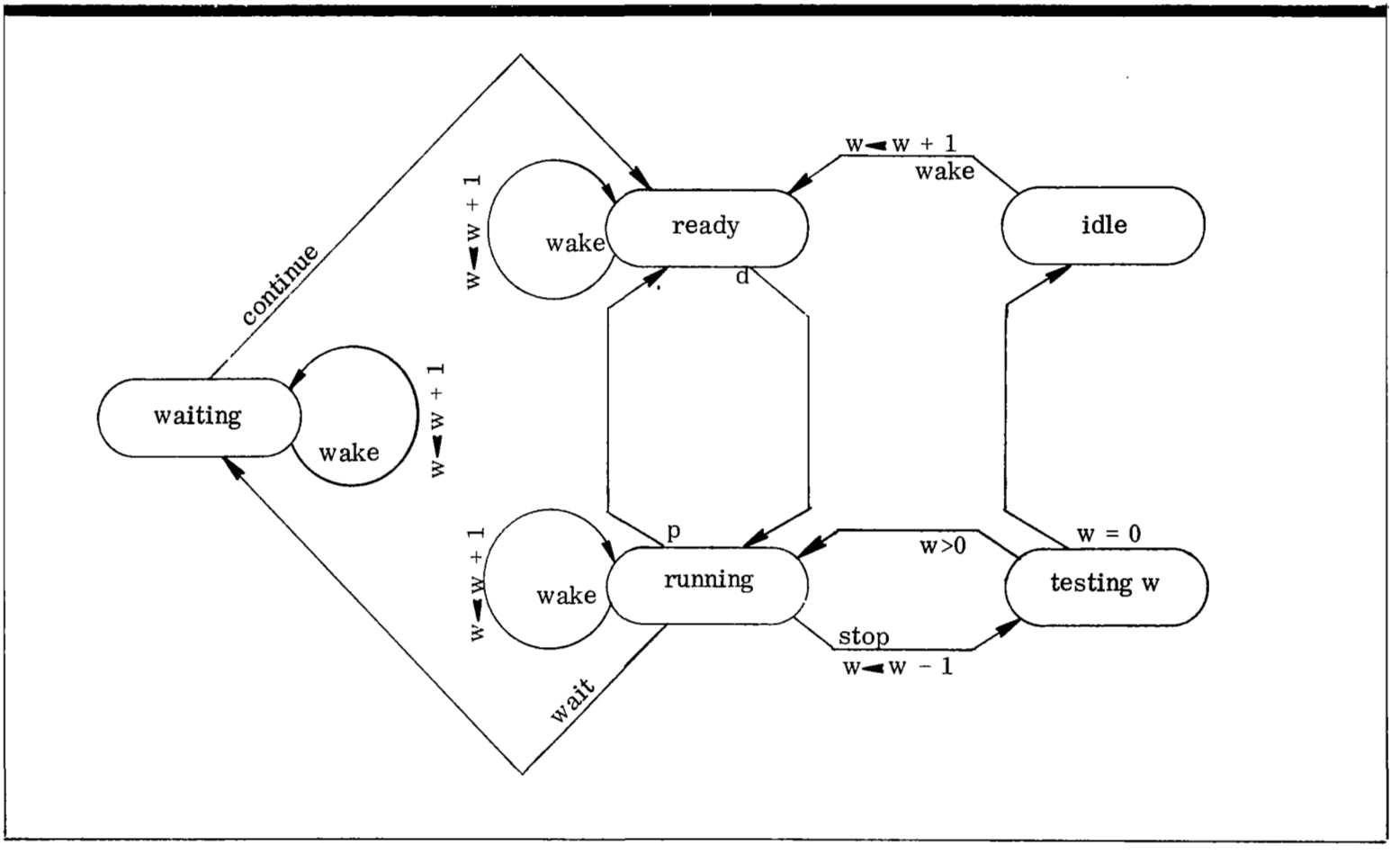


FIGURE 4. EXTENDED STATE TRANSITIONS

effect of a WAKE and CONTINUE primitive is preserved through an appropriate transition from "idle suspended" and "waiting suspended" to "ready suspended." Figure 5 shows the complete state diagram.

Suspended states can be associated with a special mode of processor operation which will support extensive observation by external processes of processing activity. This mode might be referred to as the "debug" mode. Figure 5 shows process state changes with regard to certain debugging primitives, namely "suspend" and "release." Other debugging primitives that influence processor states, but are unknown to processes, can be defined to enable the specification of a comprehensive automatic debug program as part of an executive. These concepts are properly discussed elsewhere. For the purposes of the present discussion, "suspend" can be considered to be equivalent to the (manual) depression of a console "stop" button in a single processor configuration; "release" is equivalent to a "go" button depression. In a multiprocessor configuration, the concepts assume more meaning in that actions equivalent to "stop" and "go" can be carried out under program control on one processor to "suspend" and "release" a process executing on another processor. As was mentioned before, these primitives will have meaning only when the affected processor is operating in the debug mode. While in this mode, special logic sequences can be invoked to accomplish single-step and phase-step processor operation, register content readout and alteration, breakpointing, etc. The associated fully-integrated processor/process capabilities are not discussed here.

F. Process Termination

The diagram of figure 4 contains the essential ingredients of process control and will be the basis for discussion in subsequent sections. However, it does not include the two termination states into which EXIT and ABORT take a process. The reason for not showing the termination states is that they overcomplicate the diagrams and add little to the concept. EXIT and ABORT are considered, however, in the implementation schemes to be developed.

EXIT can be invoked only by a process in the running state in its own behalf. The primitive will place the process in the "unload" state and activate a system unload procedure to return all of the resources allocated to the process. In addition, all incomplete activities, such as input and output, initiated by the process will be completed by the system. An ABORT will cause the specified process to be placed in the "post-mortem" state and will activate a system abort procedure. This procedure will perform various debug functions such as dumping main memory at machine registers. Upon completion, the abort procedure will invoke an EXIT, thus causing the process to be unloaded.

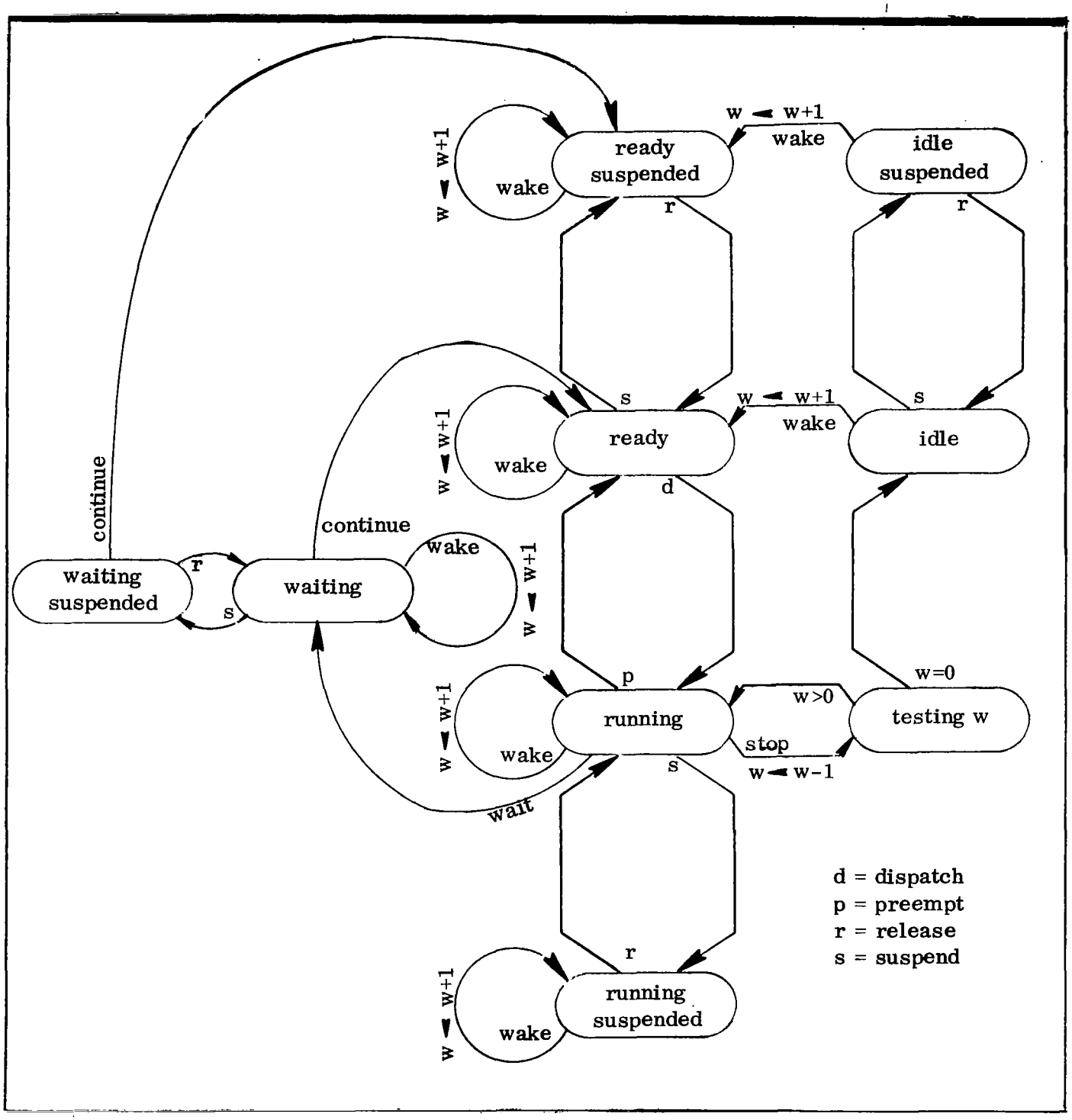


FIGURE 5. FINAL STATE TRANSITIONS

G. Example of Usage

Before proceeding with a discussion of implementation concepts, a specific programming example is offered to show the use of WAIT and CONTINUE. Shown below is an "intuitive-ALGOL" source-form listing of two cooperative procedures. One, named "free," is intended to illustrate a procedure for finding and allocating to the caller a block of contiguous main-memory words. It is a function that returns the address of the block to the caller; the caller specifies the size of the requested block as a formal parameter. "Putback" provides the means whereby a caller may return a previously allocated block to the system. In this way, blocks which are no longer needed by the caller are made available for allocation to future callers of "free."

```
                procedure free(request.size)
look:           if another.block
                then begin
                     look.at.size;
                     if too.small then go to look;
                     remove(block.address);
                     if too.big then insert.block(excess.block.address, excess.size);
found:               free:=block.address;
                     return.to.caller
                     end
                else begin
                     queue(request.size, processid, caller.return);
                ➤    wait;
                     get(block.address, caller.return);
                     go to found
                     end;
```

```
                procedure putback(block.address, block.size)
scan:           if more.queued.requests
                then begin
                     if this.queued.request.size  block.size then go to scan;
                     dequeue(this.queued.request);
                     fix.block(block.address);
                     fix.size(block.size)
                ➤    continue(processid);
                     if block.size = 0 then return.to.caller;
                     go to scan
                     end;
                insert.block(block.address, blocksize);
                return.to.caller;
```

"Free" is an algorithm that locates a block by looking through a list or map of available storage blocks until one large enough to satisfy the caller's request is found. When one is located, it may be too big, in which case the excess sub-block is placed back in the map by a call to "insert-block." In case no available block is large enough to satisfy the caller's request, the size of the requested block, along with the caller return address and a pointer to the caller's PCB, is queued to allow the system to proceed. Until the request can be satisfied, the process remains in the waiting state.

"Putback" has the job of updating the map which specifies available storage so that "free" is made aware of the returned space. "Putback" first scans the queue of requests to determine whether the block being returned will satisfy some waiting request. If a request can be satisfied, the waiting process is CONTINUED. "Putback" continues to scan the queued requests until there are no more requests to check, or all of the returned space has been used to satisfy queued requests. When scanning is complete, the remaining space is inserted in the storage map.

Several interesting points can be derived from the example. The most important is the implication that, in a multiprocessor system wherein the example code can be shared among several processors simultaneously, the code must be reentrant. The code of "free," for example, represents part of the mechanization of all processes having queued requests. This is an important example of code and data sharing.

The reentrant coding requirement could be eliminated by provision of duplicates of the code as part of the mechanization of every process. Since the algorithm is likely to be rather complicated due to "garbage collection" and memory map characteristics, the cost in storage space would probably be accessible to (shared by) all processes. This requires some form processor lockout capability such as "Test and Set" /1/9/. This same lockout capability could be used to obviate the need for multiple copies of man-reentrant code but would require special attention to process coding. Reentrant code combined with Test and Set applied to the global data is preferable.

---

[9]Blakeney, G. R., Cudney, L. F., and Eickhorn, C. R.: An Application-Oriented Multiprocessing System - Design Characteristics of the 9020 System. IBM Systems Journal, V6, N2, pp. 88, 1967.

SECTION IV. IMPLEMENTATION

This section outlines an implementation of the concepts of process control discussed previously. Two design approaches are developed for comparison purposes. The first is comprised of a combination of hardware and software; the emphasis is on digital logic. The second is predominantly software. While the hardware design is comprised of digital logic, and the software is depicted as a high-level language, it is important to realize that stored logic could replace any (or all) of the implementation media. This fact confirms the observation that there are no longer well-defined demarcations in the selection of implementation media for executive control. Such selections must be based on appropriate trade studies that include preliminary designs such as those discussed below.

Before depicting the actual implementation, it is necessary to discuss some concepts and techniques of general use as digital design aids. The applicability of transition matrices and the evaluation of control variables is outlined briefly to support the design rationale that follows.

A. Transition Matrix

The information shown in a state diagram can be represented in other forms that are often more compact. For the purposes of further discussion, an example is shown in table 2. This matrix depicts the important state transition aspects of the figure 4 state diagram with the exception of "d" and "p." (All future references to state diagrams developed in this report will imply that of figure 4 unless specifically indicated otherwise.) A state diagram is considered to be /10/11/ a representation for a finite state machine that has inputs, represented in these discussions by the primitives, and outputs, represented in report examples by increasing, decreasing, or doing nothing to the work variable. Table 3 shows an output matrix that could be used to control the value of "w."

---

[10] -: Minsky, M.: Computation: Finite and Infinite Machines. pp. 21, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1967.

[11] -: Chu, Y.: Digital Computer Design Fundamentals. pp. 375, McGraw-Hill Book Company, Inc., New York, N. Y., 1962.

TABLE 2. STATE TRANSITION MATRIX

Primitive	State / DC / BA	Idle 00	Ready 01	Running 10	Waiting 11
STOP	00	X	X	Y	X
WAKE	01	01•	01	10	11
WAIT	10	X	X	11•	X
CONTINUE	11	F	F	F	01•

F Should not happen. (Could be used to flag software/hardware error.)

X Cannot logically happen. (Could be used to flag hardware error.)

• Indicates state change.

Y Final state depends on w.

TABLE 3. WORK VARIABLE CONTROL

Primitive	State / DC / BA	Idle 00	Ready 01	Running 10	Waiting 11
STOP	00	X	X	-1	X
WAKE	01	+1	+1	+1	+1
WAIT	10	X	X	0	X
CONTINUE	11	F	F	F	0

Matrix element values:

-1 Implies decrement w.

0 Implies no change in w.

+1 Implies increment w.

X, F Same as in previous figure.

The "Testing w" state, in effect, does not exist as far as processes are concerned; it acts like a pseudo, or "transient," state in that while the machine is in this state, it is merely in the process of deciding, on the basis of "w," whether the required transition should be "10"-to-"10" or "10"-to-"00." When a STOP is invoked (in the running state), w will be decremented and tested immediately to determine which transition to make. This could have been indicated in the state diagram as shown in the illustration below.

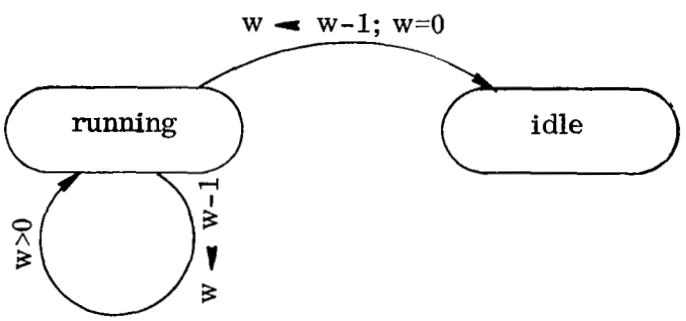


FIGURE 6. ALTERNATE FORM FOR STOP TRANSITION

WAKE is the only valid primitive that could conceivably be invoked while a process is in the "Testing w" state. There are several alternatives to controlling this situation. The simplest way would be to have an access lock placed on the value of w until it has been tested. After the proper transition has been made, the access lock would be removed. The only drawback to this scheme occurs in those cases where the process is placed in the idle state, only to have its state changed immediately to "ready" thus causing a large amount of work to be required to place it back in the running state. This problem arises only in a multiprocessor/multiprogramming environment, of course.

