PA-RISC 2.0 Firmware Architecture Reference Specification

Version 1.1E

Printed in U.S.A. July 22, 2004

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.

2. Firmware Objectives Overview

The purpose of the PA-RISC Firmware Architecture is to specify the architectural requirements for writing PDC and IODC which will allow generic software to run across a wide range of processors, some of which may be implemented in different technologies.

To accomplish this purpose, the PA-RISC Architecture specifies the functions required to accomplish the following goals:

- Initialize the hardware and boot the machine from the point a reset is received or a power on sequence is initiated until generic software is given control (PDC Entry Points).
- Respond to a hardware triggered event, such as a reset, machine check, or the detection of an impending power failure (PDC Entry Points).
- Provide processor specific services for generic software (PDC Procedures).
- Provide services to allow the identification, configuration, and initialization of I/O modules (IODC Data and IODC Entry Points).
- Allow the OS to be handed control from firmware to do further processing related to a hardware triggered event (OS Interfaces).

These specifications are contained in Part II of this manual (Chapter 3 through Chapter 6).

The PA-RISC Firmware Architecture employs many concepts from the PA-RISC Processor Architecture and PA-RISC I/O Architecture as a base for defining its specifications. The remainder of this chapter defines the concepts and specifies the Processor and I/O Architecture components necessary to specify the firmware.

The functionality described for the PA-RISC I/O Architecture assumes a specific model for the mapping of the I/O address space, and the addressing of I/O modules in that space. In the specific case where a PA-RISC processor is connected to a foreign bus through a bus adapter, the algorithm for locating I/O modules may change from the PA-RISC algorithm. However, the IODC functionality to identify, configure, and initialize those modules must still be provided. Note that the specifications in the following sections assume a fully native PA-RISC model.

2.1 Module Addressing

Module addressing refers to the address spaces used by various modules and the general characteristics of each address space. It should be noted that module addressing deals solely with physical addresses, not virtual addresses. As such, "physical" is implied whenever "address" is used by itself.

The system address space is logically subdivided into **pages**, each 4 Kbytes in size and aligned on a 4-Kbyte boundary. When these pages are associated with modules in the system, they are grouped into **address spaces** for reference purposes. There are three possible address spaces which may be recognized by a given module: the hard, extended, and broadcast physical address spaces. Every module recognizes a unique hard physical address space and the broadcast physical address space. There are two kinds of extended address spaces; bus converter ports have I/O range spaces while other module types may have a soft physical address space. These three address spaces are described in the following sections.

In addition to pages and address spaces, the architecture also divides the I/O and broadcast address spaces into two other units termed **I/O registers** (or **registers** for short) and **register sets**. The I/O address space is byte addressable; however, the architected unit of addressability in the I/O space is a word-sized, word-aligned quantity called a register. Every register set consists of 16 registers (numbered 0 - 15), and is aligned on a 16-word boundary. There are 64 register sets in each page (numbered 0 - 63).

2.1.1 Hard Physical Address (HPA) Space

Every native module responds to (i.e., slave acknowledges) a 4-Kbyte address range in the I/O address space called its **hard physical address (HPA) space**. The HPA space must be implemented and privileged for all native modules. The HPA space allows access to functionality which is architecturally required of all native modules.

HPA space need not be implemented for fixed address modules and for PCI devices. For fixed address modules, the HPA address must still be a valid address in the I/O space of the platform, however no registers need be implemented in the space, and the HPA address can simply be used as a handle to obtain IODC information using PDC_IODC.

For PCI modules, not only must the IODC information be obtained by calling PDC_IODC, but the the HPA is not an actual (64-bit) I/O address, but is a 32-bit PCI Function address (PFA). Bit 0 of the PFA must always be zero. For the format of a PFA see the *Local PCI Bus Interface Specification, Version 2.1*.

The register set layout of the HPA is shown in Figure 2-1.

The first register set in the HPA of every module is the **Supervisor Register Set** (SRS). Registers in the SRS are used to configure and control the module as a whole. For example, the registers used to identify the module and to reset the entire module are located in the SRS.

The second register set in the HPA of every module is the **Auxiliary Register Set** (ARS). The ARS contains additional module-type dependent registers related to the operation of the module as a whole. For example, the registers which control the size and location of the extended address space of a bus converter port are located in the ARS as are the extended error logging registers for memory modules.

Register sets labeled HVRS in the figure are **HVERSION-Dependent Register Sets**. No architected function can be implemented in an HVRS. These register sets contain only HVERSION-dependent registers.

Register sets labeled BSRS in the figure are **Bus Specification-Dependent Register Sets**. These register sets are defined by the bus specification on which the module resides and are typically used to implement a common functionality among all modules on a given bus. The bus specification may optionally delegate their definition to the module SVERSION or HVERSION.

Register sets labeled TRS in the figure are **Type-Dependent Register Sets**. There are two kinds of TRS:

- **SVERSION-Dependent Register Set** (SVRS)

  An SVRS contains only SVERSION-dependent registers. No architected function can be implemented in an SVRS.

- **HVERSION-Dependent Register Set** (HVRS)

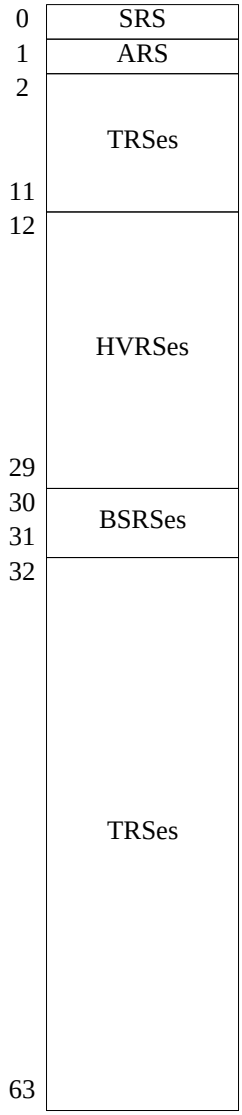
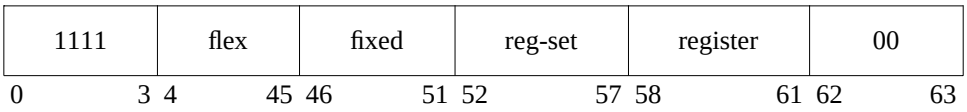


Figure 2-1. HPA Space Layout

See Section 2.2, I/O Register Properties, for a discussion of register properties, including SVERSION and HVERSION dependent.

During configuration, each native bus is allocated 256 Kbytes of HPA space in the I/O address space. This 256 Kbytes of address space is subdivided into 64 4-Kbyte units which have a one-to-one correspondence with the 64 possible modules that can be connected to each native bus. Thus, every module has one and only one of these 4-Kbyte units allocated for its HPA. The HPA space of a native bus must be 256-Kbyte aligned; the HPA space of a module must be 4-Kbyte aligned.

Hard physical addresses to architected I/O registers are of the form:



1111	address is an I/O address
flex	bus to which the module is connected
fixed	module's location on the bus
reg-set	register set offset within the HPA
register	register offset within the register set
00	all architected HPA registers are word aligned

Figure 2-2. Hard Physical Addresses

For the addresses in a module's HPA space, the *flex* field specifies the native bus to which the module is connected. All modules on any given bus have the same *flex* value. That is, the *flex* field identifies the 256-Kbyte HPA space associated with each bus. Each native bus has a unique *flex* value assigned to it by software during configuration.

