PA-RISC 2.0 Firmware Architecture Reference Specification

Version 0.36

Printed in U.S.A. March 7, 1999

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

1. PDC Entry Points

The object of PDC is to provide a uniform, architected context in which to perform processor-dependent operations. One of the two PDC mechanisms is a set of entry points that are triggered when special events are recognized by the processor hardware. These entry points relate to initialization and error recovery.

This chapter describes the PDC entry points. The PDC procedures are described in Chapter 4, PDC Procedures.

1.1 Definition of Key Terms

Boot is the process by which PDC initializes a system and brings code from the boot device into memory. Boot is performed by the monarch processor in response to specific trigger events. Boot is initiated when new software must be loaded into the system. The objectives of boot are to force the system into a known state, to configure system resources for software, and to load code from the boot device.

A **hard boot** is a boot in which memory is destructively initialized. A hard boot is triggered by a broadcast CMD_RESET or by a power-on where memory is invalid. The processor state, all pending I/O operations, and the contents of the Initial Memory Module are all lost in a hard boot. The contents of other memory modules may also be destroyed depending on the value of the *fast_size* flag.

A **soft boot** is a boot in which memory is nondestructively initialized. A soft boot is triggered by a failed TOC. The processor state, all pending I/O operations, and the contents of memory between Page Zero and the contents of MEM_FREE + 32 KBytes are lost during a soft boot. After a failed TOC, the processor state can be rebuilt from PIM.

A **Transfer of Control** saves the state of the processor in Processor Internal Memory (PIM) and begins execution of recovery software at a nonzero location specified by a special location in Page Zero called MEM_TOC. The TOC code is protected by a checksum. The processor state is saved in PIM and pending I/O operations are not disturbed.

A **Machine Check** is either a High Priority Machine Check (HPMC) or a Low Priority Machine Check (LPMC). Software is executed to locate the source of the failure and take appropriate recovery action. In an LPMC, no system state is lost, since the error was completely recovered and is reported to software only for logging purposes. In an HPMC, the amount of state lost is HVERSION dependent. As much processor state as possible is saved in PIM, but recovery is not always possible.

The **console device**, or **console**, is the I/O device used to communicate with the operator during boot. Messages written to the console device inform the operator of progress through the boot sequence. Input accepted from the console device allows the operator to select among boot alternatives. The module to which the console device is connected is called the **console module**.

A **simplex console** is a console where different I/O devices are used for the console input and output functions. The **display device** performs the output function and the **keyboard device** performs the input function.

The **boot device** is the I/O device which contains the software which is loaded into memory and executed. PDC locates the boot device and verifies that the software it contains is properly formatted. The module to which the boot device is connected is called the **boot module**.

The **monarch processor** is the processor which has been selected to perform boot, and execute IPL.

Monarch selection is the process by which all the processors compete for selection as the monarch processor.

Rendezvous is the state in which all non-monarch processors wait while the monarch processor performs the boot process. A processor leaves rendezvous when it receives a rendezvous interrupt (interrupt to EIR{0}).

The **Initial Memory Module** or **IMM** is the memory module on the central bus which is selected by PDCE_RESET to contain Page Zero, the Console and Boot Device IODC, and the IPL code. In the absence of failures, the Initial Memory Module is the memory module on the central bus with the most installed memory, the smallest HPA in the case of a tie in installed memory, and at least 256 Kbytes of memory.

An **Initial Memory Module Candidate** or **IMMC** is a memory module on the central bus which has been selected by PDCE_RESET as a possible candidate for the Initial Memory Module. The Initial Memory Module Candidate must be successfully tested and initialized before it can be chosen as the Initial Memory Module. In addition, during powerfail recovery, PDCE_RESET must verify that the IMMC is the same memory module that was identified as the Initial Memory Module during the most recent boot.

Initial Program Load (IPL) is the generic term for the first code loaded into memory from the boot device. This code is loaded and executed by the PDC boot code. IPL must be capable of loading and launching ISL.

Initial System Load (ISL) is the standard code module which is used during the startup of every Precision operating system or diagnostic utility. It provides a standard operator interface for the selection and initiation of

system software. The first code brought into the system from the boot device may be ISL. In such a case, ISL is the IPL. Alternatively, IPL may be distinct from ISL. In that case, IPL performs preliminary functions before launching ISL.

1.2 PDC Entry Points

The event-triggered entry points into PDC are defined as follows:

- **Transfer of Control (PDCE_TOC)**. The trigger is the receipt of a directed CMD_RESET. Processors can optionally provide a "TOC button" which triggers PDCE_TOC when pressed.
- **Machine Check Preparation (PDCE_CHECK)**. The trigger is the occurrence of a failure which may result in a High-Priority Machine Check or Low-Priority Machine Check.
- **Processor Reset (PDCE_RESET)**. The trigger is either power-on or the receipt of broadcast CMD_RESET. Additionally a soft boot will be initiated in response to a failed TOC. Processors can optionally provide a "reset button" which triggers PDCE_RESET when pressed. (More specifically, the reset button causes the system to respond as though both BUS_POW_VALID and BUS_SEC_VALID changed from being deasserted to being asserted.)

The relationship between these three PDC entry points and the operating system software to which they branch is shown in the following figure. Note that the figure presents an overview of the relationship, and does not show all possible conditions which may affect the flow of control.

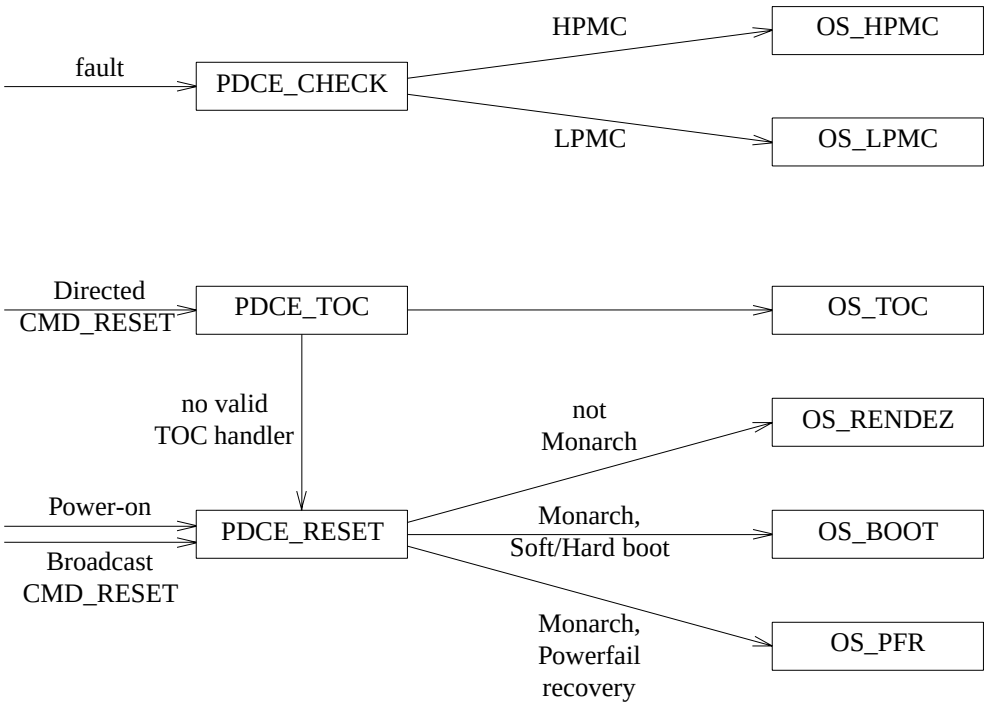


Figure 1-1. PDC Entry Points and the Operating System

There is one PDC entry point which is not event triggered, but is called by software. This is PDCE_PROC, which is a single entry point for all PDC procedures. The PDC procedures are described in Chapter 4, PDC Procedures.

The exact operation of all the PDC entry points is not specified by the I/O Architecture. Instead, PDC must satisfy specific requirements, especially at the interface to IPL or the operating system.

Algorithms illustrating the suggested or typical flow of PDC entry points are presented in programming notes adjacent to the definition of the architectural requirements. It is not required that PDC follow the exact sequence of steps presented in the examples in the programming notes.

This chapter describes the PDC entry points for all categories of processor. In general, the differences in behavior between the categories of processor are described, except where a particular feature, for example, the BOOT_ID, is only found in some categories of processor.

ENGINEERING NOTE

It may increase the supportability of a system if the instructions and data for PDCE_CHECK and PDCE_TOC are stored internal to the processor module rather than in a memory module.

PDCE_TOC

Synopsis: Transfer of Control (TOC) is used to initiate recovery when the system software is stuck in an error state.

Triggers: PDCE_TOC is triggered by the receipt of a directed CMD_RESET.

Responses: TRANSFER OF CONTROL (ENTER OS_TOC)
This option will be chosen if all of the checks that protect the OS_TOC code succeed. The OS_TOC code is located in memory at the address specified by the MEM_TOC word in Page Zero.

SOFT BOOT (ENTER OS_BOOT)
Soft boot is performed after a directed CMD_RESET if the OS_TOC code has been corrupted or is not present. The processor which received the directed CMD_RESET informs all other processors, and then all processors compete to select a monarch, and the monarch processor completes the soft boot sequence, which results in the IPL code being launched on the monarch processor.

RENDEZVOUS (ENTER OS_RENDEZ)
Rendezvous is performed by non-monarch processors whenever a soft boot, by the monarch processor, becomes necessary after a directed CMD_RESET due to the OS_TOC code being corrupted or not being present.

HALT
The processor halts if the processor selftest detects an unrecoverable error. When halted, the processor enters an idle loop waiting for a directed or broadcast CMD_RESET, or power-on.

Description: Transfer of Control is nonmaskable. Upon receiving a directed CMD_RESET, the processor module triggers PDCE_TOC independent of the state of the PSW M-bit.

If the processor detects an unrecoverable error, for example during a selftest, it must halt and wait for a directed or broadcast CMD_RESET, or power-on.

If any of the checks on MEM_TOC or the OS_TOC code fail, the processor must force all processors to perform a soft boot.

- A category B processor must use an SVERSION-dependent mechanism to force all processors on the central bus to select a monarch and soft boot.
- A category A processor must invoke soft boot in an HVERSION-dependent manner.

When a broadcast CMD_RESET command is sent to a processor while it is in the middle of PDCE_TOC, the processor must either boot (i.e., enter OS_BOOT) or halt.

When a directed CMD_RESET (TOC) command is sent to a processor while it is in the middle of PDCE_TOC, the processor must perform a TOC by entering OS_TOC or must halt.

When PDCE_TOC is interrupted by an HPMC, the only allowed exit options from PDCE_TOC are:

- Entry to OS_TOC; PSW M-bit is set to 1 at the interface to OS_TOC.

SUPPORT NOTE

In order to ensure that software does not lose machine check information, OS_TOC should check for HPMCs.

- Halt.

When PDCE_TOC is interrupted by a Group 2 interruption, the only allowed exit options from PDCE_TOC are entry to OS_TOC or to halt.

When PDCE_TOC is interrupted by a Group 3 or 4 interruption, the only allowed exit options from PDCE_TOC are entry to OS_TOC or to halt.

ENGINEERING NOTE

A recovery counter trap, or Group 3 and 4 interruptions are not expected to occur during PDCE_TOC. Consequently, these cases have been included here primarily for purposes of completeness. The intent is also to clarify that entry to the recovery counter trap handler or to Group 3 and 4 fault/trap handlers from PDCE_TOC are not allowed options.

Soft boot, and the interfaces to OS_BOOT and OS_RENDEZ are detailed in the PDCE_RESET description in Section 2.2, PDC Entry Points.

ENGINEERING NOTE

The architecture currently does not require TOC to be recoverable. However, processor designers are strongly encouraged to preserve the processor state as much as possible. This would make it easier to implement recoverable TOC in the future.

PDCE_TOC must not alter module configuration status maintained as tertiary state.

Powerfail recovery is not required if power fails during PDCE_TOC.

PROGRAMMING NOTE

Example: An example algorithm showing the suggested flow of PDCE_TOC is described below.

1. At the time of TOC, the PSW is copied into IPSW. The initiated instructions are backed out or completed, and the recovery counter is set to a consistent state. If PSW Q-bit equals 1 at the time of TOC, the IA Queues are copied into the IIA Queues, the IPRs receive HVERSION-dependent information, and the front entry of the IIA Queues points to the correct instruction which resumes execution after TOC interruption.
2. Save the processor general, control, space, and interruption registers in PIM. The CPU State word in PIM qualifies the validity of the Processor State-Save region. See the PDC_PIM description in Chapter 4, PDC Procedures, for a description of PIM.
3. Write the appropriate code (CB0x) to the chassis display to indicate that a TOC has occurred.
4. Flush the entire data cache to memory.
5. Perform a nondestructive processor test. If the processor test fails, write the appropriate code to the chassis display, and halt.
6. If the MEM_TOC word is nonzero, the OS_TOC checksum is correct, and the word at the address pointed to by MEM_TOC is nonzero, then branch to the OS_TOC code, which is the code pointed to by the MEM_TOC word. If any of the MEM_TOC checks fail, then perform the following steps:
    a. Write to the LBRS IO_FLEX register to initialize the flex address of modules on the central bus to the central bus physical address space and disable bus requestorship.

b. Perform a soft boot.

A category B processor must force all processors on the central bus to trigger PDCE_RESET, select a monarch and perform a soft boot in an SVERSION-dependent mechanism.

A category A processor may branch directly to the PDC soft boot code.

The following figure shows the PDCE_TOC algorithm for a category A and category B processors.

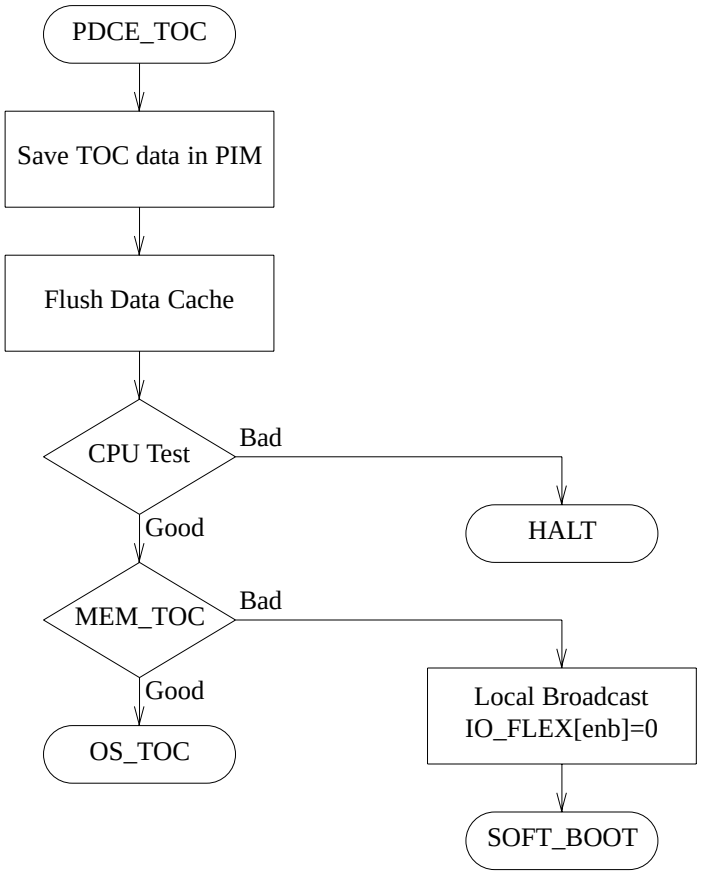


Figure 1-2. PDC Transfer of Control (PDCE_TOC) Algorithm

Synopsis: The OS_TOC interface defines the boundary between PDCE_TOC and the operating system Transfer of Control code.

Privilege: Level 0.

PSW: Q-bit is 1. E-bit is set to default endianness. W-bit is set to default width.

All other bits are 0.

CRs: CR14 (Interrupt Vector Address) contains the address of an Interruption Vector Table containing code to handle an HPMC, or is unchanged.