An alternative to the above scheme would be for each machine to set a control line indicating that it is executing a process in the "Testing w" state. All other process control machines would sense this line and delay further processing until the line is reset. For the particular conflict we are considering, this alternative has the advantage that no value access lock mechanism is required, the control line accomplishing the necessary control. However, the disadvantage of the first scheme is still inherent. This report will assume the second scheme although it is not necessarily best, and there may be other schemes. The "Y" shown in table 2 could be used to indicate that the "Testing w" control line should be set.

The "X's," shown as elements of both matrices, indicate combinations of input and state which are not valid by definition. These combinations would not normally go unrecognized, however, because of their value in detecting errors in a logic circuit designed to function like the finite machine. For this reason, a second output matrix might be constructed to control the detection of this class of error (caused by an input [e.g., primitive] or the machine [e.g., state] itself).

An "F" is used as an element to indicate those combinations which, although they could occur, would indicate another (possibly different from the "X") type of error. The relation between WAIT and CONTINUE is one which would be prohibited, by programming standards, from existing between more than one pair of processes or procedures at any given instance. Another, perhaps more precise, way of looking at this is to state that every WAIT has one and only one logically valid, matching CONTINUE. Also, every CONTINUE will always "unlock" (match) one and only one waiting process. An error would occur if some unauthorized process invoked a CONTINUE when, in fact, either the object (process) of the CONTINUE is not waiting or the event being waited upon has not occurred. The "F" indicator will flag the first of these two possible mismatches; a key of some sort would be required, as discussed previously, to flag the second.

B. Control Variables

In order to develop a valid logic device to represent the finite machine of the process control state diagram, it is necessary to define several output variables, each having an appropriate output matrix to define the variable values. The form shown in tables 2 and 3 is awkward for space reasons and a tabular form, sometimes called a "standard basis" /12/ truth table, is preferred. Table 4 is a standard basis for representing all combinations of A, B, C and D (from the transition matrix) in the first four rows. The column numbers at the top indicate the sixteen possible elements of the matrix of table 2. Although it may be confusing, these numbers are referred to, in conformance with generally accepted policy, as "state numbers." Thus, state 5 represents the occurrence of the WAKE primitive (B = 0, A = 1) when a process is "ready" (D = 0, C = 1).

---

[12] -: Digital Systems Engineering: pp. 1.28, Lecture Notes, 5th Edition, RCA Institutes, Clark, N. J., 1965.

TABLE 4. EXAMPLE STANDARD BASIS

VARIABLE	STATE			
	0 0 0 0 0 1 2 3	0 0 0 0 4 5 6 7	0 0 1 1 8 9 0 1	1 1 1 1 2 3 4 5
A	0 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1
B	0 0 1 1	0 0 1 1	0 0 1 1	0 0 1 1
C	0 0 0 0	1 1 1 1	0 0 0 0	1 1 1 1
D	0 0 0 0	0 0 0 0	1 1 1 1	1 1 1 1
X	1 0 1 0	1 0 1 0	0 0 0 0	1 0 1 0
F	- 0 - 1	- 0 - 1	0 0 0 1	- 0 - 0
N	- 1 - -	- 0 - -	a 0 1 -	- 0 - 1
G	- 1 - -	- - - -	0 - 1 -	- - - 1
H	- 0 - -	- - - -	b - 1 -	- - - 0
Y	- 0 - -	- 0 - -	1 0 0 -	- 0 - 0
W	- 1 - -	- 1 - -	1 1 0 -	- 1 - -
V	- 1 - -	- 1 - -	0 1 - -	- 1 - -

X: Invalid - flag hardware error.
F: Illogical - flag hardware/software error.
N: State change (• in transition matrix).
G: New C when N = 1.
H: New D when N = 1.
Y: Indicates entering Testing w. Set appropriate control line.
W: Change value of w (Test and Set could be used to gain access).
V: Increment (= 1) or decrement (= 0) value when W = 1.
-: Don't care.

Note: "a" is a 1 if and only if W = 0 when tested.
"b" is a 0 if and only if W = 0 when tested.

Each row below the top four represents an output variable whose possible (Boolean) values are shown at the intersections with the different states (columns). The X condition defined in the transition matrix and discussed above is an example of a condition that must be detected. Therefore, X is defined as an output variable whose value will be "1" (Boolean True) for those combinations of A, B, C and D which are invalid; i.e., cannot logically happen. Therefore, a "1" is placed at the intersection of the X row with states 0, 2, 4, 6, 12 and 14. All other state numbers are valid; X is therefore "0" (Boolean False) for these states.

Notice that for some state-number/row-value intersections in the standard basis, a "-" is shown. If a logic circuit is developed to provide values for all of the output variables, and the value provided for X turns out to be "1," the values of all other output variables are useless since some error has occurred. For this reason, "-" is used to indicate a "don't care" value for certain variables at certain state numbers. Furthermore, if a don't-care condition exists for a particular variable, it does not matter whether the logic circuit produces a "0" or a "1" for that variable in that condition. It is common practice in digital design to take advantage of don't-care conditions in order to minimize or simplify the overall logic circuitry. Simplification often occurs when a specific circuit is intentionally designed to produce, say, a "1" for certain don't-care state numbers.

C. Sum of Products

A simple procedure is available for the derivation of equations to evaluate the various output variables for all combinations of inputs. Referring to the standard basis, it is clear that X must be "1" when A, B, C and D are "0," A, C and D are "0" and B is "1," etc. It must be "0" for all other combinations.

Taking state 0 first, it is clear that if the False ("0") A, B, C and D values are negated and "ANDed" (logical product) together, the logical result is "1" (True). State 1 can result in a "1" if the False values of B, C and D are negated and ANDed together with the True value of A. State 3 suggests ANDing a negated C and D together with A and B, and so on. The input standard basis thus provides an indicator for obtaining the desired logical values for all output variables.

The logical entities thus formed are sometimes called "minterms" and are designated by a lower case "m" subscripted by the appropriate state number. Thus, /13/

$$m_0 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$$
$$m_1 = A\,\overline{B}\,\overline{C}\,\overline{D}$$
$$m_2 = \overline{A}\,B\,\overline{C}\,\overline{D}$$
$$m_3 = A\,B\,\overline{C}\,\overline{D}$$
$$m_4 = \overline{A}\,\overline{B}\,C\,\overline{D}$$
$$m_5 = A\,\overline{B}\,C\,\overline{D}$$
$$m_6 = \overline{A}\,B\,C\,\overline{D}$$
$$m_7 = A\,B\,C\,\overline{D}$$

are the first eight possible minterms. It is clear that each minterm has a True Boolean value provided the variables A, B, C and D have the values depicted in the standard basis for the subscript state number. In fact, it is also clear that for all combinations other than those of the appropriate state number, a given minterm will have a False value. That is, m_i is "1" if and only if the variables A, B, C and D have the values associated with state i.

Therefore, X can be evaluated by use of the expression

$$X = m_0 + m_2 + m_4 + m_6 + m_{12} + m_{14}$$

$$X = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,C\,\overline{D} + \overline{A}\,B\,C\,\overline{D} + \text{etc.}$$

This sum of products forms the basis, therefore, for a computer program or a logic circuit to evaluate X. Expressions can similarly be derived for all of the other output variables.

Considerable simplification of expressions such as that shown for X will accrue from factoring and the application of deMorgan's Theorem. Additional simplification can result from the use of Karnough maps, Mahoney maps, and other derivatives of the Venn diagram. These simplifications are "tricks of the trade" for digital systems designers and are not discussed here although the serious systems programmer should master at least one of the set of techniques (see, for example, reference 11). Figure 7 shows a somewhat simplified logic circuit that will evaluate X using AND and OR gates. A map is also shown with minterms shaded according to the standard basis. The value of X can be used as input to a flip-flop to interrupt further processing or to cause a trap to a fault diagnosis program.

---

[13] The logical product is denoted by simple concatenation; the logical sum is denoted by "+." "$\overline{V}$" is taken to mean "not V."

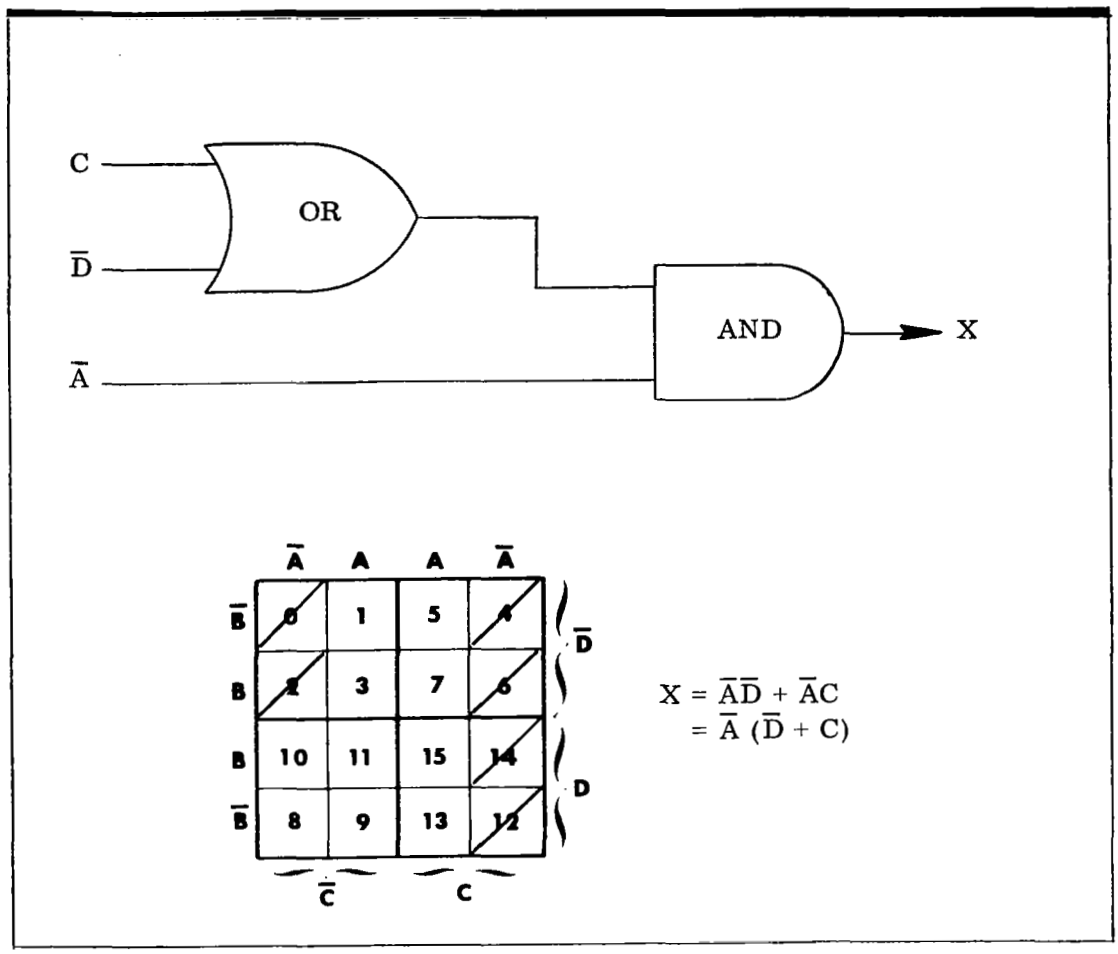


FIGURE 7. EVALUATION OF X

D. Processor Control

The output variables shown in Table 2 are defined in that table. The justification for each of these variables is clear from the discussions since they were all derived from the matrices of tables 2 and 3. Several additional input and output variables are required in order to enable a complete definition of the process control mechanism. These variable requirements are derived from further consideration of the dispatching function and processor state transitions.

1. Dispatcher. Previous discussions have mentioned a dispatching function without going into detail as to what it does. While it is not desirable to explicitly define a particular dispatcher, its main features can nevertheless be outlined. The dispatcher is an algorithm or set of rules for matching a process in the ready state with a system processor. Once a match has been made, the process can begin executing. Since there will normally be more than one process in the ready state, the dispatcher is actually an algorithm for deciding which "ready" process to dispatch and then making the appropriate change to the state of the affected processor and process to start execution if a process is dispatched.

a. Ready List. The dispatcher represents a portion (sometimes large) of system overhead. It should therefore be able to operate efficiently. It cannot be defined efficiently if it examines the various PCB's in the system to make its decisions. For this reason, a separate list containing an entry for each process that is not idle or waiting is usually defined. Entries in this list, often referred to as the "ready list" or "switch list," contain the addresses of (pointers to) the PCB corresponding to their associated processes. This list effectively isolates the dispatching algorithm from the order and structure of PCB's and thereby makes it possible to order and structure the ready list to enable optimization of the dispatching algorithm.

In order to construct the ready list, a procedure is necessary for the creation of an entry for a specified process and insertion of the entry into the ready list at the appropriate place. Each time a process makes the state transitions "idle-to-ready" and "waiting-to-ready," this procedure must be invoked by the process control mechanism. Conversely, a procedure must be defined and properly invoked to remove an entry from the ready list and destroy it.

When a process is running, it may remain in the ready list for convenience, but with its state indicator showing that it is running instead of ready. An alternative is to maintain two lists: a ready list for processes in the ready state, and an active, or running, list for processes in the running state. If

this scheme is used, the "insert" procedure discussed above inserts entries in the ready list as mentioned; but the "remove" procedure removes entries from the running list. In either case, a separate procedure is necessary and sufficient for removal and insertion, since RELEASE and SUSPEND (involving a different pair of procedures) are not part of the current discussion.

Based on the discussions thus far, it is evident that a variable is required to trigger activation of the dispatcher as a result of primitive operations. Furthermore, a variable is necessary to signal whether an entry must be placed into or removed from the list(s). The dispatcher, of course, or some other procedure, such as a priority control procedure, may manipulate and reorganize the list(s).

b. Dispatcher Overview. It was mentioned previously that special privileged primitives are required by the dispatcher to dispatch and preempt processes. Through the proper design approach, both of these primitives could be incorporated into a single primitive to switch execution from one process to another. In a single processor multiprogramming system this approach would perhaps be adequate. However, in a multiprocessor, multiprogramming system, a process might stop when there is no other ready process to which a switch can be made. For this reason, and because of the greater inherent flexibility, the two controls are kept separate in this report.

Figure 8 shows an overview to a processor allocation scheme showing the various functions of a dispatcher. Control is passed to the dispatcher through a mechanism referred to as a "trap." This mechanism, a list-driven processor capability activated by the process control mechanism, is discussed below under Trap Processing.

The basic functions of the dispatcher as shown in figure 8 are self-explanatory. A given processor is considered, for simplicity, to have only two possible states. These are referred to as "executing" and "stopped." When a processor is stopped, it can be placed in the executing state only by a DISPATCH primitive. This primitive specifies an identifier for both the processor and the dispatched process (Processid in Figure 1).

When a processor recognizes a dispatch signal, it will:

- Load the address of the associated PCB from a prespecified memory location,
- Load its volatile registers from the process-associated PCB machineregister space,
- Store its identification in the CPUnumber space of the PCB,

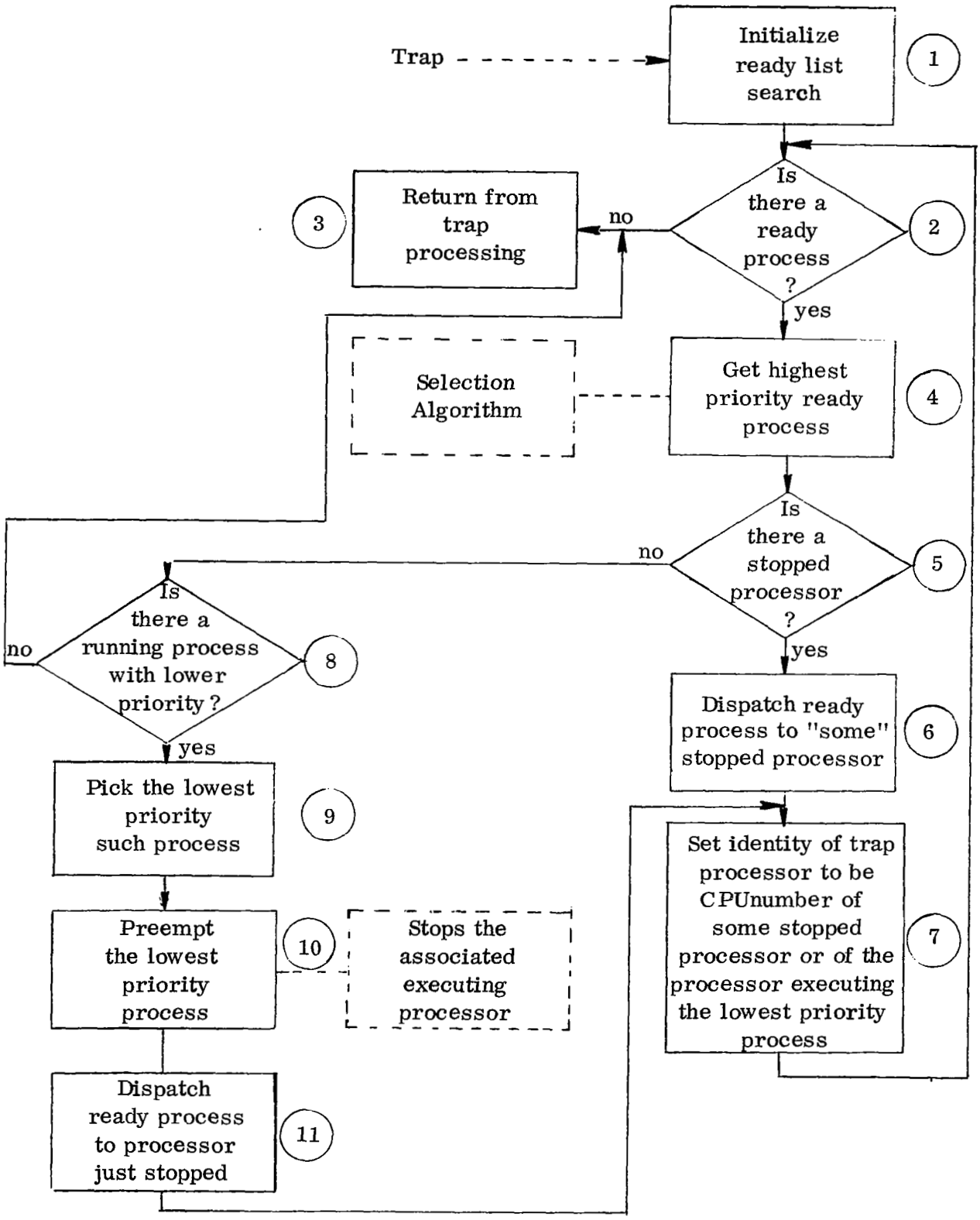


FIGURE 8. DISPATCHER OVERVIEW

- Load the memory address register from Returnaddress in the PCB,
- Start an instruction fetch cycle.