On some native busses, the *flex* field address comparison hardware may be shared by multiple modules. Before indicating a valid flex comparison, this shared hardware must also check that the address is in the I/O space (i.e., the upper four bits are ones) if the modules sharing the comparison logic do not make this check. Similarly, this shared hardware must also ensure that the address precedence rules specified in Section 2.1.4, Address Space Conflicts, are obeyed (e.g., a valid flex comparison must never be asserted for any broadcast transaction even if the flex address of the bus overlaps the broadcast address space). See the appropriate bus specification for further details.

Note that due to other architectural restrictions (PDC and broadcast address spaces), the *flex* field does not have a full range of values. In particular, the range of I/O addresses available for use by HPA spaces is 0xF1000000 00000000 through 0xFFFFFFFF FFFBFFFF The *flex* value of the central bus is always 0x3FFFFFFFFFFE (42 bits).

For the addresses in a module's HPA space, the *fixed* field distinguishes each module from other modules on the same bus (i.e., with the same *flex* value). The *fixed* field of an HPA must not vary across power failures or boots, and must be independent of configuration changes. For each bus transaction to the I/O address space, a module must compare the *fixed* values for each slave address where the *flex* values match.

The *reg-set* and *register* fields indicate the desired register set and register, respectively.

2.1.2 Extended Address Space

A module's **extended address space** allows the module to extend the range of addresses to which it responds beyond the range provided by its HPA space. The kind of extended address space available to a module is dependent on the module's type and is illustrated in the table below.

TABLE 2-1. Extended Address Space Availability

Module Type(s)	Extended Address Space
TP_B_DMA, TP_A_DMA, TP_A_DIRECT, TP_CIO, TP_MEMORY	soft physical address space
TP_BCPORT	I/O and memory range spaces
TP_NPROC	PDC address space
TP_CONSOLE	none

2.1.2.1 Soft Physical Address (SPA) Space

If an I/O , memory, or CIO adapter module requires a larger address range than that provided by its HPA space, the module needs some **soft physical address (SPA) space**. All the information needed by generic software to allocate SPA space to a module is contained in the module's IODC (see Chapter 5, IODC, for details on the IODC_SPA byte and the ENTRY_SPA entry point).

2.1.2.1.1 Memory SPAs

The SPA space of a memory module or processor-dependent interleave group must be 2^n Kbytes (n > 1) in size, and it must be aligned on a boundary which is a multiple of its size. The SPA space must reside in the memory address space.

The pages in the memory SPA space of a memory module or processor-dependent interleave group can be either implemented or unimplemented. The implemented pages must be contiguous, greater than one-half of the value of *max_spa*, and start at the base address of the SPA space.

When the SPA space of a module resides in the memory address space, all addresses to the module's SPA space are of the form:

memory-page physical memory page

offset offset within the page

Figure 2-3. Memory Soft Physical Addresses

Note that due to other architectural restrictions (e.g., the PDC space and the size of the system memory space), the *memory-page* field does not have a full range of values. In particular, the range of system memory addresses available for use by SPA spaces is 0x00000000 00000000 through 0xEFFFFFFF FFFFFFFF.

2.1.2.1.2 I/O SPAs

An I/O module (Type-B DMA, Type-A DMA, or Type-A Direct) may have up to two SPA spaces (as specified by its SVERSION). Each of the SPA spaces must be a multiple of 4K bytes in size and must reside in the I/O address space. All pages in the SPA space(s) must be implemented.

The base address, bound address, and alignment of the SPA spaces allocated to a module is SVERSION dependent.

The register set layout of an I/O SPA space is shown below:

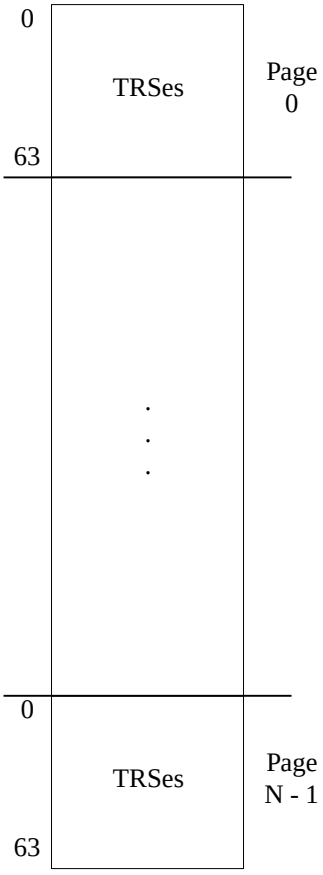
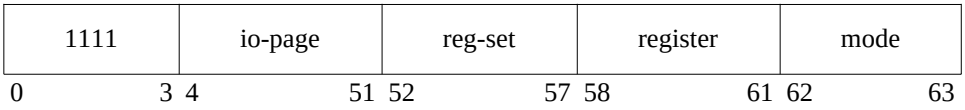


Figure 2-4. I/O SPA Space Layout

All register sets in an I/O SPA space are TRSes (see Section 2.1.1, Hard Physical Address (HPA) Space, for a description of the different kinds of TRSes).

I/O soft physical addresses are of the form:



1111	address is an I/O address
io-page	page in the I/O address space
reg-set	register set offset within the page
register	register offset within the register set
mode	addresses are aligned as per the bus operation's mode

Figure 2-5. I/O Soft Physical Addresses

Note that due to other architectural restrictions (e.g., the PDC and broadcast spaces and the HPA space of the central bus) the *io-page* field does not have a full range of values. In particular, the range of system I/O addresses available for use by SPA spaces is 0xF1000000 00000000 through 0xFFFFFFFF FFF7FFFF

The *reg-set* and *register* fields indicate the desired register set and register, respectively.

The *mode* field aligns the address as appropriate for the type of bus operation being used. All architected registers must be accessed as a word (i.e., *mode* = 0b00).

2.1.2.2 I/O Range Address Spaces

Bus converters have the I/O address spaces used by all the modules "behind" them as their extended address space, and is called the **I/O range address space**.

2.1.2.3 PDC Address Space

Native processors have the PDC address space as their extended address space.

Other modules must not prevent themselves from issuing operations to the PDC address space.

2.1.3 Broadcast Physical Address (BPA) Space

The I/O addresses in the range 0xFFFFFFFF FFFC0000 through 0xFFFFFFFF FFFFFFFF are called the **broadcast physical address (BPA) space**. All pages in the BPA space must be implemented and privileged.

