Processor Management and Initialization

3

CHAPTER 3
PROCESSOR MANAGEMENT AND INITIALIZATION

This chapter describes the facilities for initializing and managing the operation of the 80960SA/SB processor. Included is an overview of the processor-management facilities and a description of the steps required to initialize the processor. Appendix D gives a listing of the necessary 80960SA/SB code to initialize the processor.

OVERVIEW OF PROCESSOR-MANAGEMENT FACILITIES

This chapter and Chapters 5, 6, 9, and 11 describe the 80960SA/SB's processor-management facilities. These facilities are primarily software-related, although some hardware considerations are also discussed.

For the purpose of discussion in these chapters, it is assumed that the processor is going to execute a program made up of a system kernel (or executive) and application code. This program may be located in ROM or RAM.

Such a program has the following facilities available to it to initialize, communicate with, and control the processor:

- Instruction List
- System Data Structures
- Interrupts
- IACs
- Faults

These facilities allow system hardware and the kernel to initialize the processor and initiate instruction execution. They also provide software or external agents with methods of interrupting the processor to service external I/O devices.

The following paragraphs give an overview of these processor-management facilities.

Instruction List

At the most rudimentary level, the processor is controlled through a stream of instructions that the processor fetches from memory and executes one at a time. Once the processor is initialized, it begins executing instructions and continues until it is stopped.

System Data Structures

The i960 architecture defines several system-data structures that reside in memory. These data structures offer a means of configuring the processor to operate in a specific way.

The system data structures can be located anywhere in the processor's address space. The processor gets pointers to most of these data structures from the initial memory image (IMI). The IMI is described later in this chapter in the section titled "Initial Memory Image."

The interrupt table provides pointers to interrupt-handling procedures. The interrupt vector numbers act as indices into this table. For the purpose of handling interrupts, a separate interrupt stack is maintained in the address space. The interrupt mechanism is described in Chapter 5.

The fault table provides pointers to fault-handling procedures. When the processor detects a fault, it generates a fault vector number internally that provides an index into the fault table. The trace table provides pointers to trace-fault-handling procedures. The fault mechanism is described in Chapter 6.

The system-procedure table contains pointers to the kernel procedures, which are accessed using the system call (**calls**) mechanism. The system-procedure-table structure is described in Chapter 4 in the section titled "System-Procedure Table."

The processor uses two stacks for procedure calls: the local procedure stack and the (optional) supervisor stack. These stacks are described in Chapter 4.

The processor also contains a register, called the process controls register, that it uses to store information about the current state of the processor and the program it is executing. The process controls are described later in this chapter in the section titled "Process Controls."

Interrupts

The processor supports two methods of asynchronously requesting services from the processor: interrupts and IAC messages. Interrupts are the more common of the two.

An interrupt is a break in the control flow of a program so that the processor can handle a more urgent chore. Interrupt requests are generally sent to the processor from an external source, often to request I/O services. When the processor receives an interrupt request, it temporarily stops work on its current task and begins work on an interrupt-handling procedure. Upon completion of the interrupt-handling procedure, the processor generally returns to the task that was interrupted and continues work where it left off.

Interrupts also have a priority, which the processor uses to determine whether to service the interrupt immediately or to postpone service until a later time.

Faults

While executing instructions, the processor is able to recognize certain conditions that could cause it to return an inappropriate result or that could cause it to go down a wrong and possibly disastrous path. One example of such a condition is a divisor operand of zero in a divide operation. Another example is an instruction with an invalid opcode. These conditions are called faults.

The processor handles faults almost the same way that it handles interrupts. When the processor detects a fault, it automatically stops its current processing activity and begins work on a fault-handling procedure.

PROCESS CONTROLS

The process-controls word (shown in Figure 3-1) contains miscellaneous pieces of information to control processor activity and show the current state of the processor. The various functions of this field are described in the following paragraphs.

