# *SBus Specification B.0*

Written by Edward H. Frank and Jim Lyle.

Edited by Jim Lyle and Mike Harvey.

# *Contents*

# *Figures*

# *Preface*

This book describes the formal specifications of the protocols and the electrical and mechanical features of the SBus.

## Contents

This book contains five chapters, three appendices, and a glossary.

Chapter 1, "Introduction," provides general information about the SBus specification.

Chapter 2, "SBus Overview," explains the philosophy and principles upon which the SBus is based, and describes the basic SBus protocols.

Chapter 3, "Protocol Design," describes the SBus protocols to transfer information across the SBus.

Chapter 4, "Electrical and Mechanical Design," describes SBus profiles, and the electrical and mechanical specifications of the SBus.

Chapter 5, "FCode Drivers for SBus Cards," describes the FCode programming language for writing SBus device PROMs.

Appendix A, "Specification Compliance," describes how SBus slaves, masters, and systems may comply with the SBus specification.

Appendix B, "SBus Extensions," describes extensions to the SBus that may be implemented in some systems, including the new 64-bit SBus extension.

Appendix C, "FCode Reference," describes FCodes currently supported by the Open Boot PROM, as well as new 2.0 FCodes.

The glossary defines terms used in this book.

## Changes from Version A.2

The following SBus features are new with this revision of the SBus specification:

- SBus ExtendedTransfer protocols to perform 64-bit per clock cycle transfers.

  This new information appears in Appendix B.

- SBus Profiles which define a minimum SBus card/controller feature set to guarantee plug compatibility between SBus cards and hosts.

  This new information appears in Chapter 4.

The following changes are also new with this revision of the SBus specification:

- Atomic restrictions in Chapter 3.
- DC changes in Chapter 4.
- VME/Futurebus mechanical drawings in Chapter 4.
- New 2.0 FCodes in Appendix C.

## Conventions

This book uses the following conventions:

### SBus Name

The name *SBus* is always written as *SBus*, not as sbus, Sbus, S-Bus, etc.

### Signal Names

Signal names are indicated with Helvetica font. At the first mention of a signal in a chapter or appendix, the full signal name appears with its abbreviation in parentheses. After its first mention, the signal is referred to by its abbreviation. For example:

At first mention, Request* (BR*). Thereafter, BR*.

### Signal Vector

A signal name followed by a range in parenthesis, for example D(31:0), represents a vector of logically related signals. The first number in the range indicates the most significant bit.

### Asterisk

An asterisk is appended to a signal name, for example Ack(2:0)*, to indicate that the signal is asserted in the low state, and unasserted in the high state.

### Words Used with Care

Care has been taken in the use of the words *must*, *may*, *only*, *might*, *can*, *could*, *should*, and *driven*.

### must only, may only

The phrase "An SBus device (master, slave, controller, system...) *must (may only)*..." means the function is required of all SBus devices and that failure to implement the function as described will likely cause failure or malfunction of the SBus device and possibly the entire system.

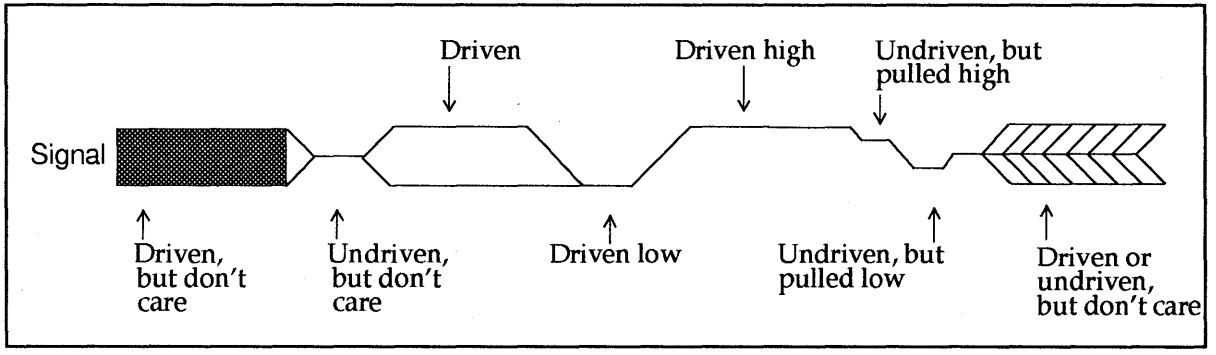
**should**

The phrase "An SBus device (master, slave, controller, system...) *should* ..." means that a designer need not implement the function, but that implementing the function is recommended because the function is valuable. Design and market trade-offs should be weighed carefully before any decision is made to omit the recommended function. A device that does not implement a recommended feature will work in any SBus system, but in some cases the device's ability to operate with other SBus devices may be limited.

**driven**

The term *driven* means that an output driver is sourcing or sinking current. *Driven low* means that an output is driving the signal to 0 volts. *Driven high* means that an output is driving the signal to 5 volts. If a vector of signals is shown as *driven* (but not specifically as high or low), each signal in the vector is stable and in its appropriate state. In the specific case of Ack(2:0)*, however, the *driven* convention indicates that at least one of the three signals in this vector is driven low — that is, Ack(2:0)* is asserted.

*Figure P-1. Timing Diagram Conventions*



**Clock Edge**

The term *clock edge* means the *rising* edge (the transition from 0 volts to 5 volts) of the Clock signal.

## Special Notes

Many parts of the specification contain three special notes, called a *profile recommendation, recommendation,* and *observation.* The information in these notes is interpreted according to the following definitions.

**Profile Recommendation**

Implementing features and functions described in a *profile recommendation* results in a device or system that is in compliance with the SBus Profile recommendations. This ensures compatibility with other systems or devices that are also compliant with the SBus specification.

**Recommendation**

Implementing features and functions described in a *recommendation* results in an SBus device or system that is higher in performance, operates better, or is more robust. SBus designers should omit these features only after careful consideration of the design trade-offs. Nevertheless, such omissions will not prevent the SBus device or system from working properly.

**Observation**

The information in an *observation* provides additional information about the operation of the SBus, constructing SBus devices and systems, and other related topics.

# 1

# *Introduction*

The computer bus described in this book is a high-performance system and I/O interconnect for use in today's highly-integrated computers. The SBus addresses the issues of high-performance, low-power, high-integration, and small mechanical form factor. Unlike previous workstation buses, the SBus is not designed for use as a backplane bus.

Typical SBus systems consist of a *motherboard* (containing the central processor and SBus interface logic), a number of *SBus devices* on the motherboard itself, and a modest number of SBus expansion connectors.

## Purpose

A principal reason for this revision of the SBus specification is the definition of an ExtendedTransfer protocol to transfer 64 bits of data each clock cycle. SBus ExtendedTransfers work only in systems and cards designed to support them.