The register set layout of the BPA is shown below:

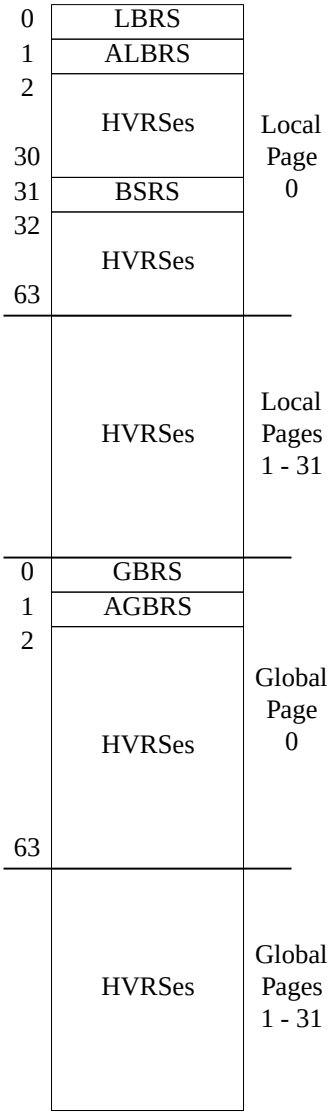


Figure 2-6. BPA Space Layout

The first register set in the local half of the BPA is the **Local Broadcast Register Set** (LBRS). Issuing a WRITE operation to this register set affects every module on the local bus. One important use of this register set is to set the flex address of each bus during configuration, using the IO_FLEX register in the LBRS. The second register set

in the local half of the BPA is the **Auxiliary Local Broadcast Register Set** (ALBRS).

The first register set in the global half of the BPA is the **Global Broadcast Register Set** (GBRS). Issuing a WRITE operation to this register set affects every module on all busses (assuming the bus converters are appropriately configured). For example, a reset command (CMD_RESET) written to the IO_COMMAND register in the GBRS is used to reset all processors in the entire system, causing the system to reboot. The second register set in the global half of the BPA is the **Auxiliary Global Broadcast Register Set** (AGBRS).

Register sets labeled HVRS in the figure are HVERSION-dependent register sets. These register sets contain only HVERSION-dependent registers (see Section 2.2, I/O Register Properties, for a discussion of register properties, including HVERSION dependent).

All addresses in the BPA space are of the form:

3FFFFFFFFFFF	address is a broadcast address
g	global (1) or local (0) broadcast space
broad-page	page in the selected broadcast space
reg-set	register set offset within the page
register	register offset within the register set
00	all BPA space registers are word aligned

Figure 2-7. Broadcast Physical Addresses

The *g* bit is used to distinguish between local and global broadcasts in the BPA space. The *broad-page*, *reg-set*, and *register* fields indicate the desired page, register set, and register, respectively, being addressed in the appropriate half of the BPA space.

The actions taken by a module with respect to the BPA space are module-type dependent.

2.1.4 Address Space Conflicts

As previously mentioned, a module may recognize as many as three different address spaces:

1. its hard physical address (HPA) space
2. its extended address space
3. the broadcast physical address (BPA) space

For each transaction, a module determines if the slave address matches the broadcast physical address (BPA) space, its hard physical address (HPA) space, or its extended address space.

In a correctly configured system, at most one of these address spaces is matched for any given transaction; if more than one matches, it is potentially due to an error in the system's configuration or because the system is being reconfigured. If overlap does occur, a broadcast flex transaction will correct any overlap of the HPA space of a module with the broadcast space. A directed CMD_RESET will correct any extended space overlap by disabling the module's extended address space, except a native processor's.

2.1.4.1 Address Decode Requirements

In order to correct any address space overlap, the following transactions must not be ignored by a module:

1. WRITE to the LBRS IO_FLEX register
2. directed CMD_RESET to the IO_COMMAND register in the SRS

In addition to ensuring that the above transactions are not ignored by a module, the integrity of these transactions must be protected from being corrupted by modules which do not normally recognize these transactions. In

particular, broadcast transactions are not required to be recognized by Type-A Direct modules on busses with a centralized "flex" decode (as described in Section 2.1.1, Hard Physical Address (HPA) Space) since only the local broadcast transaction to the IO_FLEX register would normally be required and the centralized decode mechanism relieves these modules of this responsibility.

In order to protect broadcast transactions from being corrupted, all modules with an SPA space must decode the broadcast space.

All modules on a bus which does not do centralized "flex" decoding must decode the broadcast space. On busses with the "flex" decode centralized (e.g., HP-PB), any module which does the decode must NOT assert "HPA valid" for broadcast transactions even if its IO_FLEX register is incorrectly configured to overlap the broadcast space.

If a module is configured such that its HPA or SPA space overlaps the PDC address space, the subsequent behavior of that module is HVERSION dependent, and, as a consequence, the behavior of the entire system may be affected.

2.1.4.2 Address Decode Model

In order to ensure that a system can be properly configured, the address spaces recognized by a module are prioritized to handle any case of overlap. The BPA space takes precedence over the HPA space which takes precedence over the extended space. For I/O modules, software must prevent SPA address overlaps among the SPA spaces for a given module; otherwise, the operation of the module's SPA spaces is HVERSION dependent.

SUPPORT NOTE

For diagnostic purposes, implementations detecting accesses to overlapping

---

The address decode precedence is required to guarantee that a system can be reset and new addresses correctly assigned, independent of incorrect addresses previously set by transient hardware or software errors. The address decode precedence provides this guarantee, when the following initialization sequence is assumed:

1. A broadcast IO_FLEX transaction is issued to every bus in the system to relocate the HPA spaces of all modules on each bus.
2. A CMD_RESET is written to the IO_COMMAND register in the SRS of every module on each bus to disable every module's extended address space.

The address precedence must be maintained even on modules which do not implement broadcast I/O registers. For instance, every module which has an extended address space must decode broadcast transactions, even if they do not implement broadcast transactions, in order to ensure that the precedence rules are followed. Similarly for HPA space decoding, any bus (like HP-PB) which allows for bus level address decode (e.g., HPA_VALID_L) must also ensure the address precedence rules are obeyed (e.g., HPA_VALID_L must never be asserted for any broadcast transaction even if the "flex" address of the bus overlaps the broadcast address space).

The model below shows the expected precedence behavior (but in no way implies any particular implementation):

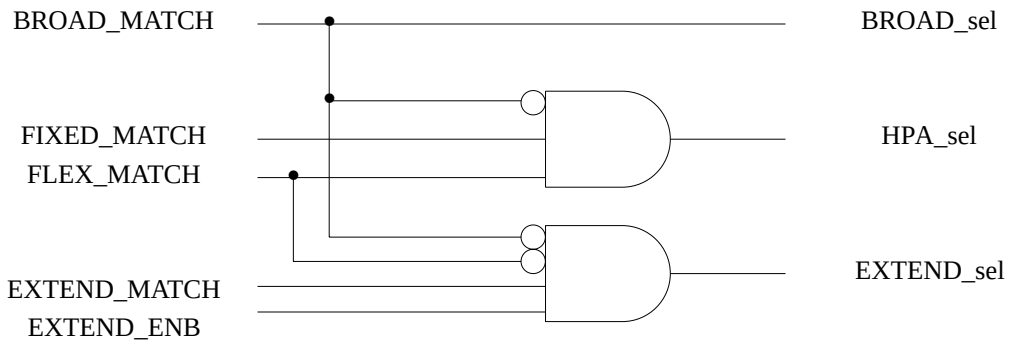


Figure 2-8. Address Decode Model

BROAD_MATCH: True when the transaction address is in the broadcast space (PATH_SADD{0..45} == 0x3FFFFFFFFFF). This line need not be implemented if the module does not recognize any broadcast transactions, does not have an extended address space, and the bus does a centralized FLEX_MATCH for the module's HPA space.

FIXED_MATCH: True when the transaction address bits 14 through 19 equal the module's fixed field (PATH_SADD{14..19} == *fixed*).