ENGINEERING NOTE

If implementations maintain the value in CR14 that was present when the TOC was triggered, the reliability may be increased if PDCE_TOC verifies the HPMC handler checksum.

CR22 (IPSW) contains the PSW in effect at the time of the interruption.

All other control registers are HVERSION dependent.

GRs: GR26 contains the address of PDCE_PROC.

On category B (multiprocessor) processor modules, GR25 contains the address of the HPA of the current processor. On category A (uniprocessor) processor modules, GR25 is HVERSION dependent.

All other general registers are HVERSION dependent.

SRs: All space registers are unchanged.

PIM: The Processor State-Save region and the CPU State word in PIM reflect the state of the processor at the time the TOC trigger was received, except that the IPSW contains the PSW and IIA Queues contain IA Queues.

Cache: The cache is clean or invalid, and contains only physical addresses.

Cache coherence is enabled.

TLB: The TLB is initialized and invalid. The hardware miss handler is unchanged.

Memory: The MEM_TOC word is nonzero, the checksum of MEM_TOC_LEN bytes of code located in memory at the location pointed to by MEM_TOC is zero, and the first word of code at the location pointed to by MEM_TOC is nonzero.

Other memory is unchanged.

PDCE_CHECK

Synopsis: Machine checks are caused by internal processor hardware failures or by certain classes of bus errors. The purpose of PDCE_CHECK is to capture information pertinent to a machine check, so that logging, analysis, and recovery can be performed.

Triggers: PDCE_CHECK is triggered by internal processor hardware failures and by certain classes of bus errors.

Responses: HPMC (ENTER OS_HPMC)
If an HPMC condition occurred, the processor enters the HPMC handler (OS_HPMC).

LPMC (ENTER OS_LPMC)
If an LPMC condition occurred the processor enters the LPMC handler (OS_LPMC).

HALT
The processor halts if the severity of the hardware failure prevents it from satisfying the architected interface to OS_HPMC or OS_LPMC. The processor halts if the checksums of OS_HPMC are invalid.

OS_TOC
The TOC handler (OS_TOC) may be entered if PDCE_CHECK is interrupted by a directed CMD_RESET (TOC).

OS_BOOT
OS_BOOT may be entered if PDCE_CHECK is interrupted by a broadcast CMD_RESET.

Description: **Interruption Vector Table**

An **Interruption Vector Table** or **IVT** is a table of interruption handling routines pointed to by the Interrupt Vector Address (IVA) (in CR14). An Interruption Vector Table must have three special characteristics before it can be used in handling HPMCs. First, the location at IVA + 32 must be established to aid the processor in preparation for an HPMC. That location must contain either the instruction returned by the PDC_INSTR procedure or a null instruction (if PDC_INSTR returned a status of -1). Second, the instruction at IVA + 36 is the start of OS_HPMC. Third, the words at IVA + 56 and IVA + 60 must contain the ADDRESS and LENGTH (in bytes) of the continuation of OS_HPMC.

The start of the LPMC handler is located in the Interruption Vector Table at IVA + 160.

An Interruption Vector Table is a critical part of the operating system. PDC and ISL must provide their own tables to deal with interruptions during boot. Thus the handler may reside either in memory or in ROM internal to the processor.

When a broadcast CMD_RESET command is sent to a processor while it is in the middle of PDCE_CHECK, the only exit options for the processor are to boot (i.e., enter OS_BOOT) or halt.

When a Directed Reset (TOC) command is sent to a processor while it is in the middle of PDCE_CHECK, the processor must choose one of the following exit options:

- Halt.
- Perform a TOC (enter OS_TOC)
- Exit PDCE_CHECK and either enter OS_HPMC or OS_LPMC.

It must be guaranteed, that the PIM update during PDCE_CHECK either completed successfully, or that the PIM contents are marked invalid.

Entry to OS_HPMC is the recommended option if it is determined in PDCE_CHECK before the occurrence of the TOC, that the nature of the failure requires an HPMC to be taken. If that determination cannot be made, entry to OS_TOC is the recommended option.

When PDCE_CHECK is interrupted by an HPMC, the only allowed exit options from PDCE_CHECK are entry to OS_HPMC or to halt.

When PDCE_CHECK is interrupted by a Group 2 interruption, the only allowed exit options from PDCE_CHECK are entry to OS_HPMC/OS_LPMC, or to halt.

When PDCE_CHECK is interrupted by a Group 3 or 4 interruption, the only allowed exit options from PDCE_CHECK are entry to OS_HPMC/OS_LPMC or to halt.

ENGINEERING NOTE

A recovery counter trap, or Group 3 and 4 interruptions are not expected to occur during PDCE_CHECK. Consequently, these cases have been included here primarily for purposes of completeness. The intent is also to clarify that entry to the recovery counter trap handler or to Group 3 and 4 fault/trap handlers from PDCE_CHECK are not allowed options.

Accessing PDCE_CHECK for an HPMC

There are two ways by which implementations may access PDCE_CHECK for HPMC:

- The occurrence of the failure condition may trigger PDCE_CHECK directly. When PDCE_CHECK is done, if the PSW M-bit was 0 at the time of the failure, the processor will vector to IVA + 32. On processors that trigger PDCE_CHECK directly, IVA + 32 will always contain a null instruction, so the next instruction after that will be fetched from IVA + 36.
- The occurrence of the failure condition may vector to IVA + 32 directly. The instruction at IVA + 32 will trigger PDCE_CHECK. When PDCE_CHECK is done, if the PSW M-bit was 0 at the time of the failure the processor will vector to IVA + 36.

ENGINEERING NOTE

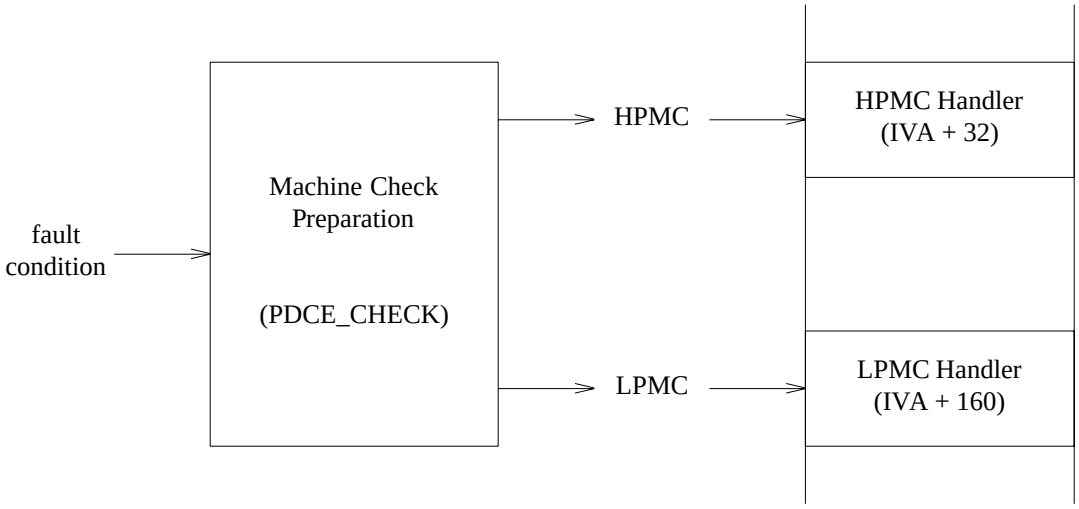
PDCE_CHECK may be implemented by code in ROM, in hardware, by a "Service Processor", or by any combination of the three.

PDCE_CHECK for an LPMC might be implemented as a simple latch and interrupt; code is not necessary.

If PDCE_CHECK entry point is implemented in code, that code may reside in a memory module. The architecture also allows PIM to be stored in Page Zero. In implementations that use memory for PDCE_CHECK code or PIM, operations to memory are required during HPMC processing, which could limit the chances for successful recovery (if a faulty bus or memory module caused the HPMC). Therefore, implementations which place a premium on fault recovery may consider executing PDCE_CHECK from ROM internal to the processor and insuring that PIM is truly internal to the processor. This consideration makes it preferable for implementations to enter PDCE_CHECK directly, rather than through an instruction at IVA + 32.

Machine Check Handlers

A High Priority Machine Check (HPMC) is a Group 1 Interruption. The HPMC occurs when the processor enters the HPMC handler (OS_HPMC). Thus, the HPMC is distinct from the failure which causes the processor to invoke the HPMC. The figure shows the difference between the failure condition and the HPMC.



Machine Check Preparation

Every processor is required to implement a means to prepare for Machine Checks. This preparation process is called PDCE_CHECK.

The following points characterize the behavior of PDCE_CHECK.

1. PDCE_CHECK is entered:
   - Either upon the occurrence of a failure, or
   - When the PSW M-bit changes from 1 to 0 and there is a pending failure condition.
2. The error information logged in PIM is information captured at the time of occurrence of the failure.
3. The processor state information logged in PIM is information captured at the time the interruption is taken.
4. If it is determined in PDCE_CHECK that the PSW M-bit was 1 at the time of the failure, then an allowed response is to return control to the process that was executing at the time of occurrence of the failure. PDCE_CHECK will be subsequently re-entered when the PSW M-bit becomes 0.

ENGINEERING NOTE

If it is determined in PDCE_CHECK that the PSW M-bit was 1 at the time of the failure, then it is strongly recommended that the processor respond by halting. Continuing with code execution subsequent to the detection of bus errors can have catastrophic consequences on data integrity.

Most of the details of PDCE_CHECK are HVERSION dependent, so long as the required interface to the machine check handlers is satisfied. If PDCE_CHECK is unable to satisfy the required interface (for example, nested failures in a processor that cannot tolerate them), then it can halt the processor (that is, enter an idle loop).

If the processor detects an unrecoverable error, for example during a selftest, it must halt, and wait for a directed or broadcast CMD_RESET, or power-on.

HPMC

For HPMC conditions, if either of the checksums of OS_HPMC are incorrect, the processor must halt, and wait for a directed or broadcast CMD_RESET, power-on.

Entry to OS_HPMC is the recommended option if it is determined in PDCE_CHECK before the occurrence of the directed CMD_RESET, that the nature of the failure requires an HPMC to be taken. If that determination cannot be made, entry to OS_TOC is the recommended option.

If the processor halted, the value of the PSW M-bit is HVERSION dependent.

During HPMC handling, the cache and cache coherence must be enabled. The processor must assert PATH_ERROR in any transaction which references a cache line containing errors or a corrupt cache. The processor may optionally perform non-coherent operations to/from the data and instruction caches until the failure has been isolated to a level where the processor can perform coherent operations, without the risk of multiple HPMCs.

SUPPORT NOTE

The availability of a multiprocessor system may be increased if OS_HPMC performs non-coherent operations to avoid the risk of multiple HPMCs.

Powerfail recovery is not required if power fails during PDCE_CHECK.

There are certain error conditions which a processor is required to detect and respond to by invoking an HPMC. Each processor will also define a set of HVERSION-dependent hardware failures to which it will respond by invoking an HPMC. The processor is free to manage those internal failures in an intelligent way before invoking the HPMC. HPMCs are characterized by the property that intervention by OS_HPMC is required before processing can continue.

A HPMC is masked when the PSW M-bit is set to 1. The processor may do its HVERSION-dependent preparation whenever it detects a failure condition. But it is only allowed to take an HPMC (that is, enter OS_HPMC) when the M-bit is 0. If the M-bit is 1 at the time a failure is recognized, the HPMC is kept pending and is then taken as soon as the M-bit is cleared to 0.

LPMC

A Low Priority Machine Check (LPMC) is a Group 2 Interruption. The LPMC occurs when the processor enters OS_LPMC. Thus, the LPMC is distinct from the failure condition which causes the processor to invoke the LPMC.

There are certain error conditions which a processor is required to detect and respond to by invoking an LPMC. Each processor will also define a set of HVERSION-dependent recoverable hardware failures to which it will respond by invoking an LPMC. LPMCs are characterized by the property that the failure has been completely recovered by the time the LPMC occurs.

A LPMC is masked when the PSW I-bit is cleared to 0. The processor may do its HVERSION-dependent preparation whenever it detects a failure condition. But it is only allowed to take an LPMC (that is, enter OS_LPMC) when the I-bit is 1. If the I-bit is 0 at the time a failure is recognized, the LPMC is kept pending and is then taken as soon as the I-bit is set to 1.

In cases where the required recovery action is handled completely by PDCE_CHECK, it is still permissible to invoke OS_HPMC.

ENGINEERING NOTE

Although the architecture allows errors completely corrected by PDCE_CHECK to be reported using an HPMC, implementations are encouraged to use an LPMC to report these errors since an HPMC allows for the possibility of having to reboot (because the PSW Q-bit might be zero).

Example:

PROGRAMMING NOTE

An example algorithm showing the suggested flow of PDCE_CHECK for an HPMC failure condition is described below.

1. Save the processor's general, control, space, and interruption registers in PIM along with any HVERSION-dependent processor state relevant to the HPMC. In addition, set the HPMC Error Parameters Region in PIM to indicate the nature of the failure to OS_HPMC.
2. Indicate that an HPMC has occurred by writing the appropriate code to the chassis display.
3. Correct the problem, if possible.
4. Test the processor. This test may change the contents of the general registers (and need not restore them). If the test detects a condition which prevents the proper fetching and execution of instructions, then write the appropriate code to the chassis display, log failure information in PIM, display tombstone on console display (if console display exists) and halt. Halting prevents any further failure analysis, but this is preferable to corrupting the rest of the system.
5. Compute the checksums of OS_HPMC, as described in the OS_HPMC Interface section. If the checksums are correct, vector to OS_HPMC, else write the appropriate code to the chassis display, and halt.

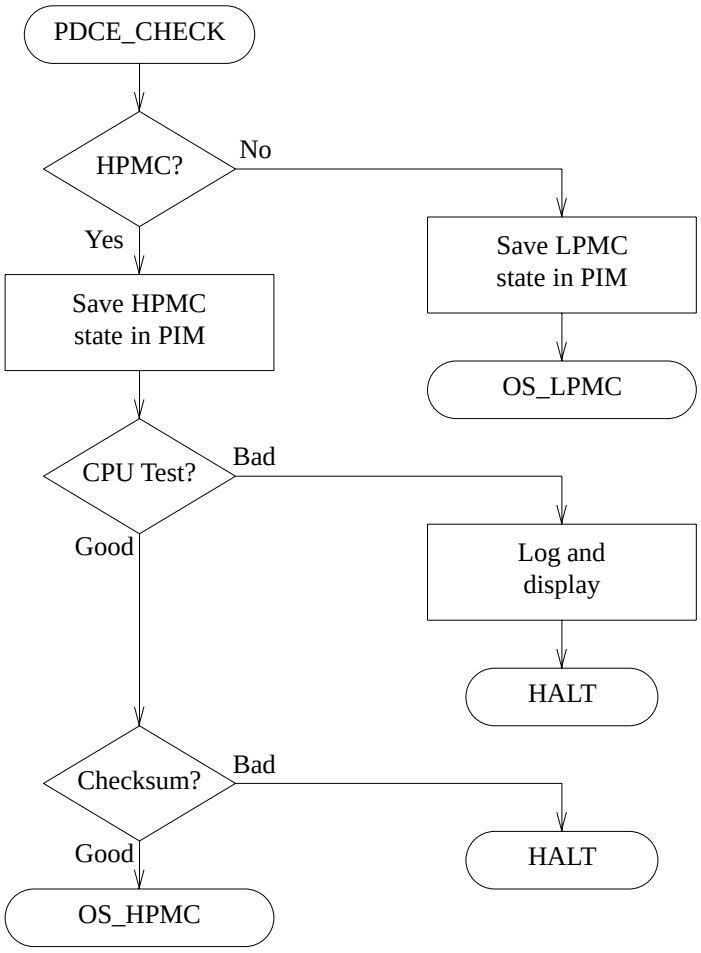


Figure 1-3. Machine Check Preparation (PDCE_CHECK) Algorithm

The following sequence describes the typical flow for the occurrence of a single HPMC failure:

1. Software is running normally, with the M-bit set to 0.
2. The processor detects an HPMC failure condition.
3. PDCE_CHECK is triggered, and saves the processor state in PIM.
4. The HPMC is presented to the HPMC handler (OS_HPMC), with the M-bit set to 1.
5. OS_HPMC calls PDC_PIM to obtain the processor state at the time of the HPMC, and then performs failure analysis and recovery.
6. OS_HPMC does an RFI. The software continues to run from the point at which it was interrupted. The RFI restored the M-bit to 0 (when the PSW was restored from the IPSW).