However, a 32-bit SBus expansion card that completely ignores the ExtendedTransfers will work equally well in existing 32-bit only SBus systems, as well as in future systems with 64-bit masters and slaves: 32-bit SBus masters can transfer data to and from 64-bit SBus slaves; 64-bit SBus masters can transfer data to and from 32-bit SBus slaves.

This book describes the logical, electrical, physical, and programming interfaces for integrated circuits, boards, and systems. Boards and systems designed according to the specifications in this book can operate in a variety of SBus environments.

## Scope

The SBus is designed for use as a chip-level interconnect between components in microprocessor-based systems. The bus is designed to span only a small physical distance; it is not designed for use as a general-purpose backplane interconnect, although such use might be possible.

The bus is optimized for use in systems employing CMOS components. Slave interfaces to the bus can be implemented using a small number of buffers and programmable logic devices.

The SBus offers the following features to bus designers:

- Up to a 64-bit data path.
- A 32-bit virtual address for masters.
- A 28-bit physical address per slave.
- No jumpers (geographical device selection).
- A 16.67 to 25 MHz master bus clock.
- Completely synchronous operation (except for interrupts).
- Up to eight masters.
- Data transfers of 1, 2, 4, 8, 16, 32, and 64 bytes.
- Error and rerun protocols.
- Compatible with CMOS components.

- Machine-independent code for autoconfiguration and boot devices.
- Implementations can have low transfer latency.
- Seven shared interrupt lines.

## SBus Objectives

The design objectives of the SBus are:

- High performance.
- Low cost.
- Low power dissipation.
- Small form factor suitable for use in desktop computers.
- Ease of use by designers and users.

# 2

# *SBus Overview*

This chapter provides an overview of the SBus. The overview is neither a specification nor definitive. Its purpose is to introduce the following concepts that support the SBus specification in the remainder of this book:

- SBus system organization.
- SBus design philosophy and principles.
- SBus signals.
- SBus basic bus cycle.

Like most computer buses, the SBus can be used in a variety of configurations. The differences between these configurations are minor in most respects. In fact, an SBus slave or DVMA master, in general, never needs to know the kind of system in which it resides. In all configurations, there are some centralized SBus functions that must be implemented by an SBus controller.

## Dominant Technologies

The SBus is optimized for the technologies expected to dominate in the late 1980s and early 1990s: CMOS and surface-mount. The SBus is designed for use as a chip-level bus, between components such as processors and memory. It can also be used as a motherboard I/O expansion bus in configurations where it is possible to control wiring distances, clock skew, noise, and capacitance.

### Protocol Concerns

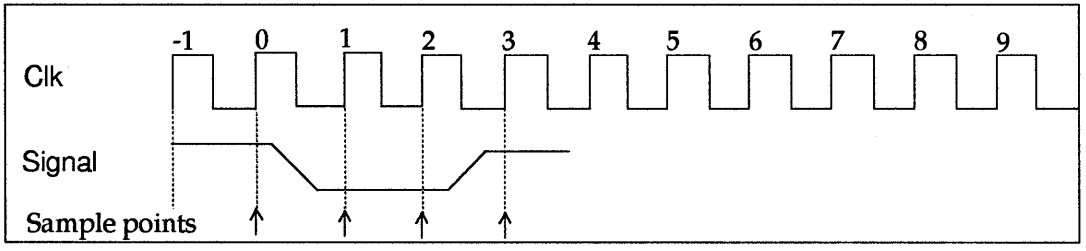
The design of the SBus protocols is based on three principles:

- Synchronous operation.
- Active drive.
- No driver overlap.

### Synchronous Operation

The SBus controller is responsible for generating a fixed-frequency Clock (Clk) in the range of 16.67 MHz to 25 MHz. All signals are sampled on the rising edge of this Clk. Signals must be driven so that they meet the SBus setup time and hold time requirements subject to the allowable Clk skew. SBus interrupts are allowed to be asynchronous. It is the responsibility of the controller to *synchronize* them to the appropriate Clk.

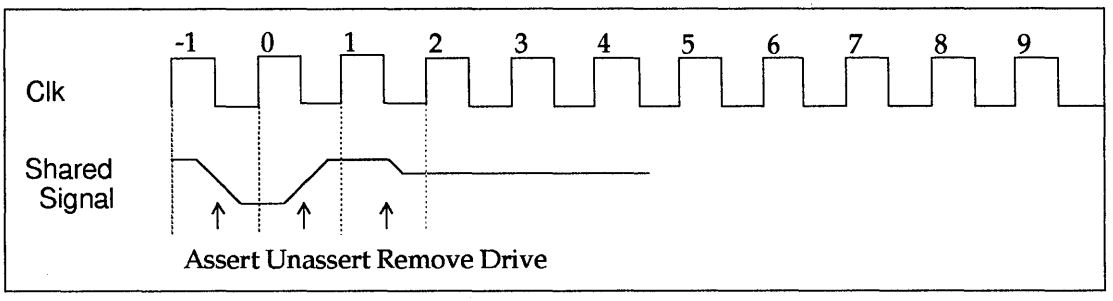
*Figure 2-1. Synchronous Operation*



**Active Drive**

A tristate control signal, which has been asserted, is actively driven to its unasserted state before the source removes its drive. Adhering to this principle facilitates the operation of the bus at speeds up to 25 MHz, without the need for low-resistance pullup resistors and output drivers capable of sinking the resulting static current.

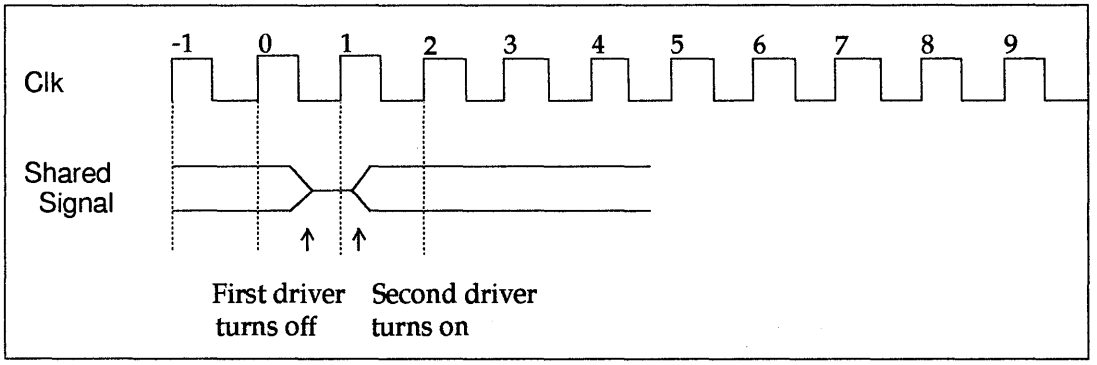
*Figure 2-2. Active Drive*




**No Driver Overlap**

No signal (except open-drain interrupts) is driven by two outputs during the same clock cycle. Adhering to this principle guarantees that output drivers never *fight*, since this can result in unreliable operation and excessive power dissipation. The alternative of using open-drain outputs is not compatible with low-power, high-performance CMOS.