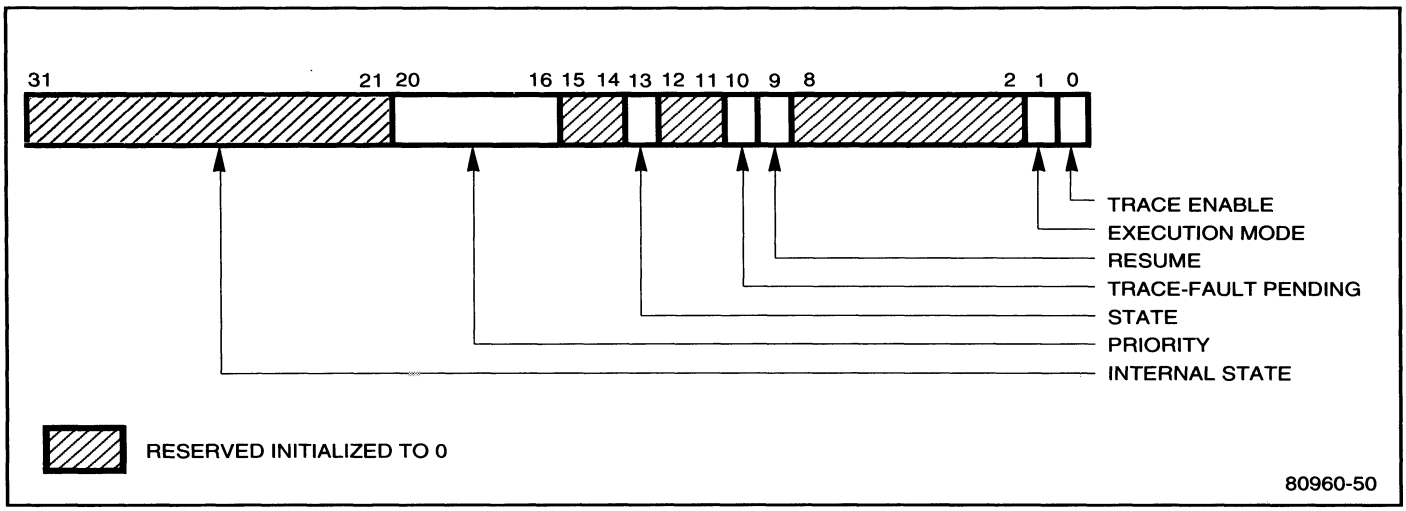

Figure 3-1: Process-Controls Word

The *execution mode* flag determines whether the processor is operating in the user mode (clear) or supervisor mode (set). The processor automatically sets this bit on a supervisor call and clears it on a return from supervisor mode.

The *priority* field determines the priority (from 0 to 31) of the processor. When the processor is in the executing state, it sets its priority according to this value.

The *state* flag determines the state of the program. This bit tells software whether the processor:

- is currently executing a program (0) or
- is servicing an interrupt (1).

The *trace-enable* and *trace-fault-pending* flags control tracing. The trace-enable field determines whether trace faults are to be generated (set) or not-generated (clear). The trace-fault-pending field is a flag that the processor uses to determine if a trace event has been detected (set) or not (clear). The use of these fields is discussed in detail in Chapter 7.

The *resume* flag signals the processor that an instruction has been suspended. The processor sets this flag whenever it suspends an instruction to handle an interrupt or fault. On a return from the interrupt or fault handler, the processor checks this flag and performs an instruction resumption action if the flag is set.

NOTE

The resume flag is implemented in the 80960SB processor, but not in the 80960SA processor. In the 80960SB, instruction suspension is used for long (≥100 clock) floating point operations.

All of the bits in the process controls are set to zero as part of the initialization procedure. Bits 2 through 8, 11, 12, 14, 15, and 21 through 31 are reserved. These bits should not be altered following initialization.

Changing the Process Controls

The kernel can change the process controls using any of the following three methods:

- Modify-process-controls instruction (**modpc**)
- Alter the saved process controls prior to a return from an interrupt handler
- Alter the saved process controls prior to a return from a fault handler

The **modpc** instruction reads and modifies the process controls cached in the processor.

In the latter two methods, the kernel changes the process controls in the interrupt or fault record that is saved on the stack. On the return from the interrupt or fault handler, the modified process controls are copied into the processor's internal process controls.

NOTE

Changing the saved process controls by means of a fault handler can only be used if the fault handler was invoked by means of an implicit supervisor call.

When the process controls are changed as described above, the processor acts on the changes as soon as it receives the new information, except for the following situation.

If the **modpc** instruction is used to change the trace-enable flag, the processor does not guarantee to act on the change until after up to four more instructions have been executed.

PRIORITIES

The processor provides a priority mechanism for determining the order in which programs, interrupts, and IACs are worked on. Priorities range from 0 to 31, with 31 being the highest priority. Each interrupt vector is assigned a priority. Also, when the processor is executing a program, it sets its priority according to the priority field of the process controls.