This sequence of operations is referred to as a dispatch cycle. The dispatch signal is transmitted to the affected processor by the processor executing the DISPATCH primitive. In the case where the affected processor is the one executing the primitive (i.e., it is executing the dispatcher), no processor-to-processor communications is required and the primitive is not recognized until trap processing is complete.

A PREEMPT primitive specifies the identification of the affected processor. When the processor recognizes a preempt signal, it will sequence through the preempt cycle as follows:

- Save its volatile registers in the PCB machineregister space for the process it is executing,
- Erase its identification from the PCB CPUnumber space, and
- Stop.

2. Trap Processing. A trap is a transfer of processor control to a specified location as a result of some event or condition requiring special attention. From the viewpoint of process control, trap processing offers a flexible and efficient means of initiating certain procedures, such as the dispatcher, or the "remove" and "insert" procedures that operate on the ready list as mentioned previously.

Each processor is required to have a one-bit trap designator register (TDR) indicating whether it is the trap processor or not. Only one processor in a multiprocessor, multiprogramming system will be designated at any given time as the trap processor. The selection and designation is controlled by the dispatcher as shown in box 7 of figure 8. Selecting in this way, on the basis of processor state and process priority, insures that trap processing, a system overhead item, interferes only with the lowest priority process and then only if there is no stopped processor.

Traps are implemented by each processor upon recognition of an event requiring software support. The mechanism consists of first placing a pre-specified main memory location (entry point of a support routine) in a list that is normally treated as a first-in-first-out (FIFO) stack; and then setting a

trap control line. The stack beginning address and ending address are stored either in common main memory or local scratch-pad memory for each processor. Also, the first word in the stack is reserved for the location of the first unused stack word.

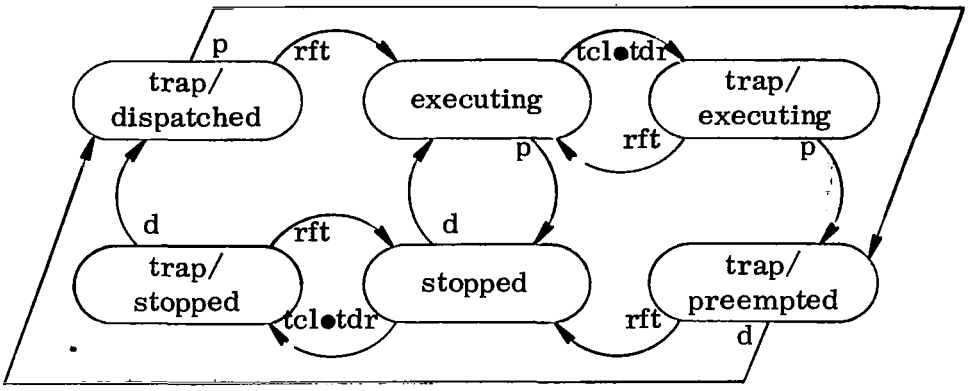
At the completion of an instruction sequence, all processors match (by use of a logical product) the trap control line (TCL) with their trap designator register; a resulting "1" will cause the designated processor to access the FIFO stack for the trap instruction fetch address (a policy may be invoked to save certain machine registers also). The trap control line is not reset until the trap stack is empty. Upon completion of the processing of each trap, the trap processor must check to see if it is still the designated trap processor, since the designation may have been changed during processing. Also, a trap processing line (TPL) must be provided to prevent a newly designated processor from taking action during trap processing. The need for this is dictated by the requirement in the reported approach to maintain the FIFO order. Perhaps this requirement could be relaxed by a different approach. To complete the scheme, it is necessary to provide a trap stack access control capability to lock-out access by other processors (attempting to place a trap entry on the stack) when the designated processor is removing an entry.

In summary, the trap processing discussion shows a need for the following controls:

- Trap designator register (TDR),
- Trap designator register set line (TDS),
- Trap processing inhibit line (TPL),
- Trap control line (TCL),
- Trap stack and manipulation capability, and
- Trap stack access lock-out capability.

3. Processor States. In order to show more clearly the relationship between processes and processors, the effect of control signals and trap processing is displayed in the state transition illustrations of figure 9. Transitions are caused by

- d - the dispatch primitive used to set processor dispatch control lines through processor-to-processor communications,
- p - the preempt primitive used to set processor preempt control lines,



a. State Diagram

control signal \ state	e	te	s	ts	tp	td	N	N
d	X	X	e	td	td	X	-	-
p	s	tp	X	X	X	tp	-	-
rft	X	e	-	s	s	e	-	-
tcl•tdr	te	-	ts	-	-	-	-	-

b. Transition Matrix

X: hardware error or illegal use of primitive or control signal
N: not used
-: not checked/not possible/don't care

FIGURE 9. PROCESSOR STATES

- TCL•TDR - for each processor this is the logical product of the trap control line and the processor's trap designator register contents, and
- rft - the return from trap processing instruction.

In these illustrations, "p" will cause a preempt cycle after which the processor state is changed; "d" causes a dispatch cycle; and "rft" tests the logical product of the trap control line level and the contents of the processor trap designator register. When "rft" finds the product to be a Boolean true, it forces the transition to either "te" or "ts" so that the corresponding "e" and "s" states become transient states in this case.

The symbols "p" and "d" play a dual role in that they are interpreted as either executed primitives or control line contents. In the case when a processor executes a "p" or "d" primitive, the control logic for that primitive is invoked. These primitives provide, as arguments, the identification of the affected processor. In the case of "d," the identification of the process to be executed is also provided as an argument.

When a processor recognizes that either its "p" or "d" control line is at the "1" level (these lines might be checked at the end of each instruction sequence), the control logic for processor state transitions is invoked. Two cases (one with two subcases) must be considered in the design of the control logic:

- Case 1: The processor has received and recognized a "1" level on either its "p" or "d" control line; in this case, its processor state transition control logic is invoked. This logic then invokes the process control state transition logic (to be developed and depicted in figures 10 and 12) if the initial processor state was "executing" (in the case of a recognized "p") or "stopped" (in the case of a recognized "d").
- Case 2: The processor has fetched and must execute a "p" or "d" primitive; the primitive logic verifies that the primitive is not an illegal instruction. Then, two subcases are considered:
- Case 2a: The "p" or "d" primitive argument processor is not that of the cognizant processor; in this case, the argument processor's "p" or "d" control line (whichever is appropriate) is set.

- Case 2b: The "p" or "d" primitive argument processor is that of the cognizant processor; in this case the processor invokes its processor state transition control logic. It then checks to see if it was initially in either the trap/executing or trap/stopped state. If so, the process control state transition logic (figures 10 and 12) is invoked. In the case of a "p," the logic shown in figure 12 will trigger a "stop processor" function (F21). F21 would be disabled in this case (only) by the processor state transition control logic prior to invoking the process state transition control logic.

When a processor executes a "p" or "d" primitive, a validity check is first made to insure that the processor is in a trap/"anything" state. Since the "p" and "d" primitives are regarded as privileged, they will be treated as illegal instructions if they appear in an instruction stream while the processor is in the executing state. Once this check has been made, the logic for these primitives checks the CPUnumber (or processor identification) associated with the primitive. If the processor's own identification does not agree with that of the primitive argument, the "p" or "d" control line associated with the argument processor is raised (set) to the "1" level. It is the recognition of this "p" or "d" control line which causes a subsequent state transition by that processor.

If the processor executing the "p" or "d" primitive finds that it is the argument processor, it invokes the processor state transition control logic as if it had recognized an associated control line set by another processor.

Although development of the control logic for processor state transitions is beyond the scope of this report, sufficient detail has been presented to clarify the impact of the "p" and "d" primitives on processors. Also, the controls for trap processing can be designed easily on the basis of the discussion.

E. Hardware Implementation

The state transition matrix shown in table 5 is an expansion of table 2 to include the primitives for dispatching, preempting, exiting, and aborting. Additional states should be included to show that a process is terminated or suspended. However, these additions expand the number of standard basis states beyond the point where mapping for Boolean expression simplification is trivial. The intent of this report is to indicate the necessary considerations, and work through a reasonable subset of the required states

TABLE 5. EXPANDED MATRIX

Primitive	State / ED / CBA	Idle 00	Ready 01	Running 10	Waiting 11
STOP	000	X	X	Y	X
WAKE	001	01●	01	10	11
WAIT	010	X	X	11●	X
CONTINUE	011	F	F	F	01●
DISPATCH	100	X	10●	X	X
PREEMPT	101	X	X	01●	X
EXIT	110	X	X	T●	X
ABORT	111	R●	R●	R●	R●

X, F, Y, ● - Same as Figure 6.
T - Stack exit and mark terminated.
R - Stack abort and mark terminated.

to show comparative results for both hardware and software approaches. It is felt that the requirements of table 5 accomplish this objective without overcomplicating the techniques.

1. Logic Network. Table 6 gives the variable designations corresponding to table 5 in a standard basis form. The maps in Appendix A were used to derive the expressions shown in table 7. Note that don't-care minterms are shaded in the maps as if they were "1s" to simplify expressions. Figure 10 is a feasible logic network that evaluates the expressions of table 7. Network elements are shown only for X, F, N and G to illustrate the scheme.

Operation is initiated by input of a suitable (logic "1") pulse, P, which goes into the R-S flip-flop (FF) on its set line, S. This FF is initially in the reset (or clear) state wherein the output line Q is set to "0" thus disabling the AND gates shown directly above it. A logic diagram of the R-S FF is shown in figure 11. The Q line is labeled "1" to show that a "1" input on the S line will cause this line to go to a "1." It will stay this way until a "1" is subsequently input on the R line to clear it (set it to "0").

The Boolean values of A, B, C, D and E are input to their respective enable/disable AND gates so that, when P arrives, they will be applied, along with their complement values, to the network to give values for X, F, etc. Inputs A, B, C, D and E are assumed to be logic levels that are held at their value (possibly by flip-flop registers). Therefore the outputs X, F, etc. and their complements are also levels and will be held at their value from the time P arrives until a reset pulse is input on the R terminal of FF. A finite time is required for the outputs to achieve their correct values after P arrives. This time delay is a characteristic of the logic and is duplicated in the dashed box labeled F2. The delay will cause the start pulse, P, to be delayed until the outputs are stabilized, at which time the P-pulse appears as an output on line P'. P' is used to signal that the network is ready to supply values for X, F, etc.

2. Process Control Sequencing. The output variables have a hierarchical or precedence relationship as follows: X, F, I, Y, W, V, N, G, H, S, M, T, R, J. For instance, if X is "1," no other variables have meaning. In this case, a trap to a hardware failure procedure is stacked and X is used to stop the processor. If X is "0," an F value of "1" is allowed to cause a trap to be stacked and the processor to be stopped. If both X and F are "0," a "1" value for I will delay the processor until the Testing w control line (indicated in succeeding figures by "L") is reset, whereupon E and D are accessed again and processing cycled. When I is "0," a Y value of "1" will cause the Testing w control line to be set, etc.

The reason for precedence is the need to place entries on the trap stack in the proper order. Therefore, the actions necessary for process control must be sequenced in the proper order. The actions are shown in

TABLE 6a. EXPANDED STANDARD BASIS

VARIABLE*	STATES							
	0000 0123	0000 4567	0011 8901	1111 2345	1111 6789	2222 0123	2222 4567	2233 8901
A	0101	0101	0101	0101	0101	0101	0101	0101
B	0011	0011	0011	0011	0011	0011	0011	0011
C	0000	1111	0000	1111	0000	1111	0000	1111
D	0000	0000	1111	1111	0000	0000	1111	1111
E	0000	0000	0000	0000	1111	1111	1111	1111
X	1010	1110	1010	0110	0000	1000	1010	1110
F	-0-1	---0	-0-1	0--0	0001	-000	-0-0	---0
N	-1--	---1	-0--	1--1	a01-	-111	-0-1	---1
G	-1--	---c	----	0--c	0-1-	-1cc	---1	---c
H	-0--	---c	----	1--c	b-1-	-0cc	---0	---c
Y	-0--	---0	-0--	0--0	100-	-000	-0-0	---0
W	-1--	---0	-1--	0--0	110-	-000	-1--	---0
V	-1--	----	-1--	----	01--	----	-1--	----
S	-1--	---0	-0--	0--1	a01-	-011	-0-1	---0
M	-1--	----	----	---0	0-0-	--00	---1	----
T	-0--	---0	-0--	0--0	000-	-010	-0-0	---0
R	-0--	---1	-0--	0--1	000-	-001	-0-0	---1
I	-0--	---0	-0--	0--0	010-	-000	-0-0	---0
J	-0--	---0	-0--	0--0	a011	1111	-0-0	---0

Note: "a" is a 1 if and only if W = 0 when tested.
"b" is a 0 if and only if W = 0 when tested.
"c" indicates terminated state.

*Variable definitions are shown in Table 6b.

TABLE 6b. CONTROL VARIABLES

Symbol	Definition
X	Invalid - flag hardware error.
F	Illogical - flag hardware/software error.
N	State change (● in transition matrix).
G	New D when N = 1.
H	New E when N = 1.
Y	Indicates entering Testing w. Set appropriate control line.
W	Change value of w (Test and Set could be used to gain access).
V	Increment (= 1) or decrement (= 0) value when W = 1.
S	Activate dispatcher.
M	Remove (= 0) or insert (= 1) a ready list entry when S = 1.
I	Check Testing w control line. If Testing w control line is set, a delay occurs. After the delay, another access of the process state is made and the control logic is invoked again.
J	Stop this processor after setting trap indicator or saving machine registers.
T	Stack exit and set state to indicate terminated.
R	Stack abort and set state to indicate terminated.
-	Don't care.