*Figure 2-3. No Driver Overlap*



## Electrical Concerns

From an electrical and performance perspective, the setup, hold, and delay requirements of the SBus are compatible with many CMOS logic families and CMOS gate arrays. Circuit modeling and lab measurements should be performed to determine the compatibility and appropriateness of use for a given CMOS logic family.

### Total Bus Capacitance

The SBus limits the total bus capacitance to 160pF for systems running at speeds up to and including 20 MHz, and 100pF for systems running above 20 MHz. SBus expansion cards may not add more than 20pF per signal.

### Static Power Dissipation

The SBus avoids static power dissipation. Input and output circuitry must have minimal leakage current.

## Mechanical Concerns

The SBus is designed as a board level expansion bus for use in desktop systems and other environments where space is limited. From a mechanical and packaging perspective, the availability of 100- to 200-pin surface-mount quad plastic flat packages provides substantial capability to be implemented in a single chip that is both small and inexpensive.

## SBus Signals

The SBus uses 82 signals for information transfer and control. If used with an expansion connector, 14 power and ground connections are added, for a total of 96 pins. The following figure summarizes these signals. The I/O column is from the perspective of a slave.

The abbreviations for signal names are for use in schematics and data sheets.

**Note:** During ExtendedTransfers, some of the Extended Transfer Information signals are multiplexed to serve additional functions: PA(27:0) is used for Data(59:32); Size(2:0) is used for Data(62:60); and Read is used for D(63). For more information, see Appendix B.

*Figure 2-4. SBus Signals*

| Name | Abbreviation | I/O | Description | Driven By |
|---|---|---|---|---|
| PhysAddr(27:0) | PA(27:0) | I | Physical Address | Controller |
| SlaveSelect* | Sel* | I | Slave Select (1 per slave) | Controller |
| Data(31:0) | D(31:0) | I/O | Data | Masters/Slaves |
| Size(2:0) | Siz(2:0) | I/O | Transfer Size | Masters |
| Read | Rd | I/O | Transfer Direction | Masters |
| Clock | Clk | I | SBus Clock | Controller |
| AddressStrobe* | AS* | I | Address Strobe | Controller |
| Ack(2:0)* | Ack(2:0)* | I/O | Transfer Acknowledgment | Slaves/Controller |
| LateError* | LErr* | I/O | Late Data Error | Slaves |
| Request* | BR* | O | Bus Request (1 per master) | Masters |
| Grant* | BG* | I | Bus Grant (1 per master) | Controller |
| Reset* | Reset* | I | Reset | Controller |
| IntReq(7:1)* | IntReq(7:1)* | O | Interrupt Request (open drain) | Slaves |
| DataParity | DtaPar | I/O | Data Parity (optional) | Masters/Slaves |
| Ground (7 pins) | Gnd | PG | Ground | Controller |
| +5V (5 pins) | +5V | PG | Power (2 A per slot) | Controller |
| +12V | +12V | PG | Power (30 mA per slot) | Controller |
| -12V | -12V | PG | Power (30 mA per slot) | Controller |

## SBus Configurations

Like most system interconnects, the SBus can be used in a number of configurations. The first design of the SBus was based on the notion that the CPU is a special participant on the SBus in that it uses a special path through the SBus controller to access the bus. This configuration is called a *host-based SBus.*

Nominally, SBus masters use the translation hardware in the SBus controller to translate the virtual address which the master has placed on the data lines into a physical address that the SBus controller places on the address lines. The controller then starts a slave cycle by asserting AddressStrobe* (AS*). What makes host-based systems special is that a translation cycle never takes place on the SBus, because of the special path of the CPU. It is assumed that the CPU has a private address translation facility (at least, logically).

Any SBus master may communicate with any other slave on the bus, regardless of system configuration. No limitations restrict an SBus master to DVMA operations into and out of system memory alone. A master may perform DVMA operations between itself and a slave in another slot, or even a slave in the same slot (most SBus masters also have slave capabilities).

If multiple, independent SBuses are attached to any one system in parallel to increase connectivity or available bandwidth, communication between a master in one SBus and a slave in another SBus depends on the system; it may or may not be supported, at the discretion of the system designer.

**Note:** The following information applies primarily to 32-bit per clock cycle transfers. For information about 64-bit per clock cycle transfers, see Appendix B.

## Host-based Systems

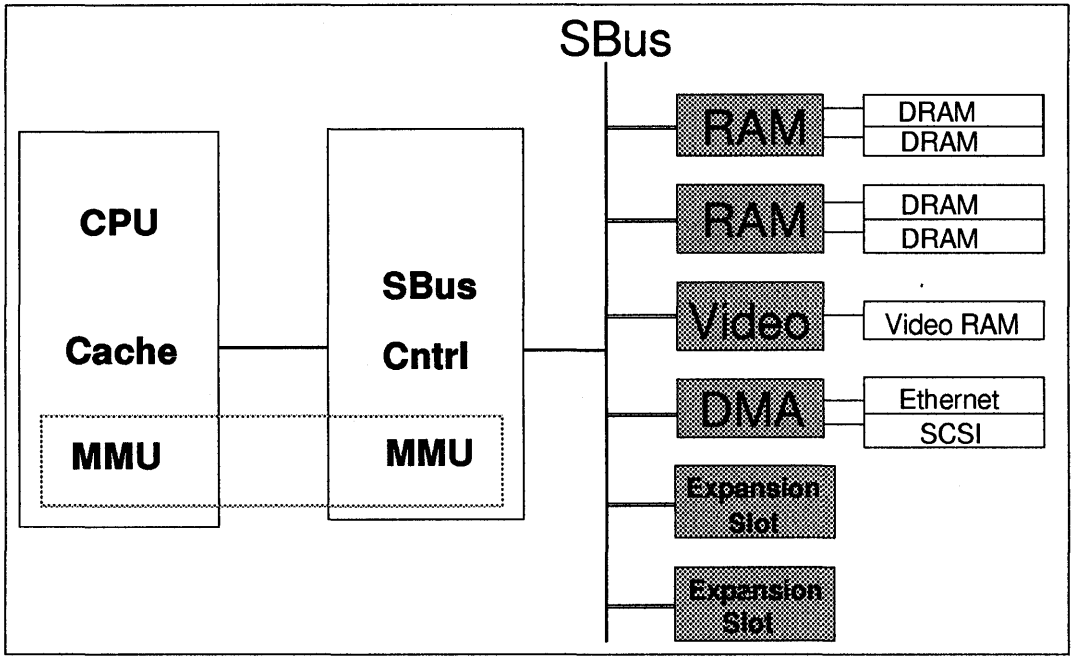
A host-based SBus is one in which the CPU (for example, a SPARC IU, SPARC FPU, cache, and MMU) uses the SBus as its principal memory and I/O bus. In most high-performance systems, the processor is connected to the SBus via a cache and memory management unit. Depending on exact implementation details, the system bus interface in the cache and MMU may be the SBus interface.