Interrupt priorities serve two functions. First, they determine if the processor will service an interrupt immediately or delay servicing it with respect to its current priority. Second, they determine which interrupt of several interrupts is serviced first.

When the processor receives an IAC, it always services it immediately (i.e., treats the IAC as if it has a priority of 31).

PROGRAM STATES

The processor can switch between the executing and interrupted states.

Software can change the state of the processor in either of the following ways:

- issue a reinitialize processor IAC
- issue an interrupt IAC

The reinitialize processor IAC forces the processor to reinitialize itself using pointers from a new IMI. The interrupt IAC generates an interrupt through the IAC mechanism.

Executing and Interrupted State

In the executing state, the processor is executing the program. If the processor is interrupted while in the executing state, it saves the current state of the program, switches to the interrupted state, and services the interrupt. Upon returning from the interrupt handler, the processor resumes work on the program.

Stopped State

In the stopped state the processor ceases all activity. The only way to get the processor out of the stopped state is to reinitialize the processor with a hardware reset.

INSTRUCTION SUSPENSION

When the processor is interrupted while it is in the midst of executing an instruction, it does one of three things before it services the interrupt:

1. It completes the instruction.
2. It terminates the instruction and sets the processor state so that it is as if execution of that instruction had not yet begun.
3. It suspends the instruction and saves the necessary resumption information so that execution of the instruction can be continued when the processor begins work on the program again. This course of action is reserved for 80960SB floating point instructions that have a long execution time (≥100 cycles) and that alter the internal and external processor state as they execute.

Which of these steps the processor takes depends on the instruction being executed. However, whichever step it takes is transparent to the software. The processor automatically saves the necessary state information so that work on the program can be resumed with no loss of information.

Refer to the section in Chapter 5 titled "Interrupt Handling Actions" for more information on how resumption information is saved when an interrupt is serviced.

NOTE

As described earlier in this chapter (in the description of the resume flag), instruction resumption is an extension to the i960 architecture that is supported in the 80960SB processor but not in the 80960SA processor.

MEMORY REQUIREMENTS

The processor provides a 2^{32}-byte address space. This address space can be mapped to read-write memory, read-only memory, and memory-mapped I/O. (The processor does not provide a dedicated, addressable I/O space.)

The address space is linear (or flat): there are no subdivisions of the address space such as segments. For the purpose of memory management, an external memory management unit (MMU) may subdivide memory into pages or restrict access to certain areas of memory to protect kernel code and data. But from the point of view of the processor, the address space is linear.

All of the address space is available for general use except the upper 16M bytes ($FF000000_{16}$ to $FFFFFFFF_{16}$), which are reserved for special functions. (One of these functions, the IAC message, is described in Chapter 11.)

An address in memory is a 32-bit value in the range 0 to $FFFFFFFF_{16}$. It can be used to reference a single byte, 2 bytes, 4 bytes, 8 bytes, 12 bytes or 16 bytes of memory depending on the instruction being used. (Refer to the descriptions of the load and store instructions in Chapter 9 for information on multiple-byte addressing.)

Memory Restrictions

The processor requires that the memory to which the address space is mapped has the following capabilities.

- It must be byte addressable.
- It must support burst transfers (i.e., transfers of blocks of contiguous bytes up to 16 bytes in length).
- It must guarantee *indivisible* access (read or write) for memory addresses that fall within 16-byte boundaries.
- It must guarantee *atomic* access for memory addresses that fall within 16-byte boundaries.

The latter two capabilities are only required when the 80960SA/SB bus is shared by other bus masters.

An indivisible access guarantees that a processor reading or writing a set of memory locations will complete the operation before another processor can read or write the same location. The processor requires indivisible access within an aligned, 16-byte block of memory.

An atomic access is a read-modify-write operation. Here external logic must guarantee that once a processor begins a read-modify-write operation on a set of memory locations, it is allowed to complete the operation before another processor is allowed to access the same location.

As described above, the processor requires that when one processor is performing an atomic operation within an aligned, 16-byte block, other processors are delayed from performing another atomic operation within that block until the first operation has been completed.

The 80960SA/SB processor provides two features to aid in implementing the memory requirements described above: BLAST line and a LOCK line on the local bus.