TABLE 7. SUM OF PRODUCTS EXPRESSIONS

Variable	Boolean Expression
X	$\bar{A}\bar{C}\bar{E} + A\bar{B}C\bar{E} + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{E} + \bar{A}DE + \bar{B}CDE$
F	$B\bar{C}\bar{E} + AB\bar{C}\bar{D}$
N	$\bar{D}\bar{E} + B + C$
G	$\bar{C}\bar{E} + \bar{B}CE + B\bar{C}$
H	$\bar{B}D + \bar{A}\bar{C}$
Y	$\bar{A}\bar{B}\bar{C}$
W	$A\bar{C} + \bar{B}\bar{C}$
V	$A\bar{C}$
S	$\bar{C}\bar{D}\bar{E} + BD\bar{E} + B\bar{D}E + B\bar{C}$
M	$\bar{C}\bar{E} + D\bar{C}$
I	$A\bar{C}\bar{D}E$
J	$C\bar{D}E + B\bar{D}E$
T	$\bar{A}C\bar{D}$
R	ABC

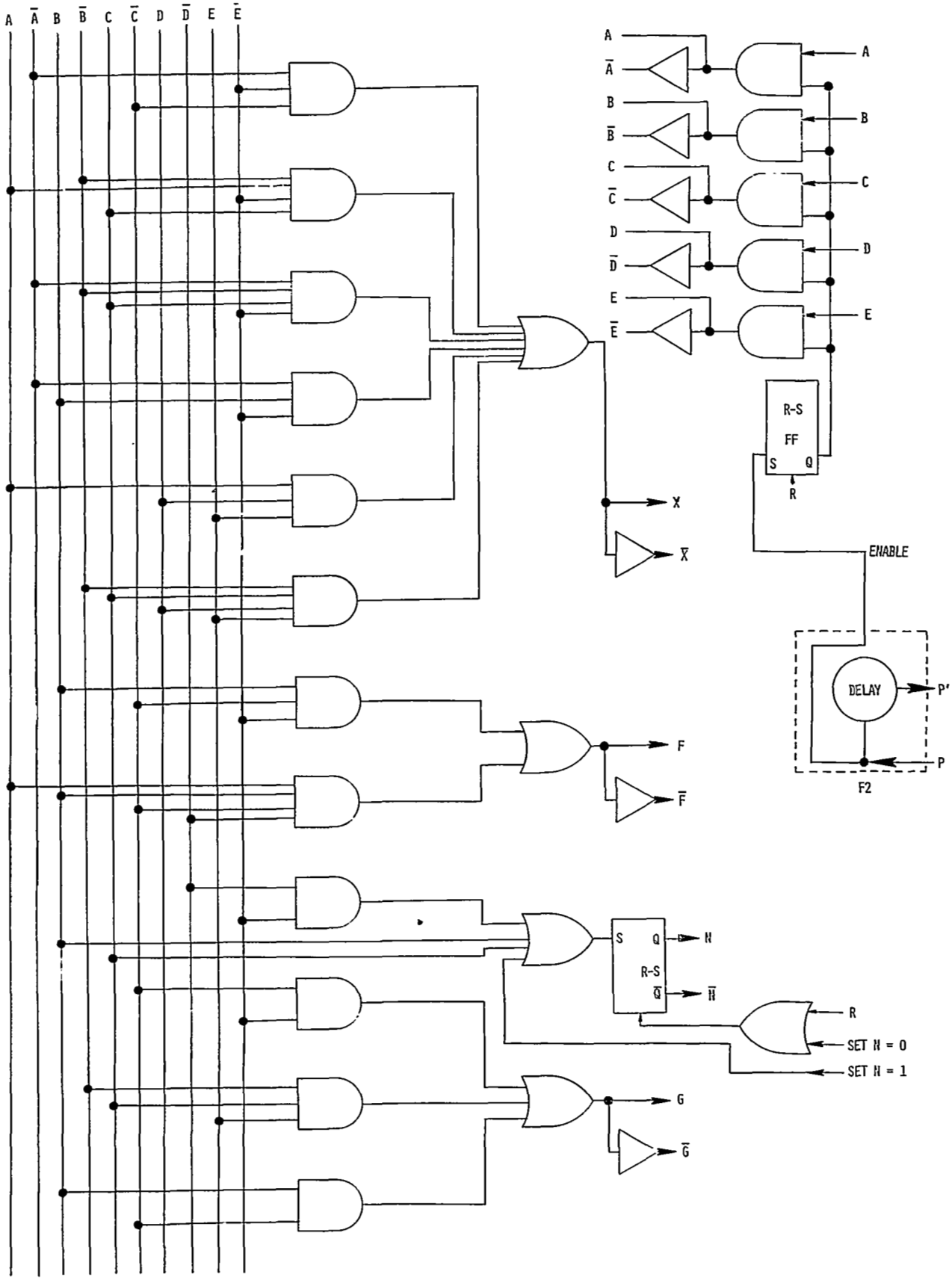


FIGURE 10. LOGIC NETWORK FOR PROCESS CONTROL

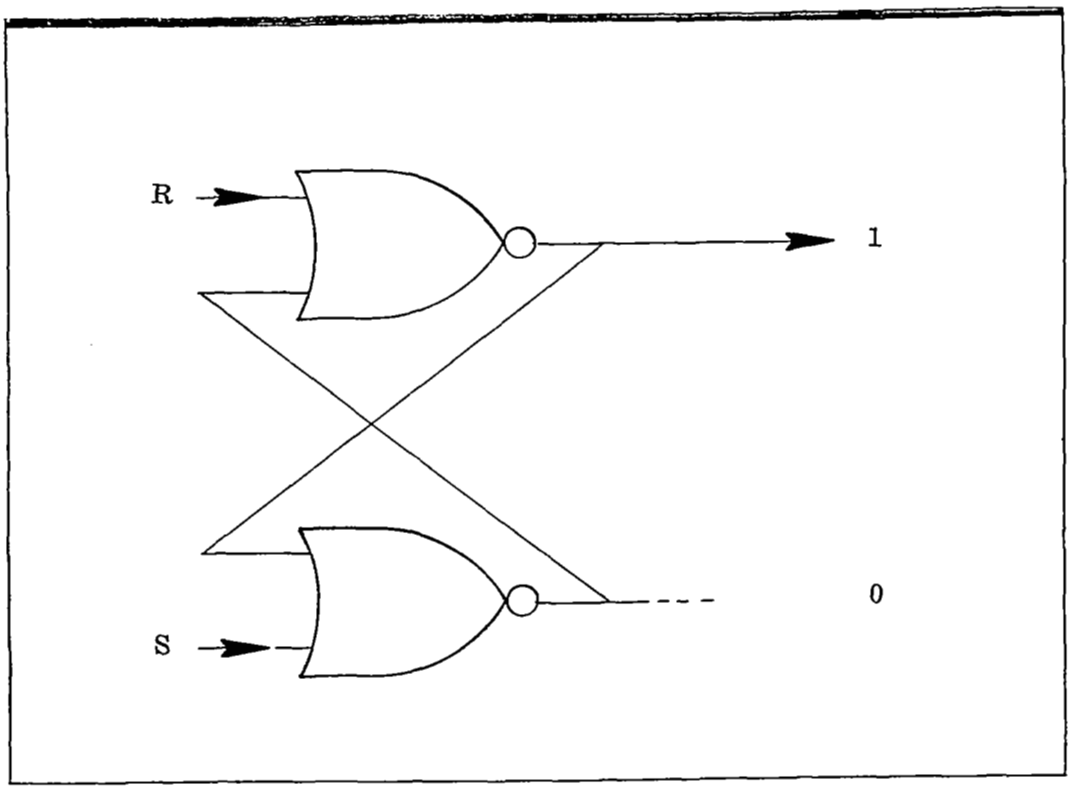


FIGURE 11. R-S FLIP-FLOP

table 8 as a set of ordered functions. Notice that the functions at steps 10 (F10) and 11 (F11) involve setting the values of certain of the variables previously evaluated by the network of figure 10. The network arrangement for the variable N shows how this capability can be implemented. The network reset signal R is applied through the OR-gate to the FF associated with N, thereby presetting it to "0." The logic network will subsequently set it to a "1" if and only if N's value is "1." When F10 or F11 is subsequently activated, a "1" pulse can be fed back to the network to insure that future samples of N will be the desired value.

Software support procedures have been specified to take action on certain conditions as follows:

- HWFAULT - A hardware fault has occurred.
- HSFAULT - A hardware or software fault has occurred.
- INSERT - Insert an entry in the ready list for the specified process.
- REMOVE - Remove an entry from the ready list for the specified process.
- DISPATCHER - Review all ready processes for possible changes in processor allocation and make all necessary changes. Designate the trap processor.
- EXITI - Initiate the removal of the specified process from the process control domain and release of all allocated resources.
- ABORTI - The specified process has aborted, or has been aborted by another process. Initiate the appropriate diagnostic action (the least should be a short dump). Preempt the process if it did not abort itself.

These support procedures are executed by the designated trap processor. If a particular application is completely predetermined with respect to one or more of the above procedures, it would be possible to implement part or all of them in digital logic or microcode depending on the processor design. It is not advisable, however, to assume that flexibility is not needed; the open-ended approach is clearly superior for general usage.

Figure 12 gives a sequence control logic diagram that shows the scheme for ordering the functions of table 8. Control is accomplished

TABLE 8. STEPS IN CONTROL SEQUENCE

STEP	FUNCTION		
0			Generate or receive start signal.
1			Fetch state of process.
2			Invoke logic of Figure 10.
3	X = 1	implies	Stack HWFAULT; continue sequence at STEP 20.
4	F = 1	implies	Stack HSFAULT; continue sequence at STEP 20.
5	IL = 1	implies	When L = 0, continue sequence at STEP 0. Delay while L = 1.
6	Y = 1	implies	Set L to "1."
7	W = 1	implies	Fetch w.
8	WV = 1	implies	Increment w. Continue at STEP 12.
9	$W\bar{V}$ = 1	implies	Decrement w.
10	Test w		(w = 0). Set S = 1, N = 1, J = 1, H = 0.
11	Test w		(w > 0). Set S = 0, N = 0, J = 0.
12	N = 1	implies	Store new state HG.
13	$\bar{S}$ = 1	implies	Continue sequence at STEP 17.
14	M = 1	implies	Stack INSERT; continue sequence at STEP 16.
15	$\overline{M}$ = 1	implies	Stack REMOVE.
16			Stack DISPATCHER.
17	T = 1	implies	Stack EXITI.
18	R = 1	implies	Stack ABORTI. Set J=0 if required.
19	Y = 1	implies	Clear L to "0."
20	X+F+S+T+R = 1	implies	Set trap control line.
21	X+F+J = 1	implies	Stop Processor.
22			End of Sequence. Reset Flip-Flops. Activate next instruction fetch cycle.

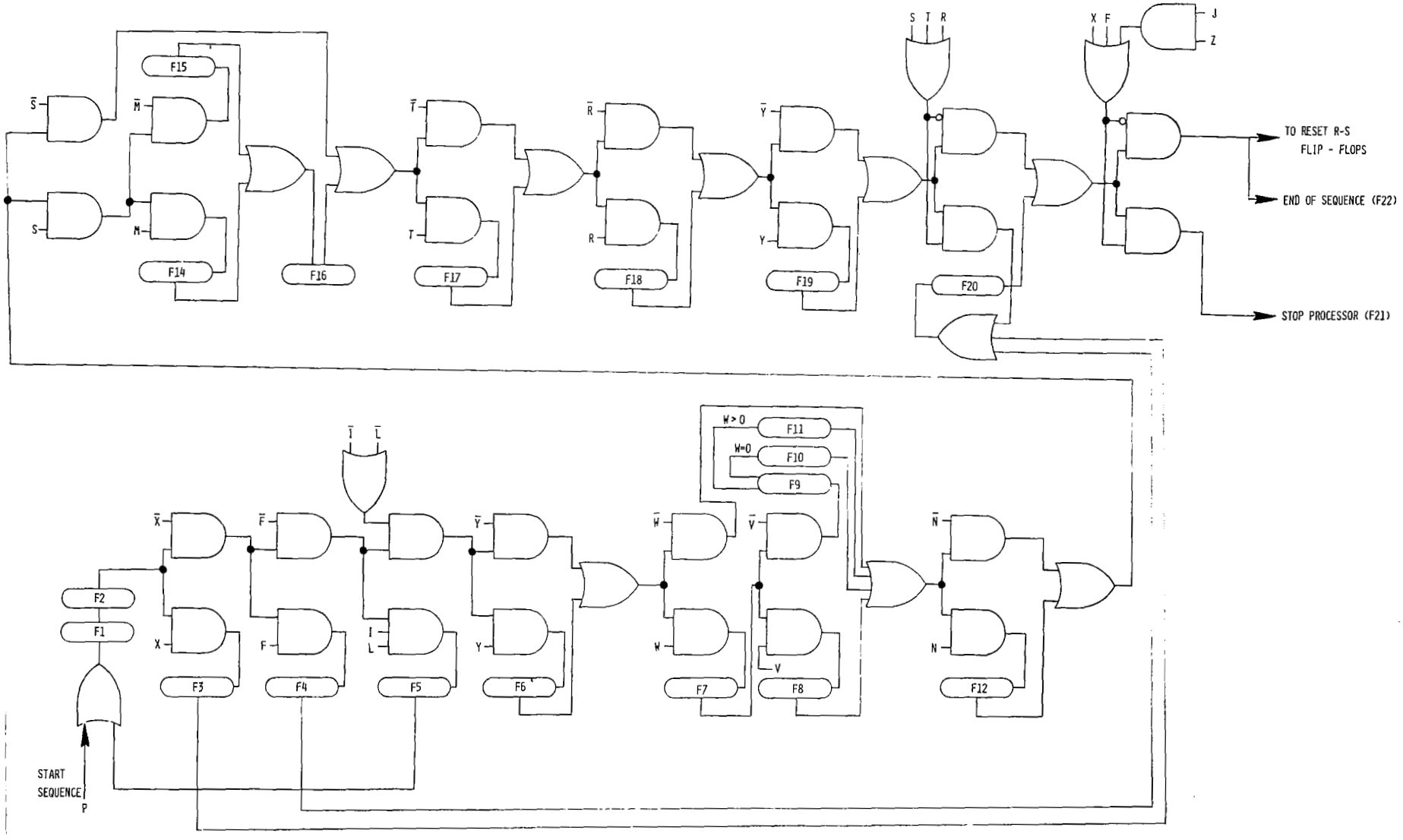


FIGURE 12. PROCESS CONTROL SEQUENCING

through propagation of a "start sequence" pulse (labeled P) through the circuit. The pulse is "steered" to the appropriate functions in proper order by pairs of AND-gates that have been conditioned by network variable values and their complements. Z is a disable/enable control to allow the processor control logic (discussed previously) to determine whether a stop is to occur when J = 1. The gate is normally enabled, but should be disabled when a preempt occurs under some processor state conditions. Logic flow for the various functions is outlined in Appendix B.

By way of summary, figure 13 shows conceptually the relation of the logic discussed with the instruction decode circuitry. It should be clear that some degree of parallelism could be incorporated; in the interest of simplicity, no attempt has been made to show advanced concepts such as instruction overlap or stacks, pipelining, parallelism, etc. Also, consideration must be given to several important questions such as: what policy prevails in the case when the trap processor detects a fault condition; and, is it reasonable to assume that the processor that detected a fault condition is the offender?

F. Software Implementation

The concepts of process control have historically been implemented with software. The reasons are numerous; the foremost, perhaps, is that they were not understood well enough to permit otherwise. Furthermore, formalism of operating systems has only recently attracted attention and, as shown in this report, can be extended to include many previously ill-defined programming techniques.

Because the author is not aware that any precedence has been set in the hardware implementation of process control, considerable attention has been given to digital logic formulation and design. This is not to imply that the hardware design techniques are in any way complex or unique; quite the opposite is true, a fact that tends to support the premise that such executive control capabilities can readily be transferred from the domain of software to that of hardware.

In order to emphasize the hardware approach by contrast, the design concepts are presented below in an abbreviated software form. The presentation is comprised primarily of high-level flow charts and source language procedures. While no particular computer is felt to be specifically designed to enhance the software approach, the UNIVAC 1108 Multiprocessor System /14/

---

[14] 1108 Multiprocessor System; System Description. UP-4046 Rev. 1, UNIVAC Data Processing Division, Sperry Rand Corporation.

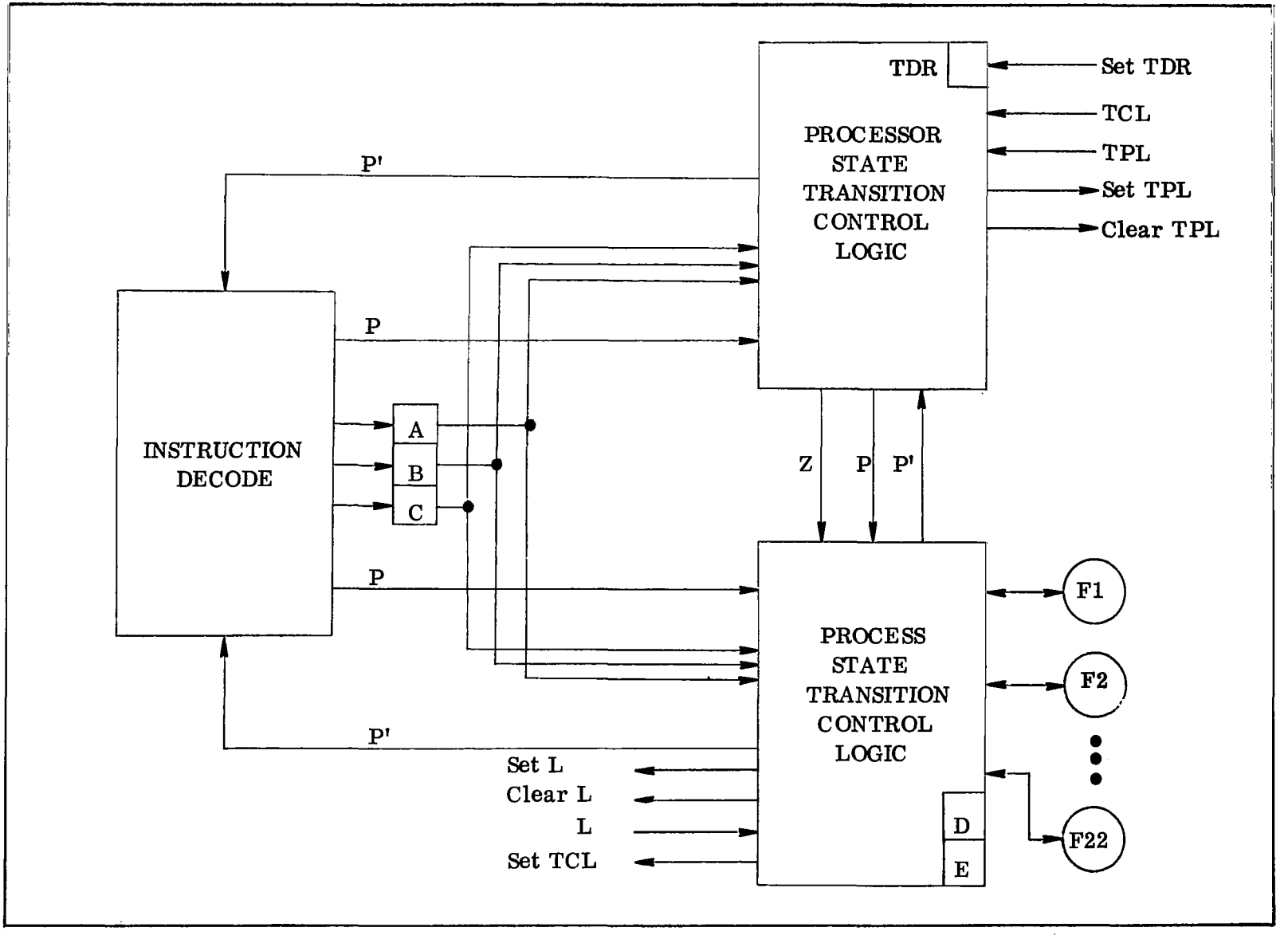


FIGURE 13. LOGIC OVERVIEW

is selected as being representative of the class of system the discussion is directed toward.