The following sequence describes the typical flow for the occurrence of two nested HPMC failures. Two HPMC check conditions are said to be nested if the second HPMC check condition occurs before OS_HPMC (invoked by the first HPMC check condition) calls PIM with ARG1=0.

1. PDCE_CHECK preparation for the first HPMC check condition may halt, and write the appropriate code to the chassis display. This scenario is possible if PDCE_CHECK determines that the two check conditions are related and the second machine check condition impairs the ability to successfully transfer control to OS_HPMC.
2. PDCE_CHECK preparation for the first machine check condition, upon noticing the second check condition, reports error critical in PIM. This scenario assumes that PDCE_CHECK has mechanisms to determine that control can be successfully transferred to OS_HPMC.
3. PDCE_CHECK preparation reports the first machine check condition in PIM. The second machine check condition is kept pending. OS_HPMC is launched and it logs the information of the first check condition and performs recovery. OS_HPMC restores the context of the machine corresponding to the time it took the HPMC interruption due to the first check condition. The machine gets interrupted by the second pending check condition. The PDCE_CHECK preparation of the second machine check condition saves the machine state in PIM (which corresponds to the context when the machine took the second HPMC interruption). The error information for the second machine check condition is also reported in PIM. OS_HPMC is launched again, recovers from the second HPMC condition, and restores the context of the machine corresponding to the time it received the second HPMC interruption. This scenario also assumes that the two HPMCs are unrelated and PDCE_CHECK has a mechanism to check this condition, and also that delaying the second HPMC does not cause any system damage.
4. A PDC_PIM call with ARG1=0 in OS_HPMC (due to the first HPMC) may notice the occurrence of a second HPMC and return archsize = 0.
5. Report and process the first machine check condition. When the M-bit goes to zero take the second HPMC interruption and report error critical. This is like step 2 except that the error critical report is delayed.

Synopsis: The OS_HPMC interface defines the boundary between PDCE_CHECK and the operating system HPMC handler (OS_HPMC).

Privilege: Level 0.

PSW: M-bit is 1. E-bit is set to default endianness. W-bit is set to default width.

All other bits are 0.

CRs: CR14 (Interrupt Vector Address) is unchanged.

CR22 (IPSW) contains the PSW in effect at the time of the interruption.

All other control registers are HVERSION dependent.

GRs: GR26 contains the address of PDCE_PROC.

On category B (multiprocessor) processor modules, GR25 contains the address of the HPA of the current processor. On category A (uniprocessor) processor modules, GR25 is HVERSION dependent.

All other general registers are HVERSION dependent.

SRs: All space registers are HVERSION dependent.

PIM: The processor state and the HPMC error parameters are stored in PIM.

Cache: Cache coherence is enabled.

TLB: The TLB is unchanged. The hardware TLB miss handler is unchanged.

Memory: The word at IVA + 32 is nonzero.

The arithmetic sum of the 8 words starting at IVA+32 plus the sum of the LENGTH/4 words starting at ADDRESS is 0. Here, LENGTH is the number stored at IVA + 60, and ADDRESS is the number stored at IVA + 56.

Other memory is unchanged.

Synopsis: The OS_LPMC interface defines the boundary between PDCE_CHECK and the operating system LPMC handler (OS_LPMC).

Privilege: Level 0.

PSW: E-bit is set to default endianness. W-bit is set to default width. All other bits are 0.

CRs: CR22 (IPSW) contains the PSW in effect at the time of the interruption.

All other control registers are unchanged.

GRs: All general registers are unchanged.

SRs: All space registers are unchanged.

PIM: The LPMC error parameters are stored in PIM.

Cache: Cache coherence is enabled.

TLB: The TLB is unchanged. The hardware TLB miss handler is unchanged.

Memory: Memory is unchanged.

PDCE_RESET

Synopsis: A processor is reset to start the configuration or reconfiguration of a PA-RISC system. The response to the reset depends on the state of the system at the time the reset trigger occurred, and the type of trigger.

Triggers: PDCE_RESET is triggered by a broadcast CMD_RESET or a hard or soft power-on. Additionally a soft boot will be initiated in response to a failed TOC.

Responses: POWERFAIL RECOVERY (ENTER OS_PFR)
This option will be chosen by the monarch processor after power-on if that processor finds that memory is valid and that recovery software has been established, and is not corrupt. The recovery software, called OS_PFR, is established by the operating system in response to a power failure interrupt.

HARD BOOT (ENTER OS_BOOT)
The system will hard boot after a broadcast CMD_RESET, or when powerfail recovery is not possible after power-on. The processor selected as the monarch completes the boot sequence, which results in IPL software being launched. During a hard boot, memory is tested destructively.

SOFT BOOT (ENTER OS_BOOT)
The system will soft boot after a failed TOC. The processor selected as the monarch completes the boot sequence, which results in IPL software being launched. During a soft boot, memory is tested nondestructively to preserve as much of the system state as possible for later dump and analysis.

RENDEZVOUS (ENTER OS_RENDEZ)
In a multiprocessor configuration, a processor will rendezvous if it is not selected as monarch when competing with the other processors during monarch selection.

HALT
When halted, the processor enters an idle loop, waiting for a directed CMD_RESET, broadcast CMD_RESET, or power-on. For example, PDCE_RESET will halt if the processor test gives an error, or if it tries to boot but encounters an error in accessing the boot device.

Description:

PROGRAMMING NOTE

PDCE_RESET may be triggered at a time when the Initial Memory Module is not initialized. Therefore, PDCE_RESET must ensure that the Initial Memory Module is initialized before using memory.

In a multiprocessor system, a monarch processor must be selected to perform powerfail recovery or boot.

The objective of monarch selection is to select a monarch processor which will perform a soft or hard boot. All other processors in a multiprocessor system, which have not halted, will wait for a rendezvous interrupt.

Category B rendezvousing processors executing PDC may issue bus operations to the memory address space, but they must monitor the BUS_POW_WARN signal.

If the BUS_POW_WARN signal is asserted, category B rendezvousing processors must flush their caches and ensure that they are not issuing bus operations to the memory address space when the BUS_POW_VALID signal is de-asserted.

ENGINEERING NOTE

This avoids the need for all processors to synchronize their powerfail preparation, and ensures that the rendezvousing processors are not performing operations to memory when BUS_POW_VALID is deasserted.

Interval timer interrupts on EIR{0} must not cause the processor to mistakenly start execution of the OS_RENDEZ code.

ENGINEERING NOTE

Regularly writing a large value to the interval timer register, CR16, is one method of ensuring that the interval timer does not cause an interrupt on EIR{0}.

Disabling all interrupts except EIR{0} while a processor is searching its local bus to determine whether it is the monarch processor or while a processor is waiting for a rendezvous interrupt avoids the need to handle spurious interrupts.

Category A processors need not implement monarch selection algorithm, since they are always the monarch processor.

The monarch selection algorithm is SVERSION dependent for category B processors, provided that a monarch is selected if at least one processor passes its selftest.

Every memory module on the central bus must have been reset before the IO_STATUS[he, se, estat] fields (or their HVERSION-dependent equivalents for processor-dependent memory modules) are read.

When a broadcast CMD_RESET command is sent to a processor while it is in the middle of PDCE_RESET, the only exit options it has are to either boot (i.e., enter OS_BOOT) or perform powerfail recovery (enter OS_PFR) or halt.

When a directed CMD_RESET (TOC) command is sent to a processor while it is in the middle of PDCE_RESET, the only exit options it has are to either boot (i.e., enter OS_BOOT) or perform powerfail recovery (enter OS_PFR) or halt.

When PDCE_RESET is interrupted by an HPMC, the only allowed exit options from PDCE_RESET are:

- Halt.
- Perform powerfail recovery (enter OS_PFR)
- Perform boot (enter OS_BOOT)

When PDCE_RESET is interrupted by a Group 2 Interruption, the only allowed exit options from PDCE_RESET are entry to OS_BOOT or OS_PFR or halt.

If the interruption is a powerfail interrupt, then control must not be transferred to the OS if it (the OS) cannot be guaranteed a full powerfail budget.

When PDCE_CHECK is interrupted by a Group 3 or 4 interruption, the only allowed exit options from PDCE_CHECK are entry to OS_BOOT or OS_PFR or halt.

ENGINEERING NOTE

A recovery counter trap, or Group 3 and 4 interruptions are not expected to occur during PDCE_RESET. Consequently, these cases have been included here primarily for purposes of completeness. The intent is also to clarify that entry to the recovery counter trap handler or to Group 3 and 4 fault/trap handlers from PDCE_RESET are not allowed options.

A failed TOC, a failed powerfail recovery attempt, and a broadcast CMD_RESET all cause PDCE_RESET to boot. Boot has two variants; hard and soft. The boot initiated in response to a broadcast CMD_RESET, or a failed powerfail recovery attempt is a hard boot, and tests memory

destructively. The boot initiated in response to a failed TOC is a soft boot, and tests memory nondestructively.

PDCE_RESET must avoid corrupting memory if power fails while it is executing. Assertion of BUS_POW_WARN at any time while PDCE_RESET is executing must cause PDCE_RESET to prepare the system for the impending power failure and enter an idle state (which does not cause any bus transactions) before the deassertion of BUS_POW_VALID.

The boot algorithm is described briefly below, and depicted graphically in Figure 3-6, Boot Algorithm.

1. Initialize Memory

   Configure and test memory so that whatever software is subsequently loaded can use memory with confidence. This is the only step that varies between hard boot and soft boot. Memory initialization is described in Section 3.3, Memory Initialization.

2. Locate and Initialize the Console Device

   Locate a candidate for the console device. Attempt to initialize and test the console device. If the initialization attempt fails, it may be necessary to find another candidate for the console device. This step is described in Section 3.4, Console Device Initialization.

3. Locate and Initialize the Boot Device

   Locate a candidate for the boot device. There are four selection mechanisms by which the candidate is chosen. Attempt to initialize and test the boot device. If the initialization attempt fails, it may be necessary to find another candidate for the boot device. This step is described in Section 3.5, Boot Device Initialization.

4. Load and Launch the IPL Code

   Read in the IPL code from the boot device. Execute that code. IPL is described in Section 3.6, Initial Program Load (IPL).

PROGRAMMING NOTE

After performing these steps, the PDC boot code may wish to send a CMD_CLEAR to all the bus converters ports that it knows of to clear the soft errors that might have been logged as a result of previous calls to PDC_ADD_VALID or PDC_IODC. Further error isolation is hampered by having multiple bus converter ports with residual soft errors.

As shown in Figure 3-6, Boot Algorithm, and described in Section 3.6, Initial Program Load (IPL), IPL then loads and launches ISL, which in turn loads and launches an operating system.

During boot, chassis displays and console messages must be used to keep the operator informed of the progress through the sequence to facilitate troubleshooting of boot failures.

PDCE_RESET must not alter module configuration status maintained as tertiary state.

Example:

PROGRAMMING NOTE

An example algorithm showing the suggested flow of PDCE_RESET for monarch selection, initial memory module selection, and response selection in a multiprocessor system of Category B processors is shown below. A system with a category A processor should implement the appropriate subset of this algorithm.

1. Test the processor. If the test fails, write the appropriate error code to the chassis display, and halt.

2. Write to the LBRS IO_FLEX register on the central bus to initialize the HPA of the processors to the central bus physical address space and to disable the bus requestorship of other modules on the bus.
3. Clear EIR{0}, ensure that the interval timer will not cause an interrupt on EIR{0}, and enable interrupts on EIR{0}. Select the monarch processor as the processor with the highest BOOT_ID, or in the case of a tie, the processor with the lowest HPA.
4. Processors which are not selected as the monarch wait for a rendezvous interrupt. Non-monarch processors, which have not halted, continue to monitor the state of the monarch and if it fails, select a new monarch processor, which will restart boot. On receipt of a rendezvous interrupt, check that the MEM_RENDEZ word is nonzero, and execute the OS_RENDEZ code.
5. The monarch processor initializes the local bus. It issues directed CMD_RESET.ST to each module on the local bus except the processor modules.
6. Read the IODC_TYPE byte of each module on the local bus to determine which ones are memory modules (TP_MEMORY). The memory module (or primary memory module of a processor-dependent interleave group) with the most installed memory, the smallest HPA in the case of a tie in installed memory, and at least 256 Kbytes of memory, is the Initial Memory Module Candidate (IMMC). There is no Initial Memory Module Candidate if there are no memory modules on the processor's local bus, or the largest has less than 256 Kbytes of installed memory, in which case the monarch processor halts. Installed size is determined using the algorithm described in Section 3.3.5, Determining Installed Memory Size.
7. If PDCE_RESET was initiated due to the receipt of a broadcast CMD_RESET, perform a hard boot. If initiated due to power-on, check to see if powerfail recovery should be performed, as described in the following steps.
8. Check IO_STATUS[sl] on the IMMC. If the *sl* bit is 0, continue with the next step, otherwise search for another IMMC (see step 12).
9. Enable the SPA of the IMMC with a base address of 0x00000000 and set its *edc_enb* bit to 1.
10. Verify that the IMMC is the same memory module that was identified as the IMM during the most recent hard or soft boot. If the IMMC was the primary memory module of a processor-dependent interleave group, then configure the satellites of that group into their proper places and enable their SPAs. If the IMMC is the same as the most recent IMM then continue with step 11, otherwise search for another IMMC (see step 12).
11. Check MEM_POW_FAIL and compute the powerfail checksum. If MEM_POW_FAIL is nonzero and the checksum is correct then powerfail recovery should be performed, so write the appropriate code (CA0x) to the chassis display, and branch to the OS_PFR code, which is the code pointed to by MEM_POW_FAIL. Before branching to OS_PFR, check BUS_POW_WARN. Branch to OS_PFR only if BUS_POW_WARN is not asserted. If BUS_POW_WARN is asserted, then loop idly as in PDC_POW_FAIL. If MEM_POW_FAIL is zero or the checksum is incorrect, search for another IMMC as described in the next step.
12. Search for another IMMC. The next candidate is chosen using the algorithm in step 6 excluding all memory modules which have already been candidates. If another candidate is found, repeat steps 8 through 12. If no more candidates exist recovery is not possible, so perform a hard boot.