The following figure shows the logical interconnections in a host-based SBus system. The system shown in the figure is representative of a typical system. Many other configurations of SBus systems are possible.

This configuration is the one for which the SBus was originally designed. Because the processor core incorporates the SBus controller, at times the processor appears to be a special SBus participant. The only reason an SBus cycle is divided into a translation cycle and a slave cycle is that, in a host-based configuration, the processor does not use the SBus controller's translation mechanism to translate virtual to physical addresses.

Instead, the processor uses a direct path to the MMU for this purpose. One reason for implementing systems in this way is that the MMU may want to provide special services to the processor, such as a larger virtual address space and the ability to handle page faults.

*Figure 2-5. Example of Host-based SBus System*




The CPU and the SBus controller can share the MMU or use independent MMUs. Additional details about this subject appear in "Direct Virtual Memory Access" later in this chapter. The choice of which of these two alternatives to use is an implementation detail, and not fundamental to the operation of the bus.

In some very high performance systems it may be desirable for the SBus to be used only as a high-performance I/O interface, and not as the CPU's channel to main memory. This configuration may be useful in systems that embody memory buses which are wider than 32 bits. In this case, a processor bus to SBus interface must support either bidirectional transfers, or the SBus must have local memory for DVMA devices.

Ultimately, it is possible to have SBus systems in which each SBus consists electrically of a single SBus device that is connected via an interface IC to some other bus, such as the system memory bus. Such a configuration may allow multiple SBus devices to be accessed in parallel, since each device is on its own SBus. This architecture allows systems to be built in which the aggregate system I/O bandwidth is higher than that provided by a single SBus.

Such configurations may place limitations on communication between SBus devices on different SBuses, and thus be designed principally for transfers between the CPU and SBus slaves, or transfers between SBus masters and system memory, or both.

As explained in "Latency and Peformance" later in this chapter, different SBus configurations may have fairly different expected latency. As a rule-of-thumb, low-end systems which use the SBus as the system memory bus (for example, a Sun SPARCstation 1) have lower latency access to system memory than larger systems in which a DVMA access to main memory may result in the traversal of an interface to some other bus.

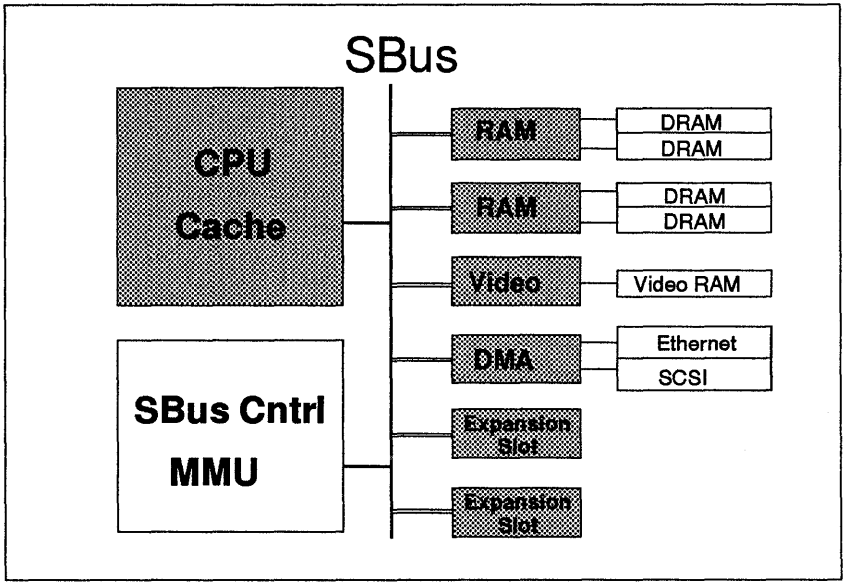
An advantage of low-latency systems is that they allow I/O devices to be built with a modest amount of local buffering, thereby reducing the cost of these devices.

## Symmetric SBus Systems

As the following figure shows, the host CPU does not have a special path to the SBus in symmetric configurations; it is identical to every other master on the bus. In this case, the SBus controller performs address translation for CPU accesses to the SBus.

Nevertheless, nothing prevents the CPU from having private memory and its own MMU for translating virtual addresses when it accesses that memory. This latter configuration is very much like the host-based configuration in which the SBus is used only for I/O expansion. (Of course, in this configuration, the MMU must be reset so that the CPU is able to load the MMU over the SBus).

*Figure 2-6. Example of Symmetric SBus System*



### Bus Bridges

In some SBus environments, it may be desirable or necessary to have more SBus slots than provided by the base system. In these cases, an *SBus bridge* can be built to extend the SBus electrically, providing additional SBus slots.

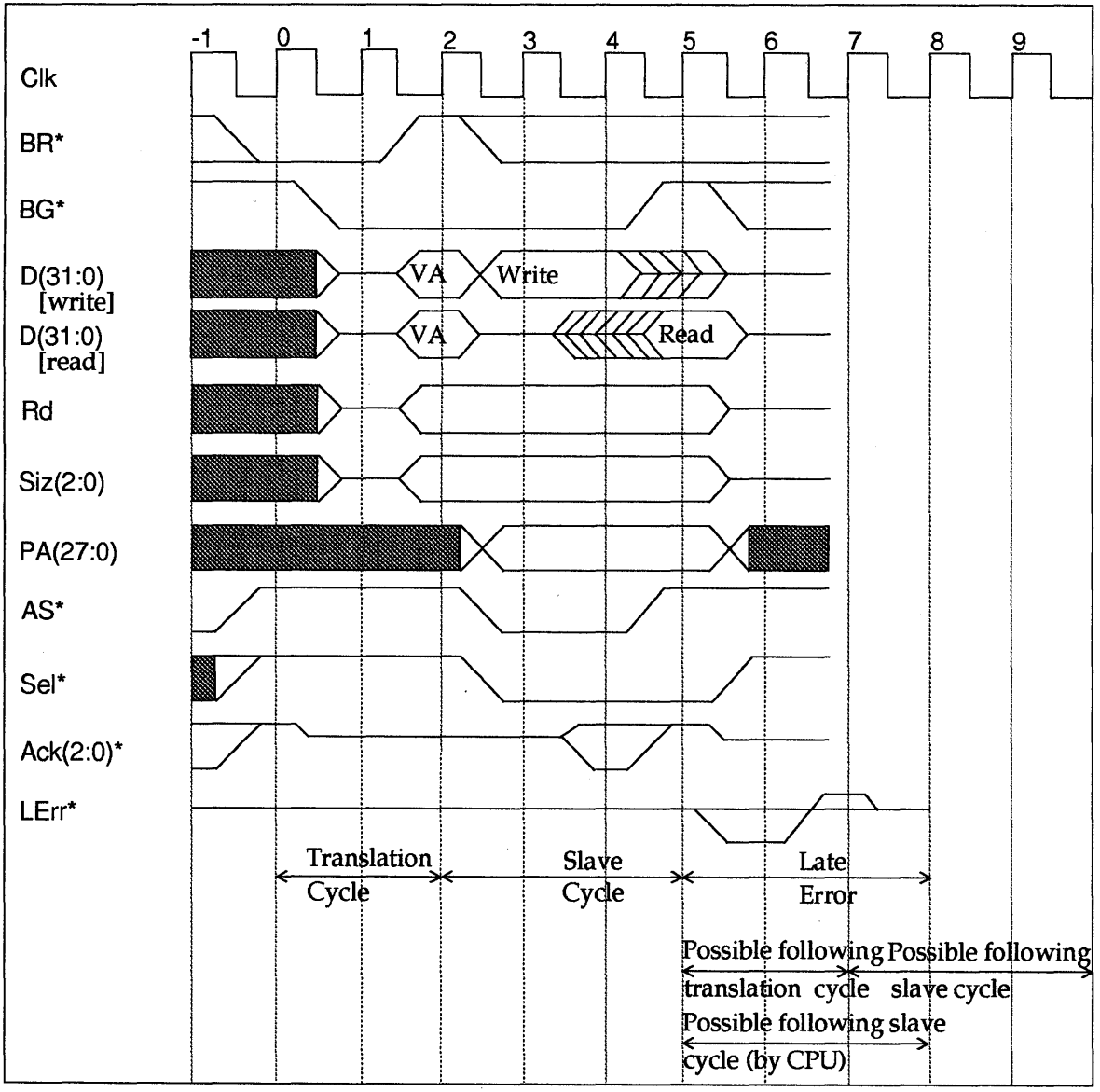
Nominally, the SBus bridge is functionally transparent to SBus devices. However, in the case where the bridge is designed to plug into an existing SBus slot, a degradation in performance is likely for those devices in the extended SBus environment.