FLEX_MATCH: True when the transaction address matches the FLEX value of the local bus. There are exactly three choices:

- PATH_SADD{0..45} == IO_FLEX{0..45}
- PATH_SADD{0..3} == 0xF and PATH_SADD{4..45} == IO_FLEX{4..45}
- PATH_SADD{0..3} == IO_FLEX{0..3} == 0xF and PATH_SADD{4..45} == IO_FLEX{4..45}

ENGINEERING NOTE

For HP-PB transactions, the HPA_VALID_L line denotes (FLEX_MATCH) and (not BROAD_MATCH).

EXTEND_MATCH: True when the transaction address matches the extended address space of the module. This indicator is only implemented by modules which have an extended address space.

For memory modules, EXTEND_MATCH is true if either of the following is true:

- PATH_SADD{0..N} == IO_SPA{0..N}
- PATH_SADD{0..N} == IO_SPA{0..N} and IO_SPA{0..3} != 0xF

where N equals 31 - IODC_SPA[shift].

For I/O modules which have an SPA space, EXTEND_MATCH is dependent on the base and bounds of each of the module's SPA spaces.

For bus converter ports, EXTEND_MATCH is based on the mode of the port, whether the port is an upper or lower port, and the values in the I/O range registers (IO_IO_LOW and IO_IO_HIGH).

EXTEND_ENB: True when the extended address space of the module is enabled. This indicator is only implemented by modules which have an extended address space.

For I/O and memory modules, EXTEND_ENB is true if their SPA space is enabled.

For bus converter ports, EXTEND_ENB is true if the bus converter port mode is not OFF (IO_CONTROL[mode] != OFF).

BROAD_sel: The module's BPA space has been selected. This line is not implemented by modules which do not recognize the BPA space.

HPA_sel: The module's HPA space has been selected.

EXTEND_sel: The module's extended space has been selected. This line is only implemented on modules which have an extended address space.

2.2 I/O Register Properties

The unit of addressability in the I/O and broadcast address spaces is the **I/O register**. Each I/O register is 32 bits wide and is word aligned. All architected registers must be accessed as a word. Non-architected (e.g., SVERSION dependent) registers may also be accessed on byte or halfword boundaries as allowed by the module's SVERSION/HVERSION.

For architecturally defined registers, the values obtained by reading from the I/O address and the effects of writing to the address are architected; however, the physical implementation of any I/O register (a microprocessor control port, MSI register, network, or memory) is not specified by the architecture.

2.2.1 Register Accessibility and Validity

A register is defined to be **accessible** if an attempt to read or write it does not generate an addressing error. A register is defined to be **valid** if, on a read, the data returned has its defined meaning, and on a write, the value written causes its defined effect.

In ST_OFF, the accessibility of any module state is HVERSION dependent and bus mastership is disabled. If BUS_POW_VALID is asserted, and the bus init time has elapsed, the accessibility of the module state is as follows:

- The BPA space must be accessible.
- The HPA space must be accessible after the first WRITE transaction to the IO_FLEX register.
- The SPA space must be accessible when it is enabled, and must be inaccessible when disabled.
- The PDC address space must be accessible.

The validity of a register is primarily determined by the module's operating state and is listed as one of the following:

A WRITEs have the architected effect; READs return the architected data.

SV WRITEs have an SVERSION-dependent effect (within the privilege of the page); READs return SVERSION-dependent data.

HV WRITEs have an HVERSION-dependent effect (within the privilege of the page); READs return HVERSION-dependent data.

\- Register is inaccessible.

2.2.2 Register Access Modes

One set of characteristics associated with each I/O register is the ability of software to read or write the register. The register access modes defined by the architecture are read (R) and write (W).

A register is **non-readable** if all READ operations issued to the register return HVERSION-dependent data; otherwise, the register is **readable**. A register is **non-writable** if all WRITE operations issued to the register have HVERSION-dependent effects; otherwise, the register is **writable**.

Even though the effect of a WRITE operation issued to a non-writable register is HVERSION dependent, the actual effects must adhere to the attributes of the page in which the register resides.

Only software which is HVERSION dependent (e.g., diagnostic software) should read a non-readable register or write to a non-writable register.

SUPPORT NOTE

While the architecture allows registers to be specified as non-readable or non-writable, a module may allow the actual storage locations to be read or written. Making all registers in a module readable and writable, even if the data involved is HVERSION dependent, improves the testability of the design.

2.2.3 Register Fields

The bits in a register are grouped by the architecture into **fields**. A field allows bits which are functionally indistinguishable by the architecture to be treated as a single unit. There are five main classifications of register fields: architected, constant, reserved, SVERSION dependent, and HVERSION dependent. In addition, there are classifications that are composites of the base types (e.g., module type dependent) which are used as shorthand notation by the architecture.

2.2.3.1 Architected Fields

An **architected field** is one whose functionality is specified by the architecture. The architected data must be returned in all architected fields for reads from readable registers. For writes to writable registers, the data written in each architected field must conform to the architectural specifications for that field.

Writes of an architected value to an architected field must perform the architected actions.

2.2.3.2 Constant Fields

A variation on architected fields in a register is the **constant field**. A constant field in a register is one which has only one architecturally allowed value. For writes to a writable register, software must write the architected value appropriate for any constant field in the register, and hardware may assume that the value written in each field is the architecturally specified value. If software writes any other value to a constant field, hardware may do anything allowed by the privilege of the page, although it is recommended that ERR_IMPROP be logged. For reads of a readable register, hardware must return the architected value appropriate for any constant field in the register.

2.2.3.3 Reserved Fields

A **reserved field** in a register is one which is not currently architected, but which may be architected in the future without further qualification. As such, the read data and write effects are specified by the architecture to allow for this expansion. If a register with reserved fields is readable, the hardware must return the value 0 for each reserved field. For writable registers with reserved fields, the software must write the value 0 to each reserved field; however, the hardware can not depend on this value being written since the field may become architected in the future, thus allowing software to write nonzero values. A write to a reserved field must have no effects on the module. In addition, a reserved field is unchanged by hard and soft power-on.

2.2.3.4 SVERSION-Dependent Fields

The architecture does not define the functionality of **SVERSION-dependent fields** in any register. SVERSION documentation is free to specify the exact usage of these fields as long as its functionality does not violate any architectural rule (e.g., the field must conform to the attributes of the page in which the register resides, and it can not implement an architected function). As such, SVERSION-dependent fields are not architected, nor will they ever be architected in the future.

2.2.3.5 HVERSION-Dependent Fields

The architecture does not define the functionality of **HVERSION-dependent fields** in any register. HVERSION documentation is free to specify the exact usage of these fields as long as each field adheres to the attributes of the page (register) in which the field resides. HVERSION-dependent fields are not architected, but they may become architected in the future with further qualification.

The architecture uses HVERSION-dependent fields to reserve space for future expansion in such a way as to maintain forward compatibility for present implementations without being as restrictive as reserved fields. When the HVERSION-dependent fields becomes architected, the existence of the fields's architected functionality will be indicated via some IODC mechanism.

In addition to the HVERSION-dependent fields specified by the architecture, SVERSION documentation can also specify HVERSION-dependent fields by choosing to reclassify an SVERSION-dependent field as HVERSION dependent.