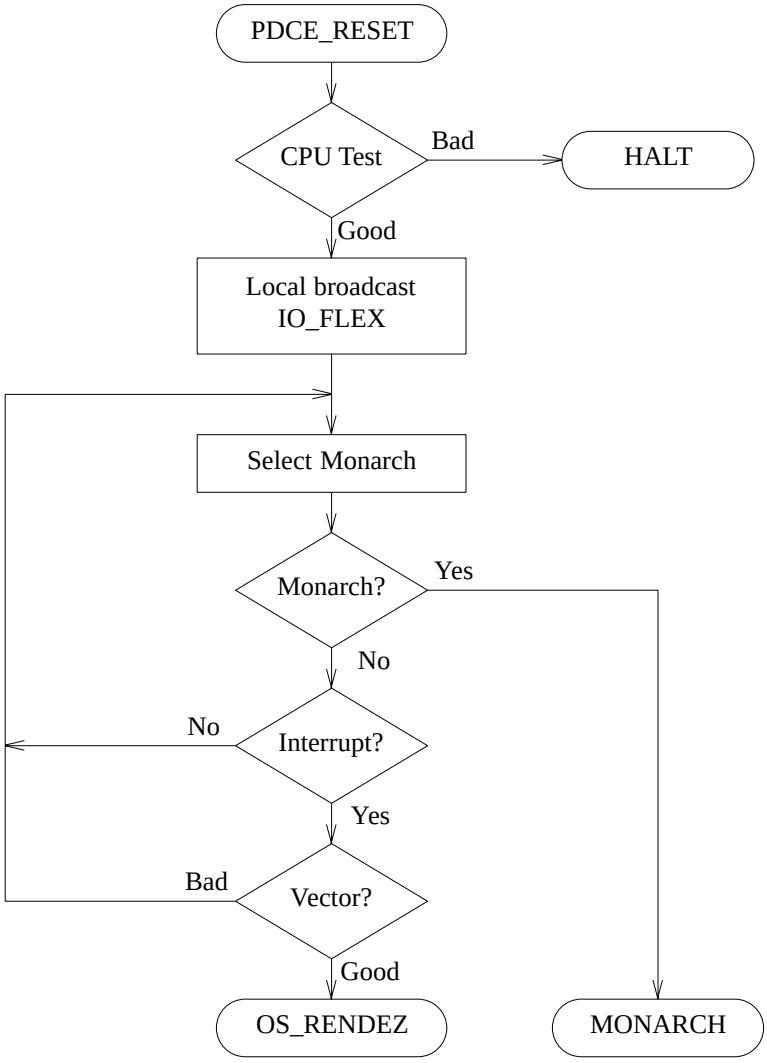


Figure 1-4. Processor Reset and Monarch Selection (PDCE_RESET) Algorithm

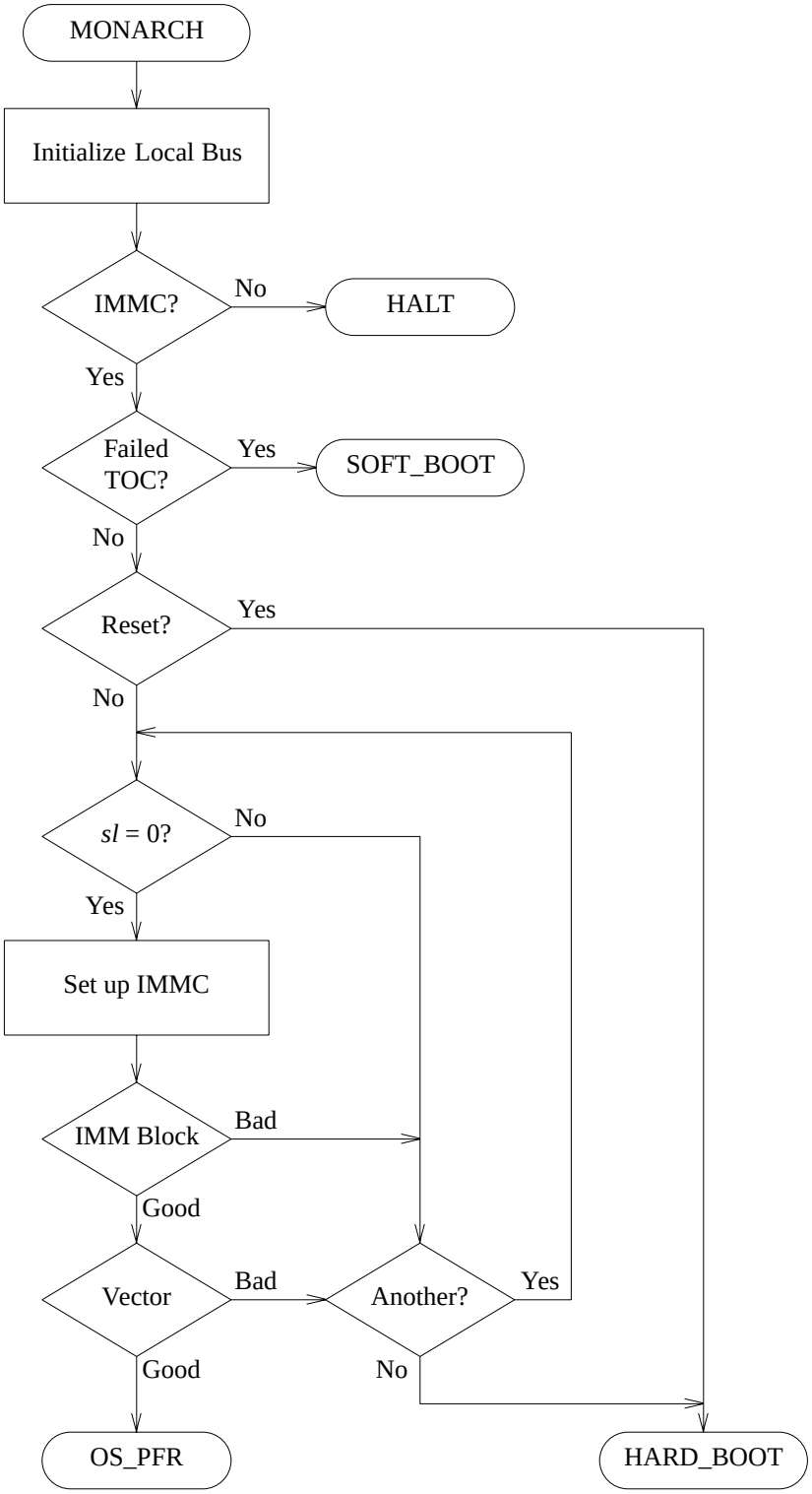


Figure 1-5. Monarch Processor Reset (PDCE_RESET) Algorithm

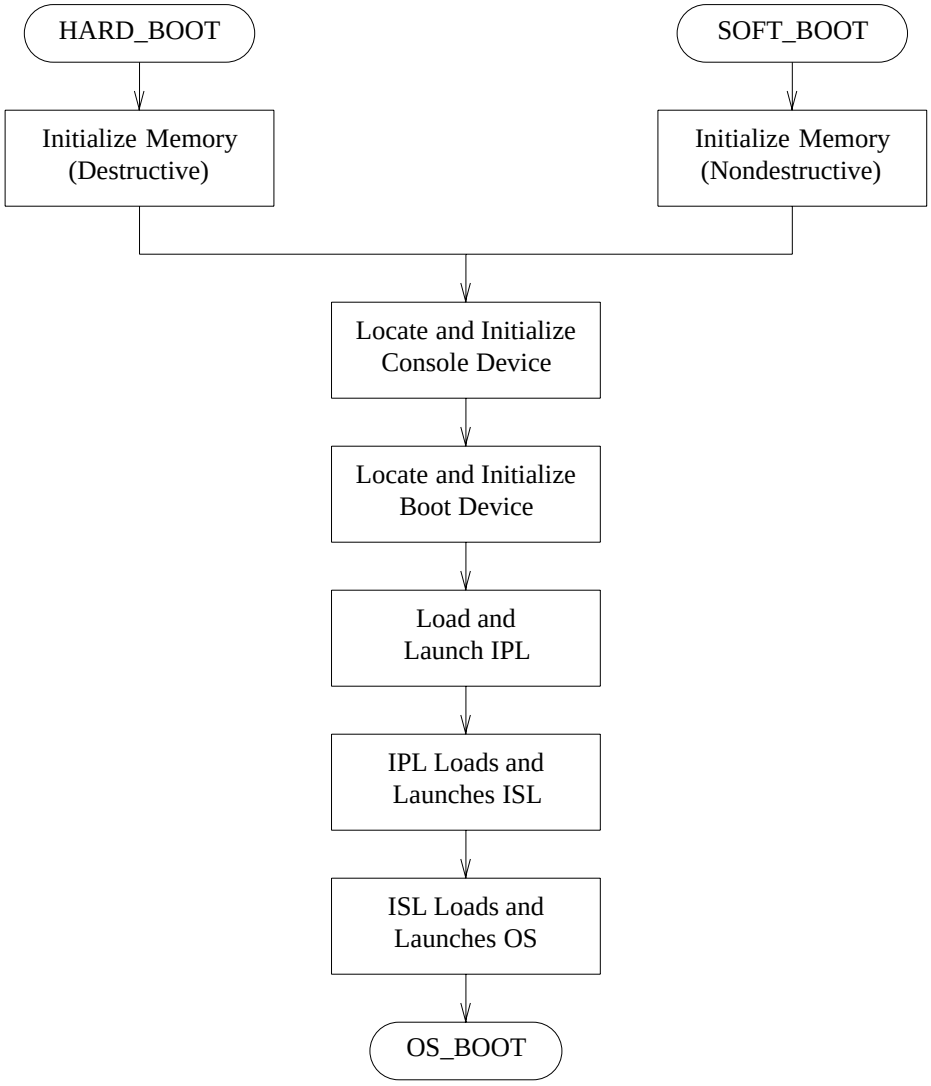


Figure 1-6. Boot Algorithm

Synopsis: The OS_PFR interfaces defines the boundary between PDCE_RESET and the operating system powerfail recovery code (OS_PFR) for the monarch processor.

Privilege: Level 0.

PSW: The Q-bit is 1. E-bit is set to default endianness. W-bit is set to default width.

All other bits are 0.

CRs: CR14 (Interrupt Vector Address) contains the address of the Interrupt Vector Table containing code to handle an HPMC.

All other control registers are HVERSION dependent.

GRs: GR26 contains the address of the monarch's PDCE_PROC.

All other general registers are HVERSION dependent.

SRs: All the space registers are HVERSION dependent.

PIM: PIM is HVERSION dependent.

Cache: The cache is clean or invalid.

Cache coherence is enabled.

TLB: The TLB is initialized and invalid. Hardware TLB miss handling is disabled.

Memory: PDC in category A processor modules must guarantee, in an HVERSION dependent-manner, that the Initial Memory Module (IMM) located during powerfail recovery is the same IMM located during the most recent hard or soft boot.

PDC in category B processor modules must guarantee, in an SVERSION dependent-manner, that the IMM located during powerfail recovery is the same IMM located during the most recent hard or soft boot.

The IMM has its SPA enabled to a base address of 0x00000000, IO_CONTROL[edc-enb] set to 1, and IO_STATUS[sl] is 0. The HVERSION-dependent equivalent of these requirements must be satisfied for processor-dependent memory modules.

The extended address spaces of other modules do not conflict with the SPA of the IMM.

The MEM_POW_FAIL word is nonzero, the checksum of MEM_PF_LEN bytes of code at the location pointed to by MEM_POW_FAIL is zero, and the first word of the code at the address pointed to by MEM_POW_FAIL is non-zero.

For all processor-dependent memory modules and interleave groups that are configurable (for example, are composed of array units which can be configured by PDC to contain different parts of the SPA) the following conditions are true:

- If the processor-dependent memory module or interleave group is the IMM, then its configuration is the same as it was prior to powerfail.
- If the processor-dependent memory module or interleave group is not the IMM, then the appropriate call to PDC_IODC "Nondestructive init" will restore its configuration to be the same as it was prior to powerfail. If the configuration can not be restored, then module's HVERSION-dependent equivalent of its IO_STATUS[sl] bit is set.

The configuration of a processor-dependent memory module or interleave group is considered to be the same if all locations in the SPA contain the same values as before power failed.

MEM_ERR is zero if no errors were detected.

The operating system must insure that all code that comprises OS_PFR (up to the point at which the SPAs of memory modules after the IMM are re-enabled) resides within the SPA space of the IMM.

PROGRAMMING NOTE

Since the I/O Architecture guarantees a minimum SPA size of only 256 Kbytes, the operating systems must verify on a product-by-product basis that the IMM actually provided is large enough for their needs.

Other memory is unchanged.

Other: All HVERSION-dependent hardware in the processor has been initialized, except possibly any coprocessors. Coprocessors must be initialized by PDCE_RESET or PDC_COPROC.

All the modules on the central bus have their HPAs initialized to the central bus physical address space and bus requestorship disabled.

A monarch processor has been identified as the processor with the highest value of BOOT_ID, or, in the case of a tie, the lowest HPA, on the central bus. The monarch processor will execute the OS_PFR code.

Other processors must be in a HALT state or waiting for a rendezvous interrupt.

BUS_POW_WARN is not asserted.

Synopsis: The OS_BOOT interface defines the boundary between PDCE_RESET and IPL, and between ISL and the operating system boot code (OS_BOOT) for the monarch processor.

Privilege: Level 0.

PSW: The Q-bit is 1. E-bit is 0 (big endian). W-bit is 0 (narrow mode).

All other bits are 0.

CRs: CR14 (Interrupt Vector Address) contains the address of the Interrupt Vector Table containing code to handle an HPMC.

CRs 0 (Recovery Counter), 8, 9, 12, and 13 (Protection IDs), 10 (Coprocessor Configuration Register), 11 (Shift Amount Register), 15 (External Interrupt Enable Mask), 16 (Interval Timer), and 23 (External Interrupt Request Register) are 0.

All other control registers are HVERSION dependent.

GRs: For the interface between PDCE_RESET and IPL.

- GR25 contains the address of the first doubleword past the end of the IPL code.
- GR26 is 0 if IPL is noninteractive, and 1 if it is interactive.
- All other general registers are HVERSION dependent.

For the interface between ISL and OS_BOOT.

- GR2 contains the return pointer ISL.
- GR24 contains the entry pointer to the utility.
- GR25 contains the boot command from ISL.
- GR26 contains the pointer to PDCE_PROC.
- All other general registers are HVERSION dependent.

All of the above pointers must be within the first 3.75 GB of memory, and must be reachable in 32-bit addressing mode (narrow mode).

SRs: All the space registers are HVERSION dependent.

PIM: If a hard boot is being performed, then PIM is HVERSION dependent.

If a soft boot is being performed, then PIM is HVERSION dependent for category A processors, and contains the state of processor at the time soft boot was triggered for category B processors.

Cache: The cache is clean or invalid, and contains only physical addresses.

Cache coherence is enabled.

TLB: The TLB is initialized and invalid. Hardware TLB miss handling is disabled.

Memory: The architected memory module or primary memory module of a processor-dependent interleave group which passed its testing, has the most installed memory, the smallest HPA in the case of a tie in installed memory, and at least 256 Kbytes of installed memory, is the Initial Memory Module (IMM).

The IMM has been tested and initialized, its SPA enabled to a base address of 0x00000000, and IO_STATUS[sl] set to 0.

The SPAs of other modules do not conflict with the IMM, other processor modules, the boot module, or the console module.

The architected words in Page Zero are set to their prescribed values (for example, the Initialize Vectors (words 0x00 through 0x3C) are all zero, MEM_FREE is initialized, and MEM_PDC

points to the monarch processor's PDCE_PROC, MEM_ERR is zero if no errors were detected).

Other memory modules on the central bus have been tested and initialized as specified by the *fast_size* flag.

See Section 3.3, Memory Initialization for a detailed description of the memory initialization requirements.

If the IODC_IO word of the Console/Display structure in Page Zero is nonzero, then the console ENTRY_IO is established in memory at the location pointed to by IODC_IO. Furthermore, a nonzero IODC_IO word implies that ENTRY_INIT has already been called, and that the module-device connection between the console module and the console device is open.

If the IODC_IO word of the Keyboard structure in Page Zero is nonzero, then the keyboard ENTRY_IO is established in memory at the location pointed to by IODC_IO. Furthermore, a nonzero IODC_IO word implies that ENTRY_INIT has already been called, and that the module-device connection between the keyboard module and the keyboard device is open.

See Section 3.4, Console Device Initialization for a detailed description of the console initialization requirements.

The IODC_IO word of the Boot Device structure in Page Zero is nonzero, and points to the location of the boot device ENTRY_IO in memory. ENTRY_INIT has already been called, and that the module-device connection between the boot module and the boot device is open.

See Section 3.4, Console Device Initialization for a detailed description of the boot device initialization requirements.

The ENTRY_IO entry points for the console device and keyboard device are located at physical addresses which are lower than IPL_START.

Other: All HVERSION-dependent hardware in the processor has been initialized, except possibly any coprocessors. Coprocessors must be initialized by PDCE_RESET or PDC_COPROC.

All the modules on the central bus have their HPAs initialized to the central bus physical address space and bus requestorship enabled.

All modules on the central bus, except for the processors, have received a directed CMD_RESET.ST.

A monarch processor has been identified as the processor with the highest value of BOOT_ID, or, in the case of a tie, the lowest HPA, on the central bus. The monarch processor will execute the IPL and OS_BOOT code.

Other processors must be in a HALT state or waiting for a rendezvous interrupt.