## Basic SBus Cycle

A complete SBus DVMA cycle consists of two major phases: a *translation cycle* which places a physical address on the bus, and a s*lave cycle* which transfers data between the master and slave (except in the case of an error). However, when a CPU master uses the SBus in host-based systems, no translation cycle occurs on the bus; only a slave cycle occurs.

The following figure shows the sequence of events during a typical bus cycle. The timing shown is for the fastest transfer possible. If the transfer is to a slave which cannot respond this quickly, the slave extends the bus cycle by not generating a Data Acknowledgment until it is ready (subject to timeout).

*Figure 2-7. Basic SBus Cycle*



## Translation Cycle

A *translation cycle* begins when the SBus controller, after detecting that some master has asserted its Request* (BR*), decides to grant bus access to that master. At this time:

- The SBus controller asserts Grant* (BG*) for that master.
- The selected master, on the following clock edge, samples BG* as asserted and must immediately place a virtual address onto Data(31:0) (D(31:0)) for exactly one clock cycle.

  The master must also drive Size(2:0) (Siz(2:0)) and Read (Rd) to their appropriate values.
- The SBus controller samples this virtual address on the following clock edge.

  If the master is writing to the slave, the master must drive D(31:0) at this time.
- The SBus controller may then take an arbitrary number of clock cycles to translate the address.

  When the SBus controller places a physical address onto the PhysAddr(27:0) (PA(27:0)) and asserts AS*, the translation cycle ends and the slave cycle begins.

## Slave Cycle

At the beginning of a *slave cycle*, the bus controller:

- Asserts AS*.
- Drives a physical address onto PA(27:0).
- Asserts SlaveSelect* (Sel*) for the designated slave.

If a CPU master in a host-based system caused the bus cycle, the CPU master also drives Rd, Siz(2:0), and D(31:0) (if performing a write) at this time. For a DVMA master, these signals are driven during the translation cycle.

The selected slave then has up to 255 clock cycles to perform the requested transfer and issue a non-idle acknowledgment on Ack(2:0)*. In the case of a burst transfer, the slave generates multiple acknowledgments, even though AS* remains asserted for the entire transfer. For single word transfers, the slave then drives Ack(2:0)* back to the idle (unasserted) state for one clock cycle and, in the following clock cycle, removes its drive. The slave may assert LateError* (LErr*) for exactly one clock cycle, exactly two clock cycles after Ack(2:0)* is asserted.

In the case of burst transfers, a slave capable of transferring a word per clock cycle keeps Ack(2:0)* asserted for each clock cycle as a word is transferred. During write operations, the slave is acknowledging data on the data lines during the clock cycle it is asserting Ack(2:0)*. Thus, the slave samples the data at the same time the master samples the acknowledgment for that data. For read operations, the acknowledgment is pipelined. Thus, the slave first generates the acknowledgment and, during the following clock cycle, drives the data lines.

Slaves requiring more time must drive Ack(2:0)* back to the idle state during the intervening time. In all cases, after the final Data Acknowledgment, the slave must drive Ack(2:0)* back to the idle state for exactly one clock cycle, and then remove its drive. There are 255 clock cycles available to transfer data during a burst transfer, not 255 clock cycles per word.

## SBus Controller

Central to every SBus system is an SBus controller. Unlike some buses in which each master contains all of the logic necessary to perform a bus cycle, the SBus controller is responsible for initiating each bus cycle. The controller does not have to be a physically distinct object, but may appear as part of the CPU's interface to the system. The SBus controller is responsible for the functions described in the following sections.

### SBus System Clock

The SBus system clock is a constant frequency signal to which all events on the SBus are synchronized. In many SBus systems, particularly host-based systems, the SBus clock is derived from an integral sub-multiple of the processor's clock to synchronize the SBus and processor. For this reason, a designer of an SBus system is allowed to select a system clock ranging from 16.67 MHz to 25 MHz. For example, a 40 MHz CPU will probably operate the SBus at 20 MHz.

### Bus Arbitration

Each SBus system is required to support one or more masters. It is the function of the SBus controller to arbitrate between masters for access to the bus. In order to meet the latency expectations (discussed in greater detail in later sections) of many masters, controllers must implement some form of fair arbitration. When access to the bus is granted to a master, the SBus controller is responsible for monitoring the transfer, in order that it can remove BG* at the appropriate time.

## Virtual Address Translation, and Page Size Restrictions

By design, SBus masters use virtual addressing. When a master acquires the bus, it places a 32-bit virtual address on the data lines D(31:0). The SBus controller is responsible for translating the virtual address into a physical address by driving the corresponding physical address on the physical address lines PA(27:0).

The SBus controller, as well as the system in general, have substantial flexibility in how they perform this translation and handle translation misses and errors. All controllers, however, should provide support for separate translation for blocks of addresses less than or equal to 64 Kbytes.