2.2.3.6 Composite Field Definitions

The following field definitions are useful for notational purposes by the architecture when some decision is required in order to determine the final (non-composite) field definition. Composite field definitions do not exist in actual

register implementations.

2.2.3.6.1 Optional Fields

Optional fields are used in architected registers to allow for functions to be optionally incorporated into different implementations. If a given implementation chooses to utilize the function, the field must adhere to its architected definition; otherwise, the module must use the other field option allowed by the architecture.

2.2.3.6.2 Module Type Dependent Fields

A **module type dependent field** is one whose architectural definition is dependent on the type of module being considered. For example, a register field may be used differently by an I/O module than by a memory module. The module type dependent field classification is used only by generic documentation and is replaced by the actual definition by the module type documentation.

2.2.4 Register Classes

Each I/O register can be classified by the allowable actions which can occur when the register is accessed. Basically, there are four main classes of registers: architected, reserved, SVERSION dependent, and HVERSION dependent. Similar to register fields, there are composite types used by the architecture as shorthand notation.

2.2.4.1 Architected Registers

An **architected register** is a register whose functionality is specified by the architecture and contains at least one architected field. A read of a readable architected register must return the architected data, while a write to a writable architected register must cause its architected effects.

All writes to architected registers by generic software (i.e., with all SVERSION/HVERSION-dependent fields set to 0) must only affect the module state specified by the architecture, or as needed to implement the architected functionality (e.g., issuing a flex disable may require some signal being sent to the device so it will stop sending requests to the module).

Reads and writes by SVERSION/HVERSION-dependent software can have any effect on the SVERSION/HVERSION-dependent state (within the limits on privileged pages) but they must not have any effect on architected state, unless explicitly allowed by the architecture.

All architected registers which are not explicitly specified as readable (writable) are non-readable (non-writable).

2.2.4.2 Reserved Registers

A **reserved register** is one which consists of a single reserved field. Software must never access a reserved register. Hardware must, however, handle accesses to these registers since in the future they may be reclassified.

2.2.4.3 SVERSION-Dependent Registers

An **SVERSION-dependent register** is one which consists of a single SVERSION-dependent field.

2.2.4.4 HVERSION-Dependent Registers

An **HVERSION-dependent register** is one which consists of a single HVERSION-dependent field.

ENGINEERING NOTE

To simplify hardware, the HVERSION documentation can specify that an HVERSION-dependent register is aliased (i.e., not uniquely decoded).

An HVERSION-dependent register is both non-readable and non-writable.

2.2.4.5 Composite Register Definitions

The following register class definitions have characteristics which must be selected. Composite registers do not exist in implementations, but rather they exist in architectural documentation as shorthand notation for the purpose of consolidated and generalized presentations.

2.2.4.5.1 Optional Registers

The architecture uses **optional registers** to denote registers whose architected functionality need not be implemented by all modules of a given module type. Here, optional implies the ability to choose between an architected register and one of the other base (non-composite) classes. If a module desires the architected functionality of the register, the register's implementation must conform to the architecture; otherwise, the module must implement the other register class allowed by the architecture.

2.2.4.5.2 Module Type Dependent Registers

A **Module type-dependent register** is one which consists of a single module type-dependent field.

2.3 Module Operating States

The operating states of a module are given below; only ST_OFF is applicable for native processors.

A non-processor module may be in any of seven operating states. This state is defined by the value of the SRS IO_STATUS register. Since the SRS of an Type-A Direct module may not have IO_STATUS register, its state must be determined in an SVERSION-dependent manner. These operating states are defined below:

- **ST_OFF**

  The BUS_POW_VALID signal is deasserted or the signal is asserted but the bus init time has not elapsed. The accessibility of a module in the ST_OFF state is HVERSION dependent. A module enters the ST_ACTIVE state when the bus init time has elapsed.

- **ST_ACTIVE**

  The module is actively processing a command, or is performing a power-on reset. IO_STATUS[ry] = 0. When the module is ready to accept a new command, the module goes to either ST_READY (if no error), ST_AERR (advisory error), ST_SERR (soft error), ST_HERR (hard error), or ST_FERR (fatal error).

- **ST_READY**

  The module is ready for a command and is not in any error or diagnostic state. IO_STATUS[estat] = IO_STATUS[se] = IO_STATUS[he] = IO_STATUS[fe] = 0, IO_STATUS[ry] = 1. When a command is received, the module enters ST_ACTIVE. If an error condition is detected in this state, the register set transitions to the appropriate error state.

- **ST_AERR**

  An advisory error has occurred and has not yet been cleared. IO_STATUS[estat] != 0, IO_STATUS[se] = IO_STATUS[he] = IO_STATUS[fe] = 0, IO_STATUS[ry] = 1. When a command is received, the module enters ST_ACTIVE. If an error condition of higher severity is detected in this state, the module transitions to the appropriate error state.

- **ST_SERR**

  A soft error has occurred and has not yet been cleared. IO_STATUS[se] = 1, IO_STATUS[he] = IO_STATUS[fe] = 0, IO_STATUS[ry] = 1. When a command is received, the module enters ST_ACTIVE. If an error condition of higher severity is detected in this state, the module transitions to the appropriate error state.

- **ST_HERR**

  A hard error has occurred and has not yet been cleared. IO_STATUS[he] = 1, IO_STATUS[fe] = 0, IO_STATUS[ry] = 1. When a command is received, the module enters ST_ACTIVE. If a fatal error condition is detected in this state, the module goes to ST_FERR.

- **ST_FERR**

  A fatal error has occurred and has not yet been cleared. IO_STATUS[fe] = IO_STATUS[ry] = 1. When a command is received, the module enters ST_ACTIVE.

Although a module in the ST_READY, ST_AERR, ST_SERR, ST_HERR, or ST_FERR states goes to the ST_ACTIVE state upon receipt of any command, not all commands are defined to execute normally when written to a register set in either the ST_HERR or ST_FERR states. Refer to the description of the IO_COMMAND register for details.

In general, the architecture defines module operation when the module is powered (BUS_POW_VALID asserted). As such, except for discussions pertaining to boot and powerfail, the possibility of a module being in ST_OFF is usually not considered and is provided only for the sake of completeness.

The following table indicates how the operating state of a module can be identified via the SRS IO_STATUS register The '---' entries indicate that the field is not used to determine the module's operating state (i.e., it's a ''don't care'').