Briefly, the maximum configuration is represented by five processors (three central processors and two input/output processors) and four memory modules of 65,000 words (36 bits) each. Central processors, under executive control, can signal one another through the execution of a special executive (privileged) instruction known as the "Initiate Interprocessor Interrupt (III)." This instruction enables the executive, while being executed by processor A, to interrupt processor B ($\neq$A) or C ($\neq$A) for the purpose of assigning them to the execution of arbitrary tasks. If the identity of A is known by the executive, the identity of the interrupted processor can be determined.

Other specific executive instructions that are needed in the approach to be depicted are:

- Load Storage Register (LSR)
- Load Processor State Register (LPS) and
- Allow All Interrupts and Jump (AAIJ).

LSR provides the means for setting the memory access boundaries to those of a given process. LPS enables the executive to set the state of a processor such that certain privileged instructions and capabilities will be prohibited to insure system protection. Finally, AAIJ is the jump instruction which, when used in conjunction with LSR and LPS, transfers processor control to a process return address or entry point. In addition to the above special instructions, each processor is equipped with a set of unique control registers which can be used by the executive to save processor-related variables such as identification, etc.

Because of the nature of the instructions available to implement the software approach, the mechanisms naturally assume a form different from that of the hardware approach. From the potential user's viewpoint, the primitives become, in the software implementation, "executive requests," i.e., "supervisor calls," rather than instructions. Furthermore, "preempt" and "dispatch" are functions initiated directly by the dispatcher, and are therefore neither instructions nor supervisor calls. And, finally, traps are implemented by use of the III instruction. Except for these changes, the process control concepts remain the same. While it is possible (and perhaps even desirable) in the software case to follow a vastly different approach than the scheme outlined in the hardware implementation, an intentional effort was made to keep the two approaches as near the same as possible for comparison purposes and to ease the transition for readers not heavily oriented toward programming.

Figure 14 gives flow diagrams of the executive requests for primitive functions considered in the hardware implementation (except PREEMPT and DISPATCH). All primitives require similar testing in order for the validity of the request to be checked. For this reason, they are organized as "set-up" procedures; each primitive is a unique procedure that performs initialization and then transfers processor control to a common procedure for validation and action. Figure 15 gives an overview to the common procedure "CONTROL."

CONTROL plays the role of the logic of figures 10 and 12. It evaluates the Boolean expressions, and tests their values in order to make branching decisions to perform the appropriate functions. In order to perform the functions implied by HWFAULT, HSFAULT, EXIT, and ABORT, the entry address and arguments for initialization routines for each of these procedures is queued in much the same way as in the hardware case. REMOVE, INSERT, and DISPATCHER calls are made directly because their frequency of usage is assumed to be higher and their program size smaller than those of the four queued entries.

In the case of a "preempt" or "dispatch" primitive, a unique code is queued by the dispatcher along with the identification of the affected cpu. Whenever entries are queued, CONTROL will initiate an interprocessor interrupt requiring the trap processor to service the queue. An overview to this capability is shown in figure 16. Notice that the interrupted processor checks first to see if it has a queued "preempt" or "dispatch" primitive. If not, it is the trap processor and therefore executes the queued initiation routines.

Figure 17 gives approximate source-form listings of the statements that make up the various procedures. The listings are sprinkled with comments to make them self-explanatory for the most part. The own declarator was used to approximate the need for all procedures to be reentrant since all processors may be executing the same code simultaneously. Those procedures not explicitly defined are defined implicitly by context or by comments. See reference /15/ for a description of ALGOL.

---

[15] Naur, P., et al: Revised Report on the Algorithmic Language ALGOL 60. Comm of ACM, V6N1, January 1963.

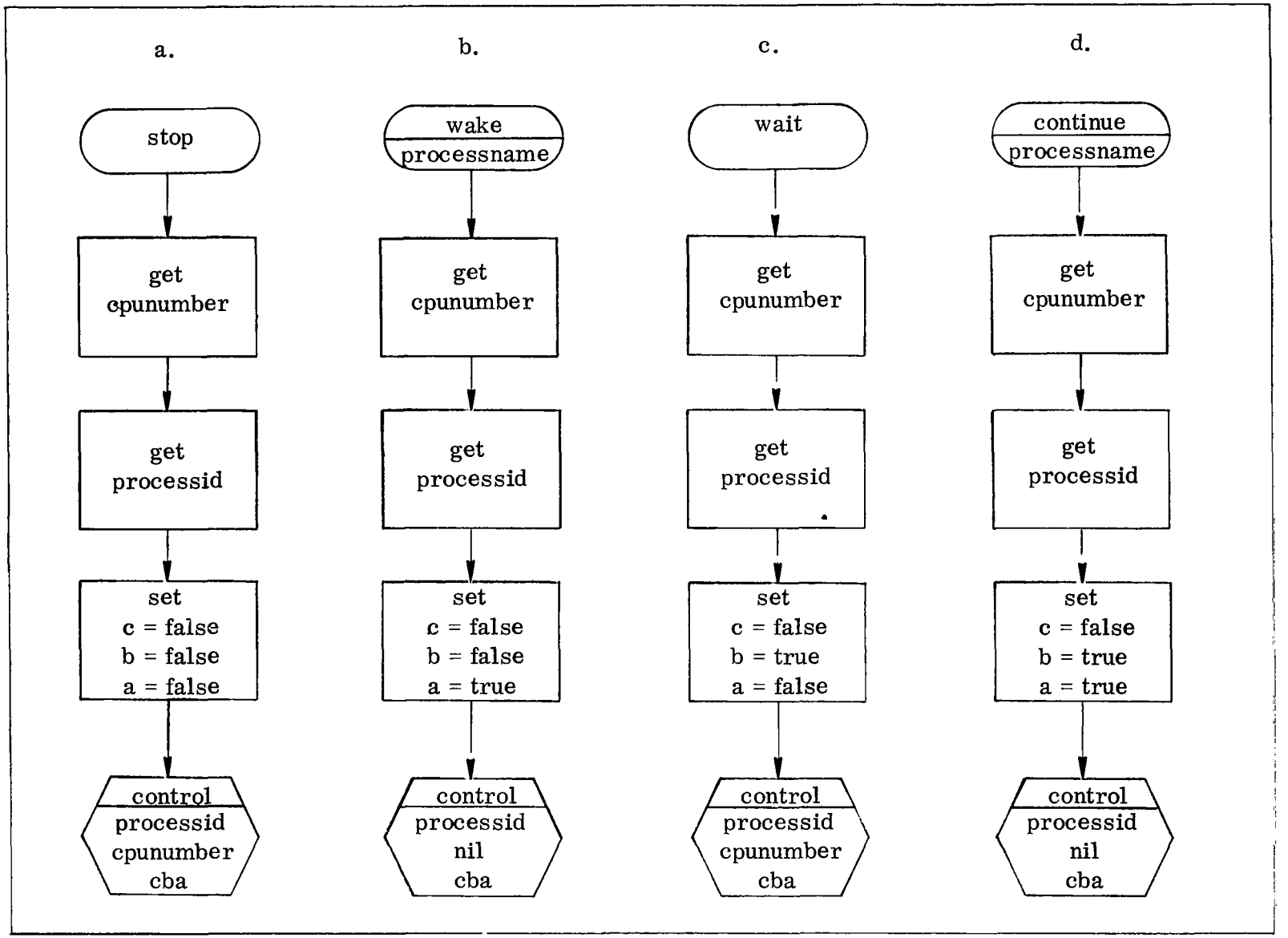


FIGURE 14. PROCESS CONTROL PRIMITIVES

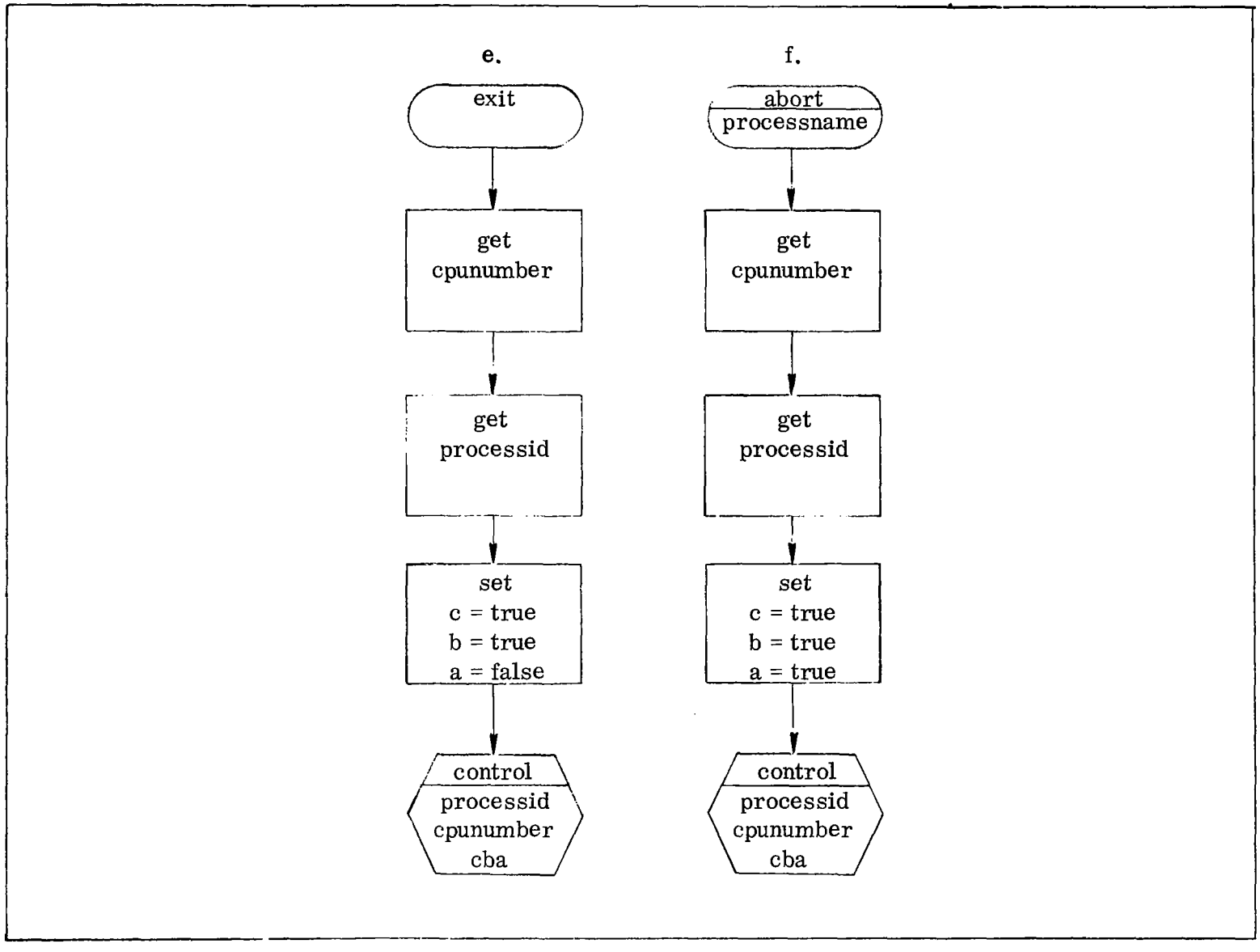


FIGURE 14. PROCESS CONTROL PRIMITIVES (Continued)