All bus converters on the paths to the console module and boot module have been initialized, and tested using ENTRY_TEST. All busses on the paths to the console module and boot module have been allocated address space. All modules on those busses have their HPAs initialized, have received a directed CMD_RESET.ST, and have bus requestorship enabled. The state of all other modules is HVERSION dependent (they cannot be presumed to have been reset). The SPA spaces of other modules may be enabled, but are guaranteed not to conflict with the address spaces of any memory module, the boot device, or the console device.

BUS_POW_WARN is not asserted.

PROGRAMMING NOTE

When IPL is invoked, a stack is not initialized and the stack pointer (GR30) is HVERSION dependent. The IPL code is responsible for establishing its own stack, subject to the following constraints:

- The memory address space from Page Zero to MEM_FREE is allocated to PDC.
- Use of the memory space from MEM_FREE to IPL_START may conflict with the IODC ENTRY_IO for the console device and/or boot device.

Synopsis: The OS_RENDEZ interface defines the boundary between PDCE_RESET and the operating system rendezvous code (OS_RENDEZ), and between PDC_PROC and the operating system rendezvous code (OS_RENDEZ) for non-monarch processors.

Privilege: Level 0.

PSW: The Q-bit is 1. E-bit is set to default endianness. W-bit is set to default width.

All other bits are 0.

CRs: CR14 (Interrupt Vector Address) is 0.

All other control registers are HVERSION dependent.

GRs: GR26 contains the address of PDCE_PROC.

On category B (multiprocessor) processor modules, GR25 contains the address of the HPA of the current processor. On category A (uniprocessor) processor modules, GR25 is HVERSION dependent.

All other general registers are HVERSION dependent.

SRs: All the space registers are HVERSION dependent.

PIM: If a hard boot is being performed, then PIM is HVERSION dependent.

If a soft boot is being performed, then PIM is HVERSION dependent for category A processors, and contains the processor state at the time soft boot was triggered for category B processors.

If a directed rendezvous is being performed via PDC_PROC, then PIM remains unchanged.

Cache: The cache is clean or invalid, and contains only physical addresses.

Cache coherence is enabled.

TLB: The TLB is initialized and invalid. Hardware TLB miss handling is disabled.

Memory: Memory in Page Zero or above MEM_FREE is unchanged by non-monarch processors.

Other: All HVERSION-dependent hardware in the processor has been initialized, except possibly any coprocessors. Coprocessors must be initialized by PDCE_RESET or PDC_COPROC.

The MEM_RENDEZ word must be nonzero, and the first word of the code at the location pointed to by MEM_RENDEZ must be nonzero.

An external interrupt has been received on EIR{0}.

PROGRAMMING NOTE

If a processor does not respond within one second of receiving the rendezvous interrupt, this may be taken as an indication that the processor has entered the HALT state.

BUS_POW_WARN is not asserted.

1.3 Memory Initialization

The objective of memory initialization is to initialize and test sufficient memory to load in ISL or an operating system. Memory initialization is destructive or nondestructive depending on whether a hard boot or soft boot is being performed.

When memory initialization is complete, the following requirements must be satisfied:

- Bus requestorship on the central bus is enabled.
- Memory initialization (destructive or nondestructive as appropriate) has been performed.
- The architected words in Page Zero are set to their prescribed values, as follows:
  - — The reserved areas of Page Zero are zero.
  - — The Initialize Vectors (words 0x000 through 0x03C) are zero.
  - — The processor-dependent areas of Page Zero are initialized. If portions of the processor-dependent areas of Page Zero are used to store PIM information, those portions should not be destroyed.
  - — The Memory Configuration structure (at address 0x350) is set appropriately.
  - — The MEM_ERR words contain information about any memory failures that occurred during boot.
  - — The MEM_HPA word contains the monarch processor's HPA.
  - — The MEM_PDC and MEM_10MSEC words are set appropriately.
  - — The Initial Memory Module structure (at address 0x390) is set appropriately.
- The IO_STATUS[sl] bit of the Initial Memory Module has been cleared to zero.

Details of destructive and nondestructive initialization and array testing, and determining the size of installed memory are described in Section 3.3.1, Destructive Memory Initialization, through Section 3.3.5, Determining Installed Memory Size.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to initialize memory.

1. Write the appropriate code (C200 {initialize}) to the chassis display.
2. Issue a local broadcast to IO_FLEX to enable bus requestorship on the central bus.
3. Read the *fast-size* flag from byte 0x5F of Stable Storage.

   If *fast-size* is not implemented, all memory on the central bus must be tested and initialized, and the Support requirements limiting memory initialization times must be satisfied.
4. For a hard boot, memory must be tested destructively as described in Section 3.3.1, Destructive Memory Initialization.

   For a soft boot, memory must be tested nondestructively as described in Section 3.3.3, Nondestructive Memory Initialization.

   If, during a soft boot, an error occurs which results in memory which would normally be preserved by a soft boot being lost, the boot must be changed to a hard boot.
5. Initialize Page Zero to the proper values.
6. Clear the Initial Memory Module's IO_STATUS[sl] bit to zero, but only after the MEM_POW_FAIL vector is cleared to zero.

1.3.1 Destructive Memory Initialization

Destructive memory initialization is performed as part of hard boot, and when complete, the following requirements must be satisfied:

- All memory up to *fast-size* has undergone the destructive array test, as described in Section 3.3.2, Destructive Array Test.
- All memory modules contributing memory to the *fast-size* amount have had ENTRY_TEST (described in Section 5.4, IODC Entry Points) or, for processor-dependent memory modules, the PDC-based equivalent of ENTRY_TEST executed and a status value >=0 or -5 has been returned at the end of the ENTRY_TEST sequence.
- If page deallocation is disabled, only those memory modules for which the destructive array test detected no uncorrectable errors have had SPA space assigned and enabled. If page deallocation is enabled, all memory modules have had SPA space assigned and are enabled, and all uncorrectable errors have been added to the bad page table. (See PDC procedure PDC_MEM).

ENGINEERING NOTE

When PDCE_RESET initializes memory with page deallocation enabled, an attempt should be made to maximize the amount of memory prior to the first bad page, either by suitable choice of the IMMC, or the HVERSION dependent hardware memory to SPA mapping.

PROGRAMMING NOTE

It is the responsibility of any function which loads software to be run to ensure there is sufficient contiguous memory without a bad page to load that software. It is the responsibility of loaded software to run within the confines of its loaded area until reads the bad page table to determine the location of inaccessible pages.

- Correctable array errors and status value -5 returned by the ENTRY_TEST sequence may optionally be logged in MEM_ERR. Not logging these errors must not halt hard boot. If page deallocation is enabled, these errors may optionally be added to the bad page table.

  Uncorrectable array errors and status values other than 0 or -5 returned by the ENTRY_TEST sequence may optionally be logged in MEM_ERR. If page deallocation is disabled, not logging these errors must halt hard boot.

SUPPORT NOTE

Initializing a memory module which has correctable array errors or for which ENTRY_TEST returned status value -5 guarantees higher system availability: the system is allowed to complete hard boot after which specialized diagnostics could be executed to further isolate the cause of the error. The OS uses the information in MEM_ERR to make the final decision as to whether the memory module stays enabled or not.

Note that the correctable error could be caused by a stuck-at fault in the memory array. The stuck-at fault increases the probability of having subsequent uncorrectable errors, causing the system to halt. Therefore, whenever it is possible to select another IMMC it is recommended that any memory module with correctable errors not be initialized as the IMM.

- Memory modules have been assigned an enabled SPA space in the order of decreasing installed memory (see Section 3.3.5, Determining Installed Memory Size, for a description of how to determine the size of installed memory) and increasing HPA.

- The soft-boot bit, *sb*, in the Initial Memory Module structure in Page Zero is 0.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to destructively initialize both architected and processor-dependent memory modules when page deallocation is disabled. The algorithm assumes that an Initial Memory Module Candidate (IMMC) has already been selected by PDCE_RESET.

1. Perform the destructive array test. If the module passed the test, log in MEM_ERR any correctable array errors found and continue with step 2. If the module failed the test, but was not the IMMC, log the failure in MEM_ERR and continue with step 2. If the module failed the test and was the IMMC, check if another IMMC exists. If there is another IMMC, repeat this step for the new IMMC. If no other IMMC exists, display an error code and halt.
2. Load ENTRY_TEST for the memory module and execute the Default Non User Input Offline Test (*list_type* = 2). If the module passed the test, log in MEM_ERR if ENTRY_TEST returned a status of -5 and continue with step 3. If the module failed the test but was not the IMMC, log the failure in MEM_ERR and continue with step 3. If the module failed the test and was the IMMC, check if another IMMC exists. If there is another IMMC, repeat this step for the new IMMC. If no other IMMC exists, the monarch processor should write the appropriate code on the chassis display, and halt.
3. Repeat steps 1 and 2 until
   - the *fast-size* limit is reached,
   - the end of contiguous memory is found and *fast-size* is less than 0xE, or
   - all memory on the central bus has been initialized.

   If any of these conditions is met, continue with step 4. The next module to initialize is determined by its installed size and HPA, as described above.
4. Log all IMMC failures in MEM_ERR. MEM_ERR can accommodate only eight errors. If more than eight memory errors need to be logged, halt.
5. Clear the soft-boot bit, *sb*, in the Initial Memory Module structure in Page Zero to 0.

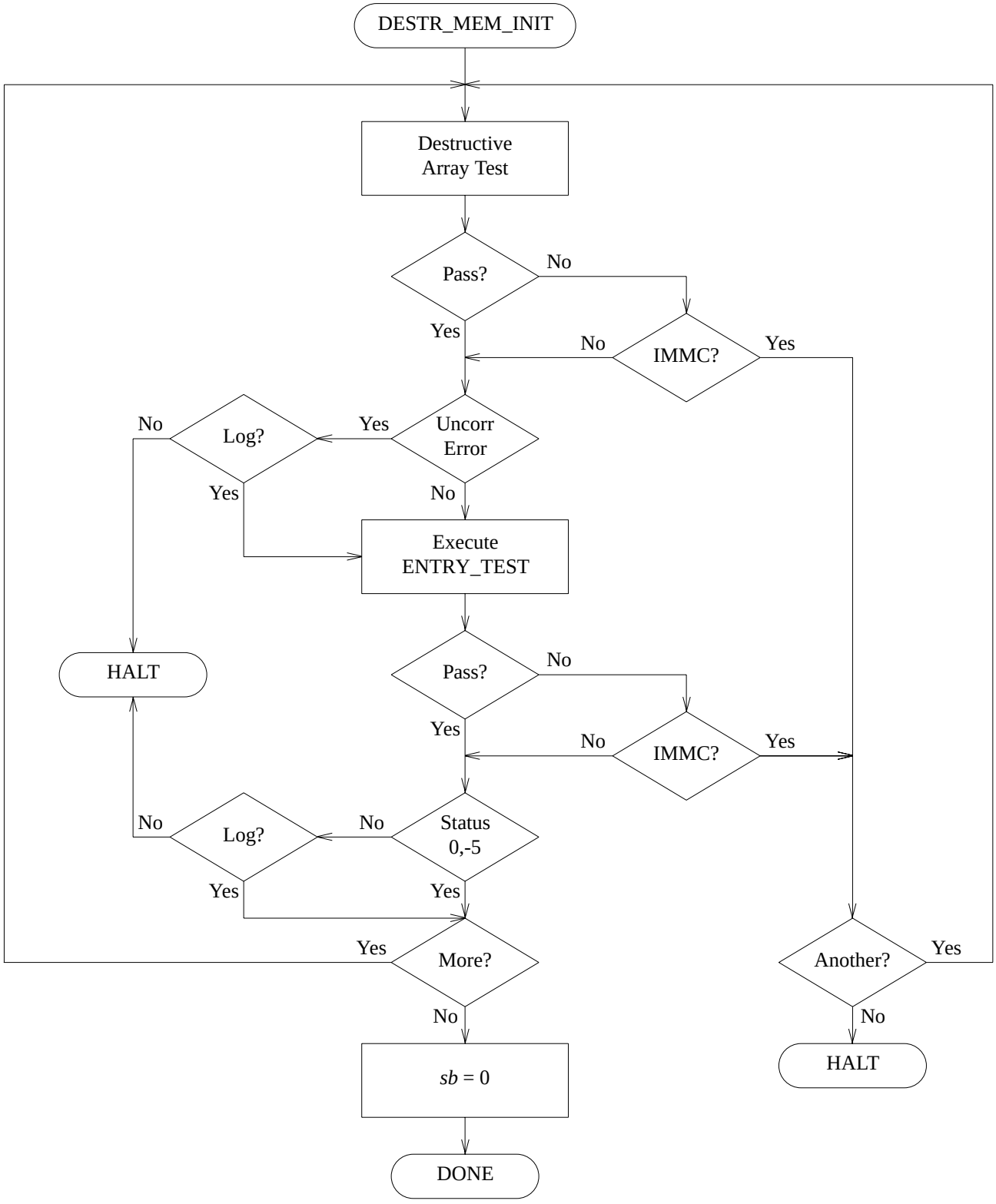


Figure 1-7. Destructive Memory Initialization Algorithm

1.3.2 Destructive Array Test

The destructive array test is done as part of destructive memory initialization, and when complete, the following requirements must be satisfied:

- All check code errors (uninitialized ECC) have been cleared from the memory array.
- All bits in the memory array can be set to both 0 and 1.
- Address aliasing problems related to "stuck-at" address bits have been detected.
- The pseudo-random pattern specified by the following algorithm has been written, read, and verified for each location from 0 through the installed size.

```
unsigned int size, i;
unsigned int seed, val1;
unsigned int mem[size]

seed ← 0x00033DCF;
val1 ← 0x0076B553;

for (i ← 0; i < size; i++) {
    mem[i] ← seed;
    if (seed & 0x80000000)
        seed ← (seed << 1) ^ val1;
    else
        seed ← seed << 1;
}
```

  The complement of the previous pseudo-random pattern has been written, read, and verified for each location from the installed size through 0, using the following algorithm:

```
unsigned int size, i;
unsigned int seed, val2;
unsigned int mem[size]

/* initial seed equals final seed from previous routine */

val2 ← 0x003B5AA9;

for (i ← size–1; i >= 0; i––) {
    if (seed & 0x1)
        seed ← ((seed >> 1) | 0x80000000)) ^ val2;
    else
        seed ← seed >> 1;
    if (mem[i] != seed)
        error();
    mem[i] ← ~seed;
}
```

PROGRAMMING NOTE

This allows the operating system to use these same patterns later when salvaging memory modules.

- If page deallocation is disabled, any memory module which failed the test has its SPA disabled. If page deallocation is enabled, the page containing the failing location must be added to the bad page table.
- The time to perform the test was limited to less than or equal to 5 minutes.
- Pattern tests are not performed; these are best left to HVERSION-dependent diagnostic software or ENTRY_TEST.
- The SPA is enabled to a properly aligned base address that does not conflict with the address space of any other module.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to destructively test the memory arrays of both architected and processor-dependent memory modules when page deallocation is disabled. For processor-dependent memory, PDC must use the HVERSION-dependent equivalents of IO_CONTROL and IO_STATUS.

1. Enable the SPA as described in Section 3.3.5, Determining Installed Memory Size.
2. Set IO_CONTROL[edc-enb] = 0. This disables assertion of PATH_ERROR so the check codes can be initialized without generating HPMCs.
3. Write a pseudo-random value to each word from 0 through the installed size, using the first algorithm described above.
4. Flush the entire data cache.
5. Set IO_CONTROL[edc-enb] = 1.
6. Read and verify the pseudo-random data and write the complement for all locations from the installed size through 0, using the second algorithm described above.
7. Flush the entire data cache.
8. Read the data in each location from 0 through the installed size or the *fast-size* limit, and verify that it matches the value written in step 6.
9. Read IO_STATUS. If no error is logged, the array test is complete with no errors. If a corrected error is logged, log the error in MEM_ERR, as described in Section 3.3.1, Destructive Memory Initialization.

If, at any point in the array test, there was an HPMC, the verification of a location failed, or the read of IO_STATUS indicated an uncorrected error, disable the SPA space of the memory module, and proceed as described in Section 3.3.1, Destructive Memory Initialization.

