DATE: March 29, 1976 PE-T-232

Programming and Engineering Staff

FROM: M. L. Grubin

SUBJECT: P-4CC PROCESS EXCHANGE AND NEW PROTOCOLS

I. Process Exchange

A. Data Bases
1. Ready List
2. WAIT Lists
3. Process Control Block (PCB)

B. Instruction Primitives
1. WAIT
2. NOTIFY

C. Dispatcher and Register File Management
1. Ready List Maintenance
2. Register Set Assignment
3. Fetch Cycle Trap

II. Traps, Interrupts, Faults, Checks

A. External Interrupts
1. Operation
2. Special Instructions (IRTN, INOTIFY)

B. Faults
1. Data Bases
2. CALF
3. Fault Handler

C. Checks

III. Register Files

IV. Control Panel

V. CP Timer

I. PROCESS EXCHANGE

The Process Exchange mechanism is composed of three data bases and two basic instruction primitives. The data bases are the ready list, wait lists, and Process Control Blocks (PCB). The basic instruction primitives are WAIT and NOTIFY. In addition, there is an independent mechanism for controlling the usage of two register sets which is related to, but not part of, the ready list data base.

A. Data Bases

1. Ready List

The ready list is a two-dimensional list structure used for priority scheduling and dispatching of processes. The entire ready list data base (excluding live registers) and all PCB's are contained in a single segment. The segment number of this segment is contained in a 16-bit register called OWNERH. Within the segment, all pointers and addresses (except fault vectors and wait list pointers) are 16-bit word number quantities.

The two-dimensionality of the ready list is achieved with a linear array of list headers for each priority level composed of a Beginning of List (BOL) pointer and an End of List (EOL) pointer.

Each pointer is the 16-bit word number address of a PCB (in the same segment as the ready list). The PCB's on each priority level list are forward-threaded through a 16-bit link word, and as many PCB's as desired can be threaded together on each priority level to form the ready list. A process' priority level is both determined by and encoded as the address of a BOL pointer in the ready list. Priority order is determined by arithmetic comparison, i.e., smaller numbers (addresses) are higher priorities. As a result, priority level list headers must be allocated in contiguous memory at system startup time.

The end of the ready list is determined by a BOL containing a 1 (PCB addresses must be even). An empty level is indicated by a BOL containing 0. Figure 1 is a picture of the ready list structure. The 32-bit registers PPA (Pointer to Process A) and PPB (Pointer to Process B) are a speed-up mechanism for locating the next process to dispatch. PPA always contains both the level (BOL pointer) and PCB address (designated level A and PCBA) of the currently active process. PPB points to the NEXT process to be run when process A 'goes away'. PPA not only points to the currently active process, but, by definition, level A is the highest level in the system. It is possible for PPB and PPA to be 'invalid'. This condition is indicated by a PCB address of 0. It is important NOT to disturb the level portions, especially level A since, even if invalid, level A indicates the highest level that WAS in the system and therefore determines where in the ready list to begin a scan, if necessary (PPB invalid), for th next process to run. In a completely idle system, both PPA and PPB will be invalid and, upon completion of the ready list scan, the u-code will go into a 'wait for interrupt' loop.

It is important to notice that there is no word number pointer to the first priority level in the ready list. The ready list allocator, ch starts the process exchange mechanism, knows where the list egins and, during execution, level A (in PPA) will always point to either the highest level currently in the system or the last known highest level and, hence, acts as an effective ready list begin pointer. In addition, level B will always be higher than the second level to run. That is, a PCB can never be on a level higher than level B unless it is the only PCB higher than level B (i.e., level A).

Two 'queuing' algorithms will be implemented for the ready list, either FIFO or LIFO queuing.

2. WAIT Lists

Every PCB in the system will always be somewhere. If it is not on the ready list, then, by definition, it will be on a wait list. A wait ist is defined by a 32-bit semaphore consisting of a 16-bit counter .C) and a 16-bit word number BOL pointer. Since the ready list and all PCB's reside in one segment (OWNERH), and only PCB's go onto wait lists, a segment number is not needed in the semaphore. However, semaphores themselves can be anywhere and, in general, are NOT in the PCB segment. The structure of a wait list is shown in Figure 2. Notice that the last block on the wait list contains a 0 link word. tice also that the semaphore contains only a BOL pointer.

ne 'queuing' algorithm for wait lists is process priority queuing. That is, the priority level of a PCB will determine where on the wait list the PCB will be queued. For PCB's of equal priority, the algorithm becomes FIFO.

3. Process Control Block (PCB)

The contents of the PCB are shown in Figure 3. The PCB can be broken ito the following logical sections which are ordered as shown:

a. Control
   - 0 - level (pointer to BOL in ready list)
   - 1 - link (pointer to next PCB or 0)
   - 2,3 - SN/WN of Wait List this block is currently on (SN=0 indicates on ready list)
   - 4 - abort flags used to generate Process Fault when PCB is dispatched.
     - Current bit assignments 1-15: MBZ
     - 16: process interval timer overflow
   - 5,7 - reserved

b. Process State
   - 8,9 - Process elapsed timers (must be maintained by software that resets the interval timer)

10,13 - DTAR2 and DTAR3 (never saved, always restored)
14 - Process Interval Timer with 1.024 msec resolution
15 - Reserved
16 - Save mask - used to avoid saving and restoring registers = 0
Bits 1- 8: GR0-GR7 (2 words each)
9-12: FP0-FP1 (4 registers, 2 words each)
13-16: Base Registers(PB,SB,LB,XB)
17 - Keys
18,33 - GR0-GR7
34,41 - FP0-FP1
42,49 - Base Registers (PB,SB,LB,XB)

Note that although all the registers are assigned locations within the PCB, only non-zero registers will actually be saved which results in a compacted list which can only be determined by the bits in the save mask. In general, the saved registers (those not equal to 0) will be between words 18 and 49. The order of the registers, however, is fixed as above.

c. Fault (See section on Faults for a description of the use of this portion of the PCB)

50,59 - Fault Vectors: SN/4N pointers to fault tables for Ring 0, Ring 1, Page Fault and Ring 3 fault handlers
60,62 - Concealed Fault Stack Header
63,... - Concealed Stack - 6 word entries. (This stack need not start at word 63).

B. Instruction Primitives

There are two basic instruction primitives for the process exchange mechanism: NOTIFY and WAIT. In addition, NOTIFY has two variants. These instructions, similar to Djikstra's P and V operators, are essentially 'interlock' mechanisms. These instructions are three-word (48-bit) 'instructions' as follows:

Instruction (16-bit universal generic)
32-bit pointer to semaphore address

As suggested by the names, WAIT is used to wait for an event (CP, time, unit record device available, whatever) and NOTIFY is used to signal that an event has occurred. In particular, a WAIT is used to wait for a NOTIFY and a NOTIFY is used to alert a process which is waiting.

Coordination is achieved by means of a semaphore containing a counter and a BOL pointer. The semaphore and the PCB's waiting for the event of that semaphore constitute a wait list. The counter, if greater than 0, indicates the number of PCB's on the wait list. If negative, it indicates the number of processes that can obtain the resource. Semaphores fall into two categories: public and private. A public semaphore is used to coordinate several processes together or control a system resource. Private semaphores are used by a single process to

coordinate its own activities. For example, if a disk request is made, process will wait on a private semaphore for the disk operation to complete. The disk process will then notify the semaphore upon completion. The distinguishing characteristics of a private semaphore is that only 1 PCB can ever be on that wait list. A public semaphore can have many different PCB's on its wait list.

1. WAIT

The operation of wait is as follows: the semaphore counter is incremented and, if greater than 0, (resource not available/event has not occurred), the PCB is removed from the ready list and added to the specified wait list. If the counter is less than or equal to 0, the process continues. If the PCB is put on the wait list, the general registers are NOT saved and the register set is made available. Therefore, a process can NEVER depend on the general registers being intact after a WAIT. In fact, from the point of view of an executing process, a WAIT appears as a NOP which destroys the registers. In addition, WAIT will turn off the process timer. Figure 4 is a detailed flow chart of the WAIT instruction.

2. NOTIFY

The NOTIFY instruction has two flavors:

NFYE: use FIFO queuing op code Bit 16 = 0
NFYB: use LIFO queuing op code Bit 16 = 1

The instructions differ ONLY in the ready list queuing algorithm used. The operation of NOTIFY is as follows: the semaphore counter is decremented and the notifying process continues. If the counter is less than 0, no action is taken, but if greater than or equal to 0, a PCB is removed from the top of the wait list and added to the ready list. No explicit action is ever taken on the notifying process, only the notified semaphore. If a notified process is of higher priority than the notifying process, the latter will be effectively 'interrupted', but will remain on the ready list. Figure 5 is a detailed flow chart of the NOTIFY instruction.

C. Dispatcher and Register File Management

The dispatcher is the root of the process exchange mechanism and is responsible for determining the next process to run (be dispatched), and assigning that process a register set. There is considerable overlap with NOTIFY and WAIT in functionality related to maintaining the ready list. For example, both NOTIFY and WAIT update PPA and PPB as appropriate, but the dispatcher scans the ready list if PPA is invalid. Register file management, including any necessary saves and restores, are the sole province of the dispatcher. Figures 6 and 7 are detailed flow charts of the dispatcher.

1. Ready List Maintenance

Upon entry, the dispatcher first asks if PPA is valid. If it is, the process is assigned a register set and dispatched. If PPA is not valid, a scan of the ready list is initiated. If a PCB is found, PPA is adjusted and the process dispatched. If the ready list is empty, the dispatcher idles. Whenever a process is dispatched the process timer is turned on.

2. Register Set Assignment

In each register set, a register, designated OWNER, contains a pointer to the PCB of the process which owns the set. OWNER is a full 32-bit pointer and OWNERH is used throughout the system to determine the segment number of the ready list and PCB's. Obviously, OWNERH must be the same in both register sets. In addition, the low order bit of the keys register (KEYSH) is used to indicate whether the register set is available. The bit is called the Save Done bit and, if set, indicates that the register set and its copy in the owner's PCB are identical (a save has been done). This bit is set by the save routine (called from WAIT or the dispatcher) and reset when a process is dispatched. Whether a register set is available (SD=1) or not, it is always owned. Therefore, if a process goes away (either as a result of a WAIT or the notification of a higher level process) and comes back agai immediately and, if that process still owns the register set, a restore operation is not necessary. If a register set switch is necessary, the process timer is turned off. The details of selecting which register set to assign to a process being dispatched is shown on the right of Figure 6. The dispatcher is the only code which switches register sets.

3. Fetch Cycle Trap

At various points in the dispatcher (indicated by I on the flow chart) a check for interrupt pending (fetch cycle trap) is made. As a result, interrupts can occur either in the fetch cycle or in the dispatcher. The possible Fetch Cycle traps are:

1. External Interrupt (See Part II-A)
2. CP-timer increment and possible overflow (See Part V)
3. Power Failure (See Part II-C)
4. Halt switch on control panel (See Part IV)
5. End-of-Instruction Trap

The end-of-instruction trap occurs either from an ECC corrected error or from a missing memory module, memory parity, or machine check during I/O. In all cases, if the check handling software returns (via LPSW instruction), the possible destinations are either the fetch cycle o the dispatcher (PB in PSW not a real program counter). In order to guarantee the proper destination, bit 15 of the keys (KEYSH) is used to incicate if the trap was detected by the dispatcher (bit 15=1).

This bit is set by the dispatcher upon detecting a trap and is reset when a process is actually dispatched (return to fetch cycle).

11. TRAPS, INTERRUPTS, FAULTS, CHECKS

Four words used frequently are 'trap', 'interrupt' (or 'external interrupt'), 'fault', and 'check'. The meanings of these terms are carefully distinguished for the P-400/500. Software breaks in execution are divided into three main categories referred to as 'interrupts', 'faults', and 'checks'. The word 'trap', on the other hand, refers to a break in execution flow on the u-code level.

Traps can occur for many reasons, not all of which cause software visible action, and are always processed on the u-code level. Some traps may directly or indirectly cause breaks in software execution, but not all software breaks are the result of a trap.

On the PRIME 300, interrupts, faults, and checks used the same protocol to get to their respective software handlers, namely they caused a vector through a dedicated sector 0 location (JST* vector). On the P-400/500, when process exchange mode is enabled, the three categories use different protocols both from the P-300 and each other. Roughly, the three terms are used to describe:

1. Interrupt - a signal has been received from a device in the external world (including clocks) indicating that the device either needs to be serviced or has completed an operation. In general, an interrupt is not the result of an operation initiated by the currently executing software and will not be processed by that software (though, of course, it may).