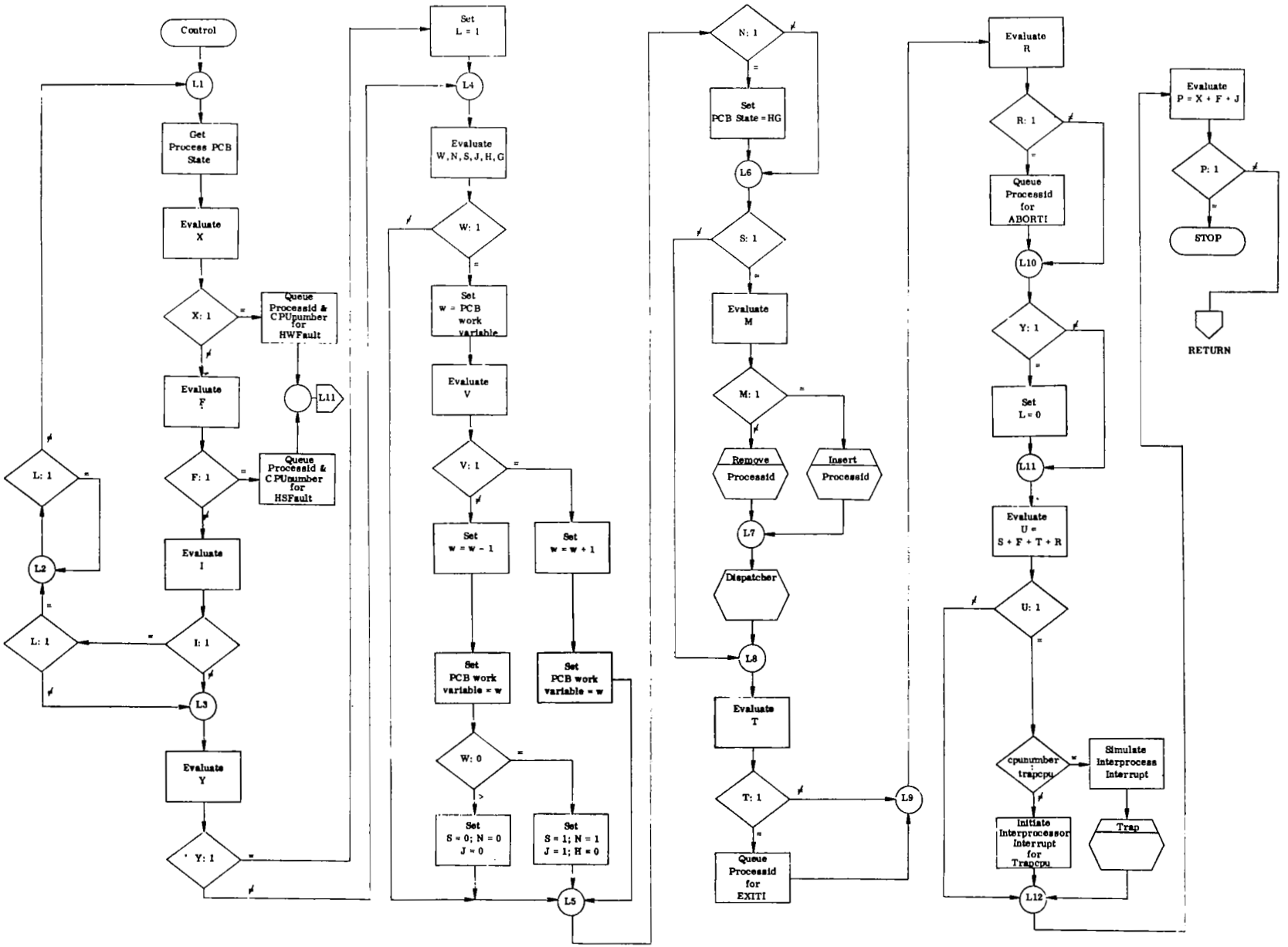


FIGURE 15. CONTROL FLOW DIAGRAM

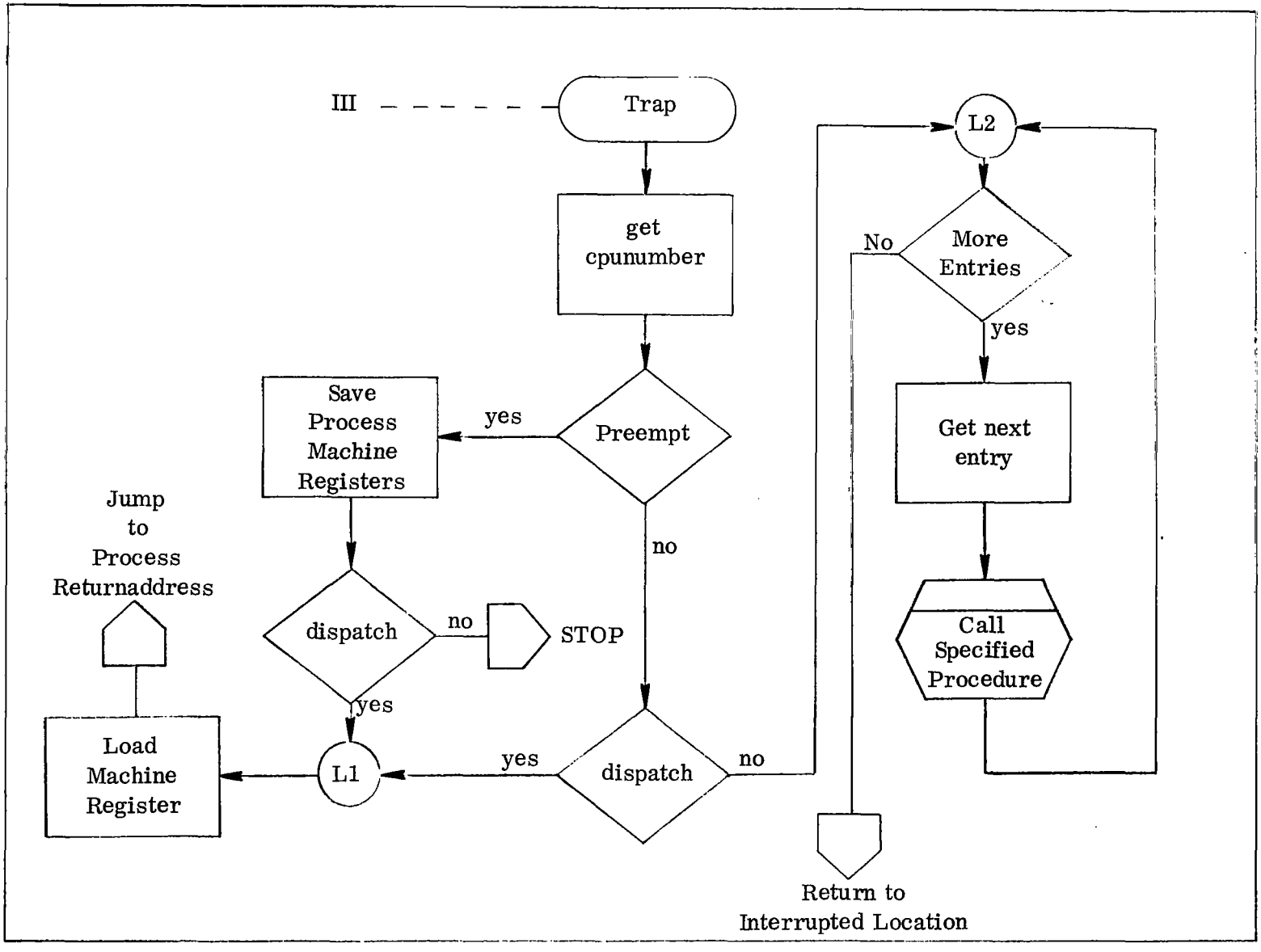


FIGURE 16. INTERPROCESSOR INTERRUPT PROCEDURE (TRAP)

```
primitives begin
        own integer processid, cpunumber, processname;
        own boolean a, b, c;
        integer procedure cpuid, idhashname;
        procedure control;
        comment cpu and process are assumed to be special control register
                addresses containing the implied identification data;

        procedure stop begin
        cpunumber:= cpu;
        processid:= process;
        a:= false;
        b:= false;
        c:= false;
        control (cpunumber,processid,a,b,c)
        end stop;

        procedure wake (processname) begin
        processid:= idhashname (processname);
        comment idhashname searches a hash-coded table correlating
                processname with a unique processid;
        cpunumber:= cpuid (processid);
        comment cpuid can be a special function that returns the cpunumber
                from the PCB;
        a:= true;
        b:= false;
        c:= false;
        control (cpunumber,processid,a,b,c)
        end wake;

        procedure wait begin
        cpunumber:= cpu;
        processid:= process;
        a:= false;
        b:= true;
        c:= false;
        control (cpunumber,processid,a,b,c)
        end wait;

        procedure continue (processname) begin
        processid:= idhashname (processname);
        cpunumber:= cpuid (processid);
        a:= true;
        b:= true;
        c:= false;
        control (cpunumber,processid,a,b,c)
        end continue;

        procedure exit begin
        cpunumber:= cpu;
        processid:= process;
        a:= false;
        b:= true;
        c:= true;
        control (cpunumber,processid,a,b,c)
        end exit;

        procedure abort (processname) begin
        processid;= idhashname (processname);
        cpunumber:= cpuid (processid);
        a:= true;
        b:= true;
        c:= true;
        control (cpunumber,processid,a,b,c)
        end abort
        end primitives;
```

```
nucleus begin boolean L;
        integer trapcpu;

        procedure control (cpunumber,processid,a,b,c) begin
        own boolean a,b,c,d,e,x,f,i,e,y,W,n,s,j,h,v,g,m,t,r,u,o,p;
        own integer cpunumber, processid, w;
        procedure getstate,hwfault,hsfault,storeworkvariable,remove,
                insert,dispatcher,exiti,aborti,trap,queue,
                storestate,simulate;
        integer procedure workvariable;
L1:     state (processid,d,e);
        x:= ā.c.ē + a.b̄.c.ē + ā.b̄.c.d̄ + ā.b.ē + ā.d.e + b̄.c.d.e;
        if x then queue (hwfault,cpunumber,processid);
        f:= b.c̄.ē + a.b.c̄.d̄;
        if f then queue (hsfault,cpunumber,processid);
        i:= a.c̄.d.e;
        if i then       begin
                        if L then       begin
L2:                                     if L then go to L2
                                            else go to L1
                                        end
                        end;
L3:     y:= ā.b̄.c̄;
        if y then L:= 1;
L4:     W:= a.c̄ + b̄.c̄;
        n:= d̄.ē + b + c;
        s:= c̄.d̄.ē + b.d.ē + b.d̄.e + b.c̄;
        j:= c.d̄.e + b.d̄.e;
        h:= b̄.d + ā.c̄;
        g:= c̄.ē + b̄.c.e + b.c̄;
        if W̄ then go to L5;
        w:= workvariable (processid);
        v:= a.c̄;
        if v then       begin
                        w:= w + 1; storeworkvariable (processid,w);
                                go to L5 end;
        w:= w - 1;
        storeworkvariable (processid,w);
        if w=0 then     begin
                        s:= true; n:= true; j:= true; h:= false
                        end
             else       begin
                        s:= false; n:= false; j:= false;
                        end;
L5:     if n then storestate (processid,g,h);
L6:     if s̄ then go to L8;
        m:= c̄.ē + d.c̄;
        if m then insert (processid)
            else remove (processid);
        dispatcher ( );
L8:     t:= ā.c.d̄;
        if t then queue (exiti,cpunumber,processid);
L9:     r:= a.b.c;
        if r then queue (aborti,cpunumber,processid);
L10:    if y then L:= 0;
L11:    u:= x + f + t + r;
        if ū then go to L12;
        if cpunumber = trapcpu
            then        begin simulate (trapcpu); trap ( ) end
            else        iii (trapcpu);
        comment iii is assumed to be a special code sequence for
                setting up and executing an "initiate interprocessor
                interrupt";
L12:    p:= x + f + j;
        if p then halt;
        return
        end control;
        comment halt will stop the processor and return transfers control
                back to the calling process;

        procedure trap begin integer queue;
        procedure save, load, nextentry;
        own integer cpunumber, processid;
        boolean array preempt, dispatch [ 0:2 ];
        cpunumber:= cpu;
        processid:= process;
        if preempt [ cpunumber ]
            then        begin
                        save (cpunumber,processid);
                        if dispatch [ cpunumber ] then go to L1;
                        halt;
L1:                     load (cpunumber,processid);
                        jump
                        end;
                        comment jump transfers control to the process;
        if dispatch [ cpunumber ] then go to L1;
L2:     if queue = 0 then return;
        nextentry (routine,cpunumber,processid);
        routine (cpunumber,processid);
        comment this call to a queued routine has to be "faked-in" since
                the language doesn't permit this exact sequence;
        go to L2
        end trap

        procedure dispatcher begin ... etc.

        end nucleus
```

FIGURE 17. SOURCE FORM STATEMENT LISTINGS FOR PROCESS CONTROL

SECTION V. MONITORING

Through the use of the PCB as a store for special data and main memory addresses, a powerful monitoring and/or debugging facility can be implemented as an integral part of the logic for process control. Three areas of usage are discussed here in order to expand on the application of the control mechanism to include program debugging, dynamic scheduling, and system stability measurement.

A. Program Debugging

As shown in the PCB of figure 1, the Breakpointaddress provides an address which is loaded, when a process is dispatched, into a special processor compare register. Every instruction fetch address is compared with Breakpointaddress; a match causes a trap to the associated BPAtrapaddress.

Breakpointoperand is similar in that it is also loaded into a special processor register. The effective address for all operand fetches is compared; a match causes a trap to BPOtrapaddress. In the case of post-indexing, every address in the (possibly infinite) sequence is compared.

Other parameters could be included in the PCB to facilitate capabilities such as "traces," or to specify special processor mode flags to indicate "debug," "phase step," and "single step" modes of processor execution.

In addition to the obvious aids to debugging provided by breakpointing, etc., sampling of the work variable as mentioned previously provides a basis for dynamic operation scheduling and detection of a trend toward system instability.

B. System Stability

As mentioned previously under the subheading on Process States, sampling the work variable, w, for all system processes provides a means of measuring the scheduled system workload. If attention is restricted to "closed" real time systems, definitions for backlog, degradation, and time-bounded stability can be given. The discussion that follows is intuitive in nature; concepts are not rigorously developed and are intended primarily to suggest possible further study.

1. Backlog. This name is given to an amount of work which has been scheduled but has not been completely processed by the system. Backlog is clearly a complicated function of time; in the context of process control, w provides a

good linear measure for it. Suppose that at time t, process P_i has a non-zero w. The value is denoted by $w_i(t)$. Since the system is closed, the average values of memory space and processor time for P_i can be measured or calculated and stored as constants in the associated PCB. Assuming they are constant and denoting the memory space and processor time as s_i and p_i, respectively, for the i^{th} process, the space-time constant for P_i is

$$c_i = s_i\, p_i\ .$$

Therefore, if n is the number of processes known to the system,

$$B(t) = \sum c_i\, w_i(t), \text{ summed over all } [\, i = 1, \ldots, n | w_i(t) > 0\,]$$

is the system backlog at time t. If because of the nature of the system it is known that the backlog at time t must be eliminated by time $t_o = t + \Delta t_o$, then an estimate of the system's ability to accomplish the backlog work in the allowable interval can be made as follows: Suppose there are M equivalent processors in the system and S is the total amount of (shared) main memory space. Then, the fraction of system capability, measured as the available system space-time, required to eliminate the backlog is

$$F(t, \Delta t_o) = \frac{B(t)}{eSM\Delta t_o} = \frac{B(t)}{SM\Delta t_o - h}, \text{ where}$$

system efficiency is represented by $e < 1$ and h is overhead. Clearly, if $F > 1$, the system cannot complete the backlog in the allotted time interval. If the configuration allows spare processors to be switched into active operation, F would be used as a first approximation to determine when spares should be switched in or out in anticipation of the expected loading. In the case of many relatively simple systems, the number of processes is small and their compute cycle space-time constant can be accurately measured or calculated. If these systems are repetitive or cyclic in nature, it is possible to measure backlog and plot it as a time function. To find the minimum number of processors that will allow completion of the backlog, B(t), set F = 1 giving

$$M_{min} = \text{ greatest integer in } \frac{B(t)}{eS\Delta t_o} + 1.$$

$$= \left[\frac{\sum_i c_i w_i(t)}{eS\Delta t_o} \right] + 1.$$

2. System Degradation. Suppose the processes of a system are ordered such that $l > m$ implies that the priority (relative importance) of $P_l \geq$ priority of P_m for all l, $m \leq n$. Suppose further that when the maximum

number of available processors, M, is in use, F>1. If the system has the ability to schedule only the higher priority processes such that F can be made less than 1, then the system is said to be operating in a degraded mode and degradation, D(t), is defined to be

$$D(t, \Delta t) = \sum_i c_i w_i(t), \quad i = 1, \ldots, k,$$

$$\text{where } k+1 = \min \left\{ l \left| \frac{\sum_i c_i w_i(t)}{eSM\Delta t} < 1, \; i=1, \ldots, m; \; l=n, \ldots, 1 \right. \right\}.$$

That is, degradation is simply that portion of the backlog at time t which cannot be eliminated in the allotted Δt interval.

If a system exhibits the property that the degradation is strictly increasing on Δt, then it is clearly unstable on Δt. On the other hand, if $D(t, \Delta t)$ is monotonically decreasing on Δt, then the system can be said to exhibit the property of time-bounded stability (over the interval Δt).

C. Dynamic Scheduling

Development of a scheduling philosophy per se is beyond the scope of this report. However, several points should be clarified. The process control mechanism provides inherently for what might be termed "demand scheduling." The primitives will insure that the dispatcher gets control whenever certain process state transitions occur. However, this is not sufficient in the general case since a process could conceivably gain control of a processor and not relinquish it.

This problem can be resolved by defining a system process that is periodically given control because of its intentionally assigned high priority. Once this process gains control, it requests activation (that it be placed in the ready state) at some prespecified future time and then invokes the STOP primitive. The sequence is repeated in order to insure that the dispatcher is given an opportunity to reschedule all processes. This scheme supports what might be termed "time-slice scheduling."

If a process is dispatched and its work cannot be completed in some predetermined time interval, it must be preempted, assuming other processes of higher or equal priority are not executing. Indiscriminate application of this kind of policy can result in processes that are never quite caught up on the work requested of them. A scheme that gives preferential treatment to processes having a greater "age x work variable" product may be applied in special cases such as this. For instance, a low priority process might be

dispatched when its age x work variable product exceeds some threshold value. This will tend to give a periodic boost to processes that add to the backlog, but don't have a high enough priority or work variable value to be dispatched.

SECTION VI. COMPARISONS

This section compares the conventional software approach to the hardware approach on the basis of reasonable (although crude) estimates. Execution time and hardware cost are the comparison factors shown here; it is believed that a similar trend can be shown for other factors such as weight, volume, reliability, and power.

Reference /16/ indicates that cost/bit of core memory is 3 cents and cost/gate is 10 cents. On the basis of the logic and software shown, it is estimated that 20K bits (based on an estimated program length of 490_{10} [40 bit] words) of memory are required for the depicted software, while 201 gates are estimated for the hardware logic. This yields a cost comparison of

Memory for software	-	$600.00
Logic for hardware	-	20.00.

An execution time comparison is even more startling: assuming a gate time of 14 nanoseconds (1 nanosecond is achievable) and an average instruction operation time of 2 microseconds, a conservative comparison is possible.

In the case of the hardware approach, it seems that a reasonable upper boundary on the time is 10 main memory cycle times (assume 10 microseconds). A reasonable lower boundary on the software approach time is obtained by assuming that only half the main memory words are fetched. This gives a time of 490 microseconds.

The two comparisons, admittedly gross but believed to show the correct relationships, result in the following illustration:

TABLE 9. COST AND TIME COMPARISON

	Cost ($)	Time (μs)
Hardware	20	10
Software	600	490
Ratio (H/S)	1/30	1/49

[16] Kerner, H. and Gellman, L.: Memory Reduction Through Higher Level Language Hardware. NASA TM X-53962, Research Achievements Review, V. III, N. 9, 1969.

SECTION VII. CONCLUSIONS

The effort reported upon had as its principal thesis the belief, shared by many computer system specialists, that portions of the executive system can be pared away from the total system and formalized through a supporting theory. The objective of the report is to outline an approach, based on current trends in the NASA space program and ideas within the computer systems community, to describing a significant aspect of an executive routine, and showing how this aspect can be formalized.

Much discussion has taken place in the spaceborne computer systems area regarding the feasibility and definition of a "hardware executive." Some have coined this phrase in the context of a capability for system fault detection, isolation and reconfiguration control, while others have outlined "hardware executives" that embrace the entire functional domain of a real time monitor. The author feels that the former interpretation is too restricted in the sense that the specified functional responsibility is a subset of that of an executive routine. The latter interpretation is valid from the point of view of functional responsibility, but implementation schemes invariably make extensive use of all the capabilities of a general purpose computer resulting principally in an extraction of the executive from the "application" computer, and the corresponding dedication of a second "executive" computer.