The 80960SA/SB processor can access memory in cycles of 1-, 2-, 4-, 8-, 12-, or 16-bytes. When making a multiple-byte access, the processor thus sends the memory controller a base address on the address lines. BLAST is asserted on the last data transfer to signal the end of the multiple word cycle.

The $\overline{\text{LOCK}}$ line is used to synchronize atomic operations. When a processor performs an atomic operation, it first examines the $\overline{\text{LOCK}}$ line. If it is asserted, the processor waits until the line is not asserted (i.e., spins on the $\overline{\text{LOCK}}$ line). If the line is not asserted, the processor asserts the $\overline{\text{LOCK}}$ line when it is performing an atomic read and deasserts the line when it performs the companion atomic write.

The $\overline{\text{LOCK}}$ line mechanism allows only one atomic operation to be carried out in memory at one time.

SOFTWARE REQUIREMENTS FOR PROCESSOR MANAGEMENT

The processor-management facilities described earlier in this chapter allow the processor to be configured and operated in several ways. This section lists the data structures that the kernel must supply to operate the processor.

To use the processor, the kernel must provide the following items:

- IMI
- Other System Data Structures
- Address Space
- Stacks
- Code

The IMI comprises the minimum data structures that the processor needs to initialize the system.

As part of the initialization procedure, a more complete set of system data structures are established in memory. These data structures include an interrupt table and a fault table. If the system call mechanism is going to be used, a system procedure table is required.

Two stacks are also required: an interrupt stack and a local (or user) procedure stack. The initial stack pointer for the interrupt stack is given in the IMI. The initial stack pointer (SP) for the local-procedure stack is given in local register r1; the initialization code is required to establish the SP value in this register.

If the supervisor call mechanism is to be used, a supervisor stack must also be provided. The initial stack pointer for this stack is given in the system-procedure table. The supervisor stack can be placed anywhere in the address space.

NOTE

The section in Chapter 4 titled "Hints on Using the User-Supervisor Protection Model" describes an application of the user-supervisor protection model, in which the processor is always in supervisor mode. When using this application, the local stack and the supervisor stack are the same. The processor gets the initial stack pointer for this stack from register r1.

Finally, three levels of code are used: initialization code, kernel code, and applications code. The initialization code is part of the IMI. (Appendix D gives an initialization code example.) The starting IP for the initialization code is also provided in the IMI.

PROCESSOR INITIALIZATION

This section describes how to initialize the 80960SA/SB processor. It defines the mechanism that the processor uses to establish its initial state and begin instruction execution. It also describes some general guidelines for writing code to complete the initialization of the processor for specific applications.

NOTE

The i960 architecture does not define an initial memory image or an initialization procedure. The following initialization requirements are specific to the 80960SA/SB processor.

Initial Memory Image

The IMI performs three functions for the processor: (1) it provides check-sum words that the processor uses in its self-test routine at start-up, (2) it provides pointers to the system data structures, and (3) it provides scratch space that the processor uses to perform certain internal functions. Figure 3-2 shows the structure of the IMI.

The IMI is made up of four parts: the check-sum word, the system adress table (SAT), the processor control block (PRCB), and the initialization code. In an embedded application, all of the parts of this image will generally be held in ROM. The possible exception to this is the PRCB, which contains scratch space which should be located in RAM. For this reason, the PRCB should be copied from ROM to RAM after system initialization. (The reinitialize processor IAC, described in Chapter 11, is used to give the processor the PRCB pointer for the relocated PRCB.)

Check-Sum Words

The check-sum words must be in memory locations 00000000_{16} to $0000001F_{16}$. The first of these words is a pointer to the base of the SAT. The second word is a pointer to the base of the PRCB. The fourth word is the instruction pointer to the first instruction of the initialization code.

The remaining words (word 3 and words 5 through 8) are check words. The values of these words must be chosen such that when the check sum is computed (as shown in the initialization algorithm in Figure 3-3), the result is equal to 0.