2. Fault - a condition has been detected that requires software intervention as a direct result of the currently executing software. In general, faults can be handled by the current software, though in many cases common supervisor code within the current process handles the fault. Also, in general, an external device in the real world is not directly involved in either the cause or cure of a fault condition. Often, however, external devices are involved indirectly as, for example, in performing a page turn operation as a result of a page fault.

3. Check - an internal CP consistency problem has been detected which requires software intervention. The condition could be either an integrity violation, reference to a memory module which does not exist, or a power failure. By contrast, a reference to a page which is not resident or an arithmetic operation which causes an exception is a FAULT condition.

A. External Interrupts

1. Operation

External Interrupts operate in either of two modes depending upon whether process exchange is turned on. If process exchange is off, all interrupts are treated as P-300 interrupts. In all cases, except memory increment, the address presented by the controller (or '63 if in standard interrupt mode) is used as the address in segment 0 of a 16-bit vector. This vector, in turn, points to interrupt response code (IRC), also in segment 0, which is entered via a simulated JST* through the vector. Thus, the current P-counter (RPL) is stored in (vector) and execution begins at location (vector) +1 with interrupts inhibited, but with no other keys or modals changed. If in vectored interrupt mode, it is the responsibility of the software to do a CAI. In either mode, the full RP is saved in the register PSWPB.

If process exchange mode is on, an entirely different mechanism operates. In all cases, except memory increment, the address presented by the controller is used as a 16-bit word number offset into the interrupt segment (#4). This segment is guaranteed to be in memory, but STLB misses may occur. The current PB (actually RP) and KEYS (keys and modals) are saved in the u-code scratch registers PSWPB and PSWKEYS. The machine is then inhibited and the IRC begins execution in 64V mode. It is the responsibility of the IRC to issue a CAI. It is important to note that the IRC in the interrupt segment does not belong to any process. PPA points to the PCB of the interrupted process and, in fact, no PCB exists for the IRC. Also, except for PB and KEYS, no registers are saved. In fact, even PSWPB and PSWKEYS are in the register file and not in memory. As a result, the IRC cannot do an enable and must return to the process exchange mechanism (i.e., the dispatcher) as soon as possible. Because of all these restrictions on what the immediate IRC can do, as well as the fact that it does not belong to any process, it is referred to as phantom interrupt code. Unless the job of servicing an interrupt is very simple, phantom interrupt code can do little more than turn off the controller's interrupt mask, issue a CAI, and NOTIFY the real IRC.

A memory increment interrupt is handled the same regardless of the state of process exchange. The address presented by the controller is used as the 16-bit word number in segment 0 (I/O segment) of a 16-bit memory cell to be incremented. If the counter does not overflow (-1->0), the u-code simply returns. With process exchange off, the return is always to the fetch cycle. With process exchange on, the return is either to the fetch cycle or the dispatcher, depending upon where the interrupt was detected. When detecting an interrupt, the dispatcher always insures that RP=PB and that all live keys=KEYS. If memory increment returns, it does so to the top of the dispatcher without having touched PB or KEYS. In this way, memory increment is guaranteed not to destroy any vital information needed by the dispatcher. If the memory cell counter does overflow, an End-of-Range interrupt is generated and then memory increment returns. The

bsequent EOR interrupt will then be treated like any other external
rupt. Figure 8 is a detailed flow chart of the external interrupt
ler.

2. Special Instructions (IRTN, INOTIFY)

Phantom interrupt code has two options for the actions it can take. If the servicing required by the interrupt is very simple, phantom code can completely process the interrupt and return to the dispatcher. If the servicing required is more complex, the phantom code must turn off the controller's interrupt mask and NOTIFY the remainder of the IRC. In the first case, PB and KEYS must be restored from PSWPB and PSWKEYS and then the dispatcher must be entered directly. Since PB cannot be restored in phantom code (the P-counter will point to the instruction in phantom code) and the dispatcher cannot be entered directly (no such instruction exists), the special instruction, IRTN, a 16-bit generic, is executed to perform these functions. After entering the dispatcher via an IRTN, the dispatcher does not know that an interrupt occurred.

In order to NOTIFY a process, phantom code must insure that PB and KEYS are restored before issuing the NOTIFY. The special instruction, INOTIFY, performs the restore and then does the NOTIFY. As NOTIFY, INOTIFY is a three-word generic with two flavors, INOTIFYB and INOTIFYE e the beginning of list option has bit 16=1 and the end of list ion has bit 16=0 in the opcode.

Phantom Interrupt code can issue a CAI in one of two ways. Either an explicit CAI instruction may be issued or the IRTN/INOTIFY instructions can issue it. Bit 15 of the IRTN/INOTIFY instructions is interpreted as follows:

Bit 15 = 0 do not issue CAI
1 issue CAI

In all, there are four INOTIFY instructions as follows:

Name	Bit 15	16	Function
INEC	1	0	End + CAI
INEN	0	0	End + no CAI
INBC	1	1	Beginning + CAI
INBN	0	1	Beginning + no CAI

Figure 9 is a detailed flow chart of the IRTN and INOTIFY instructions.

Faults

aults are CPU events which are synchronous with and, in a loose sense, aused by software. Eleven fault classes have been defined for the P-400. Several of these classes are further subdivided into distinct types. Of the eleven, three are completely new for the P-400 and, of

the other eight, three have expanded meaning when in P-400 mode. The eleven fault classes and their meanings are:

Fault	P-400	P-300
RXM	Restrict mode violation	same
Process	Abort flags word .NE. 0 in PCB on dispatch	N.A.
Page	Page Fault (Page not in memory)	same
SVC	N.A.	Supervisor Call
UII	Unimplemented instruction	same
ILL	Illegal instruction	same
Access	Violation of segment access rights	Page write violation
Arithmetic	All FLEX + IEX (Integer Exception)	FLEX
Stack	Stack overflow/underflow	Procedure Stack (S-Reg) Underflow
Segment	1: Segment # too big	N.A.
	2: Missing segment (SDW fault bit set)	N.A.
Pointer	Fault bit in pointer set	N.A.

The fault handling mechanism consists of two data bases and the CAL instruction. The u-code is in turn divided into a set of 'front-ends' for each fault class and a common fault handler.

1. Data Bases

The fault data bases consist of the fault vectors and concealed stack in the PCB and the fault tables pointed to by the PCB vectors. Figure 10 shows these data bases as well as the mapping of P-300 faults to P-400 faults. Also shown in this figure is the differential action taken according to fault class (e.g., what ring to process the fault in) and the set up the u-code 'front end' must do before going to the common fault handler.

The underlying philosophy of the four fault vectors is that while some faults may need to be processed by ring 0 code, others may be adequately handled in the current ring of the faulting process. The vectors are in the PCB to allow different processes to have different fault handlers. For example, process A may wish to use a system fault routine to handle pointer faults (dynamic linker) while process B may wish to define its own algorithms for resolving pointer faults. Notice that it is always possible for a 'current ring' fault handler to call a ring 0 procedure if the need arises. Note also that page fault has its own vector despite the fact that ring 0 is entered. For the special case of page fault, only a single, system-wide processor will be use and all PCB page fault vectors will point to the same place.

The concealed stack, also in the PCB, is used to allow fault on fault

onditions. For example, it is quite possible to get a segment fault while processing a segment fault. The only fault which cannot cause another fault of any type is page fault. Each frame of the concealed stack contains the PB and keys (KEYSH) of the faulting procedure as well as a fault code (to distinguish different types within each class) and a fault address, if appropriate. The stack itself is circular and must have allocated sufficient frames to handle the longest possible sequence of fault on fault that can occur in ring 0. Such a sequence might be: Pointer (link) fault -> Segment fault -> Stack fault -> Segment fault -> Page fault. Note that this particular sequence occurs before any software fault handler is entered. Also, the first segment fault enters ring 0, so at least a five-level stack is necessary if the original link fault is to be processed correctly.

The second data base consists of four distinct fault tables, each pointed to by a PCB fault vector. Each entry in the table consists of four words of which the first three must be a CALF instruction. Only the page fault table must be locked to memory and only the ring 0 table must be in a pre-defined (SDW exists) segment (otherwise, segment fault might recurse infinitely). Naturally, the ring 0 table, as well as the PCB, is carefully audited by ring 0 procedures.

2. CALF

CALF instruction has two major functions. First, to avoid holding off interrupts for too long, the CALF instruction defines a restart point in fault handling since it has a PB (i.e., it is a macro-machine instruction). As a result, it is quite possible to suspend a process in the middle of getting to a software fault handler. Second, it allows a straightforward mechanism to simulate a procedure call from the faulting procedure (at the instruction causing the fault) to the fault handler.

The instruction itself is a three-word generic in which the second and third words are a 32-bit pointer to the fault handler. To simulate the procedure call, the PB and KEYS from the concealed stack are placed in the fault handler's stack frame along with the other base registers (only the PB and KEYS have been changed to point to the CALF and to enter 64V addressing mode) to be used by the standard procedure return (PRTN) instruction. In addition, the fault code and address are placed in the fault handler's stack as if they were arguments passed by a standard procedure call (PCL) instruction. After the information is moved from the concealed stack it is popped. In all other respects, CALF is identical to PCL.

3. Fault Handler

A fault handler is a u-code routine that is entered from the various fault class 'front ends' and, based on process exchange mode, either simulates a P-300 type fault (JST* through segment 0 fault vectors) or performs the P-400 fault protocol which includes setting up a concealed stack frame, switching to 64V mode, and determining, on the basis of

information provided by the 'front end', which fault vector to use and setting PB to point to the proper CALF in the fault table. Figure 1 is a detailed flow chart of the fault handler and Figure 10 contains a table of the necessary setup performed by each fault class 'front end'. Note that for P-300 faults, the full RP is also saved in the u-code scratch register PSWPB and the machine is inhibited for one instruction if in Ring 0.

C. Checks

Checks, unlike faults, are CPU events which are asynchronous with, and are not caused by, normal instruction execution. Rather, they are events which are either invisible (e.g., an ECC corrected error) or fatal (e.g., missing memory module) to the currently executing procedure and perhaps the CPU entirely (e.g., machine check). Checks essentially represent processor faults as opposed to process or procedure faults. Four check classes have been defined as follows:

Check	P-400	P-300
Power Fail	Power Failure	same
Memory Parity	ECC corrected	
	ECC uncorrected	Memory Parity
Machine Check	Fatal CPU error	same
Missing Memory Module	Memory module does not exist	same

Unlike faults which can be stacked and interrupts which cause a process to be suspended, each check class has a single save area (check block) consisting of eight words in the interrupt segment (#4) in which PB and KEYS (high and low) are saved in the first four locations (check header) and the remaining four locations contain software code (probably a JMP). Figure 12 is a picture of the check data base as well as a description of the necessary u-code setup required before going to the common check handler. In addition to the memory data base, three 32-bit registers are used as a diagnostic status word (DSW) to help a software check handler sort out what happened. Figure 13 shows the format of the DSW.

Check reporting (traps) is controlled by the two low order bits in the modals (KEYSL). The possible modes are:

MCM = 0 no reporting
1 report memory parity (uncorrected) only
2 report unrecovered errors only
3 report all errors

The check trap can result in two possible actions depending upon the type of check that occurred and the type of u-code which was trapped. If the trapped code was either DMX, PIO, or external interrup processing (unless the error was a machine check for RCM parity), or if the check was for an ECC corrected (ECCC) error, the end-of-instruction flag is set, REOIV is set to the proper

offset/vector, MCM is set to 0 (except ECCC which sets it to 2), and a u-code RTN to the trapped step is executed. In this way, the IO bus is always left in a clean state. In all other cases, the check to software occurs immediately. Figure 14 is a detailed flow chart showing the operation of the check trap handlers.

The common check handler is entered from various check 'front ends' and, based on process exchange mode, either simulates a P-300 type check (JST* through segment 0 check vectors) or performs the P-400 check protocol which includes setting up the check header, inhibiting the machine, and switching to 64V addressing mode. In either mode, MCM is set to 0 before going to software. Figure 15 is a detailed flow chart of the check handler and Figure 12 contains a table of the necessary setup performed by each check class 'front end'.