There is little doubt that an extraction of the executive is desirable; but, it is not desirable to merely transliterate the function from the domain of software into either the domain of digital logic or stored logic. It seems reasonable to assume that a reformulation of the control concepts is necessary in order to insure a cost-effective transformation. That is, the control functions should be reexamined from the point of view of the target domain, and analyzed with the engineering techniques found most successful to that domain.

The report has attempted to display the concepts of extraction and reformulation as applied to the digital logic domain. The executive functions discussed are extensions of functions needed in all real time systems and, as a superset, all multiprogramming systems. The report defines the functions and inductively derives a state diagram for the necessary control in a multiprocessor environment.

The state diagram represents a sequential finite state machine which relates program dynamic requirements for computation time with the means for determining the validity of the requirements and the means for allocating the time. Having defined a machine, a mechanization for it is discussed in tutorial form. The rationale for this approach is that it should strengthen the computer hardware designer's understanding of a systems programming

view of executive control; in addition, it should better equip the systems programmer to devise such transformations for himself. It is expected that the overall result of this approach (aside from substantiating the results) will be a better mutual understanding between the programmer and hardware architect.

It should be clear that many details were excluded, and many questions left unanswered. It was not intended that a final detailed design be displayed. Naturally, the lack of detail in certain areas vitiates the comparison of software and digital logic approaches. However, a comparison was made and the results are felt to display the general relationships with fidelity. Obviously a more in-depth analysis is required to substantiate this supposition.

SECTION VIII. RECOMMENDATIONS

The report has outlined an approach to hardwired digital logic implementation of a nucleus for executive control of processes in a multiprogramming, multiprocessor computer complex. While the approach appears promising on the basis of intuition and the gross comparisons given, more detailed analysis is required to strengthen the credibility of the concept and to more rigorously define the actual requirements. Based on these observations, the following recommendations are offered to direct further effort:

- Further analysis is required to complete the specification of the multiprocessor control logic. This includes expansion to embrace all process and processor states, and transition activation inputs (primitives and control lines).

- Based on the complete specification, a careful evaluation should be made and documented.

- The concepts should be reduced to operable form for the case of a single processor multiprogramming system with similar evaluations.

If the resulting evaluations prove favorable to a digital logic implementation scheme, the following recommendations are offered:

- An analysis should be performed to determine the feasibility and effectiveness of implementing some or all of the supporting software procedures (such as HWFAULT, INSERT, etc.) in the form of microprogrammed stored logic.

- In the case of both the Space Shuttle and Space Station, there is a strong likelihood that some aspects of the support functions can be rigorously defined. If so, they become candidates for possible stored or discrete logic implementation. It is recommended that feasibility be determined and that the appropriate trade studies be performed to ascertain the most effective means of implementation.

- Finally, the feasibility and effectiveness of an all-microprogrammed approach to implementation should be determined for the case of the NASA Space Ultrareliable Modular Computer.

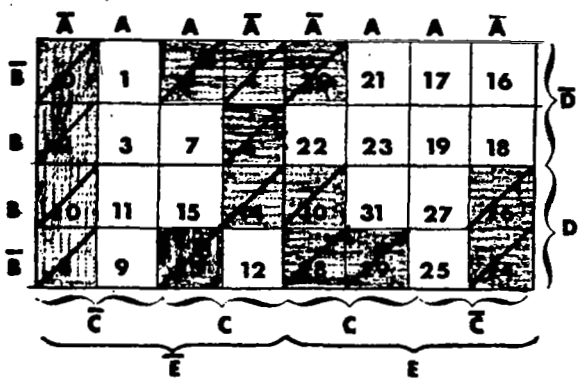
APPENDIX A

LOGIC MAPS FOR PROCESS CONTROL

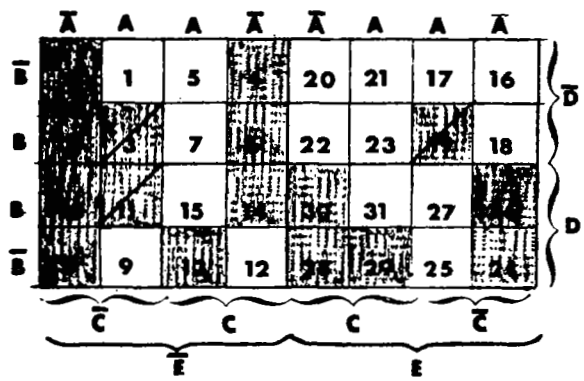
The maps shown on the following pages (figure A1) aid in the derivation of Boolean expressions for the evaluation of the output variables for process control. The "/" is used to mark those minterms that are "1." The shaded area is the union of all "1" and "don't-care" minterms. Unshaded areas represent "0" minterms.

The process of reading the map, i.e., determining the variables to be used in composing the expression, is normally guided by a set of design objectives. Since many different but logically equivalent expressions can usually be developed for each variable, the design objectives are used as criteria for the selection of a particular expression. Some typical criteria are: desired degree of redundancy; the number of gate inputs; whether inverters are desirable; whether each input variable and its complement are both available; what type of logic devices are to be used, e.g. nand/nor, and/or, etc.

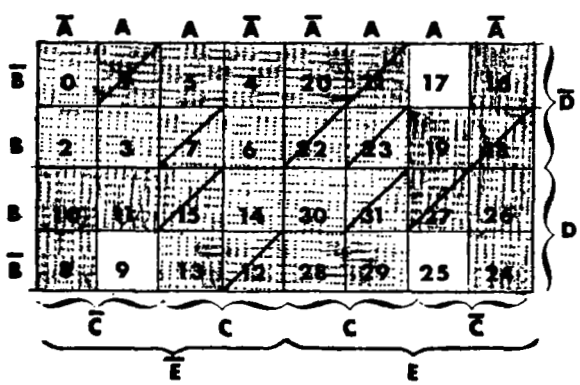
In the development of expressions for this report, no specific logic circuits were assumed, and the main objective was to adhere to a "transparent" reading philosophy for simplicity. (The author does not claim to have experience in the application of digital design techniques; tricky derivations were accordingly avoided intentionally.) An attempt was made, however, to couple the largest number of variables in each term. Also, all maps were read to satisfy "true" minterms with "don't-care" minterms forced to "true" values to eliminate terms where possible.



$$X = \bar{A}\bar{C}\bar{E} + A\bar{B}C\bar{E} + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{E} + \bar{A}DE + \bar{B}CDE$$

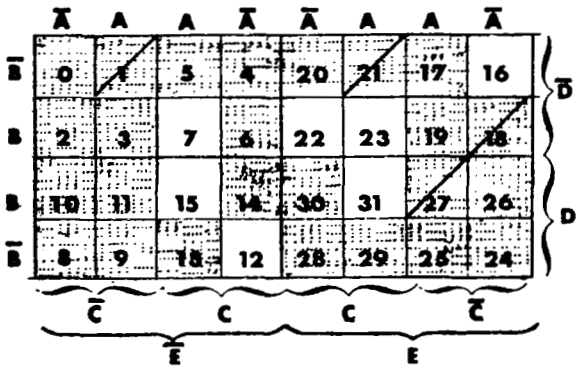


$$F = B\bar{C}\bar{E} + AB\bar{C}\bar{D}$$

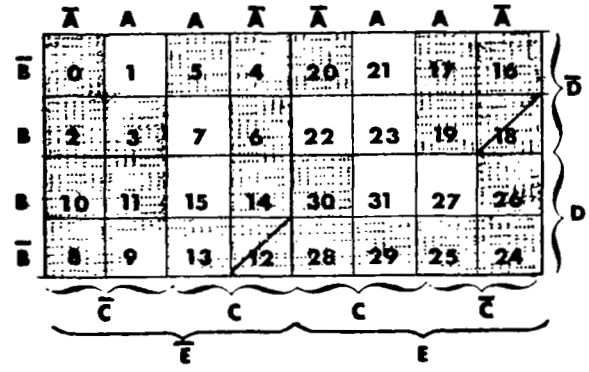


$$N = \bar{D}\bar{E} + B + C$$

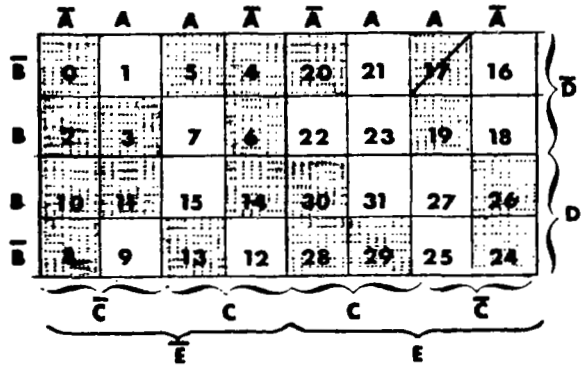
FIGURE A1. MAPS FOR DERIVATION OF BOOLEAN EXPRESSIONS (Sheet 1 of 5)



$G = \overline{C}\overline{E} + \overline{B}CE + B\overline{C}$
("c" is taken to be "0" for worst case.)

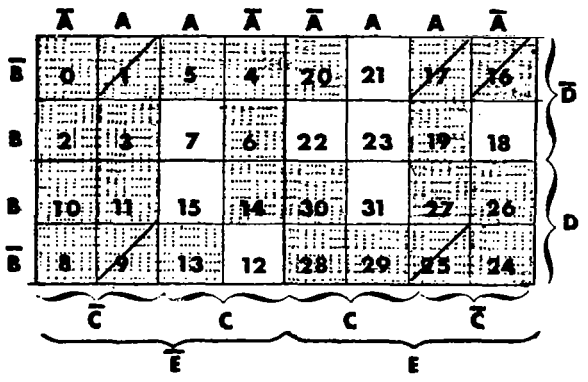


$H = \overline{B}D + \overline{A}\overline{C}$

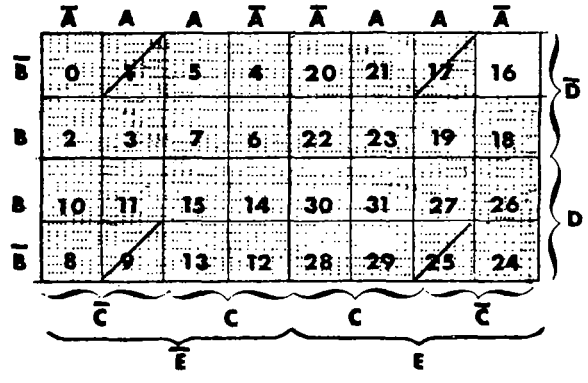


$I = A\overline{C}\overline{D}E$

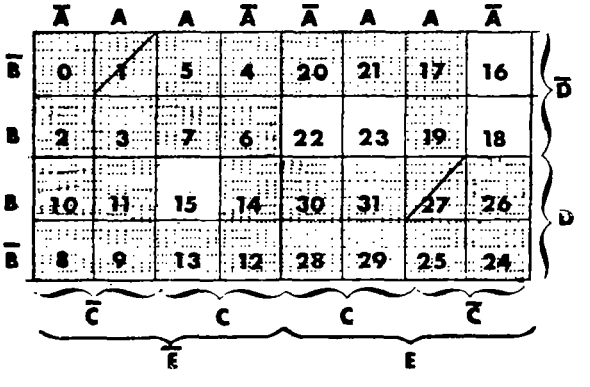
FIGURE A1. MAPS FOR DERIVATION OF BOOLEAN EXPRESSIONS (Sheet 2 of 5)



$W = A\bar{C} + \bar{B}\bar{C}$

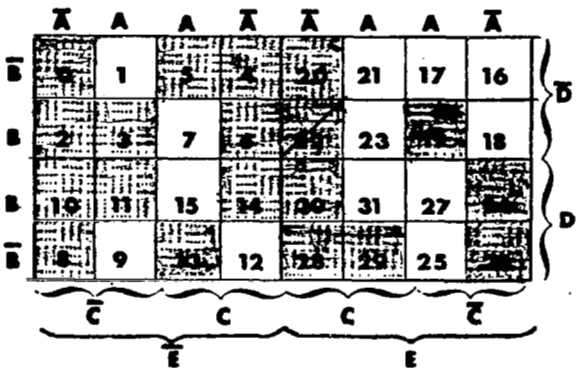


$V = A\bar{C}$

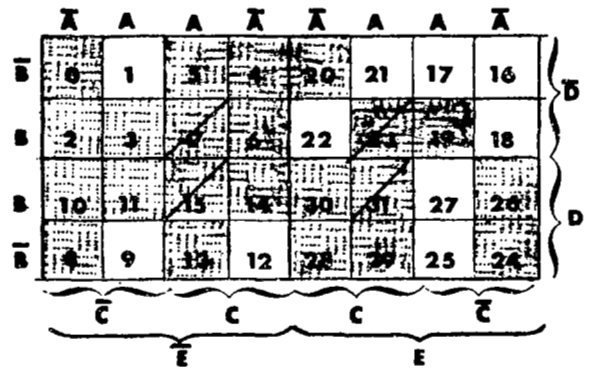


$M = \bar{C}\bar{E} + D\bar{C}$

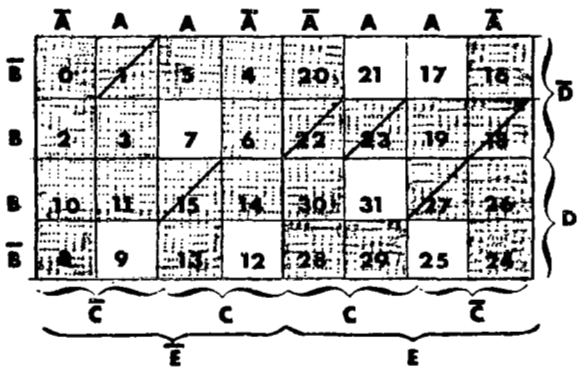
FIGURE A1. MAPS FOR DERIVATION OF BOOLEAN EXPRESSIONS (Sheet 3 of 5)



$T = \bar{A}C\bar{D}$

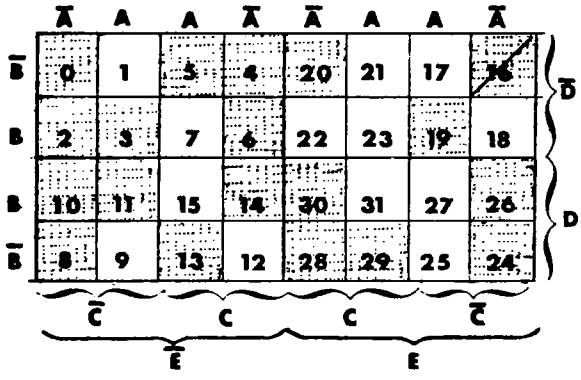


$R = ABC$

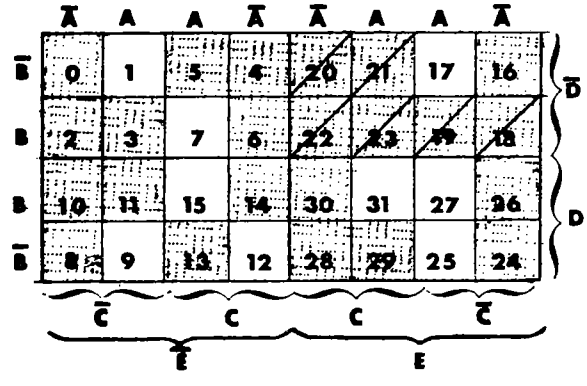


$S = \bar{C}\bar{D}\bar{E} + BD\bar{E} + B\bar{D}E + B\bar{C}$

FIGURE A1. MAPS FOR DERIVATION OF BOOLEAN EXPRESSIONS (Sheet 4 of 5)