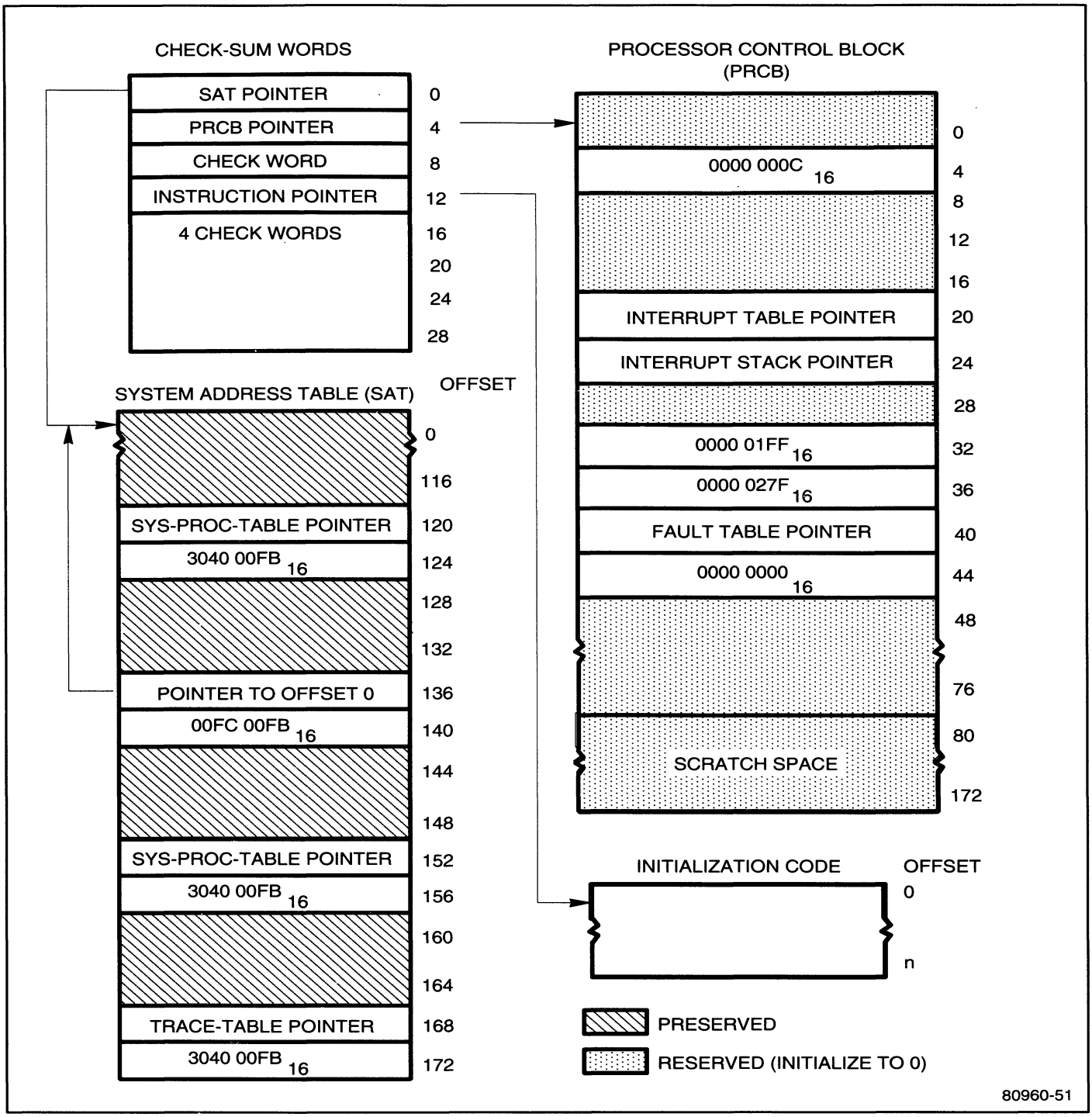

Figure 3-2: Initial Memory Image

```
assert BLAST/FAIL pin;
perform self test;
if self test fails
    then halt;
    else
        deassert BLAST/FAIL pin;
        enter predefined state;
                x ← memory(0);  read 8 words beginning
                                at address 0
            AC.cc ← 000₂;
            temp ← FFFFFFFF₁₆ add_with_carry x(0);
            temp ← temp add_with_carry x(1);
            temp ← temp add_with_carry x(2);
            temp ← temp add_with_carry x(3);
            temp ← temp add_with_carry x(4);
            temp ← temp add_with_carry x(5);
            temp ← temp add_with_carry x(6);
            temp ← temp add_with_carry x(7);
            if temp ≠ 0
                then
                    assert BLAST/FAIL pin;
                    halt;
                else
                    prcb_address ← x(1);
                    IP ← x(3)
                    fetch prcb;
                    priority ← 31;
                    process_controls.state ← interrupted;
                    FP ← prcb.interrupt_stack_pointer;
                    clear any latched external interrupt
                      signals;
                    begin execution;
            endif;
endif;
```

Figure 3-3: Algorithm for First Stage of Initialization Procedure

System Address Table

The SAT is used by the processor to locate the system data structures which may not be cached on-chip. It is 176 bytes in size and can be located anywhere in the address space aligned on a 4K byte boundary. It has three required entries, and one optional one.