III. REGISTER FILES

The PRIME 400/500 contains four distinct register files. Each file is further divided into halves, each 32 locations (registers) long, and each 16 bits wide. One half is referred to as the high half and the other as the low half. Since both halves are addressed together, each register file contains 32, 32-bit register or 64, 16-bit registers. The register files, numbered from 0, are used as follows:

RF0 - u-code scratch and system registers
RF1 - 32 DMA channels
RF2 - User register set
RF3 - User register set

This layout of register files allows easy expansion to eight register files, thus adding four new user register sets. All user register sets have the same internal format and the DMA register file simply consists of 32 channel registers. Channel register '20 within RF1 is equivalent to the P-300 DMA registers '20 and '21. Channel register '22 is mapped to '22 and '23. In this way, the mapping proceeds for each even register in RF1 to channel register '36, mapped to '36 and '37. All other RF1 registers represent additional DMA channels over the P-300. Figure 16 shows the internal structure (usage) of RF0 and the user register sets (RF2, RF3). Note that all user register sets contain the segment number of the Ready List/PCB segment (OWNERH) and a cell for the modals (KEYSL). It is necessary, before entering process exchange mode, to set OWNERH in ALL register sets to the proper value and to NEVER alter it thereafter. Although all register sets contain a cell for the modals, only the current register set (CRS) contains the valid modals. It is therefore necessary, whenever register sets are switched, to copy the modals into the new register set. Currently, only the Dispatcher switches register sets. CRS is defined and specified by the three bit field labeled 'CRS' in the modals. Since this field can span up to eight register files, but two are used for u-code scratch and DMA, user register sets are numbered from 2 - 7. Of course, only 2 and 3 are currently implemented. Thus, for the P-400/500, the CRS field must always have bit 9 off, bit 10 on, and bit 11 selects the register set (as if 0 and 1 were the numbers). In fact, the u-code will only look at bit 11.

Direct register file addressing (not using CRS) is accomplished either with the LDLR/STLR instructions or via the control panel. The Registe Files are ordered sequentially with an absolute address of 0 addressing RF0-register 0 (u-code scratch/system file), '40 addressing RF1-register 0 (DMA file), '100 addressing RF2-register 0 (user set 2), and '140 addressing RF3-register 0 (user set 3).

Beside each register name, where appropriate, is the PRIME-300 mode mapping from address traps to registers (e.g., the X register is the high half of GR7).

IV. CONTROL PANEL

The control panel for the P-400/500 is the same physical panel used for the P-100/200/300. It's functionality was enhanced by improving the u-code in the CP. All switches and selectors operate exactly as for the P-300 with the exception of the sense switches in the up position. Figure 17 is a diagram of the functionality of the switches. Notice that with all switches down, any FETCH/STORE operations are to/from memory-mapped. As long as segmentation mode is not turned on, mapped and absolute are the same, thus preserving compatibility. If SS4 down were absolute, address traps could not occur and would thus be incompatible. Notice also that SS5-16 in the up position changes meaning depending upon SS4. When mapped, all 12 switches are read as a 12-bit segment number. When absolute, SS11-16 are used as the 6 high order bits of the 22-bit physical address. To address any P-300 registers, all sense switches should be placed in the down position and addresses between 0 and '37 specified.

P-400/500 registers are accessed by raising SS1. Then, if SS2 is down, the low order 5 bits of the address are used to access 32-bit registers 0-'37 within CRS. If SS2 is raised, the full 7 bit address is used to access any register in any register file. The addresses, as shown in Figure 16, are 0-'37=u-code scratch/system, '40-'77=DMA, '100-'137=User set 2, and '140-'177=User set 3. SS4 is used to access either the high half (up) or the low half (down) of the selected register. For all register accesses, the Y+1 functions will advance the register address before the access, exactly as for memory accesses. Wrap around will occur on the appropriate number of bits, since any bits of higher order are ignored for the access.

The control panel data register is TR2H and the address register is TR3. Upon entering the control panel routine, RP is saved in TR3 and (RP) is saved in TR2H. In addition, the keys (KEYSH) are updated to reflect accurately the live keys. Thereafter, TR3H is not altered by the control panel itself so RPH is always remembered. However, on exit, PSH is used to update RPH and KEYS is used to update all the keys. As a result, single stepping can change segments as well as keys and modals. Figure 18 is a detailed flow chart of the control panel routine.

The only exception to the control panel entry protocol is that if a Fault, Check, or external Interrupt attempts to vector through a vector containing 0 in P-300 mode, the following registers will contain:

RP: address of 'trapped' instruction
PBH: SN of 'trapped' instruction
KEYSH: proper keys
TR2H: (data) 0
TR3: (address) 0|0
TR2L: address, in segment 0, of the 'vector' containing 0

V. CP TIMER

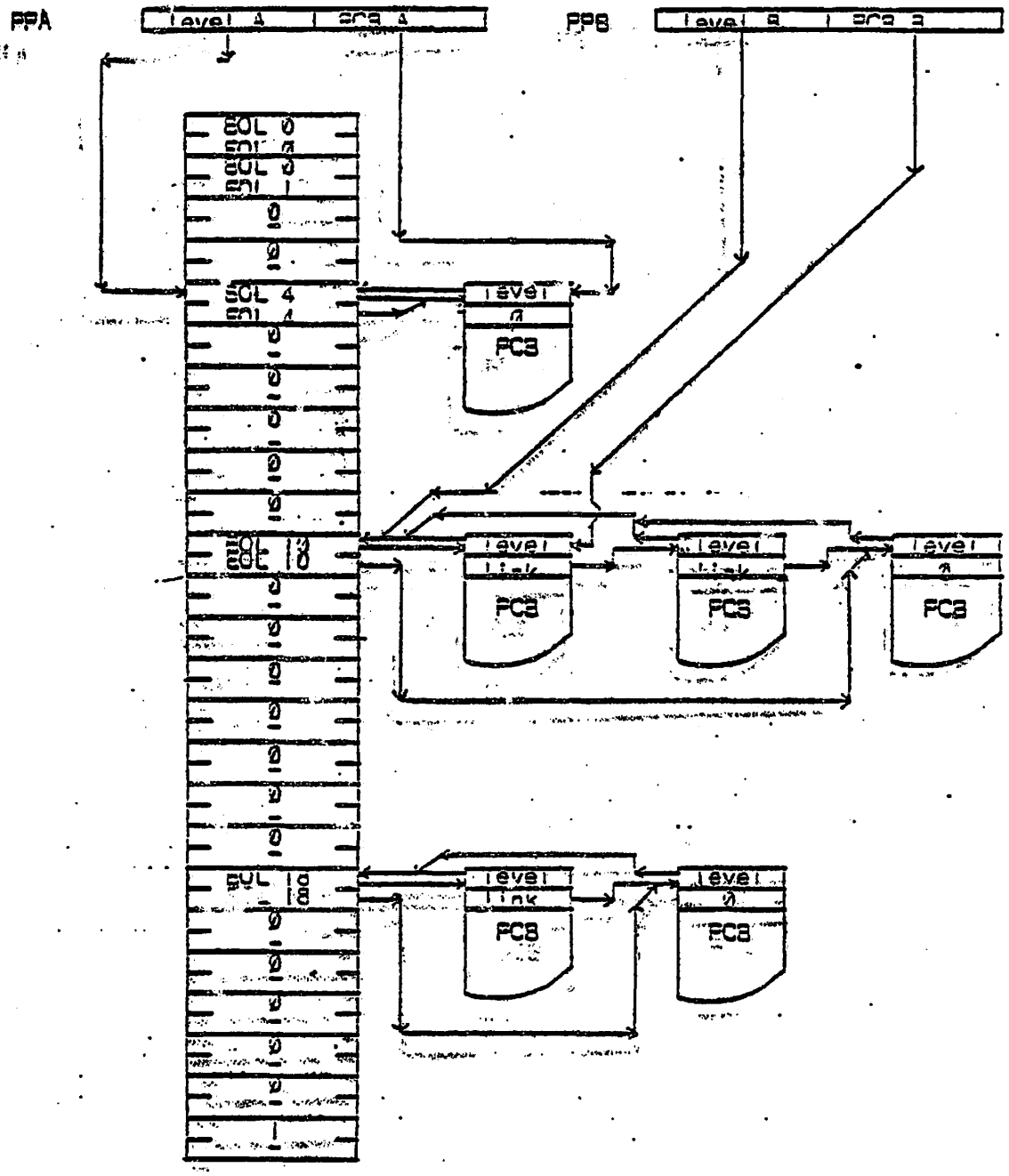
Resolution = 1024 u-sec

Turned on by DISPATCHER before dispatch.

Turned off by:
WAIT after/during save
DISP before changing CRS

On tick, u-code increments the interval timer (TIMER) in RF(CRS). When that overflows, bit 16 in the PCB abort flags (memory) is set to cause a process fault.

It is the responsibility of software that resets the interval timer to maintain the elapsed timer.



Ready List: All pointers are 16-bit word number pointers within the PCB segment. The segment number is contained in the high portion of the OWNER pointer within each register set.

All PCB start addresses must be even (bit 16 = Ø). The end of the ready list is marked with a EOL entry = 1.

FIGURE 1.

WAIT LIST STRUCTURE

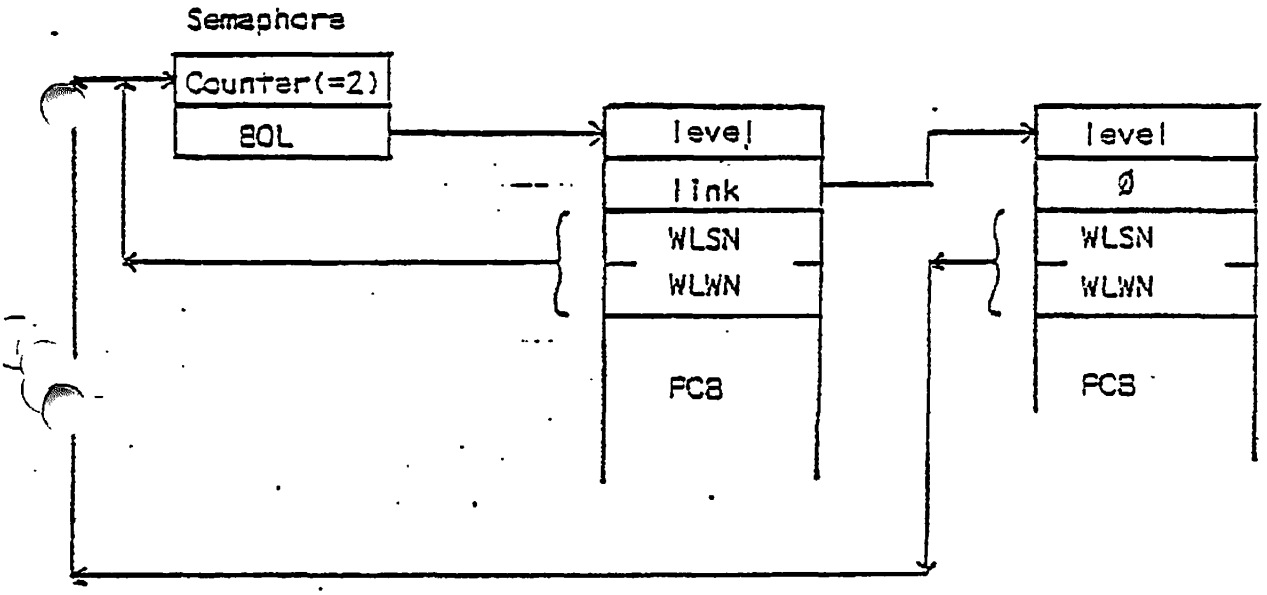


Figure 2.