TABLE 2-2. Non-Processor Module Operating States

State	estat	se	he	fe	ry
ST_OFF	---	---	---	---	---
ST_ACTIVE	---	---	---	---	0
ST_READY	0	0	0	0	1
ST_AERR	Nonzero	0	0	0	1
ST_SERR	---	1	0	0	1
ST_HERR	---	---	1	0	1
ST_FERR	---	---	---	1	1

The transitions between operating states caused by commands are defined by the following figure:

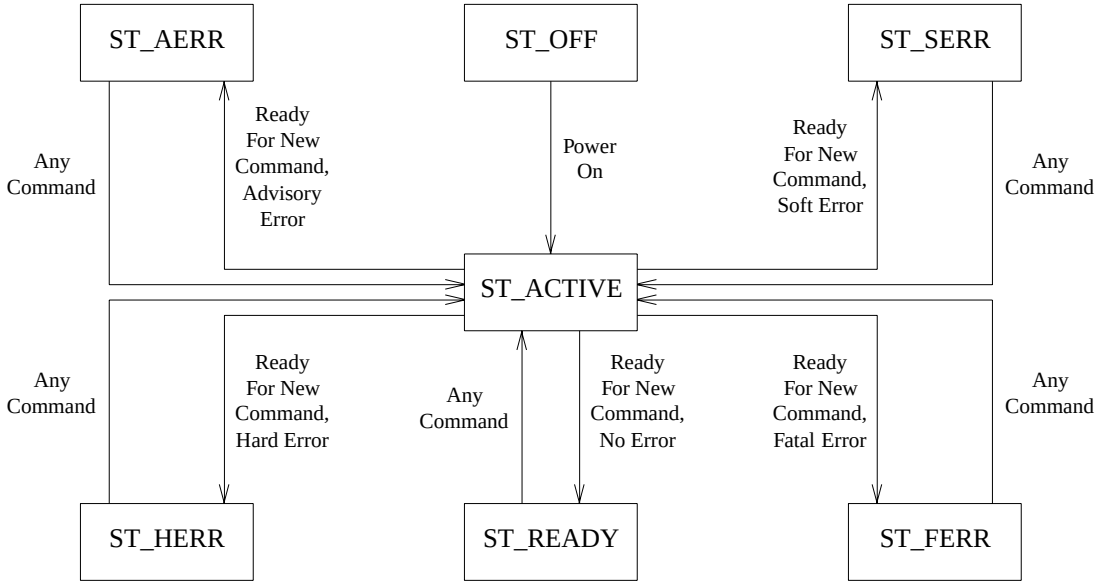


Figure 2-9. Non-Processor Operating State Transitions

In addition, the following transitions are also possible:

- from ST_READY, ST_AERR, and ST_SERR to ST_HERR
- from ST_READY, ST_AERR, ST_SERR, and ST_HERR to ST_FERR
- from any state to ST_OFF when BUS_POW_VALID is deasserted
- from ST_READY to ST_AERR and ST_SERR as well as from ST_AERR to ST_SERR when a bus transaction error is logged as a slave

2.3.1 HVERSION-Dependent Operation

Certain modules test their error detection circuitry using an HVERSION-dependent operating mode. For example, ENTRY_TEST on a memory module might place the module in an HVERSION-dependent operating mode that inserts double-bit errors.

Whenever a module enters an HVERSION-dependent operating mode using an architected mechanism (e.g., ENTRY_TEST), the following requirements must be met:

- The operations that are required to complete normally when the module is in ST_FERR must still do so while the module is in an HVERSION-dependent operating mode.
- While the module is in an HVERSION-dependent operating mode, both CMD_RESET and power-on must return the module to normal operation.

- The module must be returned to normal operation when the HVERSION-dependent operating mode is exited.

When normal operation is restored the following are true:

- The HPA of the module is accessible.
- IO_STATUS indicates the true operating state of the module.
- The operations that are required to complete normally when the module is in ST_FERR must work.

2.4 Command Properties

2.4.1 Command-Dependent Values

Values in the command-dependent field of the IO_COMMAND register are classified as follows:

DEFAULT — Value 0 is the DEFAULT architected functionality which is always required to be implemented by the module.

A — Values which have functionality specified by the architecture are classified as architected (A) command variants and must be implemented by the module.

R — Values which are not currently architected, but may be architected in the future are classified as reserved (R). The module must execute the DEFAULT architected functionality when a reserved (R) value is written.

OR — Values which were once reserved (R) and are now defined with extended architected functionality are classified as optional reserved (OR). The module must either execute the extended architected functionality or execute the DEFAULT architected functionality. The choice is SVERSION dependent.

SV — Values that are SVERSION dependent (SV) do not have any architected functionality. It is the responsibility of the SVERSION documentation to define the actions taken by the module when this value is written.

HV — Values that are HVERSION dependent (HV) do not have any architected functionality. It is the responsibility of the HVERSION documentation to define the actions taken by the module when this value is written.

2.4.2 Interaction Between *cmd* And *cmd-dep* Fields

When an architected *cmd* and *cmd-dep* pair is written with the SV field set to zero, no SVERSION-dependent effects are allowed as a direct result of the command, unless explicitly stated by the architecture (e.g., CMD_RESET).

In the case where an architected *cmd* and *cmd-dep* pair is written with the SV field set to a nonzero value, SVERSION-dependent effects on the SVERSION-dependent state are allowed in addition to the required functionality of the architected command.

When an architected *cmd* is written with the *cmd-dep* field set to an SVERSION-dependent value, SVERSION-dependent effects on the SV state are allowed in addition to the required functionality of the DEFAULT architected command (i.e., IO_COMMAND[cmd-dep] = 0).

If a value that is defined to be SVERSION dependent is written to the IO_COMMAND[cmd] field, SVERSION-dependent effects on the SVERSION-dependent state are allowed.

The relationship between the register fields is summarized below.

TABLE 2-3. Relationship Between the SV, *cmd*, and *cmd-dep* Fields in IO_COMMAND

SV field	*cmd-dep* field	*cmd* field	effect
0	A	A	Architected effect, no SV effect
≠ 0	A	A	Architected effect and SV effect on SV state
X	SV	A	Architected DEFAULT and SV effect on SV state
X	X	SV	SV effects on SV state, no architected effects

Where: X = any possible value

A reserved (R) value written to IO_COMMAND[cmd] is a special case of an architected (A) command. All of the actions which are allowed/required when an architected command is written to IO_COMMAND[cmd] still apply in the case of reserved commands (e.g., the module may temporarily clear the IO_STATUS[ry] bit, an interrupt may optionally be generated if IO_COMMAND[ie] is set (for those modules with an *ie* bit), and SVERSION-dependent effects are allowed on the SVERSION-dependent state when the IO_COMMAND[SV] field is nonzero).

However, for reserved commands the module is required to execute a null command (i.e., NOP) and may optionally log an error (ERR_IMPROP). During the course of executing a null command, no architected effects are allowed unless explicitly stated by the architecture.

In all instances where a reserved value is written to the IO_COMMAND[cmd] field, all values of IO_COMMAND[cmd-dep] are reserved.

When a reserved value is written to IO_COMMAND[cmd-dep], the module must execute the DEFAULT architected functionality. If the IO_COMMAND[SV] field is nonzero, SVERSION-dependent effects on the SVERSION-dependent state are allowed in addition to the DEFAULT architected functionality. When the IO_COMMAND[SV] field is zero, the DEFAULT architected functionality must occur and no SVERSION-dependent effects are allowed.

2.4.3 Non-Overwriting Commands

Non-overwriting commands must only be written to the IO_COMMAND register when IO_STATUS[ry] is 1. All reserved commands are non-overwriting.

The general structure of a non-overwriting command is:

```
if (IO_STATUS[ry] == 1) {
   IO_STATUS[ry] ← 0;

   /* delay until interrupt posting from previous cmd completes */

   if (IO_STATUS[fe] == 0 && IO_STATUS[he] == 0) {
      IO_STATUS[se,estat] ← 0;
      /* command specific processing goes here */
   } else
      nop();

   IO_STATUS[ry] ← 1;

   if (interrupting command) {
      /* module type specific interrupt posting goes here */
   }
} else
   SV();
```

Figure 2-10. Non-Overwriting Command Structure

2.4.4 Overwriting Commands

Overwriting commands may be written to the IO_COMMAND register regardless of the state of IO_STATUS[ry].