The first entry begins at byte 120, and contains a pointer to the system procedure table. The following word, beginning at byte 124, contains $304000FB_{16}$, a descriptor which tells the processor that this is a pointer to a procedure table. This entry is required, and is used for compatibility with other members of the 80960 architecture.

The second entry, beginning at byte 136 of the SAT is a pointer to the base (first byte) of the SAT. This pointer is usually the same as the pointer given in the first word of the check-sum words. If the pointer is different (the SAT has been relocated to RAM, for example), the processor essentially switches to the value found in this entry (i.e., this pointer is used upon reset, and is used for all subsequent accesses to the SAT). The following word, beginning at byte 140, contains $00FC00FB_{16}$, a descriptor which tells the processor that this is a pointer to the SAT. This is also a required entry.

The third entry in the SAT begins at byte 152 followed by the descriptor beginning at byte 156. It is identical to the first SAT entry at byte 120 and its descriptor at byte 124. The third entry is the pointer used by the processor to find the system procedure table, and is required.

The fourth entry in the SAT begins at byte 168 and is optional. It is a pointer used by the processor to locate the trace table. The following word, beginning at byte 172 contains the value $304000FB_{16}$, a descriptor which tells the processor that this entry is a procedure table entry. For more information on the trace table, see the section "Trace Fault Handling", in Chapter 6, "Faults".

All of the remaining words in the SAT are preserved and can be used by software.

Processor Control Block

The PRCB is 176 bytes long and can also be located anywhere in the address space aligned on a SALIGN boundary. For more information on SALIGN, see Appendix F, "SALIGN Parameter". It has seven required entries, and the rest is reserved.

The word beginning at byte 4 must contain $0000000C_{16}$.

The *interrupt table pointer* points to the first byte of the interrupt table. The *interrupt stack pointer* points to the top (first available byte) of the interrupt stack.

The word beginning at byte 32 must contain $000001FF_{16}$ and the word at byte 36 must contain $0000027F_{16}$.

The *fault table pointer* points to the first byte of the fault table.

The word beginning at byte 44 must contain all zeros for normal operation. To turn off prefetch, the word may contain $FFFFFF10_{16}$. This value may slow down operation of the processor. Prefetch is the processor looking ahead and prefetching values which may be needed in the instruction cache.

The processor uses the *scratch space* in the IMI for internal functions. This field should be set to all zeros at initialization or reinitialization of the processor and not accessed by software thereafter.

The remaining fields in the PRCB (bytes 8 through 19, bytes 28 through 31, and bytes 48 through 79) are reserved. They should be set to all zeros at initialization or restart and not accessed by software thereafter.

Initialization Code

The initial instruction list that the processor begins executing following its self test can be located anywhere in the address space.

Changing the Initial Memory Image

At initialization or on a reinitialize processor IAC, the processor reads the pointers from the IMI in memory and caches them.

In general, to change any of the IMI fields that have been cached on the processor chip, the kernel must first modify the IMI in memory, then reinitialize the processor using the reinitialize processor IAC. The processor then rereads the IMI and reloads the cached fields in its internal cache.

Building a Memory Image

The IMI shown in Figure 3-2 contains the minimum data structures required for the processor to initialize itself and begin executing code. To build a useful system, however, additional data structures are required, such as an interrupt table, a fault table, a system procedure table, a set of kernel procedures, a set of stacks, and a heap. Some of these data structures can be located in ROM along with the IMI; however, others must be in RAM because they must be writable.

Table 3-1 lists the various system data structures and shows which can be in ROM and which must be in RAM. The following paragraphs give the system limitations if a data structure is included in ROM.

All of the PRCB except the scratch space area must be in ROM. The scratch space must be in RAM.

The interrupt table must generally be in RAM for the processor to operate properly, because it contains the interrupt pending fields, which the processor must be able to write to. If the pending interrupts function of the interrupt-handling facilities are not going to be used, the interrupt table may be placed in ROM.