This enables designers of SBus cards to group registers in separate 64 Kbyte pages to protect them through the VA to PA mappings. This does not prohibit support for larger page sizes, but requires that support for the smaller page size be provided as well.

## Slave Selects

The SBus is a geographically-addressed bus. This means that each SBus slave receives a unique unary encoded address signal, called Sel*. If asserted, this signal indicates that, if and after a bus cycle is initiated, the given slave is addressed. It is the responsibility of the SBus controller to drive the Sel* appropriate to the translation of the virtual address presented by the master.

## Address Strobe

The SBus controller initiates a slave cycle by asserting address strobe. Thus, AS*, not Sel*, indicates that a slave cycle is in progress.

## Bus Timeouts

By definition, a master (in a properly functioning SBus) which initiates a bus cycle is guaranteed to receive a non-idle acknowledgment before AS* is unasserted. Thus, a properly designed slave must terminate the bus cycle after it has been selected. However, if an SBus device is selected but no physical device is present, the SBus controller terminates the bus cycle by generating an Error Acknowledgment. This case is the only one in which timeouts should occur. Hence, in most systems, timeouts will occur only during system configuration.

**Profile recommendation:** SBus devices that do not use Open Boot, or SBus devices installed in systems that do not implement Open Boot, should assert timeout themselves after 512 clocks have elapsed following the assertion of AS*.

**Note:** The 512 clock device timeout is not the same as the 256 clock SBus controller timeout. The clock device timeout is set to 512 clocks to allow for delays that may occur in some bus bridges. If an SBus device determines that the SBus controller will not be issuing a timeout and must therefore issue an Error Acknowledgment, the timeout generated by the SBus device must follow the 256 clocks, during which the SBus controller may also assert timeout.

SBus cards that use and are installed in a host with Open Boot may optionally check to determine whether the host requires this type of behavior, and act accordingly.

## Other Functions

The SBus controller is also responsible for the following functions:

- ❑ Generating Reset* (Reset*) on power-up.
- ❑ Supplying power to SBus expansion slots.

## Masters and Slaves

All other SBus functions are the responsibility of the masters and slaves on the SBus. These functions include:

- Bus sizing.
- Normal bus cycle termination.
- Error detection (including optional parity error detection).

The proper operation of the SBus depends on the proper operation of the masters and slaves on the bus. This is critical to the design of the SBus, since an improperly designed SBus device may cause a system to operate improperly, just as a properly designed (but malfunctioning) device may cause the system to operate improperly.

Although the designer is occasionally tempted to make the controller responsible for coping with certain kinds of protocol violations, the SBus specification explicitly removes this responsibility from the controller by requiring that properly working masters and slaves not violate protocol. Protocol violations are one of thousands of ways a malfunction might become evident.

Thus, it is unreasonable to devote special treatment to a few malfunctions that are as likely or unlikely to occur as the bulk of the malfunctions for which no special timeouts or treatment is provided.

## Direct Virtual Memory Access (DVMA)

A principal objective in designing the SBus is to provide support for Direct Virtual Memory Access (DVMA). DVMA allows all masters (whether the host CPU or I/O devices) to use virtual addressing in performing bus cycles. The most important advantage of DVMA is that it simplifies operating system and software memory management.

At the beginning of a bus cycle, a master places a virtual address on the bus, which the controller translates into a physical address and places on the bus. In the case of the SBus, the master places the virtual address on the data lines. The MMU may either be dedicated exclusively to the SBus controller, or shared between the CPU and the SBus controller (which the dotted boundary line represents in the figure showing a host-based SBus system earlier in this chapter).

After the controller translates the virtual address into a physical address, it places the physical address on the address lines and begins a slave cycle. The SBus supports a 32-bit virtual address space. Whether each slot uses its own virtual address map or all SBus slots use a common map depends on the implementation.

Among other capabilities, DVMA allows scatter/gather operations on a memory page by page basis. For example, a laser printer needing a megabyte of memory per printed page can specify the page as contiguous in virtual address space (even though the memory pages are not contiguous in physical address space) by setting up the MMU properly.

One other potential benefit of using virtual addressing for I/O devices is that it may allow the use of demand paging during I/O. In some systems (for example, those with TLBs) it may be desirable to let the MMU *walk* page tables. Although this book does not define how such methods should be implemented, the definition of the translation cycle, together with Rerun Acknowledgments, make implementation of such methods possible. At the same time, other systems may choose to treat translation faults as an error, and abort the transfer.

## Latency and Performance

Another principle objective in designing the SBus is to fulfill the need for high aggregate throughput and low-latency transfers. SBus performance is important because it is a critical part of overall system performance. The SBus is designed with this recognition in mind: a high performance bus can reduce the cost of bus masters by allowing the masters to have a minimum of private buffer memory. To realize this capability, of course, a system built around the SBus must include high-performance system memory capable of satisfying the raw bandwidth requirements of the devices on the bus.

Carefully implemented SBus systems should have no trouble keeping up with I/O devices, such as Ethernet or FDDI. Of course, since many of these systems will use the SBus as the CPU's memory bus, system and board designers must consider the impact of a master using a substantial fraction of the sustained bus bandwidth on CPU performance. SBus implementors should be aware that there are circumstances on desktop and especially server systems, where real-time response (latency), cannot be guaranteed.

### Clock Rates

Raw SBus performance is provided by allowing the SBus clock to operate at frequencies up to 25 MHz. In view of the boards and systems built by many different manufacturers, 25 MHz provides a good balance between high performance and ease of system design and integration.

### Transfer Rates

Potentially, a slave can transfer a word per clock cycle. Thus, the peak data rate at 25 MHz is 100 MB per second. Regarding sustainable transfer rates, the SBus provides for burst transfers of up to 16 words (64 bytes), with the opportunity for the slave to transfer one word per clock cycle.

In the case of a host-based SBus — where a CPU master can overlap address translation with earlier bus cycles — as few as two clock cycles of overhead are possible, thereby providing a burst transfer rate of 64 bytes every 18 clock cycles or 88 MB per second. In practice, memory subsystem overhead may make it difficult to sustain this transfer rate.

DVMA masters will incur at least two additional clock cycles of overhead for the translation cycle, resulting in a minimum of 20 clock cycles to transfer 64 bytes, or 80 MB per second at 25 MHz. Again, the inability of a memory system to provide data at this rate and the inability of the MMU to translate an address in a single clock cycle would result in reduced performance.

For example, the Sun SPARCstation 1 implements a 20 MHz SBus and has burst transfers of 16 bytes only. As a result of the implementation of the SBus controller and system memory, a 16 byte burst transfer by the CPU takes 11 clock cycles, and has a burst transfer rate of approximately 29 MB per second. DVMA masters require two additional clock cycles for address translation, and thus have a burst transfer rate of about 25 MB per second.

**Note:** The foregoing information applies primarily to 32-bit per clock cycle transfers. For information about 64-bit per clock cycle transfers, see Appendix B.