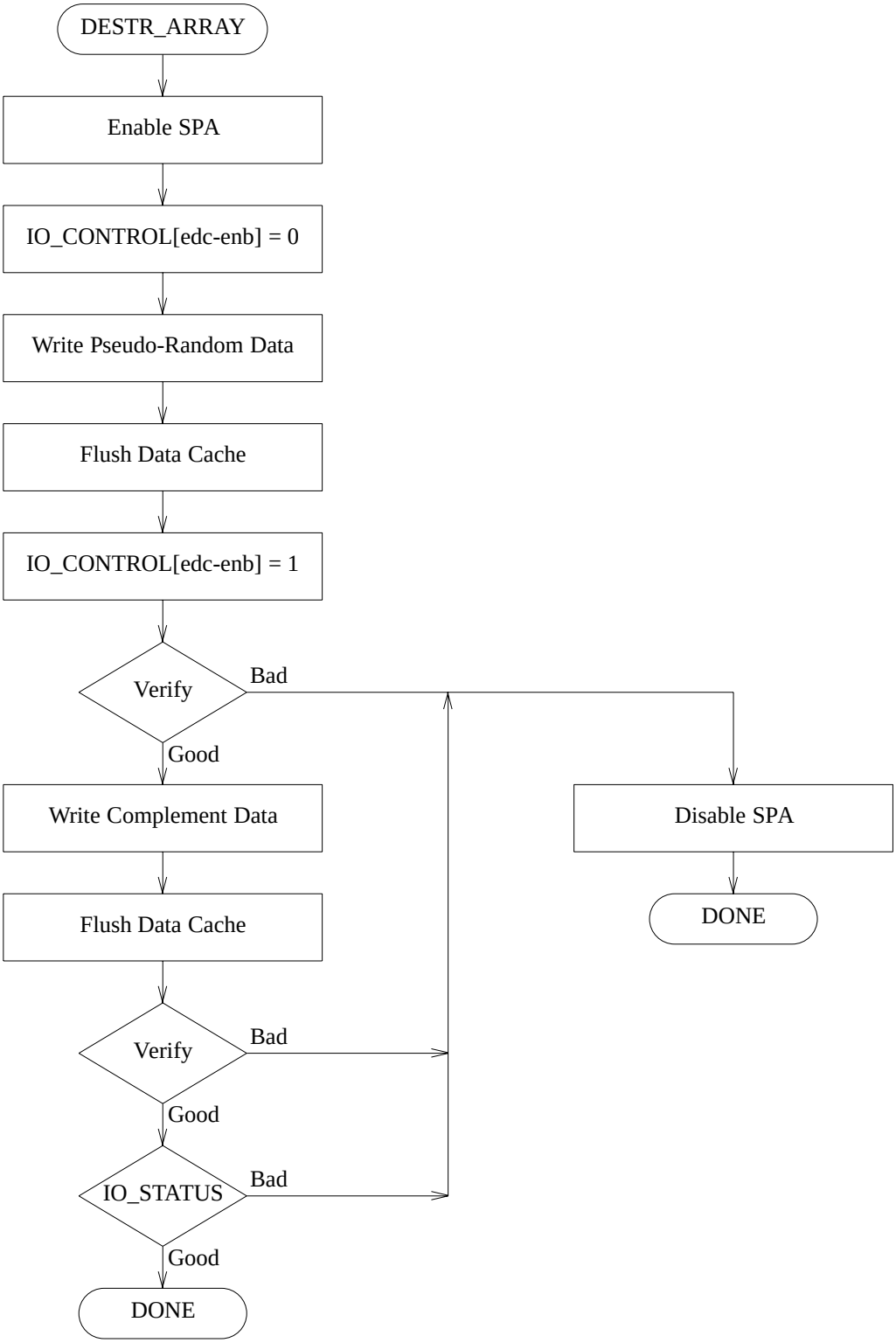


Figure 1-8. Destructive RAM Array Test Algorithm

1.3.3 Nondestructive Memory Initialization

Nondestructive memory initialization is performed as part of soft boot, and when complete, the following requirements must be satisfied:

- All memory up to *fast-size* has undergone the nondestructive array test, as described in Section 3.3.4, Nondestructive Array Test.
- Every memory module has undergone a nondestructive array test. Soft boot initializes the memory module if and only if the nondestructive array test detected no uncorrectable errors.

  Correctable errors may optionally be logged in MEM_ERR. Not logging these errors must not halt soft boot.

  Uncorrectable array errors may optionally be logged in MEM_ERR. Whether the error is logged or not, soft boot must either switch to hard boot or halt.

SUPPORT NOTE

Initializing a memory module which has correctable array errors guarantees higher system availability: the system is allowed to complete soft boot after which specialized diagnostics could be executed to further isolate the cause of the error. The OS uses the information in MEM_ERR to make the final decision as to whether the memory module stays enabled or not.

Note that the correctable error could be caused by a stuck-at fault in the memory array. The stuck-at fault increases the probability of having subsequent uncorrectable errors, causing the system to halt. Therefore, whenever it is possible to select another IMMC it is recommended that any memory module with correctable errors not be initialized as the IMM.

- Memory modules have been initialized in the order of decreasing installed memory (see Section 3.3.5, Determining Installed Memory Size, for a description of how to determine the size of installed memory) and increasing HPA.
- The soft-boot bit, *sb*, in the Initial Memory Module structure in Page Zero is 1.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to nondestructively initialize both architected and processor-dependent memory modules.

1. Perform the nondestructive array test. If the module passed the test, continue with step 2. If it failed, log in MEM_ERR the information relating to the memory module's HPA, the type of error, and the fact that a soft boot attempt failed, so that the OS can determine that a soft boot failed. Branch to PDCE_RESET to force a HARD_BOOT to be attempted.
2. Repeat step 1 until
   - the *fast-size* limit is reached,
   - the end of contiguous memory is found and *fast-size* is less than 0xE, or
   - all memory on the central bus has been initialized.

   If any of these conditions is met, continue with the next step. The next module to initialize is determined by its installed size and HPA, as described above.
3. Set the soft-boot bit, *sb*, in the Initial Memory Module structure in Page Zero to 1.

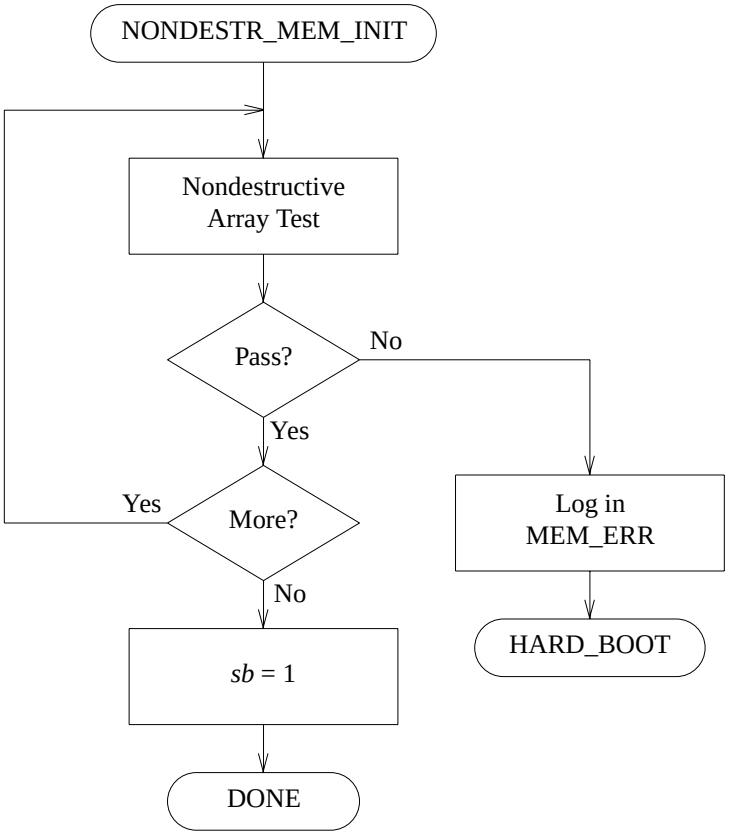


Figure 1-9. Nondestructive Memory Initialization Algorithm

1.3.4 Nondestructive Array Test

The nondestructive array test is done as part of nondestructive memory initialization, and when complete, the following requirements must be satisfied:

- The original value of each location is left unchanged.
- All latent correctable errors in the RAM array have been detected.
- All uncorrectable errors in the RAM array have been detected.
- The SPA is enabled to a properly aligned base address that does not conflict with the address space of any other module.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to nondestructively test the memory arrays of both architected and processor-dependent memory modules. For processor-dependent memory, PDC must use the HVERSION-dependent equivalents of IO_CONTROL and IO_STATUS.

1. Enable the SPA as described in Section 3.3.5, Determining Installed Memory Size.
2. Set IO_CONTROL[edc-enb] = 1.
3. Flush the entire data cache.
4. Read every location from 0 through the installed size or the *fast-size* limit. The processor data cache line size may be used as the stride.

5. Read IO_STATUS. If no error is logged, the array test is complete with no errors. If a corrected error is logged, log the error in MEM_ERR, as described in Section 3.3.3, Nondestructive Memory Initialization.

If, at any point in the array test, there was an HPMC, or the read of IO_STATUS indicates an uncorrected error, proceed as described in Section 3.3.1, Destructive Memory Initialization.

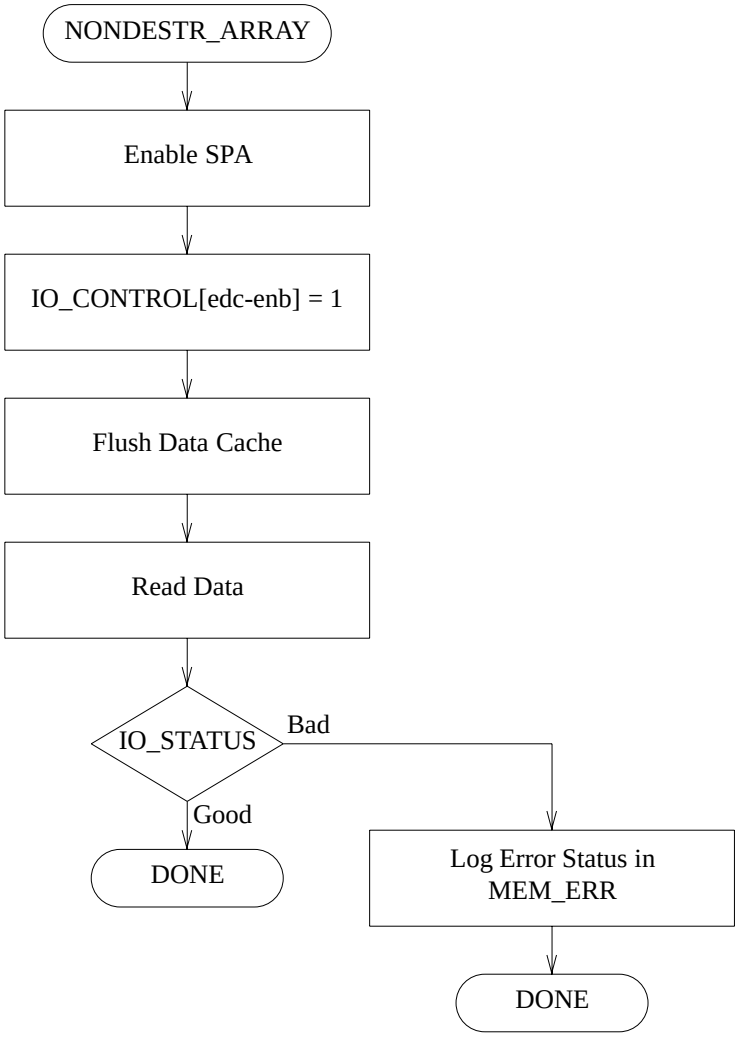


Figure 1-10. Nondestructive RAM Array Test Algorithm

1.3.5 Determining Installed Memory Size

When the installed memory size has been determined, the following conditions must have been satisfied:

- For a given memory module, the amount of memory actually installed in that module has been determined. The installed memory size is less than or equal to the SPA size of the module.
- Load word instructions were used to probe memory locations.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to determine the installed memory size.

- If the memory module is processor-dependent:
  1. Call the "Nondestructive init" option (ARG1=2) of PDC_IODC.
  2. The return parameters *max_spa* and *max_mem* contain the SPA size and installed memory size, respectively.
- If the memory module is architected:
  1. Set IO_CONTROL[edc-enb] = 0. This disables assertion of PATH_ERROR, and prevents logging of ERR_UNCORR.
  2. Read IODC_SPA to determine the SPA size of the module.
  3. Set IO_SPA[address] = *base_addr* and IO_SPA[iodc-spa] = IODC_SPA. The value of *base_addr* must be chosen so that the SPA space is aligned on the appropriate boundary and does not overlap the address space of any other module. A *base_addr* of 0 meets all alignment restrictions.
  4. Perform a binary search of the memory space between the last known installed location and the first nonexistent location. Initially, the last known installed location is *base_addr* + 1/2 * $2^{IODC\_SPA[shift]}$ and the first nonexistent location is *base_addr* + $2^{IODC\_SPA[shift]}$. The binary search is done by probing the location which is halfway between the last known installed location and the first nonexistent location. Terminate the search when the last known installed location is within 4 Kbytes of the first nonexistent location.

     To probe a location do the following:

     a. Read the location.

     b. Read IO_STATUS and check the *estat* field for ERR_UNIMPL.

     c. If ERR_UNIMPL is found, that memory location does not exist. The probed location is now the first nonexistent location.
     Write CMD_RESET.SI to IO_COMMAND to clear the error, and set up IO_SPA again as described in step 3.

     d. If ERR_UNIMPL is not found, that memory location (and every other location on that page) is installed. The probed location is now the last known installed location.
  5. Set IO_SPA = 0 to disable the SPA space.

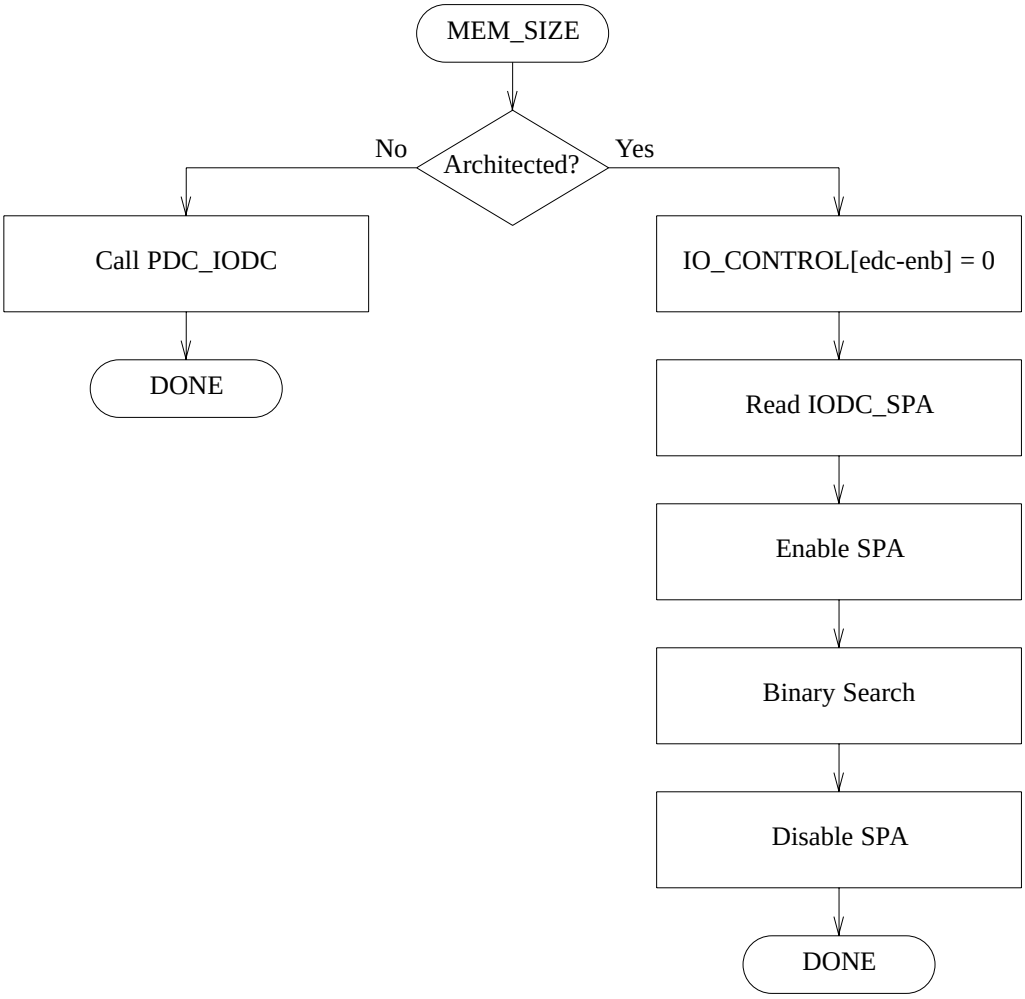


Figure 1-11. Determine Installed Memory Size Algorithm

1.4 Console Device Initialization

The objective of console device initialization is to locate the console module and device, and then to open a module-device connection to the console device. This connection is used by PDC during later stages of the boot process, and remains open for use by IPL.