The fault table can be in ROM, providing it will never be necessary to relocate the fault handler routines.

The kernel procedures can be in either ROM or RAM or both, depending on the design of the kernel.

Table 3-1: ROM and RAM Resident Data Structures

Data Structure	May Be In ROM	May Be In ROM with Limitations	Must Be In RAM
IMI	X		
PRCB		X	
SAT	X		
Interrupt Table		X	
Fault Table	X		
Kernel Procedures	X		
Stacks and heap			X

Typical Initialization Scenario

Initialization of the 80960SA/SB processor typically is handled in two stages. In the first stage of initialization the processor performs a self test and reads pointers from the IMI. During the second stage, the processor executes initialization code designed to build the remainder of the memory image so that execution of application code can begin.

First Stage of Initialization

Figure 3-4 shows the steps that the system hardware and the processor go through in the first stage of initialization. The algorithm in Figure 3-(4) gives the details of this procedure.

1. Hardware asserts the $\overline{\text{RESET}}$ pin on the processor.
2. The processor asserts the $\overline{\text{BLAST}}/\overline{\text{FAIL}}$ pin and performs a self test. If the processor passes the self test, it deasserts the $\overline{\text{BLAST}}/\overline{\text{FAIL}}$ pin.
3. The processor reads the 8 check-sum words and checks that the sum is 0.
4. Using the contents of the check-sum words, the processor determines the location of the SAT, the PRCB, and the first instruction to be executed.
5. The processor sets its process priority to 31 (highest possible) and its state to interrupted.
6. The processor clears any latched external interrupts. This means that the processor will not service any interrupts prior to beginning instruction execution.
7. The processor begins executing the initialization instruction list.