Process Control Block (PCB)

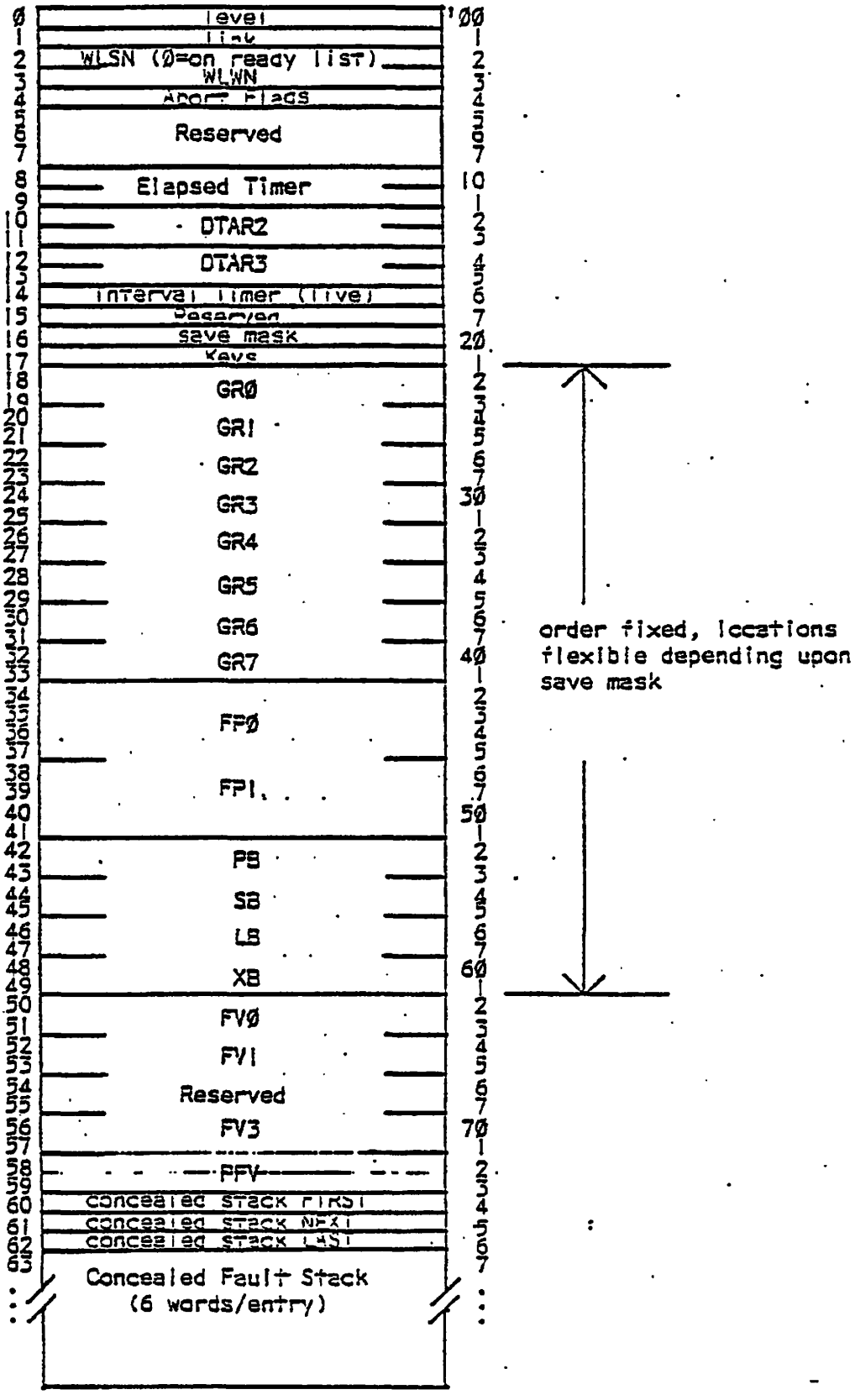


Figure 3.

saved in register file

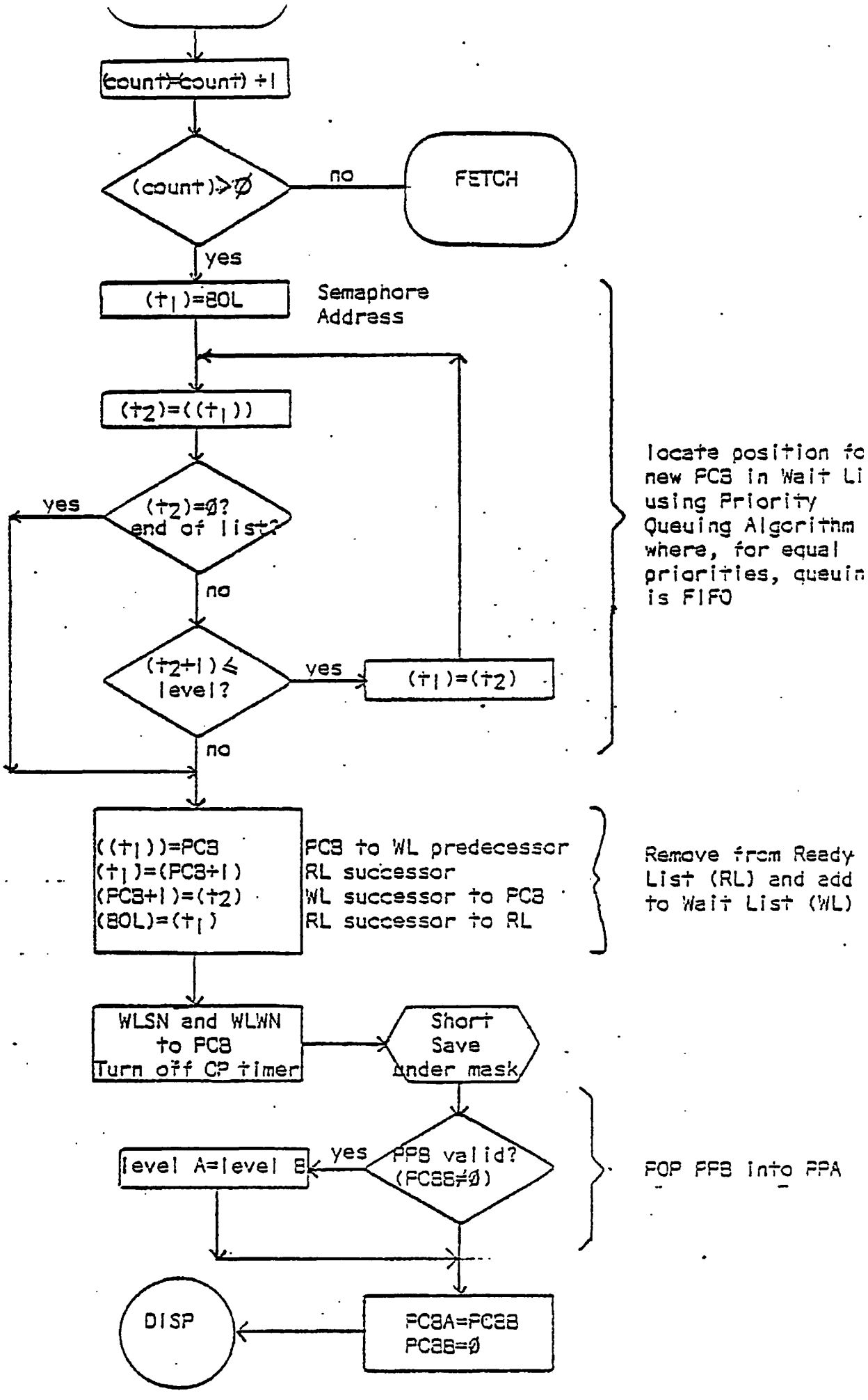


Figure 4.

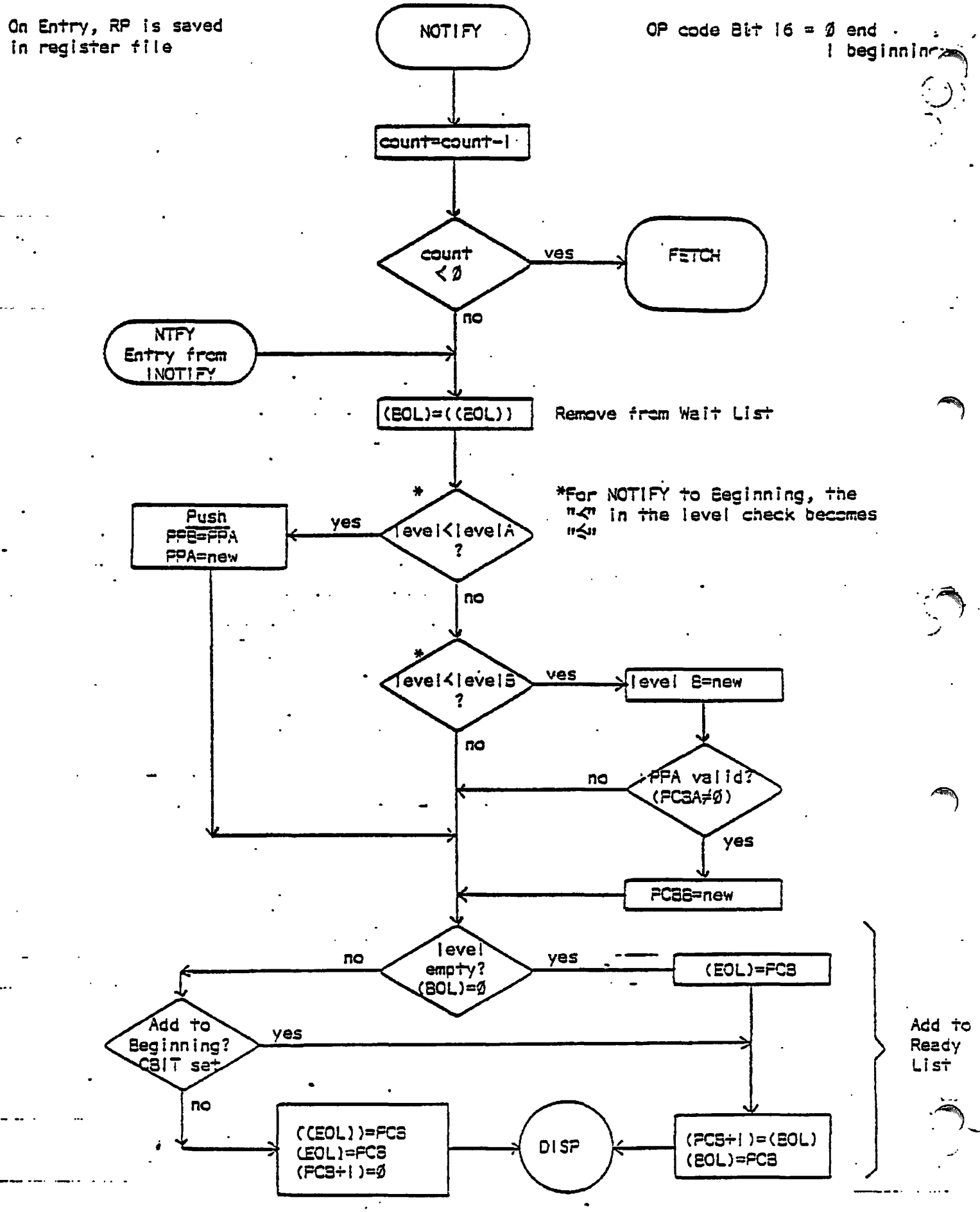

On Entry, RP is saved in register file
NOTIFY
OP code Bit 16 = Ø end
1 beginning
count=count-1
count <Ø
yes
FETCH
no
NTFY
Entry from INOTIFY
(EOL)=((EOL))
Remove from Wait List
*
level<levelA ?
yes
Push
PPB=PPA
PPA=new
*For NOTIFY to Beginning, the "<" in the level check becomes "≤"
no
*
level<levelB ?
yes
level B=new
no
PPA valid? (PCBA≠Ø)
no
yes
PCBB=new
level empty? (BOL)=Ø
no
yes
(EOL)=PCB
Add to Beginning? CBIT set
yes
no
((EOL))=PCB
(EOL)=PCB
(PCB+1)=Ø
DISP
(PCB+1)=(BOL)
(BOL)=PCB
Add to Ready List

Figure 5.