## Latency

Latency is another important parameter affecting performance. Depending on the type of device connected to the SBus, designers must consider *expected latency*, as well as *worst case latency*. The former applies to devices requiring sustained throughput, but which can tolerate an occasional underrun or overrun (network interfaces and fast disks are typical examples). The latter applies to devices which may malfunction if they underrun or overrun, such as real-time data capture devices or space shuttles.

Latency is a function of the following factors:

- The number of masters.
- The arbitration method.
- The time it takes to translate a virtual address — that is, length of a translation cycle.
- The time it takes for the addressed slave to complete the transfer — that is, the length of the particular slave cycle.

The SBus has a limit of 8 masters, requires fair arbitration, and slaves must respond within 255 clock cycles. The length of a translation cycle is the only parameter not bounded by definition.

Even ignoring the time it takes to translate an address, a DVMA master that wants to work under all latency assumptions needs to cope with a potential request to request latency of approximately 8*256 clock cycles, or about 120 µs at 16.67 MHz. Worst case translation cycles in some high-end systems may be as bad as (or even worse than) 10 µs, which would result in a worst case latency of 200 µs or more. Thus, a master requiring a guaranteed response time may need a substantial amount of private buffering.

Expected latency in SBus systems which have high-performance memory on the SBus (for example, the Sun SPARCstation 1) may be one to two orders of magnitude better. In these systems, a DVMA access to memory will likely require:

- Two to three clock cycles for the translation cycle.
- One to two clock cycles per word of data.
- Two clock cycles of overhead.

If 8 masters are all performing 64 byte (16 word) burst transfers, a master might incur approximately 300 clock cycles (8*(3+32+2)) or approximately 18 µs of latency between the start of successive transfers. Because it is unlikely that all 8 masters are requesting use of the bus, expected latency will generally be significantly less.

**Note:** Worst case latency numbers are based on 8 masters, and reflect the worse case delay from the start of a particular master's transaction to the start of the same master's following transaction. In many machines, there will be fewer than 8 masters (The Sun SPARCstation 1 has a maximum of 4, including the CPU), and thus the worst case latency in a particular system may be substantially lower than these figures.

Designing a DVMA master requires careful consideration of the performance requirements of the device, as the considerations in this section make clear. It is not possible to offer a firm recommendation about how much buffering the master should include, though some guidelines can be provided. For typical low-end systems (for example, desktop systems), a master should be able to tolerate around 5 µs of latency.

Thus, a master which needs to sustain a 5 MB per second transfer rate should have at least 32 bytes of buffer, and use burst transfers whenever possible. For high-end systems (for example, servers), masters may need to tolerate latency several times this figure.

## Addressing and Configuration

One final objective in designing the SBus is to make system configuration and the installation of new SBus devices as trivial as possible. Because each SBus slave device has its own private Sel*, the slave does not need to *know* its own address in system physical address space. As a result, no address jumpers are required.

Each SBus device is also self-identifying. Beginning at location 0 of each SBus slave's address space is a string of bytes which describes the device. Nominally, these bytes are a special header followed by ASCII text which includes such information as the name of the card's manufacturer and the model number.

In more complex cases, the bytes following the header can consist of an executable byte-coded program that configures the card at system power-up, and provides information to the operating system about the type of the device, as well as which device driver to load. For such devices as disk drive interfaces, network interfaces and frame buffers, if this program is written to adhere to the boot-time programming interface, the system can boot from the device or display boot-time information before the operating system and its drivers are loaded.

## FCodes

As explained in Chapter 5, the byte-coded instruction set used with the SBus is an extended version of the FORTH programming language. There is nothing magical about using FORTH for this application. The language already exists, has reasonably machine-independent semantics, and is easily interpreted.

To take advantage of these capabilities, the host CPU must have a FORTH interpreter in its own boot ROM. Some system designers may also wish to provide a user interface to the interpreter for use in system debugging. The nature of such a user interface is beyond the scope of this document.

## ExtendedTransfers

A new addition to the SBus specification is a protocol for transferring 64-bits of data each clock cycle. Systems and SBus devices that implement the ExtendedTransfers are fully backwards compatible with 32-bit SBus systems and devices. Conversely, current 32-bit systems and devices are fully forwards compatible with the new ExtendedTransfer protocols.

SBus ExtendedTransfers provide substantially greater system performance by doubling the peak system bandwidth from 100 MB to 200 MB per second, and increasing the sustainable bandwidth for 64 byte transfers from 80 MB per second to 133 MB per second at 25 MHz. The maximum sustainable performance using a 128 byte ExtendedTransfer at 25 MHz is 160 MB per second, or twice what is available using 32-bit transfers.

The following chapters describe the specifications for the protocol, electrical, mechanical, and programmatic operation of SBus slaves, masters, and controllers in detail.

# 3

# *Protocol Design*

This chapter describes the specifications for the SBus protocols used to transfer information across the SBus. Devices and systems can conform to this specification without implementing all features of the protocols. Devices must be designed to work even if the optional features of the protocol are not implemented by other parts of the system.

The protocols described in this chapter apply primarily to 32-bit data transfers. Although the function of many signals is the same for 32- and 64-bit transfers, there are sequencing differences between the signals. For information about the ExtendedTransfer protocol, see Appendix B.

## Signal Determination

All SBus signals — except shared interrupt lines — must be driven so that they meet the SBus setup time requirement of 15 ns with respect to the SBus clock. They must also be driven so that they meet the SBus hold time requirement of 2.5 ns with respect to the SBus clock. The setup and hold time requirements must be met at all allowable frequencies of the SBus clock.

Signal names ending with an asterisk are asserted by driving them to $V_{OL}$ (known as the low state), and unasserted by driving them to $V_{OH}$, (known as the high state). All other SBus signals are asserted by driving them to the logic 1 state and unasserted by driving them to the logic 0 state.

With the exception of the shared interrupt signals, an SBus device must not drive a signal during any clock cycle in which the signal is driven by another device. This restriction includes those clock cycles during which the other device's output drivers are turning off (becoming tristated).

Certain shared signals must be driven to their inactive (unasserted) state before being tristated, as described in what follows.

## SBus Controller Signals

Every SBus must contain logic which controls the overall operation of the bus. This logic is called the SBus controller.

### Clock

Every SBus must have a free-running oscillating signal, called Clock (Clk), which provides a master clock reference for all SBus devices. For any given SBus system, the clock frequency must be fixed within the range of 16.67 MHz to 25 MHz. Every SBus expansion device must be fully functional throughout this range of clock frequencies.

The physical and electrical design of the SBus must restrict the total clock skew between any two clock inputs to no more than 2.5 ns. The SBus clock must have rise and fall times not exceeding 3 ns into 160pF. For more timing information, see Chapter 4.

SBus devices must sample bus signals only on the rising edge of Clk (hereafter referred to as the *clock edge*).