Two types of console devices are supported. In a duplex console device, the same device provides display output and keyboard input. In a simplex console device, different devices, which may be connected to different modules, provide display output and keyboard input.

When console device initialization is complete, the following requirements must have been satisfied:

- The Console/Display structure (at address 0x3A0) in Page Zero is properly initialized. If a simplex console is used, the Boot Console/Display structure points to the display device.
- If the Console/Display[IODC_IO] word is nonzero, the console device ENTRY_IO code is established in memory at the location specified by the IODC_IO word, and a module-device connection to the console device has been opened by ENTRY_INIT (see Section 5.4, IODC Entry Points, for a description of ENTRY_INIT and ENTRY_IO).
- If a simplex console is being used, the Keyboard structure in Page Zero (at address 0x400) is properly initialized.
- If the Keyboard[IODC_IO] word is nonzero, the console keyboard device ENTRY_IO code is established in memory at the location specified by the IODC_IO word, and a module-device connection to the keyboard device has been opened by ENTRY_INIT.
- All bus converters in the path(s) to the console module(s) have been initialized, tested using ENTRY_TEST, and allocated address space.
- The SPA of the console module(s) has been initialized, and does not conflict with the address space of any other module.
- The console module(s) has bus requestorship enabled.

1.4.1 Determining the Console Module Path

The path to the console/display module is stored in the first 32 bytes of the Console/Display structure, and must be determined from:

- The optional Console/Display Path structure in Stable Storage
- A set of system-specific default paths, also called the "hardwired" paths.
- A set of null values in the Console/Display structure in Stable Storage indicating that no console is present. (That is, the fields BC(0) through BC(5) and MOD contain values in the range 128 to 255.)
- The result of a search for a console.

If a simplex console is used, then the path to the console keyboard must also be determined. The path to the keyboard module is stored in the first 32 bytes of the Keyboard structure, and must be determined from:

- The optional Keyboard Path structure in Stable Storage (at address 0xA0).
- A set of system-specific default paths, also called the "hardwired" paths.
- A set of null values in the Keyboard structure in Stable Storage indicating that no keyboard is present. (That is, the fields BC(0) through BC(5) and MOD contain values in the range 128 to 255.)
- The result of a search for a console keyboard.

1.4.2 Searching for a Console

If PDC searches for a simplex console, the system must ensure that the search finds the console display first, and the console keyboard second.

If the optional Console/Display Path in Stable Storage is implemented and contains a path that can be successfully initialized, this path must be used as the first candidate for the console path.

If the optional Console/Display Path in Stable Storage is not implemented or contains a path that cannot be successfully initialized, a set of hardwired paths may optionally be tried as candidates for the console path.

Searching for the console can be complicated if there are many levels of bus converters in the system, and so the following requirements must be satisfied:

- When a bus is being searched, all modules must be checked in order from lowest HPA to highest HPA.
- Busses are searched breadth first, i.e., in the following order:
  1. The processor's local bus.
  2. Busses at a depth of 1 (there is only 1 bus converter between the bus and the processor), in the order of ascending HPAs of the corresponding bus converters. All busses and bus converters must be initialized, and bus converters tested using ENTRY_TEST, before they can be searched.
  3. Busses at a depth of 2, and so on.
  4. The search is terminated when there are no busses left to try, or at a maximum depth determined by the PDC implementation.

ENGINEERING NOTE

The status of the search so far (for example, the path to the bus currently being searched, the HPA of the last module that was tried) should be maintained as internal state.

If a particular console candidate is unsuitable, subsequent searches return a path for the next console candidate to try. If the search was terminated, a null path is returned. It is suggested that the path be returned in a 32 byte structure with the same format as Console/Display Path.

ARCHITECTURAL NOTE

asks if console search is allowed. There are concerns that a console search compromises system security.

PROGRAMMING NOTE

The following algorithm illustrates the sequence of steps to determine the path to the console module. For a simplex console, this algorithm must be repeated for the console keyboard using the appropriate data structures in Page Zero and Stable Storage.

- If the system provides Console/Display Path in Stable Storage:
  1. Write the appropriate code (C40x {initialize}) to the chassis display.
  2. Read Console/Display Path from Stable Storage and write it into the Console/Display structure in Page Zero.
- If the system provides a set of hardwired paths for the console module:
  1. Write the appropriate code (C60x {initialize}) on the chassis display.
  2. Write the next hardwired path candidate into Console/Display. If all hardwired paths have already been tried, write null values into Console/Display.
- If the system searches for a console:
  1. Search for the next console path to be tried.

2. Write the next console path candidate into Console/Display.

The following algorithm shows the suggested sequence of steps to initialize the console device, once its path has been found.

1. If the first 32 bytes of the Console/Display structure are null, skip to boot device location and initialization (as described in Section 3.5, Boot Device Initialization) and attempt to boot without a console.
2. Write the appropriate code (Cp0x {initialize}, p = 4 when using the primary console; p = 6 otherwise) on the chassis display.
3. For all bus converters (if any) on the path to the console/display module:
    a. Test both ports of the bus converter using the ENTRY_TEST read from the upper port.
    b. Initialize the bus converter.
    c. Allocate address space for the remote bus.
    d. Initialize HPAs and disable bus requestorship on the remote bus.
    e. Reset all modules on the remote bus.
    f. Enable bus requestorship on the remote bus.
4. Initialize the SPA (if any) of the console/display module.
5. Call PDC_IODC to load the console module's ENTRY_INIT code into memory.
6. Write the appropriate code (Cp4x {initialize}, p = 4 when using the primary console; p = 6 otherwise) on the chassis display.
7. Call ENTRY_INIT with ARG1=6 to initialize and test the module. Call ENTRY_INIT with ARG1=5 to initialize and test the device.
8. Call PDC_IODC to load the module's ENTRY_IO code into memory.
9. Complete the last 16 bytes (HPA, SPA, IODC_IO, and CLASS) of the Console/Display structure in Page Zero.
10. If the class of the console device is CL_DUPLEX, the console is completely initialized. If the class of the console device is CL_DISPL, perform the suggested sequence of steps later in this Programming Note to initialize the keyboard device.

If an error occurs during the initialization of a console candidate, try an additional hardwired path if there are any, search for another console module candidate, or place the appropriate null values in the Console/Display structure and attempt to boot without a console.

The following algorithm shows the suggested sequence of steps to initialize a simplex console. The algorithm assumes that the processor provides a Keyboard Path in Stable Storage. It is also possible for the processor to use a hardwired path for the keyboard, or to search for the keyboard.

1. Read Keyboard Path from Stable Storage into the Keyboard structure in Page Zero.
2. For all uninitialized bus converters (if any) on the path to the keyboard module:
    a. Test both ports of the bus converter using the ENTRY_TEST read from the upper port.
    b. Initialize the bus converter.
    c. Allocate address space for the remote bus.
    d. Initialize HPAs and disable bus requestorship on the remote bus.
    e. Reset all modules on the remote bus.

f. Enable bus requestorship on the remote bus.

3. Initialize the SPA (if any) of the keyboard module.
4. Call PDC_IODC to load the keyboard ENTRY_INIT code into memory.
5. If the keyboard module is different from the display module, call ENTRY_INIT with ARG1=6 to initialize and test the module. Call ENTRY_INIT with ARG1=5 to initialize and test the device.
6. If the class of the device is not CL_KEYBD, that is an error.
7. Call PDC_IODC to load the keyboard ENTRY_IO code into memory.
8. Complete the last 16 bytes (HPA, SPA, IODC_IO, and CLASS) of Keyboard structure in Page Zero.

1.5 Boot Device Initialization

The objective of boot device initialization is to locate the boot device, and then to open a module-device connection to the boot device. This connection is used by PDC during the later stages of the boot process, and remains open for use by IPL.

When boot device initialization is complete, the following requirements must have been satisfied:

- The Boot Device structure (at address 0x3D0) in Page Zero is properly initialized.
- If the Boot Device[IODC_IO] word is nonzero, then the boot device ENTRY_IO code is established in memory at the location specified by the IODC_IO word, and a module-device connection to the console device has been opened by ENTRY_INIT (see Section 5.4, IODC Entry Points for a description of ENTRY_INIT and ENTRY_IO).
- All bus converters in the path to the boot module have been initialized, tested using ENTRY_TEST, and allocated address space.
- The SPA of the boot module has been initialized, and does not conflict with the address space of any other module.
- The boot module has bus requestorship enabled.

1.5.1 Determining the Boot Module Path

The path to the boot module is stored in the first 32 bytes of the Boot Device structure, and must be determined from:

- The Primary Boot Path structure in Stable Storage (at address 0x00).
- The optional Alternate Boot Path structure in Stable Storage (at address 0x80).
- A manual boot path entered through the console.
- The result of a search for a boot device.

The mechanism used to determine the boot device is specified by the *as* and *ab* bits of the *flags* byte of the Primary Boot Path, as shown in the following table:

ab	*as*	Name	Mechanism
0	0	Manual Boot	Console interaction with operator is used to obtain path to boot device.
0	1	Boot Search	Search for boot device candidate using search options of ENTRY_INIT. If candidate does not contains valid IPL code, search for next candidate. If no candidate can be found, or if no candidate contains a valid IPL image, execute the manual boot selection mechanism.
1	0	Auto Boot	Try to boot from the boot device specified in the Primary Boot Path. If that device does not exist, or does not contain valid IPL code then execute the manual search selection mechanism.
1	1	Auto Search	Try to boot from the boot device specified in the Primary Boot Path. If that device does not exist, or does not contain a valid IPL image, then execute the boot search selection mechanism.

1.5.2 Searching for a Boot Device

Searching for the boot device can be complicated if there are many levels of bus converters in the system, and so the following requirements must be satisfied:

- When a bus is being searched, all modules must be checked in order from lowest HPA to highest HPA.
- Busses must be searched breadth first, i.e., in the following order:

1. The processor's local bus.
2. Busses at a depth of 1 (there is only 1 bus converter between the bus and the processor), in the order of ascending HPAs of the corresponding bus converters. All busses must be initialized before they can be searched.
3. Busses at a depth of 2, and so on.
4. The search is terminated when there are no busses left to try, or at a maximum depth determined by the PDC implementation.

ENGINEERING NOTE

The status of the search so far (for example, the path to the bus currently being searched, the HPA of the last module that was tried) should be maintained as internal state.

If a particular boot device candidate is unsuitable, a subsequent search returns a path for the next boot device candidate to try. If the search was terminated, then a null path is returned. It is suggested that the path be returned in a 32 byte structure with the same format as Primary Boot Path.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to determine the path to the boot device.

1. Read the Primary Boot Path from Stable Storage and copy it into the Boot Device structure in Page Zero. If the read returned an error, then put system-specific default values into the Boot Device structure.
2. If the console is not initialized and autoboot is enabled, then initialize the boot device, as described later in this Programming Note. The console is not initialized if Console/Display[IODC_IO] = 0. Autoboot is enabled if Boot Device[flags[ab]] = 1.
3. If the console is not initialized and autoboot is disabled, then autosearch is the only possibility. If autosearch is enabled and the system supports autosearch, then search for the boot device, else write the appropriate code (??88 {fault}) to the chassis display, and halt.
4. Write the '*PDC revision*' message on the console. See Section 3.7, Sample Text for Console Messages for the suggested text of each console message.
5. Write the '*paths*' messages on the console.
6. If autoboot is enabled, then write the '*autoboot enabled*' message on the console.

   If no key is pressed within 10 seconds, then write the '*time expired*' message on the console, and initialize the boot device.

   If any key is pressed within the 10 seconds, continue with step 7.

   If autoboot is disabled, but autosearch is enabled, then search for the boot device, as described later in this Programming Note.
7. Write the '*primary path*' message on the console.
8. Read a string from the console; echo it back.

   If the string started with "y" or "Y", then initialize the boot device.

   If the string started with "n" or "N", then continue with step 9.

   If any other string was entered, then write the '*illegal response*' message on the console and return to step 7.
9. Write the '*alternate path*' message on the console.

10. Read a string from the console; echo it back.

    If the string started with "y" or "Y", then read the Alternate Boot Path from Stable Storage into Boot Device in memory and initialize the boot device.

    If the string started with "n" or "N", then continue with step 11.

    If any other string was entered, then write the '*illegal response*' message on the console and return to step 9.

11. Write the '*enter path*' message on the console.

12. Read a string from the console; echo it back.

    If the string contains a valid boot path, then complete the appropriate fields of Boot Device and initialize the boot device. The format is as specified by {path} in Section 3.7, Sample Text for Console Messages (omitted leading BC fields are taken to be null; omitted trailing layer fields default to zero).

    If the string contains a command recognized by PDC then execute it and return to step 11 (if possible).

    If the string started with "?", then write the '*menu*' message on the console, write the '*paths*' on the console, and return to step 11.

    If the string contains anything else, then write the '*illegal response*' message on the console, and return to step 11.