RP and live keys are invalid

Note: All interrupt ...aks result in a return ... h... top of the dis-

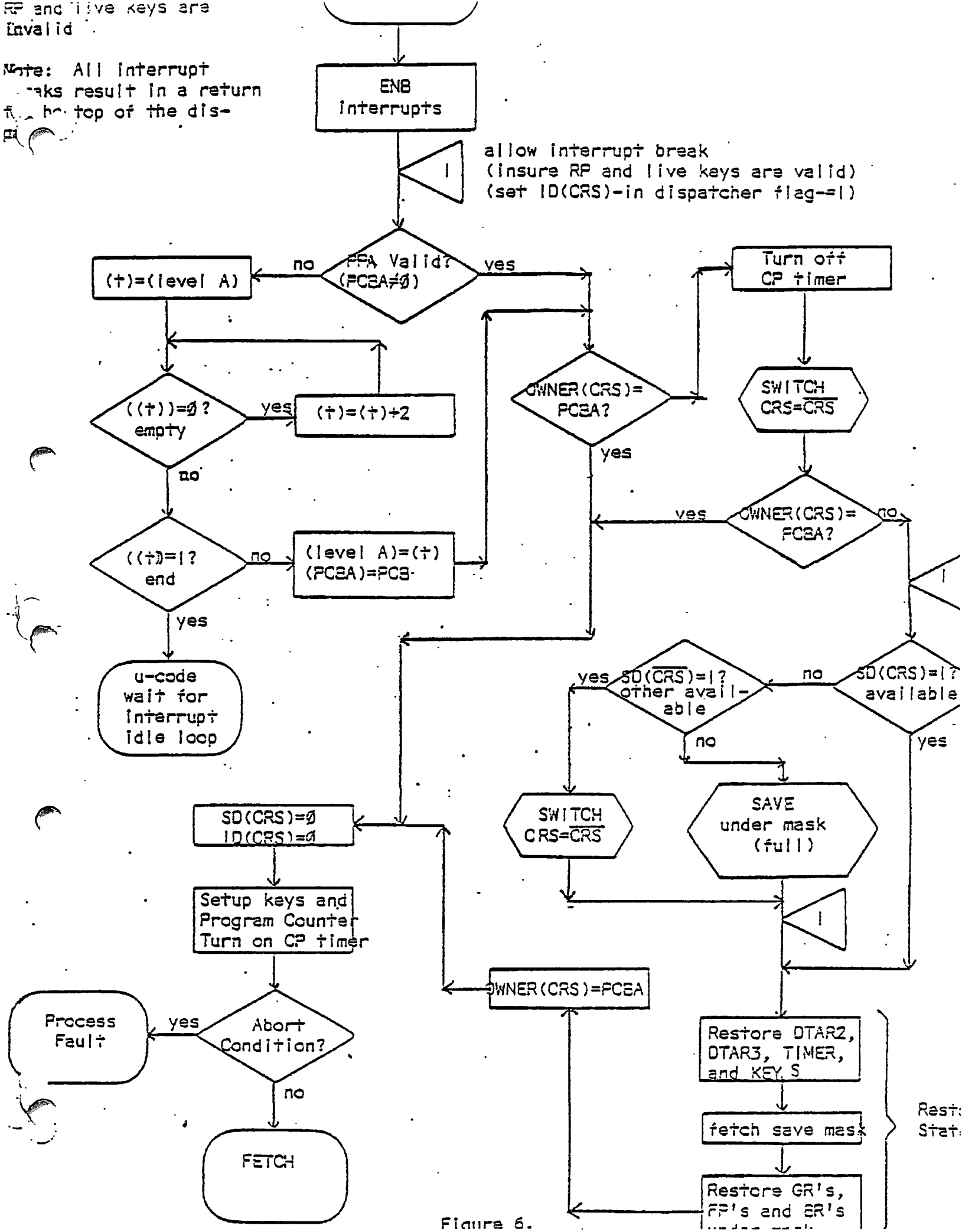


Figure 6.

*The registers to be saved are a parameter passed as a starting RF address in (TRØ,L)

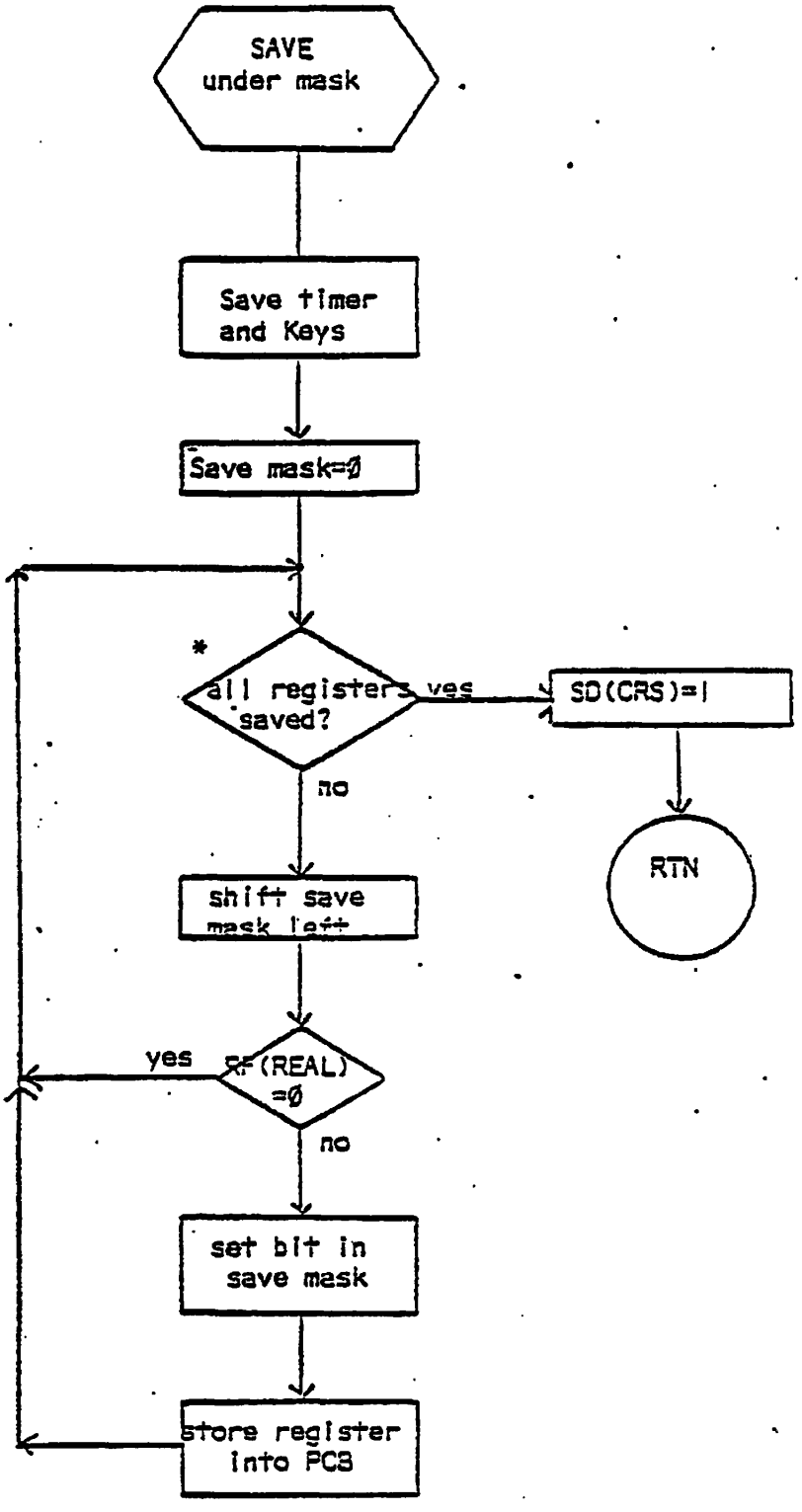


Figure 7.

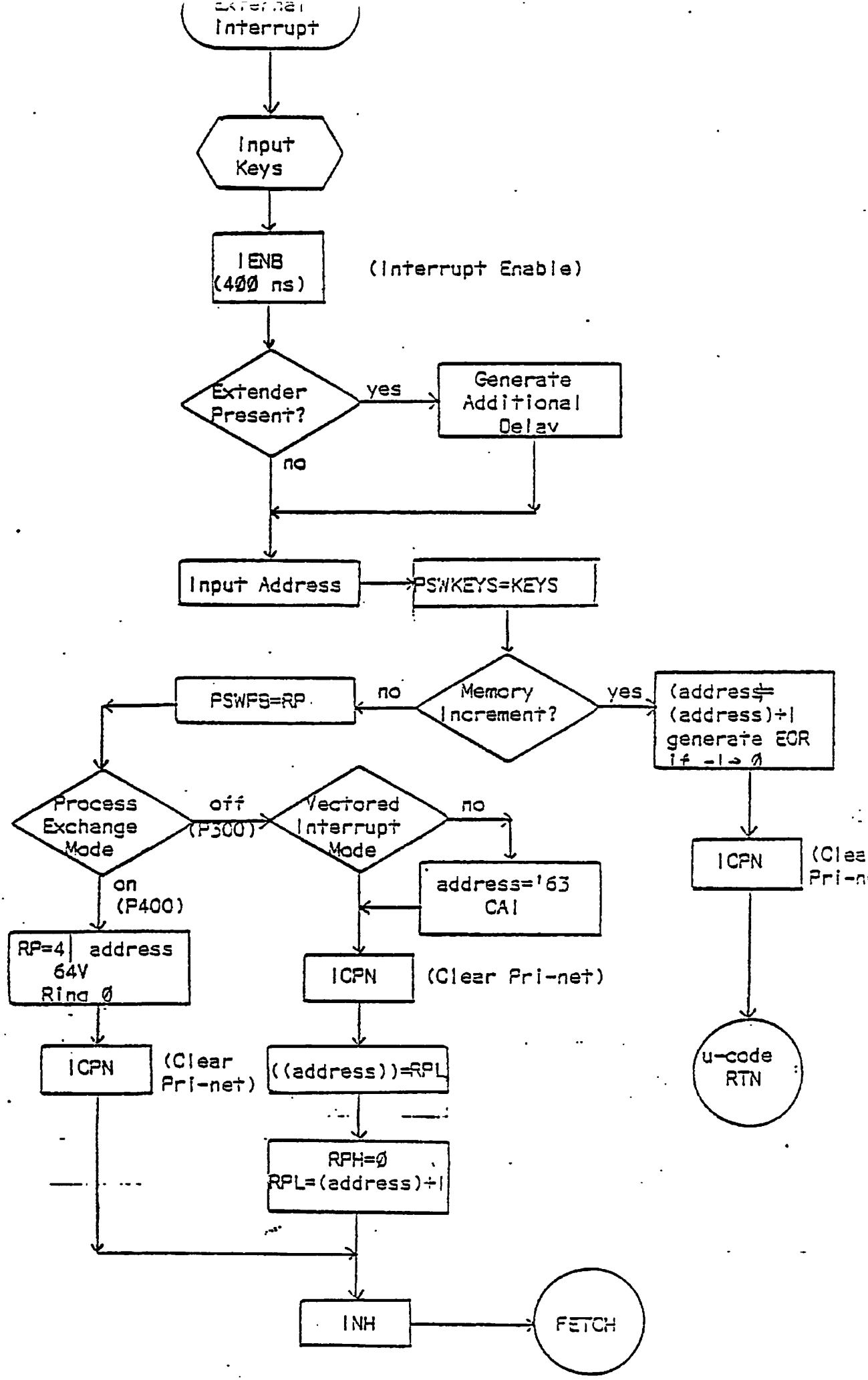
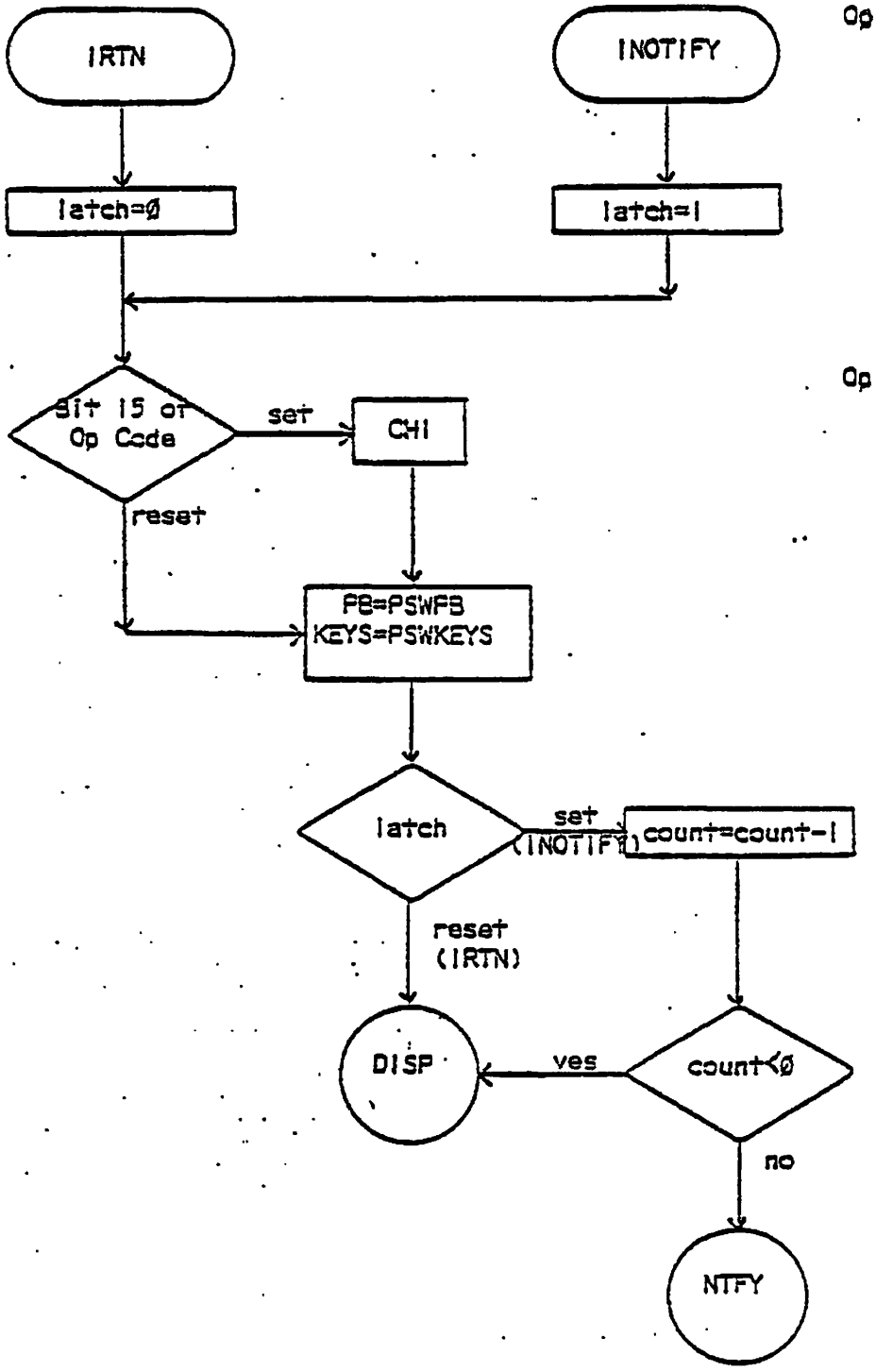