All bits in the overwriting command overwrite and take precedence over all bits in the overwritten command. So, for example, the *ie* (interrupt enable) bit of an overwriting command takes precedence over the *ie* bit of the previous command.

When a second command overwrites the first, the *ry* bit must not be falsely set to 1 by the completion of the first command.

The fetching and decoding of the command by the module hardware and/or firmware must be atomic. Any interruption in the fetching and decoding must be detected before any action is taken.

ENGINEERING NOTE

A simple sequence would be for the module to clear a bit each time a command is written to the IO_COMMAND register. Clearing the bit and changing IO_COMMAND must be atomic. The backend would then set that bit, read the command out of the command register, and then retest the bit. Repeat this sequence of operations until the read of the bit returns a 1.

Note that the command may need to be saved in temporary storage while it is being executed.

ENGINEERING NOTE

The fetching and decoding of any register must be atomic with respect to the system; no action must be taken by the module on an inconsistent register. For most registers, because of the system to module protocol, there is no race between the system writing a register and the module fetching and decoding it. That is, the module is not trying to fetch and decode a register while the system is writing it. There are, however, situations where races may exist. If, for instance, an overwriting command also requires other registers to be set up before the command is given (for example, the address of a DMA chain must be written to the IO_DMA_LINK register before CMD_CHAIN is written to the IO_COMMAND register) and the registers to be set up are being used by a currently active command, races may exist in fetching and decoding those registers.

Any overwriting command must be defined such that it functions correctly irrespective of the overwritten command being decoded or executed before the overwriting command was received.

ENGINEERING NOTE

For example, suppose that CMD_XXX is intended to modify but not abort a previously initiated CMD_CHAIN. In this case, a CMD_XXX may need to act as a CMD_CHAIN-AND-XXX in case it overwrites a CMD_CHAIN before the CMD_CHAIN is decoded and chaining started. It is particularly important that software can determine whether the module is still working on any of the outstanding DMA chains.

When a command is written to the IO_COMMAND register, the interrupt posting activity from a previous command must be completed before the new command is started. Note that this means that when the *ry* bit is set for a command, the interrupt from the previous command is guaranteed to be complete.

Overwriting commands start executing before the previous command is complete, but they complete the activity from the previous command before initiating any new actions. Therefore, an overwriting command will also display the property of not completing before all activity from the previous command is complete.

When an overwriting command is defined, it must state what commands it can overwrite, and what effect it has if the command it overwrote has not been decoded yet.

2.5 Module State

The **soft power-on state** of a module is its state at the instant a soft power-on occurs. The **hard power-on state** of a module is its state at the instant a hard power-on occurs.

The soft or hard power-on value for some module state must either be Constant, Unchanged, Defined (as a function of some other state), SVERSION dependent, or HVERSION dependent. A value of Unchanged means that the state is the same as it was just before BUS_POW_VALID was deasserted. A value of SVERSION dependent or HVERSION dependent means that the SVERSION/HVERSION documentation has the responsibility of defining the soft or hard power-on value. The SVERSION documentation must specify Constant, Unchanged, Defined, or HVERSION dependent. The HVERSION documentation must specify Constant, Unchanged, Defined, or Random.

A soft reset must leave the architected module state in the same state as its soft power-on state, except where required functionality precludes it. In addition, a hard reset must leave the architected module state in the same state as its hard power-on state, except where required functionality precludes it.

All state in a module is divided into three types of state: primary state, secondary state, and tertiary state.

Primary state is defined to be that state whose soft power-on value is either Constant, Random, or Defined. If Defined, the state must be a function of other primary state, and secondary or tertiary state. Primary state may be lost when BUS_POW_VALID is deasserted.

Secondary state is defined to be that state whose soft power-on value is Unchanged and whose hard power-on value is Constant, Random, or Defined. If Defined, the state must be a function of secondary and tertiary state. Secondary state is independent of BUS_POW_VALID.

Tertiary state is defined to be that state which is neither primary state nor secondary state. Tertiary state is independent of both BUS_POW_VALID and BUS_SEC_VALID (i.e., the soft and hard power-on value of tertiary state must be Unchanged). An example of tertiary state is the contents of a module's IODC.

SUPPORT NOTE

If a power failure occurs during an update of a module's tertiary state, tertiary state should be recoverable to the condition it was in prior to the power failure if the update fails.

The contents of the memory SPA space may be either primary or secondary state. The type of state of the memory IO_STATUS[sl] bit must be the same as the type of state of the memory SPA space.

An operation is said to have an **effect** on a module if any software observable module state is changed as a result of that transaction. For I/O modules, the software observable module state includes any software observable state from anything "behind" the module (e.g., device state, controller state, link state, etc.). In general, READ operations must have no effect on the slave module. The only exception is a read of an SVERSION/HVERSION-dependent register in a Type-A module; these reads have SVERSION/HVERSION-dependent effects on the module.

2.6 Power

All modules must perform a reset when they are powered on. Power-on is indicated when the BUS_POW_VALID signal is asserted.

A module which is reset by power-on is required to have interrupts disabled, not request any operation until enabled by software, and enter the active state, ST_ACTIVE.

ENGINEERING NOTE

Architecturally, the HPA must be immediately accessible and the *ry* bit valid at power-on. In reality, each bus specification defines a time, the bus init time, during which the bus is guaranteed to be idle. Modules may use this time to do any initialization required to make the HPA accessible and the *ry* bit valid.

Modules which do not know when the bus power was restored must wait the entire init time for that bus before initiating a system operation.

ENGINEERING NOTE

Implementations which wish to scan state into modules prior to running a system may define the bus init time to accommodate this activity, or use additional bus signals prior to BUS_POW_VALID to coordinate the initialization of parts of the system.

Modules which do not slave acknowledge transactions sent to them after the bus init time has elapsed are not architecturally visible. Architecturally, these modules do not exist.

For a module not in a module set, if the secondary state of the module survived the loss of primary power, the power-on reset performed must be a module-level soft reset with test, equivalent to CMD_RESET.ST. If the secondary state of the module did not survive the loss of primary power, the power-on reset performed must be a hard reset with test, equivalent to CMD_RESET.HT.

The same aliasing rules for command triggered module-level resets apply to resets triggered by power-on. If the module has no test, a reset with initialize (CMD_RESET.SI) is done. If the module has no secondary state, a hard reset with test (CMD_RESET.HT) is done. If the module has neither test nor secondary state, a hard reset with initialize (CMD_RESET.HI) is done.

For a module in a module set, if the secondary state of the module survived the loss of primary power, the power-on reset performed must be a card-level soft reset with test, equivalent to CMD_RESET.CST. If the secondary state of the module did not survive the loss of primary power, the power-on reset performed must be a hard reset with test, equivalent to CMD_RESET.CHT. The same aliasing rules apply to CMD_RESET.Cxx as for CMD_RESET.xx.

No status bits are preserved through a loss of primary power. As with other commands, *se*, *he*, *fe*, and *estat* indicate the success or failure of the power-on.

A module may use the BUS_POW_WARN signal to do powerfail preparation. The mechanisms used by a module to do its powerfail preparation are almost completely SVERSION dependent. The only requirement is that the module cannot request an operation when IO_FLEX[enb] bit is 0.