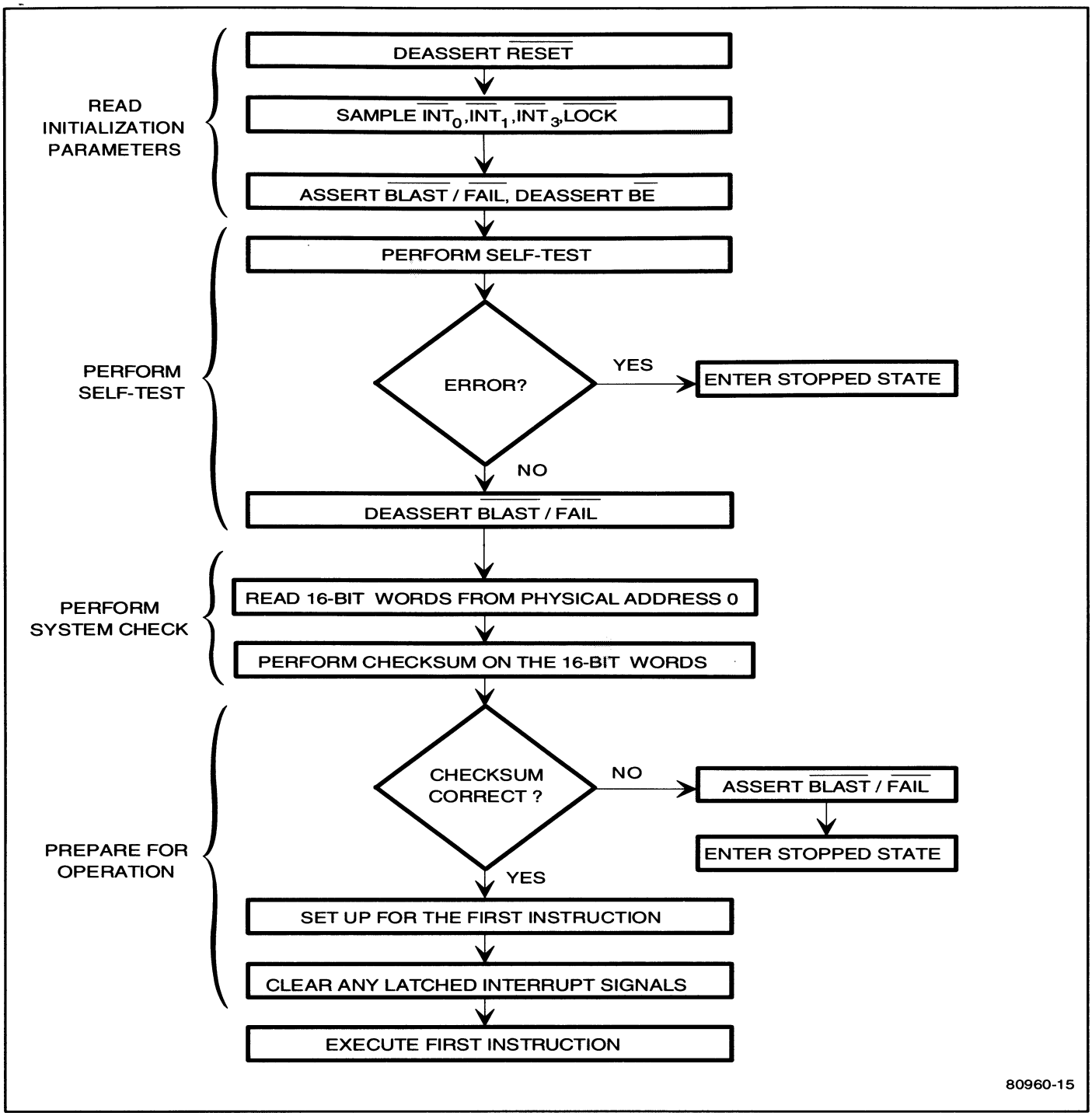

Figure 3-4: Initialization Flow Chart

After self test, the processor establishes its initial state. For the 80960SA/SB processor this state is interrupted. Also at initialization, the trace controls are set to zero; the process controls are set to zero (except for the execution mode, which is set to supervisor, and the priority, which is set to 31); and the breakpoint registers are disabled.

80960SA/SB Self-Test

The 80960SA/SB processor implements an automatic self-test following the receipt of any system RESET command. Before any bus activity occurs, the processor executes enough self-checking tasks to guarantee the proper operation of 50 percent of its internal logic, including the following major component blocks:

- Instruction Fetch Unit (IFU)
- Integer Execution Unit (IEU)
- Micro-Instruction Sequencer (MIS)

This section describes the action of the on-chip self-test logic as it tests the above logic.

Scope of Self-Test

The self-test directly tests or indirectly tests the elements of the 80960SA/SB components below:

- Directly tests:
  - Micro-Instruction ROM and control logic
  - Instruction Fetch Unit Cache
  - Instruction Execution Unit RAM Array and Literals
- Indirectly tests:
  - Access to the Micro-Instruction Bus
  - Access to the Data Bus
  - Control logic and data paths of the Micro-Instruction Sequencer, the Instruction Execution Unit, and the Instruction Fetch Unit.

The test does not check the Instruction Decoder Logic, the Floating-point unit for the 80960SB, nor the IFU Instruction Pointer and its fetch and control logic.

Test Algorithm and Operation

Before the test begins, the 80960SA/SB processor tri-states its local bus drivers. The processor then uses a checksum method to compare expected data and actual data results. If the test encounters a failure at any point, the processor asserts the BLAST/FAIL signal and deasserts both BE signals to notify the system.

The test performs its test algorithm in the following sequence:

1. Write data to the Integer Execution Unit (IEU) and the Instruction Fetch Unit (IFU). Then, verify the integrity of data with read instructions. This task follows the sequence below:
   - Write the IEU.
   - Write the IFU.
   - Read the data written to the IEU RAM array and literals.
   - Read the data written to the IFU cache.
2. Repeat the process above.
3. Read the Micro-Instruction Sequencer ROM and compare the actual results with the expected results.

Second Stage of Initialization

The processor activity during the second stage of initialization, which occurs once the processor begins instruction execution, is up to software. In general, this stage of initialization is used to copy or create additional data structures in memory, such as the interrupt table, the system-procedure table, and the fault table (if not in the initial memory image), and the kernel procedures.

Once these jobs are completed, the processor can then begin executing application code.

Appendix D gives an example of the 80960SA/SB code that might be used to carry out this second stage of initialization.

A common initialization technique is to create a new PRCB and interrupt table in RAM along with the other system data structures that are placed in memory in the second stage of initialization. The processor is then reinitialized to point to the new PRCB and interrupt table. (The code in Appendix D uses this technique.)

The processor is reinitialized using the reinitialize processor IAC. This reinitialize processor IAC message includes new pointers to the SAT and PRCB. The processor reads the new PRCB, then begins instruction execution according to the control information contained in the PRCB.