External Interrupt
Input Keys
IENB (400 ns)
(Interrupt Enable)
Extender Present?
yes
Generate Additional Delay
no
Input Address
PSWKEYS=KEYS
Memory Increment?
no
PSWPS=RP
yes
(address)=(address)+1 generate EOR if -1 -> 0
Process Exchange Mode
off (P300)
Vectored Interrupt Mode
no
on (P400)
address='163 CAI
ICPN
(Clear Pri-net)
RP=4 address 64V Ring 0
ICPN
(Clear Pri-net)
((address))=RPL
RPH=0 RPL=(address)+1
ICPN
(Clear Pri-net)
u-code RTN
INH
FETCH

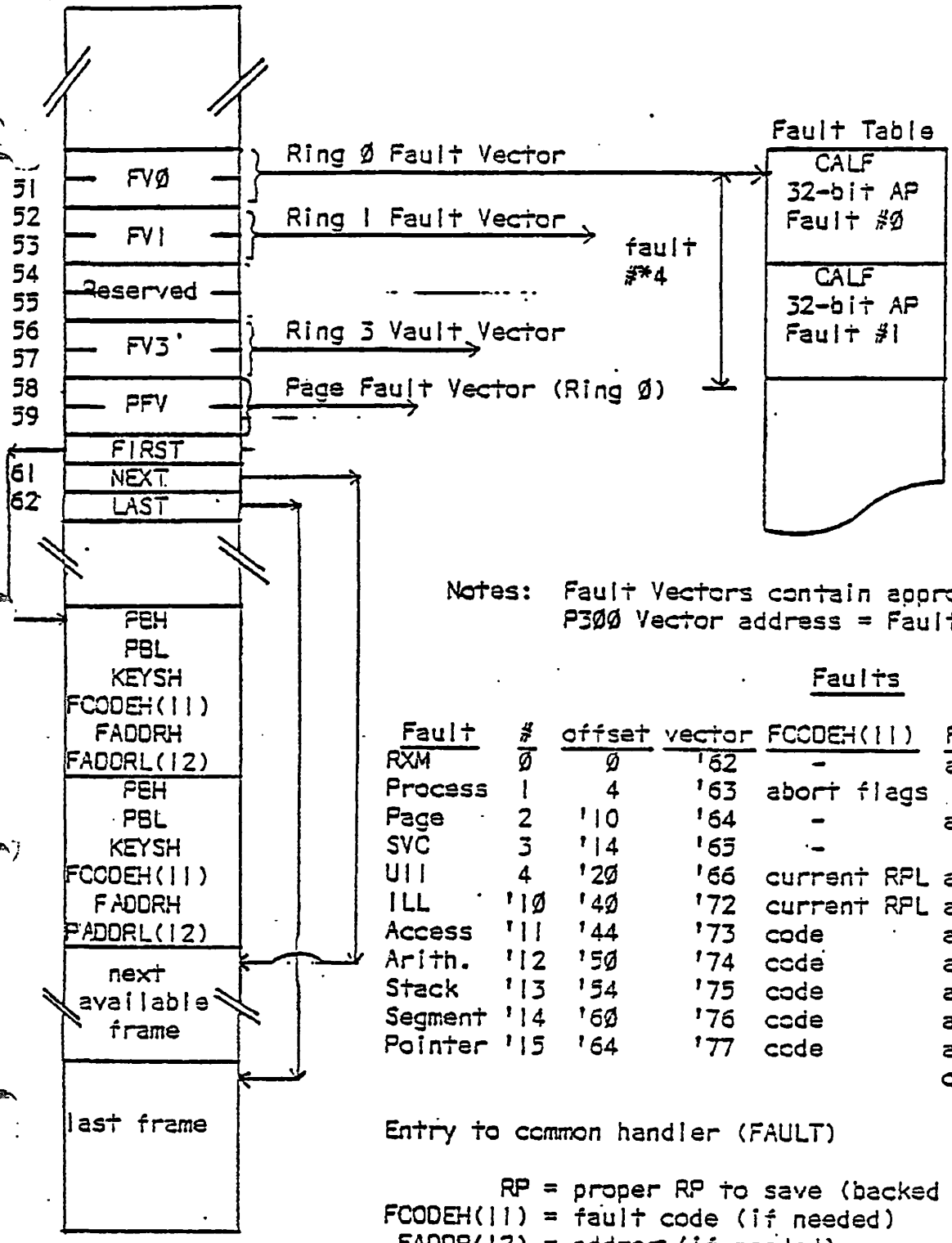
Figure 8



Op:Code Bit 16=Ø end
1 beginning

Op.Code Bit 15=Ø no CAI
1 issue CAI

Figure 9.



Notes: Fault Vectors contain appropriate ring numbers
P3ØØ Vector address = Fault # +'62

Faults

Fault	#	offset	vector	FCODEH(11)	FADDR(12)	Ring	Save
RXM	Ø	Ø	'62	-	address	current	bac
Process	1	4	'63	abort flags	-	Ø	cur
Page	2	'10	'64	-	address	Ø	bac
SVC	3	'14	'65	-	-	current	cur
UII	4	'2Ø	'66	current RPL	address	current	bac
ILL	'1Ø	'4Ø	'72	current RPL	address	current	bac
Access	'11	'44	'73	code	address	Ø	bac
Arith.	'12	'5Ø	'74	code	address	current	cur
Stack	'13	'54	'75	code	address	Ø	bac
Segment	'14	'6Ø	'76	code	address	Ø	bac
Pointer	'15	'64	'77	code	address of pointer	current	bac

Entry to common handler (FAULT)

RP = proper RP to save (backed up if necessary)
FCODEH(11) = fault code (if needed)
FADDR(12) = address (if needed)
FCODEL = fault #*4=P4ØØ fault table offset
LATCH6 = Ø fault
1 page fault (LATCH7 must=Ø)
LATCH7 = Ø go to ring Ø
1 use current ring

Figure 1Ø.

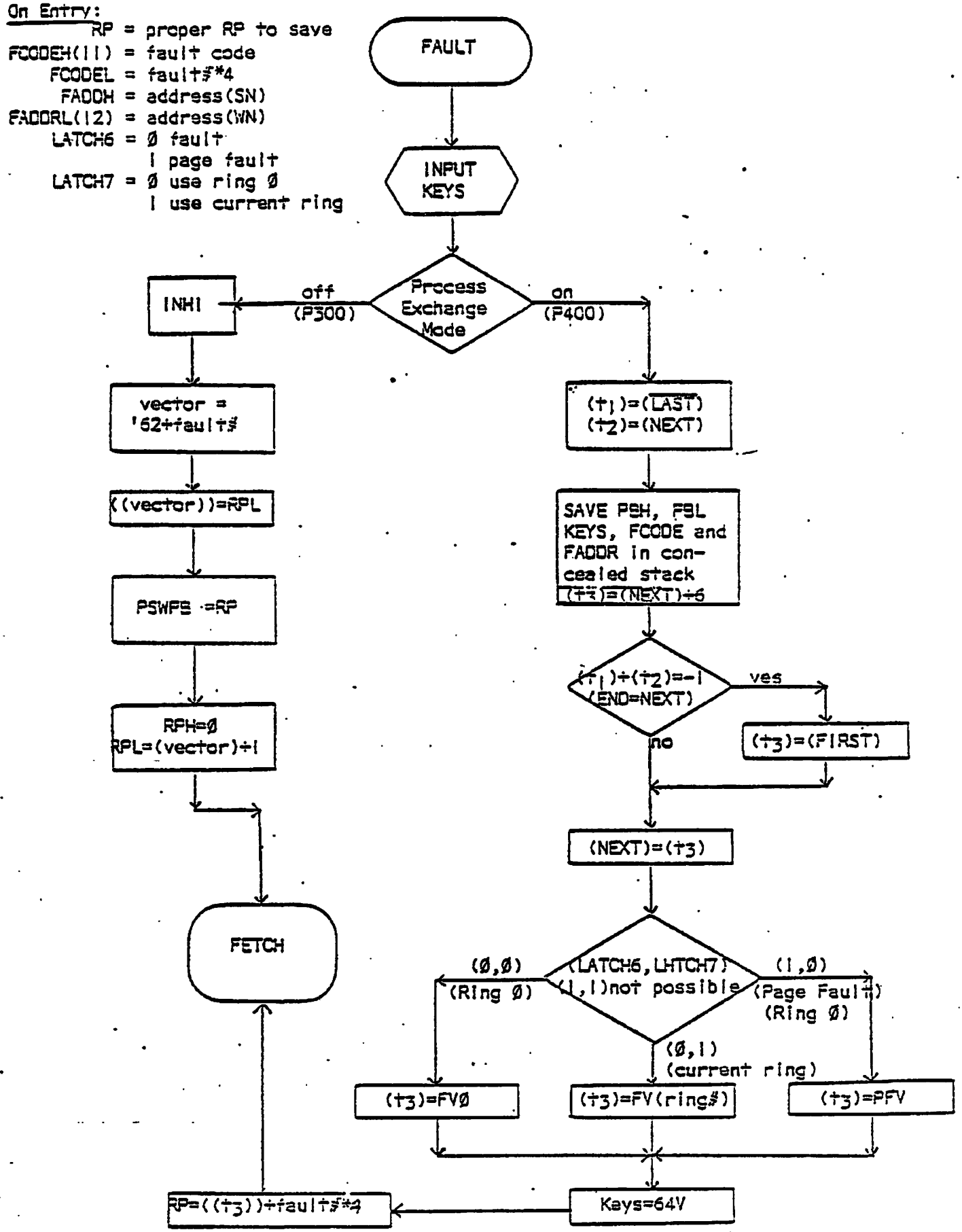
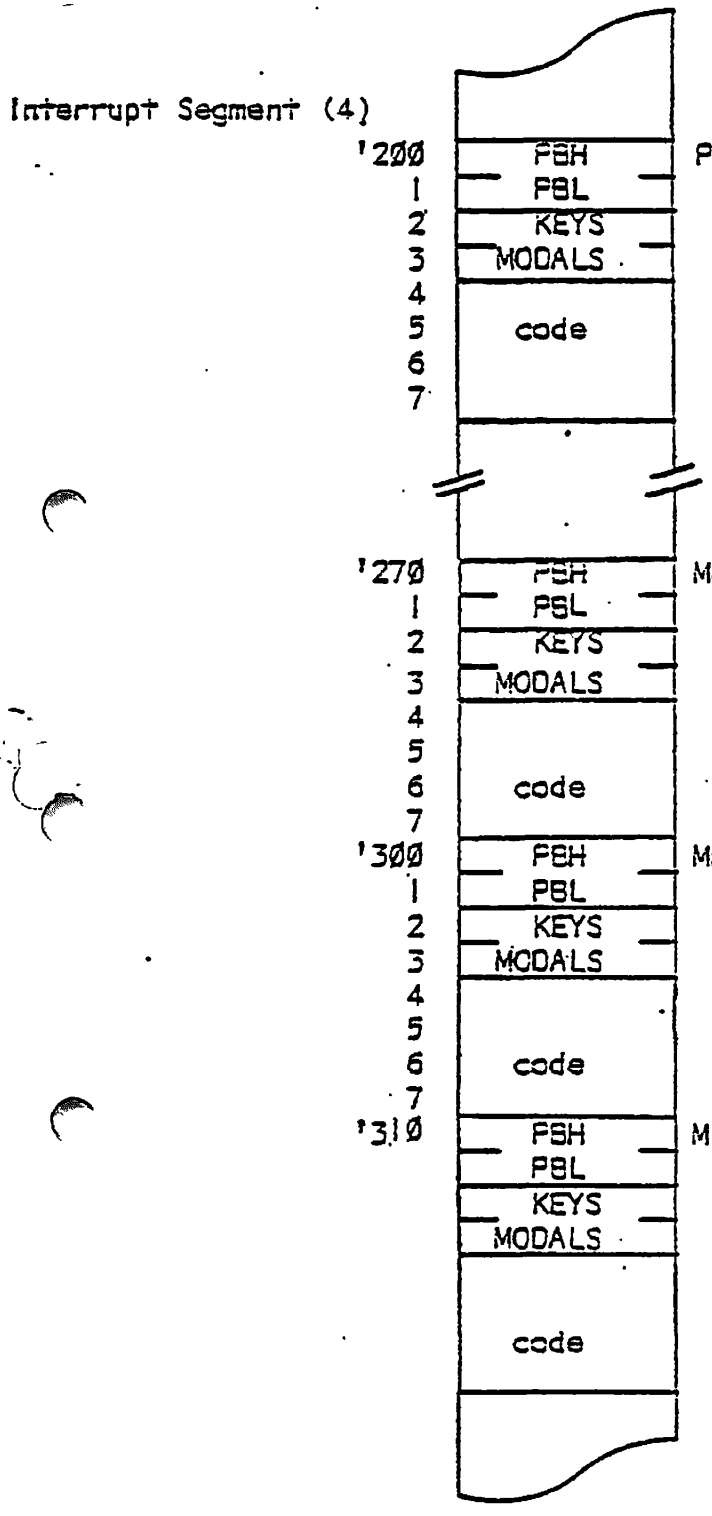