$Y = \bar{A}\bar{B}\bar{C}$



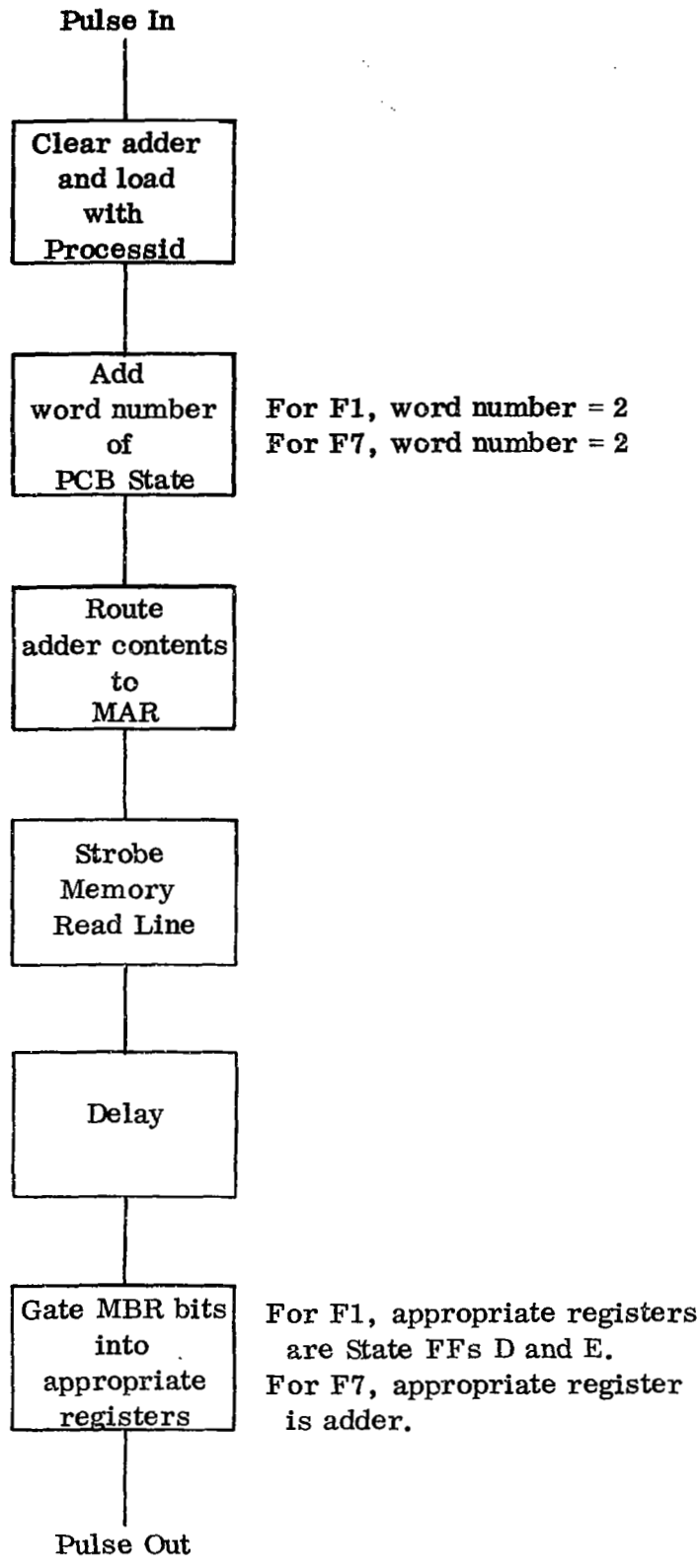
$J = C\bar{D}E + B\bar{D}E$

FIGURE A1. MAPS FOR DERIVATION OF BOOLEAN EXPRESSIONS (Sheet 5 of 5)

APPENDIX B

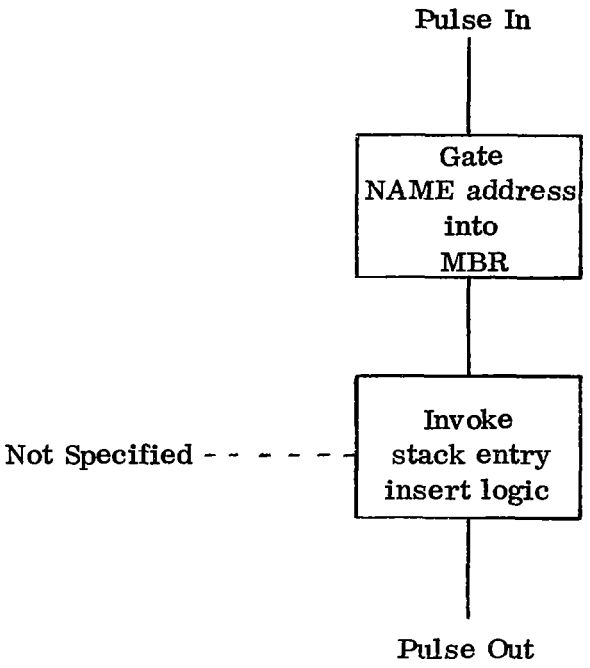
The functions outlined in figure B1 are those required as indicated in figure 12, "Process Control Sequencing." Only an overview is shown; no attempt was made to give details since the peculiar architecture of a candidate processor must be considered for proper definition.

It is clear from the following diagrams that all functions are simple. Showing this simplicity is the main objective of the appendix. (Note that function F2 is shown in figure 10 and is not repeated here.)



Functions F1 and F7

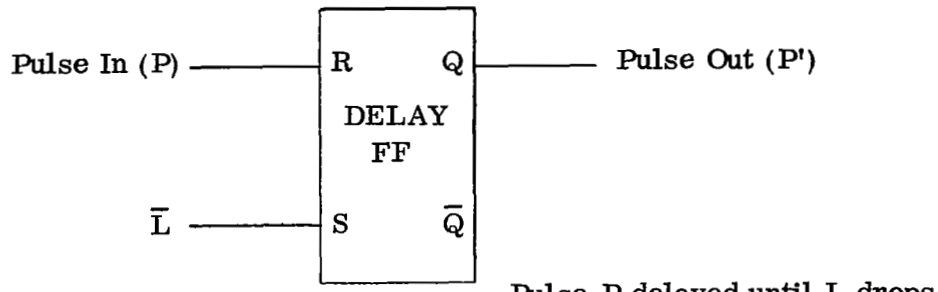
FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 1 of 10)



NAME	FUNCTION
HWFAULT	F3
HSFAULT	F4
INSERT	F14
REMOVE	F15
DISPATCHER	F16
EXITI	F17
ABORTI	F18

PROCEDURE STACKING FUNCTIONS

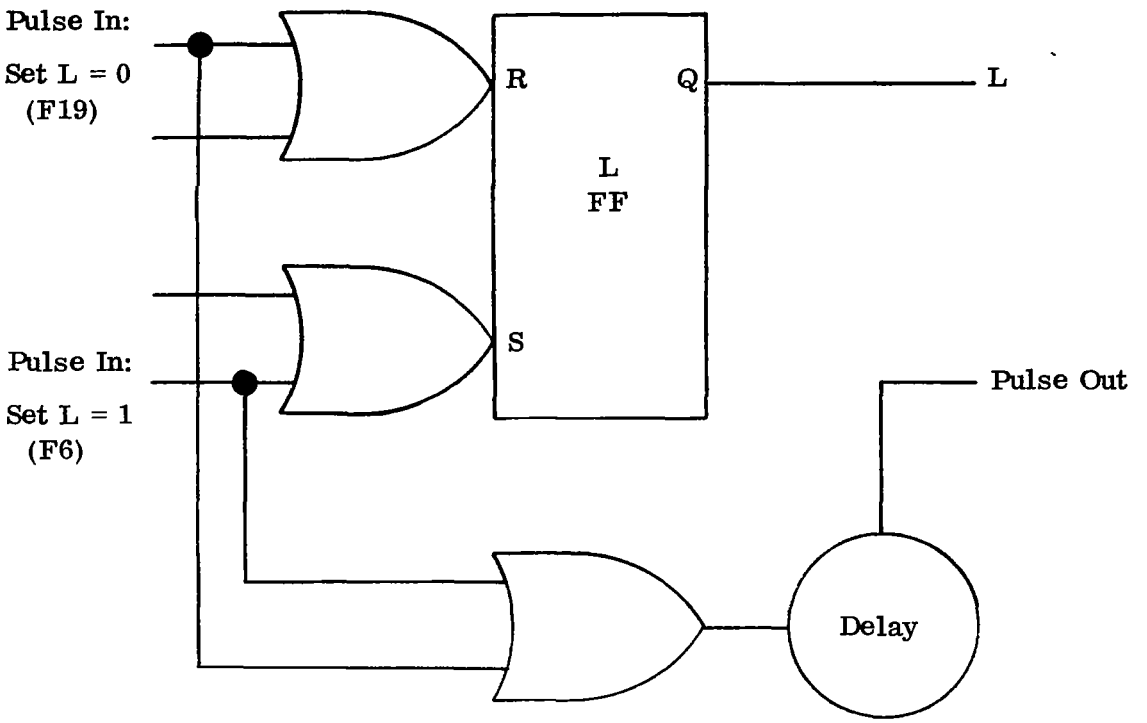
FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 2 of 10)



Pulse P delayed until L drops to false.

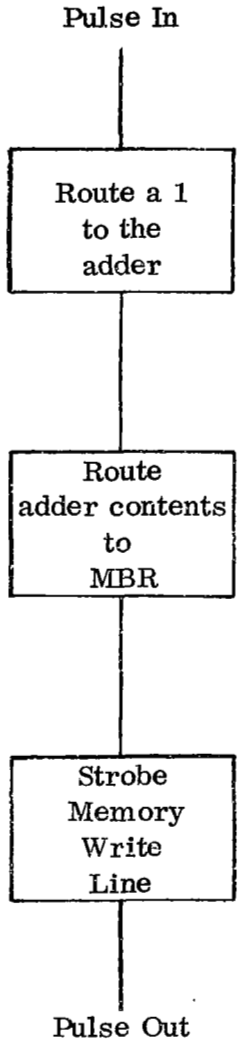
FUNCTION F5

FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 3 of 10)



FUNCTIONS F6 AND F19

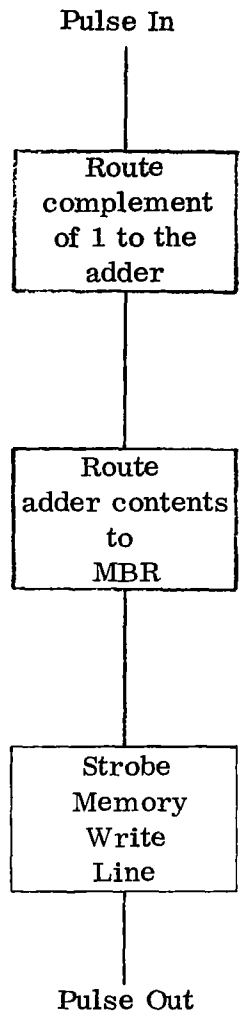
FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 4 of 10)



If w is imbedded in a word, additional packing is required.

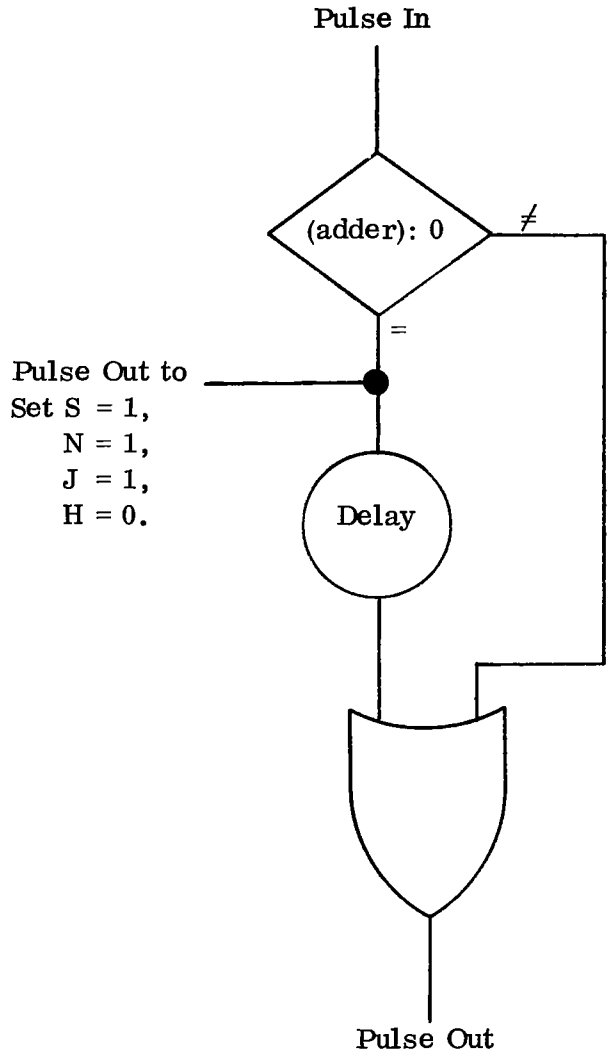
FUNCTION F8

FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 5 of 10)



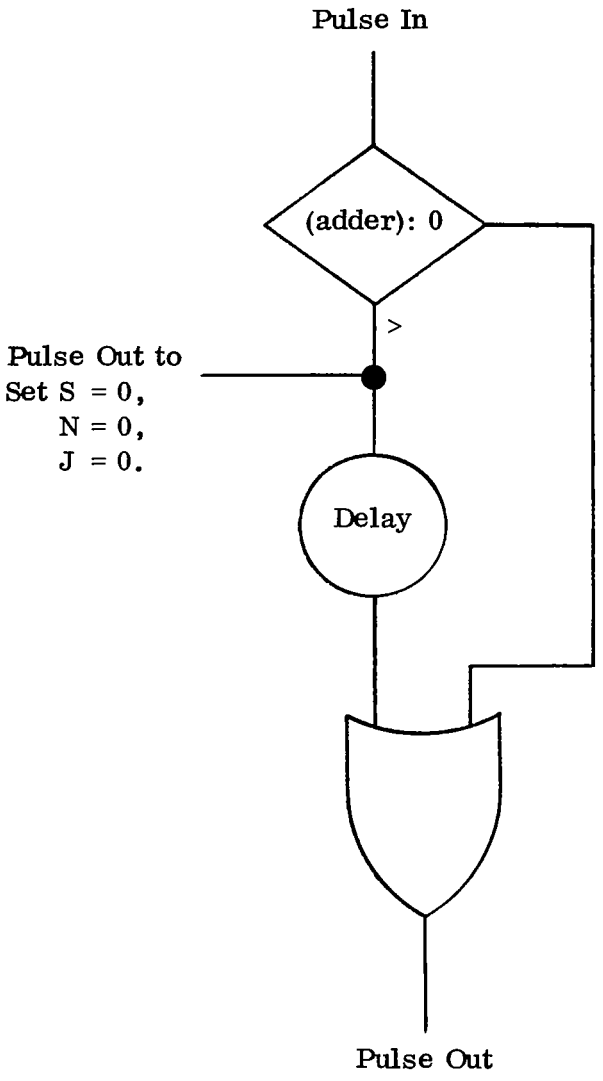
FUNCTION F9

FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 6 of 10)



FUNCTION F10

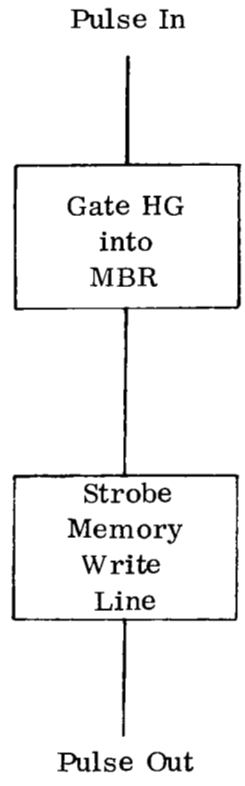
FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 7 of 10)



Note: F10 and F11 could be combined.

FUNCTION F11

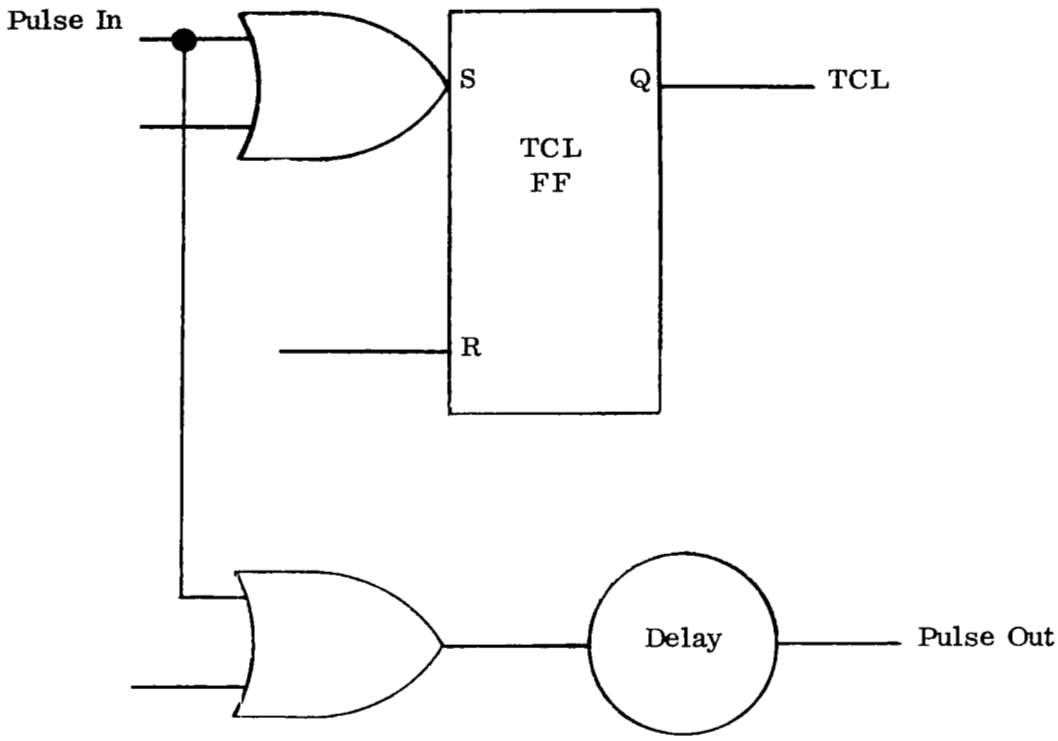
FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 8 of 10)



Packing may be required.

FUNCTION F12

FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 9 of 10)



FUNCTION F20

FIGURE B1. PROCESS CONTROL FUNCTIONS (Sheet 10 of 10)

NASA-Langley, 1971 —— 8