The following two figures illustrate this sequence, with and without a console present:

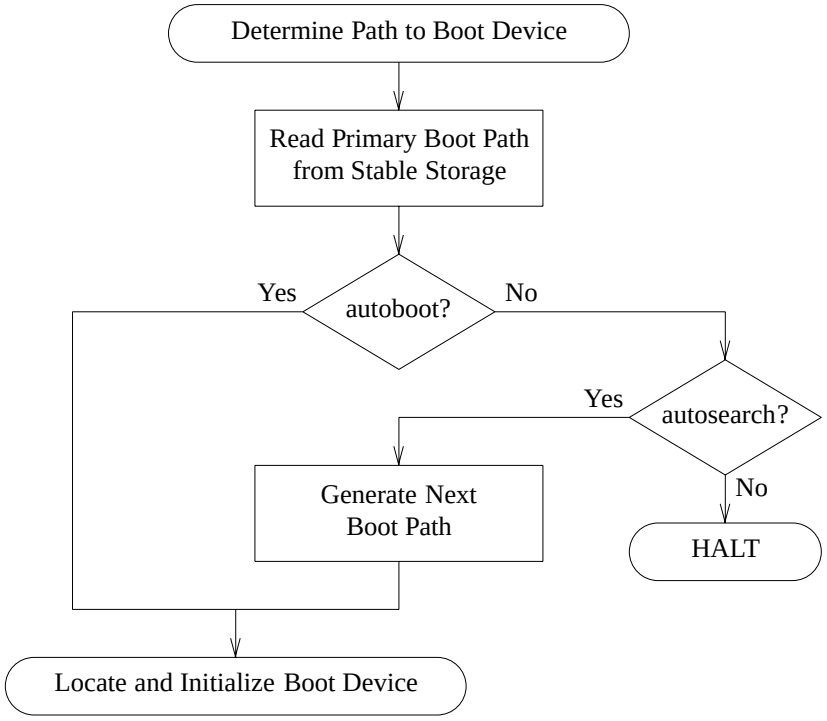


Figure 1-12. Determine Path to Boot Device Algorithm (No Console Present)

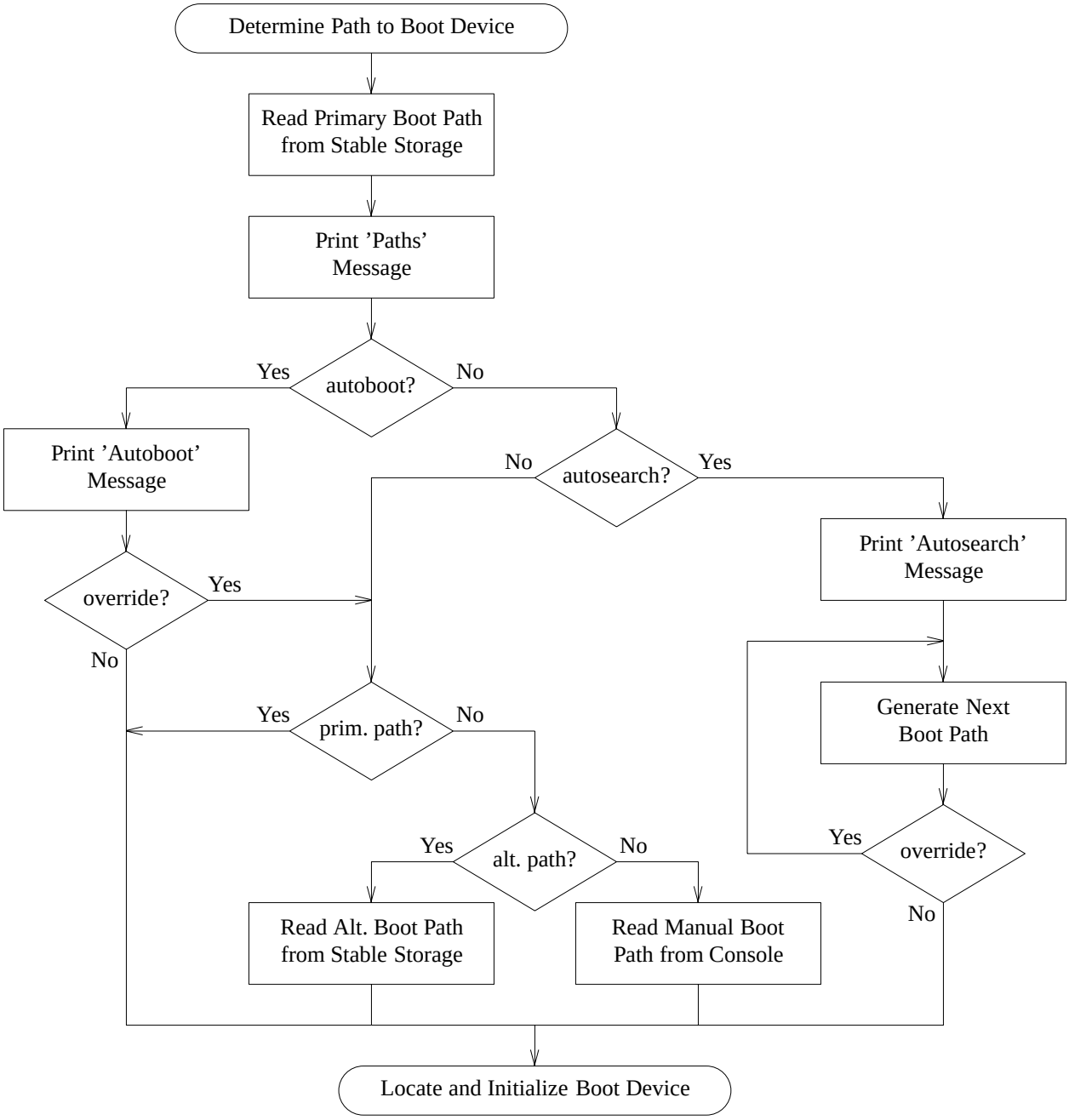


Figure 1-13. Determine Path to Boot Device Algorithm (Console Initialized)

The following algorithm shows the suggested sequence of steps to search for a boot device.

1. Perform this step only the first time this code is entered, and only if the console is initialized:

   Write the '*autosearch enabled*' message on the console.

   Read input from the console to determine if the operator wants to have approval over each boot path before it is tried.

2. The next boot path to be tried is generated and written into Boot Device. If there are no more modules left to be tried, then write the appropriate code (???? {fault}) to the chassis display, and

halt.

3. If the operator wants to approve each boot path, write the path on the console and ask for approval.

   Read input from the console to determine if the path is approved. If not approved, return to step 2.

4. Continue by initializing the boot device, as described below:

The following algorithm shows the suggested sequence of steps to initialize the boot device:

1. Write the appropriate code (Cp0x {initialize}, p = 5 when using the primary boot path; p = 7 otherwise) to the chassis display.
2. If the console is initialized, then write the '*booting*' message on the console.
3. For all uninitialized bus converters (if any) on the path to the boot module:
   a. Test both bus converter ports using the ENTRY_TEST code read from the upper port.
   b. Initialize the bus converter.
   c. Allocate address space for the remote bus.
   d. Initialize HPAs and disable bus requestorship on the remote bus.
   e. Reset all modules on the remote bus.
   f. Enable bus requestorship on the remote bus.
4. Initialize the SPA of the boot module.
5. Call PDC_IODC to load the boot module's ENTRY_INIT code into memory.
6. Write the appropriate code (Cp4x {initialize}, p = 5 when using the primary boot path; p = 7 otherwise) to the chassis display.
7. If the boot module is different from the console module, call ENTRY_INIT with ARG1=6 to initialize and test the module. Call ENTRY_INIT with ARG1=5 to initialize and test the device.
8. Call PDC_IODC to load the boot device ENTRY_IO code into memory.
9. Complete the last 16 bytes of Boot Device (HPA, SPA, IODC_IO, and CLASS).
10. If the console is initialized, then write the '*iodc revision*' message on the console.

If an error occurs during autosearch, then search for another boot device candidate so that the next path can be tried. If an error occurs while attempting an autoboot from the primary path, then it is permissible to try to boot from a system-dependent hardwired boot path. In other error cases, ask the operator for a manual boot path. If no recovery is done, the system can simply halt.

1.6 Initial Program Load (IPL)

The objective of the boot process is to load and launch the IPL code. By definition, IPL is whatever code PDC first loads in through the boot device. The objective of IPL is to launch ISL (which in turn is able to launch an operating system or a diagnostic program). While it is possible for PDC to launch ISL directly, there can be advantages to completing the intermediate step.

One advantage relates to preserving the contents of memory before it is overwritten by the text of the ISL program. The memory that would be overwritten by ISL can be saved in this way: The IPL code is a dump routine that dumps the first part of memory to disc. Then IPL continues by loading and launching ISL. This is precisely the strategy that is used in MPE-iX systems. By contrast, in HP-UX systems the IPL code is actually ISL itself.

When the IPL code is invoked, the boot device and the intermediate busses are configured. The console and the intermediate busses may be configured, depending on the boot mechanism used. No other devices or busses are configured. The initial configuration can be used unchanged by IPL. However, if the configuration is changed by IPL, the addresses of all bus converters and all soft physical addresses must be reinitialized, to avoid potential conflicts with the previous assignments.

The IPL code is loaded into memory from the boot device using ENTRY_IO. ENTRY_IO is intended to be compact rather than efficient, so the IPL code may contain a more complex driver to increase the efficiency of subsequent accesses to the boot device.

Some boot devices may have random access capabilities. PDC may use ENTRY_IO to access such a device in a strictly sequential fashion.

PROGRAMMING NOTE

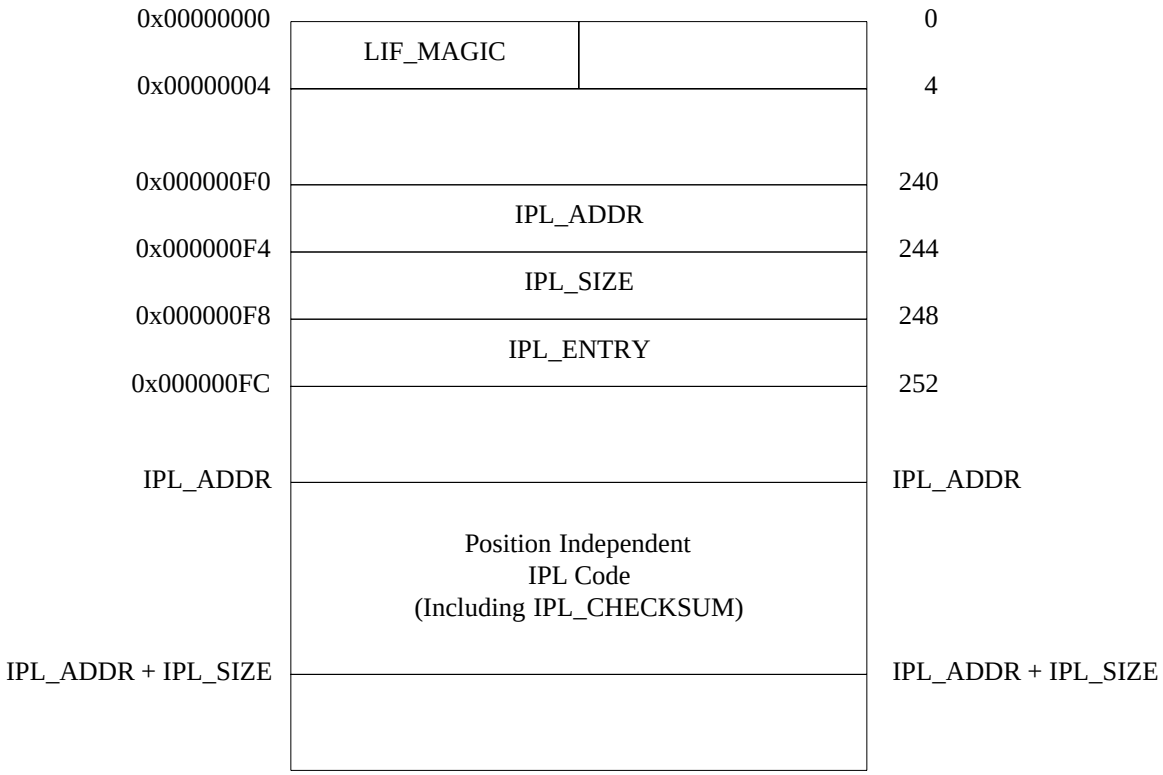
While PDC may access random access devices in a sequential fashion, PDC implementations are encouraged to access random access devices in a random fashion for optimal performance and reliability.

If IPL is aware of the random access capabilities of the boot device, it may access the device in a random fashion.

IPL must verify that *fast-size* = all before launching any operating system. If *fast-size* is not all, IPL can launch other utilities, but must not launch any operating system. IPL can distinguish operating systems from other utilities by the file type which is in the file header.

1.6.1 Data Format of IPL (LIF)

The data format of an IPL image is specified in a fashion compatible with the existing LIF (Logical Interchange Format) standard. Note that this is a 32-bit format, and has not been changed for PA-RISC 2.0. This format is shown below:



All addresses are byte offsets from the start of the boot device medium. The unlabelled areas are not used by PDC.

This format is interpreted as follows. The LIF_MAGIC halfword is read and compared to the magic number for LIF images, 0x8000. If this match succeeds, the three words IPL_ADDR, IPL_SIZE, and IPL_ENTRY are read and checked as follows:

Word	Checks
IPL_ADDR	2 Kbyte aligned, nonzero.
IPL_SIZE	Multiple of 2 Kbytes, nonzero, less than or equal to 256 Kbytes.
IPL_ENTRY	Word aligned, less than IPL_SIZE.

1.6.2 Loading IPL

If all the IPL checks are satisfied, the IPL code from address IPL_ADDR to IPL_ADDR + IPL_SIZE is transferred to the page-aligned memory location IPL_START.

The arithmetic sum of the words in the IPL code is computed. If this sum is zero, the LIF file format has been verified, and the IPL code is executed starting at IPL_START + IPL_ENTRY. IPL_ENTRY contains the offset from IPL_ADDR of the first executable instruction in the IPL code.

When the IPL image is read from a sequential access device, the record size must be a multiple of 2 Kbytes and not greater than 64 Kbytes. Also, IPL_ADDR must be the address of the first byte of a record.

PROGRAMMING NOTE

The following algorithm shows the suggested sequence of steps to load and launch the IPL code.

1. Write the appropriate code (Cp8x {initialize}, p = 5 for booting from primary path; p = 7 otherwise) to the chassis display.
2. Call boot device ENTRY_IO to read the LIF volume label (at least the first 256 bytes).

3. If LIF_MAGIC is not 0x8000, or IPL_ADDR is not page aligned, or IPL_ADDR is zero, or IPL_SIZE is not a multiple of 2K, or IPL_SIZE is zero, or IPL_SIZE is greater than 256K, or IPL_ENTRY is not word aligned, or IPL_ENTRY is not less than IPL_SIZE, then write the appropriate code (CpF0 {fault}, p = 5 for booting from primary path; p = 7 otherwise) to the chassis display, and recover by searching for another boot device candidate.
4. Call boot device ENTRY_IO to load the IPL code into memory at IPL_START. (If the class of the boot device is CL_RANDOM, then it it not necessary to sequentially read the bytes on the boot device medium that precede IPL_ADDR.)
5. Compute the IPL checksum (the arithmetic sum of the words in the IPL code). If the IPL checksum is nonzero, then write the appropriate code (CpF8 {fault}, p = 5 for booting from primary path; p = 7 otherwise) to the chassis display, and recover by searching for another boot device candidate.
6. Write the appropriate code (CpFF {initialize}, p = 5 for booting from primary path; p = 7 otherwise) to the chassis display.
7. If the boot was automatic (autoboot or autosearch), clear the interact flag so that IPL will be automatic.

   If the operator has entered any data through the console, then write the '*interact*' message on the console. Read the input from the console and adjust the interact flag accordingly.
8. If the console is initialized, then write the '*booted*' message on the console.
9. Set GR26, the interact flag to 1 if IPL is interactive, and to 0 otherwise.
10. Set GR25 to the address of the first doubleword past the end of the IPL code.
11. Branch to the IPL code.

If an error occurs, then determine the path to another boot device candidate (as described in Section 3.5, Boot Device Initialization), or alternatively, halt.

1.7 Sample Text for Console Messages

This section describes some suggested text for each of the console messages used in the programming notes earlier this chapter.

PDC revision	"<CR><LF><LF>Processor Dependent Code (PDC) revision x<CR><LF>"
paths	"<CR><LF>Console path = {path} Primary boot path = {path} Alternate boot path = {path}<CR><LF>" Where {path} above has the format: "BC0/BC1/BC2/BC3/BC4/BC5/MOD.L1.L2.L3.L4.L5.L6<CR><LF>" If leading BC fields are null, they can be omitted. The alternate boot path is omitted if not implemented.
autoboot enabled	"<bell>Autoboot from primary boot path enabled.<CR><LF> To override, press any key within 10 seconds.<CR><LF><LF>"
time expired	"10 seconds expired.<CR><LF> Proceeding with autoboot.<CR><LF>"
primary path	"Boot from primary boot path (Y or N)?>"
alternate path	"Boot from alternate boot path (Y or N)?>"
enter path	"Enter boot path, command, or ?>"
illegal response	"<bell>Illegal response.<CR><LF>"
menu	"<LF>Use 1.2.3 format for boot path,<CR><LF><LF> The following commands are available:<CR><LF>"
booting	"<LF>Booting.<CR><LF>"
iodc revision	"<LF>Console IO Dependent Code (IODC) revision x<CR><LF> Boot IO Dependent Code (IODC) revision x<CR><LF>"
interact	"Interact with IPL (Y or N)?>"
booted	"<LF>Booted.<CR><LF>"
device status	"<CR><LF>Boot device status<CR><LF> {status}" Where {status} is RET[16] through RET[31] in hexadecimal, the 16 words separated by <CR><LF>.
autosearch enabled	"<bell>Autosearch for boot device enabled.<CR><LF>"

This page intentionally left blank

TABLE OF CONTENTS

LIST OF FIGURES