On Entry:
RP = proper RP to save
FCODEH(11) = fault code
FCODEL = fault#*4
FADDH = address(SN)
FADDRL(12) = address(WN)
LATCH6 = Ø fault
1 page fault
LATCH7 = Ø use ring Ø
1 use current ring
FAULT
INPUT KEYS
Process Exchange Mode
off (P300)
on (P400)
INH!
vector = '62+fault#
((vector))=RPL
PSWPB =RP
RPH=Ø
RPL=(vector)+1
FETCH
(†1)=(LAST)
(†2)=(NEXT)
SAVE PSH, PSL KEYS, FCODE and FADDR in concealed stack
(†3)=(NEXT)+6
(†1)+(†2)=-1 (END=NEXT)
yes
no
(†3)=(FIRST)
(NEXT)=(†3)
(LATCH6,LHTCH7)
(Ø,Ø) (Ring Ø)
(1,1)not possible
(1,Ø) (Page Fault) (Ring Ø)
(Ø,1) (current ring)
(†3)=FVØ
(†3)=FV(ring#)
(†3)=PFV
Keys=64V
RP=((†3))+fault#*4

Figure 11.

Software check catchers reside in the interrupt segment (4) and are 8 words each. The first 4 words are used as a PSW save area as:



The check offsets and correspondi:
P300 vectors are:

Check	Offset	Vecto
Power Fail	'200	'60
Memory Par.	'270	'67
Machine Chk.	'300	'70
Missing Mem.	'310	'71

In all cases, the saved PS is the current PS when the check occurre

Entry to common handler (CHECK)

REGIV = P400 offset
P300 vector=(offset-'200)

LATCH5 = Ø RP is proper RP to save
= 1 proper RP is in PSSAVE
(Note: PSSAVE=Ø implies in dispatcher)

Figure 12.

Diagnostic Status Word (DSW)

80 bits, Registers '34,'35&'36 (named DSWRMA, DSWSTAT, and DSWPS)
Bits 1,32: DSWRMA
33,48: DSWSTATH
49,64: DSWSTATL
65,80: DSWPS

Valid on all checks except Power Fail as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	DSWSTATH
CI	MC	MP	MM	Machine Check Code			RCM	ECCU	ECCC	Bup Inv	RP Backup Count			OMX	IO Bus	

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	DSWSTATL
RMA Inved	Reserv	ECCC Syndrome					Mod #	Reserved		u-Verify test #						

33: CI=Check Immediate
34: MC=Machine Check
35: MP=Memory Parity (ECC)
36: MM=Missing Memory
37,39: Machine Check Code
0=Peripheral Data (EPD) Output
1=Peripheral Address (EPA) Input
2=Memory Data (EMD) Output
3=Cache Data (RCD)
4=Peripheral Address (EPA) Output
5=ROX-EPD Input
6=Memory Address (EMA)
7=Register File
40: Not RCM Parity (Reset for RCM Parity error - XCS only)
41: ECCU=ECC Uncorrectable Error
42: ECCC=ECC Corrected Error
43: Bup Inv=RP backup count (44-46) invalid
44,46: RP Backup Count-amount RPL (DSWPS) was incremented in current instruction
47: OMX, set if check occurred during OMX
48: IO Bus, set if check occurred during OMX, PIO or interrupt u-code
49: RMA Inv=DSWRMA invalid (Possible from ECCU and MM only)
50: Reserved
51,55: ECCC Syndrome=5 syndrome bits on a corrected error
56: Mod #=low order address bit of memory module causing the error
57,58: Reserved
59,64: u-Verify test # set on failure during Master Clear or VIRY instruction

Validity:
Always :1-33,43,47-48,59-80
If bit 34 set :37-40
35 :41-42,56 If bit 42 set:51-55
36 :56
If bit 43 reset:44-46

It is the responsibility of the check handling software to clear the DSW after a check has been processed.

Figure 13.

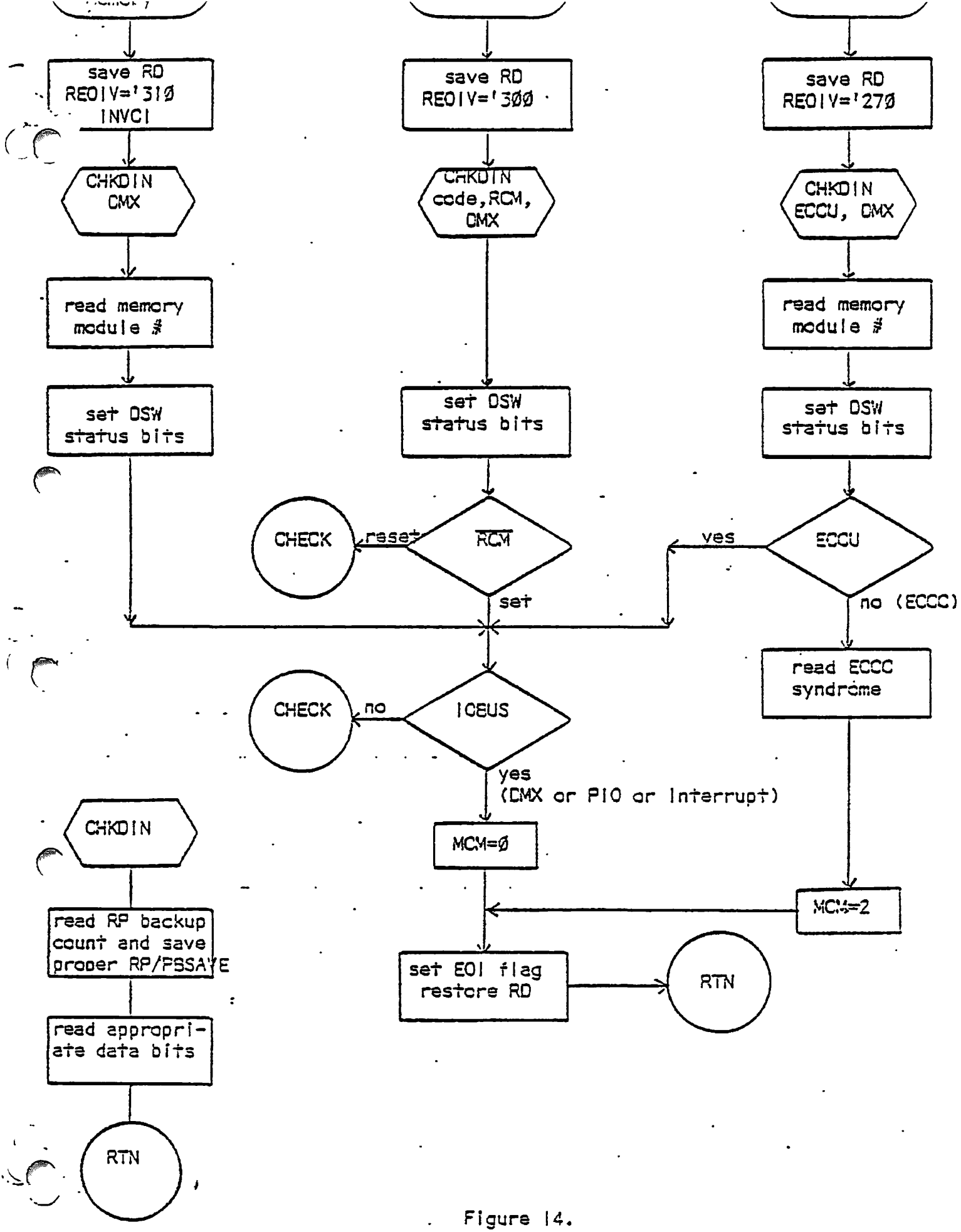
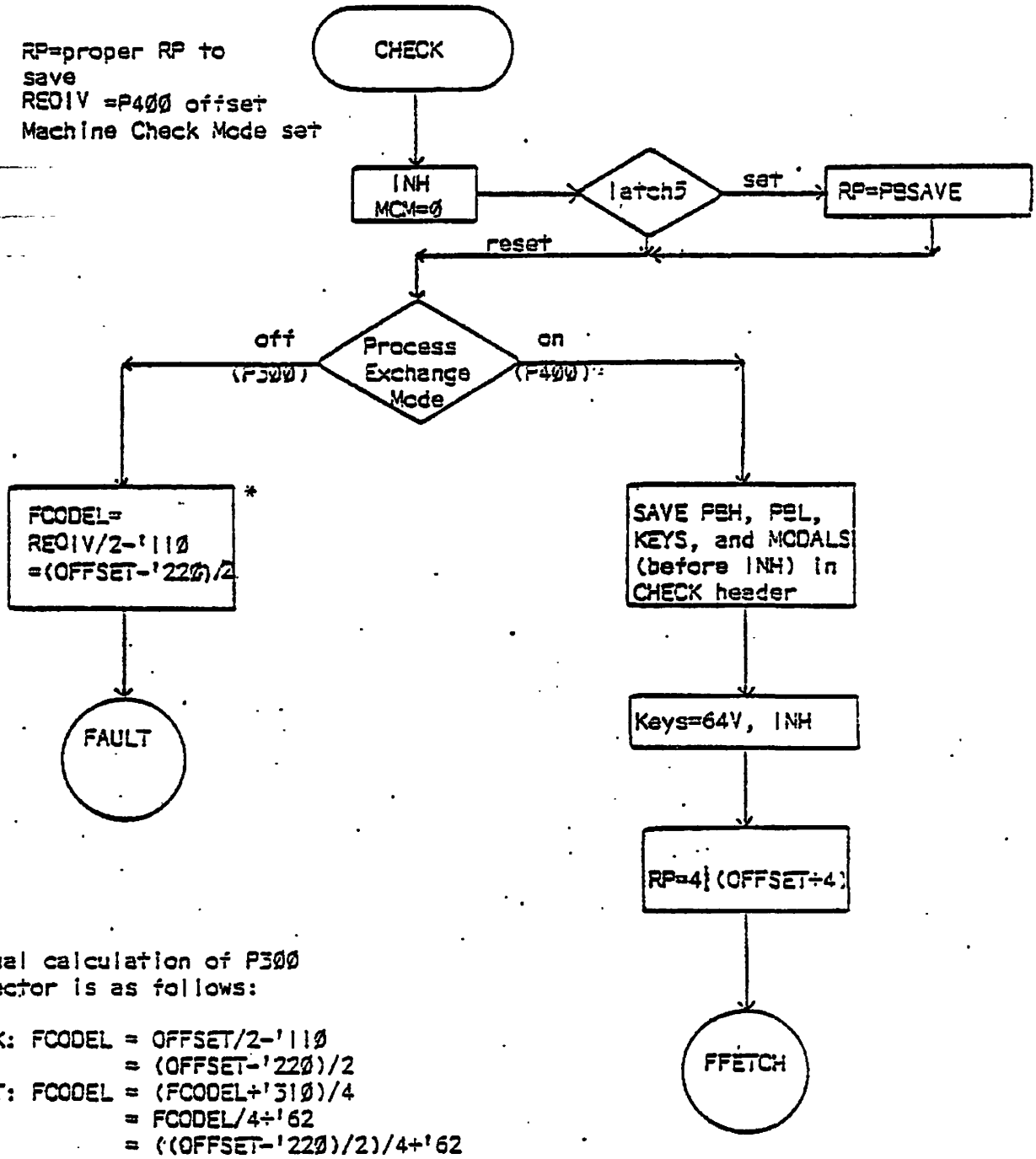

save RD
REQIV='31Ø
INVCI
CHKDIN
CMX
read memory
module #
set DSW
status bits
save RD
REQIV='3ØØ
CHKDIN
code,RCM,
CMX
set DSW
status bits
save RD
REQIV='27Ø
CHKDIN
ECCU, CMX
read memory
module #
set DSW
status bits
CHECK
reset
RCM
set
yes
ECCU
no (ECCC)
read ECCC
syndrome
CHECK
no
ICBUS
yes
(CMX or PIO or Interrupt)
MCM=Ø
MCM=2
set EOI flag
restore RD
RTN
CHKDIN
read RP backup
count and save
proper RP/PSSAVE
read appropri-
ate data bits
RTN

Figure 14.