Modules must be able to detect the deassertion of BUS_POW_VALID during a transaction, and take the appropriate action to prevent data corruption before primary power is lost.

If primary power fails while a module is issuing a transaction, the module must either guarantee that its secondary state is unaffected, or it must indicate that secondary state was lost via an SVERSION-dependent mechanism (see Section 2.6, Module State).

If primary power fails while a module is slave to a transaction, the requirements on the module depend on its type. Furthermore, the slave must guarantee that its secondary state is not modified by a partial transaction which was not addressed to that module, but which was corrupted by the deassertion of BUS_POW_VALID.

ENGINEERING NOTE

On a bus with asynchronous BUS_POW_VALID, this can be implemented by providing a parity check to detect most bad addresses, or by delaying the effects of a transaction until the address is verified.

On a bus with synchronous BUS_POW_VALID, this can be implemented by nullifying the address compare if BUS_POW_VALID is deasserted during the SADD phase.

A DMA module must stop all DMA activity until it is explicitly restarted by its driver. If the module had a semaphore before power failed, it must lose that semaphore. (A reset always causes a module to lose any semaphore it holds.)

2.6.1 Modules with Secondary State

Modules with secondary state must be prepared for the possibility that bus operations were lost during the power failure. Those that have established checkpoints will roll back to the last valid checkpoint.

For modules with secondary state, an SVERSION-dependent mechanism must exist to indicate the validity of the secondary state.

When power fails somewhere in the system, a module may receive some indication or it may receive no indication. A module will receive the BUS_POW_WARN signal if power is failing on its local bus. However, it is possible that a module will receive no indication if power fails on some other remote bus.

On an I/O module with secondary state, if BUS_POW_VALID is deasserted while the module is slave to a transaction, the module must either guarantee that the secondary state is unaffected or set an SVERSION-dependent indication that secondary state is invalid.

If the module sees the BUS_POW_WARN signal, it must finish its current bus activity and save state. (Note that BUS_POW_WARN does not disable bus requestorship.) If the module receives no powerfail indication, it will most likely receive a bus error when it tries to read through a bus converter which has lost power on its adjacent port. The bus error will cause a fatal error to be set and bus activity to stop in that register set. (A module might want to set a hard error in the SRS and stop all module activity, or it might allow activity to occur on its other register sets. Further bus activity would most likely result in other bus errors.) If the system experiences partial powerfail, and the module does not receive the BUS_POW_WARN signal, and the module has a timer running against the driver, the timer may expire.

A normal powerfail cycle for an I/O module with secondary state is: receipt of BUS_POW_WARN, saving state, loss of primary power, restoration of primary power and power-on reset, establishing initial state, setting IO_STATUS[ry] in the SRS, and waiting for a command from its driver. A module with secondary state may wish to treat a soft reset similar to a powerfail cycle. (Note that a module must be able to handle multiple powerfails and soft resets. These may be handled equivalently.)

A soft reset forces the module to a known state but does not cause loss of secondary state: the module stops being a bus master, disables interrupts and SPA space, discards any owned semaphores, and waits for the driver to command the module. Architected registers which are HVERSION dependent after a soft reset may be considered part of the module's secondary state and therefore keep their previous values. Although the architecture places no requirement on backing up these registers, a module may choose to do so.

2.6.2 Implications of Lost Operations

WRITE operations initiated by DMA modules may be lost during powerfail. If the DMA module has no secondary state, there is no problem. The module will be given a soft reset, which will cause it to lose all knowledge of the operation active at the time of the powerfail. If the DMA module does have secondary state, that state will be preserved throughout the powerfail and after the soft reset.

Operations besides DMA writes can be lost during powerfail. Some cases cause no problems. For example, an interrupt message from an I/O module to a processor can be lost without consequence, as the recovery process will simulate an interrupt on each EIR bit. Similarly, a CLEAR system operation that fails (where the bus operation on

the bus with the memory module completes, but a bus operation on one of the intermediate busses fails) is also no problem. Losing the operation causes a semaphore to be lost, but the recovery process will restore all semaphores in any case.

Commands from a driver to its module can also be lost. Recovery procedures must allow for that eventuality.

2.7 Module Visual Indicators

Native processor modules may optionally implement visual indicators in the form of chassis displays. The number of chassis displays, the rules governing their use and interpretation, and the mechanisms by which they are shared in a multiprocessor system are defined by the *Chassis I/O Standard*.

Although implementation of chassis displays is not required, all processor modules are required to implement the PDC procedure PDC_CHASSIS. PDC_CHASSIS is the only mechanism that allows access to the chassis displays (see Chapter 4, PDC Procedures).

Other module types may optionally implement visual indicators, but their use and interpretation is HVERSION dependent.

SUPPORT NOTE

The Support organizations may require the implementation of visual indicators on a particular module. The implementation is based on the diagnosability and error coverage requirements that Support places on each module and/or system.

The interpretation of the module's visual indicators is HVERSION dependent and is described in the module's HVERSION documentation. However, module designers should refer to the *Chassis I/O Standard* and the module's bus specification where standardized visual indicators may be defined.

2.8 I/O Registers Required for Firmware

The PA-RISC I/O Architecture requires several I/O Registers to be accessed by PDC and IODC during system configuration. These registers which are used by the firmware are described in the following paragraphs.

2.8.1 HPA Space Registers

2.8.1.1 IO_DC_ADDRESS and IO_DC_DATA

The IO_DC_ADDRESS Register is a writable register and IO_DC_DATA is a readable register. Both registers for a particular reside at module HPA word offset 2 (that is: HPA + 0x8). These registers are used to return to the processor module dependent data and coded needed to configure and initialize the module. An address is written to IO_DC_ADDRESS which is an offset into the modules IODC space. A read from IO_DC_DATA, then returns the information at that location. For a complete description of the layout of a module's IODC space, see Chapter 5, IODC.

The IO_DC_ADDRESS and IO_DC_DATA Registers are required only for add-on I/O modules which may or may not be attached to a particular system. For modules which are hardwired to the system board they are optional. If they are not implemented, they must be emulated through the PDC_IODC procedure.

2.8.1.2 IO_COMMAND

The IO_COMMAND Register is a writable register at HPA word offset 12 (HPA + 0x30). This register is used to reset a module into a known state, either during module initialization, or after an error has occured. It can also be used to perform selftest of the module.

2.8.1.3 IO_STATUS

The IO_STATUS is a readable register at HPA word offset 13 (HPA + 0x34). It is used to determine the operating state of an I/O module. It is accessed during system intitialization and error recovery.

2.8.2 BPA Space Registers

2.8.2.1 IO_FLEX

The IO_FLEX Register is located at word offset 8 in the Local Broadcast Physical Address Space (LBPA + 0x20), and is used to determine the physical address of all I/O modules on a particular bus. The IO_FLEX register value is set during bus initializaton.

2.8.2.2 IO_COMMAND

There is an IO_COMMAND register at word offset 12 in both the Local and Global Physical Address Spaces (LBPA + 0x30 and GPBA + 0x30). The LBPA IO_COMMAND Register is used to reset all modules on a particular bus during system initialization, and the GBPA IO_COMMAND Register is used when a broadcast CMD_RESET is used to reboot a machine without powering it off.

LIST OF FIGURES

LIST OF TABLES

This page intentionally left blank