On Entry: RP=proper RP to save
REQIV =P400 offset
Machine Check Mode set



*The actual calculation of P300 check vector is as follows:

In CHECK: FCODEL = OFFSET/2-'110
= (OFFSET-'220)/2
In FAULT: FCODEL = (FCODEL+'310)/4
= FCODEL/4÷'62
= ((OFFSET-'220)/2)/4+'62
= (OFFSET-'220)/8+'62
= (OFFSET-'200-'20)/8+'62
= (OFFSET-'200)/8-2+'62
= (OFFSET-'200)/8÷'60

This circuitous calculation is used to avoid dividing a negative number on a power fail check.

Note: '200 (Power fail offset)-'220 = -'20.

Figure 15.

	TR0	-	0			40	0	GR0	-	100	
	TR1	-	1			41	1	GR1	-	101	
	TR2	-	2			42	2	GR2(1,A,LH)	-(2,8,LL)	102	
	TR3	-	3			43	3	GR3(EH)	-(EL)	103	
	TR4	-	4			44	4	GR4	-	104	
	25	-	5			45	5	GR5(3,S,Y)	-	105	
	TR6	-	6			46	6	GR6	-	106	
	TR7	-	7			47	7	GR7(Ø,X)	-	107	
	RCMX1	-	10			50	10	FRØ(13)	-	110	
	RCMX2	-	11			51	11	-	-	111	
		RATMPL	12			52	12	FR1(4)	-(5)	112	
	RSGT1	-	13			53	13	-(6)	-	113	
	RSGT2	-	14			54	14	FB	-	114	
	RECC1	-	15			55	15	SB(14)	-(15)	115	
	RECC2	-	16			56	16	LB(16)	-(17)	116	
		REDIV	17			57	17	XB	-	117	
	ZERO	ONE	20	(20)	(21)	60	20	DTAR3(10)	-	120	
	FSSAVE	-	21			61	21	DTAR2	-	121	
			22	(22)	(23)	62	22	DTAR1	-	122	
			23			63	23	DTAR0	-	123	
			24	(24)	(25)	64	24	KEYS	(modals)	124	
			25			65	25	OWNER	-	125	
			26	(26)	(27)	66	26	FCODE(11)	-	126	
			27			67	27	FADDR	-(12)	127	
	FSWPB	-	30	(30)	(31)	70	30	TIMER	-	130	
	FSWKEYS	-	31			71	31	-	-	131	
	FPA:PLA	PCEA	32	(32)	(33)	72	32			132	
	FPB:PLB	PCEB	33			73	33			133	
	ISWRMA	-	34	(34)	(35)	74	34			134	
	IWSTAT	-	35			75	35			135	
	OSWPB	-	36	(36)	(37)	76	36			136	
			37			77	37			137	

KEYSH

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	D	L	Adr	F	I	C	C					I	S		
	P	I	Mode	L	E	C	C					0	0		
I		N		E	X	L	E								
T		K		X		T	Q								

KEYSL (Modals)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	V							CRS		M	P	S	MCM		
N	I									I	X	E			
B	M									C	M	G			

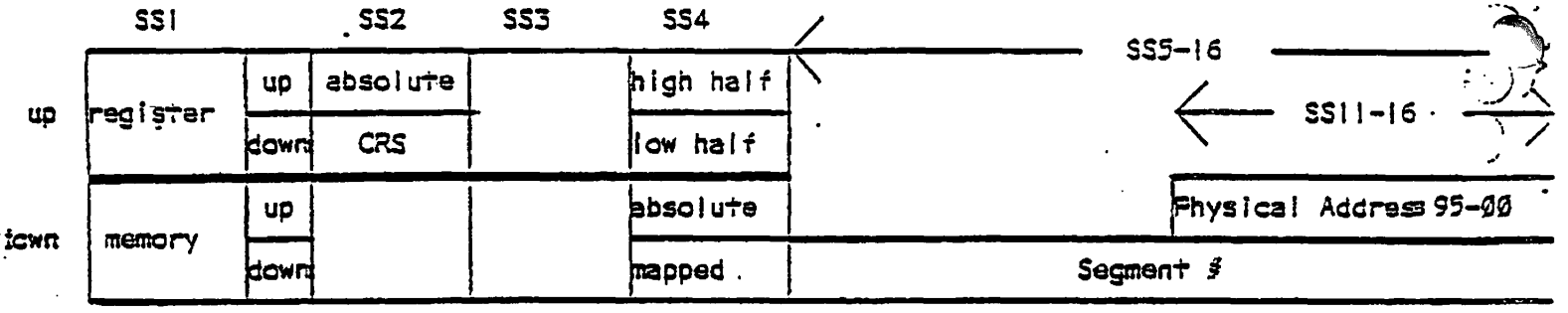
Adr.	Mode
Ø	16S
1	32S
2	64R
3	32R
4	32I
5	
6	64V

FLEX=Ø allows FLEX Faults

ENB: Set=enable interrupts
VIM: Set=Vectored interrupt mode
CRS: Current Register Set
MIO: Set=mapped I/O
PXM: Set=Process Exchange Mode
SEG: Set=Segmentation Mode
MCM: Machine Check Mode

1 Dispatcher
ave Done

Figure 16.



Notes: With all switches down, control panel works exactly as for the P-300 following either a Master Clear or a HALT if not running in segmented mode. It is necessary to make mapped memory accesses if address traps are to be generated. If running segmented, memory accesses will be mapped to segment 0 unless an explicit segment number is entered in SS5-16.

Registers: Register address is in address register (switches down) For CRS, only low order 5 bits are used; for absolute, only low order 8 bits are used Y+1 (STORE/FETCH) operates exactly as for memory with the address being pre-incremented.

Null Vector: In P-300 mode, if an external interrupt, fault, or check attempts to vector through a memory location containing a 0, the following action is taken:

HALT
data and address lights cleared
RP = address trapped
PBH = RPH
TR2L = address of vector

Figure 17.

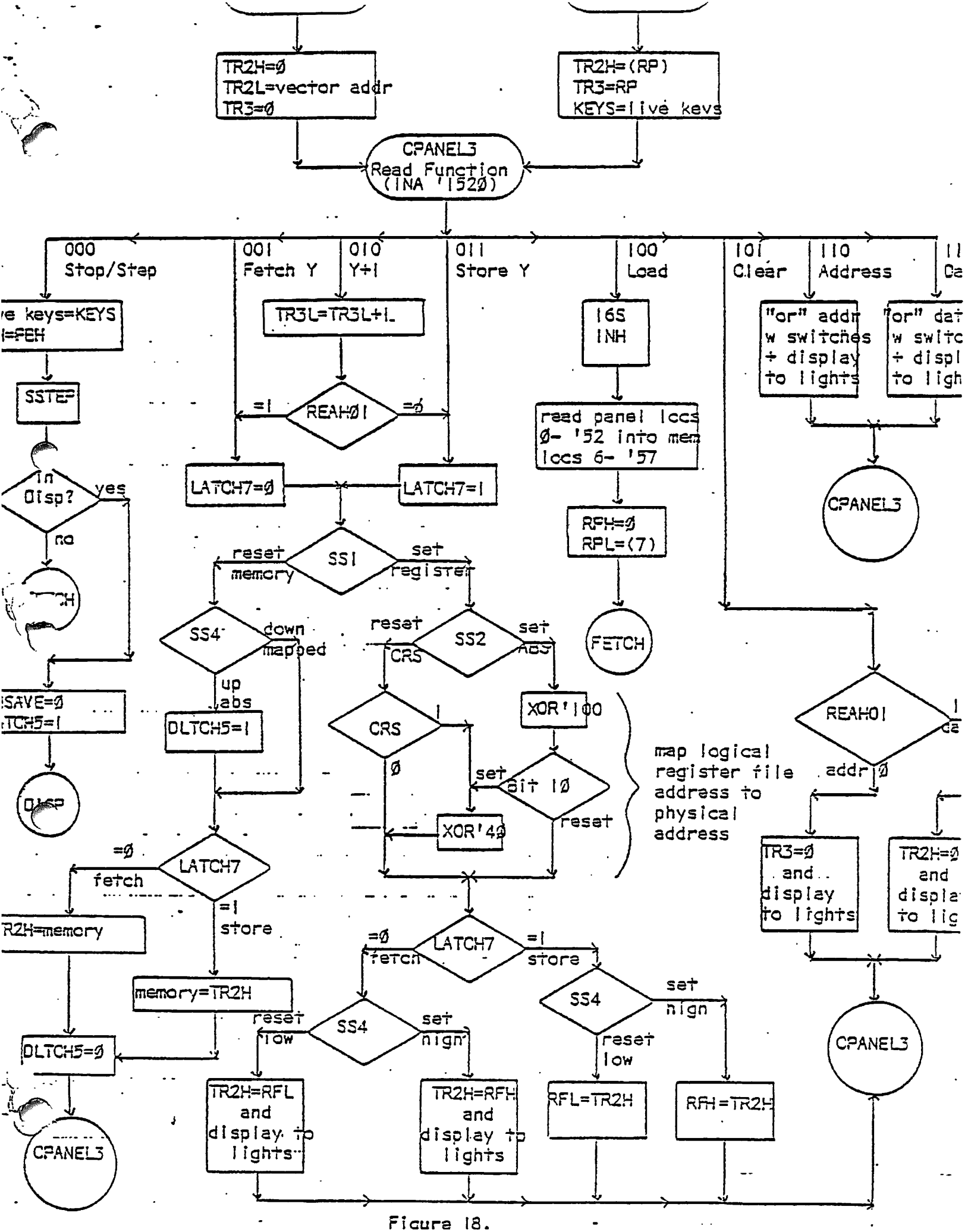

TR2H=Ø
TR2L=vector addr
TR3=Ø
TR2H=(RP)
TR3=RP
KEYS=live keys
CPANEL3
Read Function
(INA '152Ø)
000
Stop/Step
001
Fetch Y
010
Y+1
011
Store Y
100
Load
101
Clear
110
Address
e keys=KEYS
=FEH
SSTEP
In
Disp?
yes
no
SAVE=Ø
TCH5=1
DISP
TR3L=TR3L+1
=1
READØ1
=Ø
LATCH7=Ø
LATCH7=1
SS1
reset
memory
set
register
SS4
down
mapped
up
abs
DLTCH5=1
reset
CRS
SS2
set
abs
CRS
1
Ø
set
bit 1Ø
reset
XOR'1ØØ
XOR'4Ø
=Ø
LATCH7
fetch
=1
store
R2H=memory
memory=TR2H
DLTCH5=Ø
CPANEL3
=Ø
LATCH7
fetch
=1
store
SS4
reset
low
set
high
TR2H=RFL
and
display to
lights
TR2H=RFH
and
display to
lights
SS4
set
high
reset
low
RFL=TR2H
RFH=TR2H
16S
INH
read panel locs
Ø- '52 into mem
locs 6- '57
RFH=Ø
RPL=(7)
FETCH
"or" addr
w switches
+ display
to lights
"or" dat
w switc
+ displ
to ligh
CPANEL3
map logical
register file
address to
physical
address
READØ1
addr Ø
TR3=Ø
and
display
to lights
TR2H=Ø
and
displa
to lig
CPANEL3

Figure 18.