**Recommendation:** Since the SBus Clk may vary in frequency from system to system, designers should not use Clk as a *known* frequency source. Doing so may restrict the systems in which the device can be used — that is, if a device needs a clock that always runs at 10 MHz, the device should have its own 10 MHz oscillator. It should not divide the SBus Clk.

**Observation:** Using the falling edge of Clk to sample signals is dangerous: do not use the signal for this purpose.

Some SBus devices may be able to change the number of clock cycles required to perform various activities, such as memory reads and writes, as a function of the SBus Clk frequency. These devices can use FCodes to determine the frequency of the Clk signal in the current system, and initialize themselves accordingly. For more information about FCodes, see Chapter 5.

Although sampling must be done only with the rising edge of Clk, SBus devices may take advantage of the SBus setup time (15ns) by placing some amount of combinatorial logic either before the sampling register or after a flow-through latch.

There is no restriction on which edge of Clk a device may use to drive output signals. However, at 25 MHz, meeting the setup time into a 100 pF load makes it difficult to begin driving the signal at the trailing edge of Clk. Also note that the Clk duty cycle is not guaranteed to be symmetrical.

## Reset*

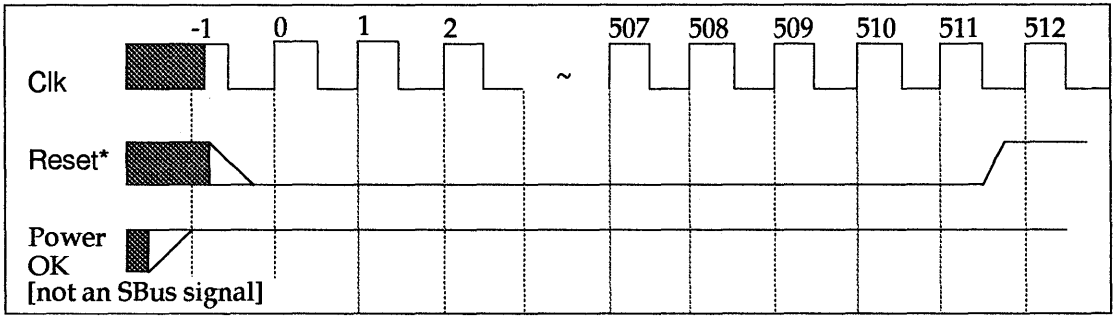
Every SBus must have a signal called Reset* (Reset*) which properly initializes all SBus devices after power-up. The SBus controller must assert Reset* after power-up. The SBus controller may be designed to allow Reset* to be asserted at other times by software or a push-button.

In all cases, Reset* must be asserted for at least 512 clock cycles before being unasserted. In the case of system power-up, power must be stable before these 512 clock cycles begin. The leading edge of Reset* may or may not meet setup times with respect to Clk. The trailing edge of Reset* must meet setup and hold times with respect to Clk. The SBus controller may keep Reset* asserted for more than 512 clock cycles.

After detecting the assertion of Reset*, an SBus device must perform whatever internal operations are required to initialize itself. While Reset* is asserted, an SBus device must not assert any bus signal. When Reset* becomes unasserted, masters may assert Request*, and slaves may assert IntReq(7:1)* (IntReq(7:1)*).

Because Reset* may be generated as a result of a software reset, an SBus device must not rely on internal *power-OK* detection alone to perform initialization.

*Figure 3-1. Example of* Reset* *Timing*



**Recommendation:** Designers of SBus controllers should provide a software reset mechanism. In most systems, it is desirable to provide an input to the SBus controller that connects either directly to the power supply's *power OK* signal or to some other hardware reset signal generated by another part of the system.

SBus controllers should assert Reset* as soon as possible after power is applied to the system.

**Observation:** The SBus does not provide a direct method for arbitrary SBus devices to reset a system. In many systems, software may force a reset via the SBus controller after certain internal error conditions, such as watch-dog timer interrupts.

## PhysAddr(27:0), SlaveSelect*, and AddressStrobe*

Every SBus controller must have:

- 28 physical address lines, called PhysAddr(27:0) (PA(27:0)).
- One or more decoded address lines, called SlaveSelect* (Sel*).
- A signal called AddressStrobe* (AS*).

These signals must be driven by the SBus controller only.

**Note:** During ExtendedTransfers, PA(27:0) is also used for D(59:32).

The SBus controller is responsible for driving PA(27:0) and Sel*, given either a virtual address from a DVMA master or, in host-based systems, a physical address from the CPU master. The SBus controller must generate a separate Sel* for each slave on the SBus. The decoding of physical addresses to generate slave select depends on the system.

PA(27:0) and every Sel* must be driven to a valid state by the SBus controller no later than the clock cycle during which AS* is asserted. PA(27:0) and Sel* must remain stable until the clock cycle after AS* becomes unasserted. Sel* may or may not remain asserted for additional clocks; its behavior is not guaranteed except where qualified by AS*. Thus, slaves must qualify Sel* with AS* to determine whether a transfer is in progress.

A slave must not rely on Sel* alone, and controllers are under no obligation to keep Sel* (Read (Rd), PA(27:0)) stable, except when AS* is stable. Sel* must be stable from a setup time before the clock edge following the assertion of AS* until the clock cycle following AS* being unasserted.

In all cases, the controller must keep AS* asserted at least until the clock cycle following the final Data Acknowledgment for the transfer, a rerun, or Error Acknowledgment. After one of these acknowledgments, the controller must unassert AS* for at least one clock cycle.

After a slave has been selected — that is, AS* and the slave's Sel* are asserted — the slave must generate an acknowledgment to terminate the bus cycle within 255 clock cycles. If the physical address presented to the slave is inappropriate, the slave must generate an Error Acknowledgment. Although most controllers unassert AS* during the clock cycle following the final acknowledgment, slaves must not make this assumption. Thus, a slave must check for AS* becoming unasserted to delineate successive bus cycles.

The following figure shows the general relationship among these signals.

In general, all accesses addressed to a particular SBus slot generate a slave select for that slot. The only exception to this rule is a write to address 0 within a slot's address space. These writes are reserved for bus expansion hardware (bridge hardware) which may not be visible within the address space of the slot that accommodates the bridge. Bridge hardware may intercept these writes to update their own state or to send DVMA information to a known location. The slave is not required to decode this location specially. It should handle any writes that it receives.

To allow for multiple bridges, the upper byte of data is defined as a key. To coexist with other hardware that intercepts writes to address 0, the bridge or other hardware must compare its key with the key in the data field, and intercept the write only if the keys match. A mechanism that implements this feature will be defined to provide unique keys to hardware.

Slaves should map registers that are read-only into the lower 64 Kbytes of their address space. By eliminating the need for the software to map the lower 64 Kbyte block as writable, inadvertent writes to address 0 are prevented. If there is a device that holds boot code which must be written, the boot code can be decoded in two locations within the slot's address space: one read-only and one read-write.

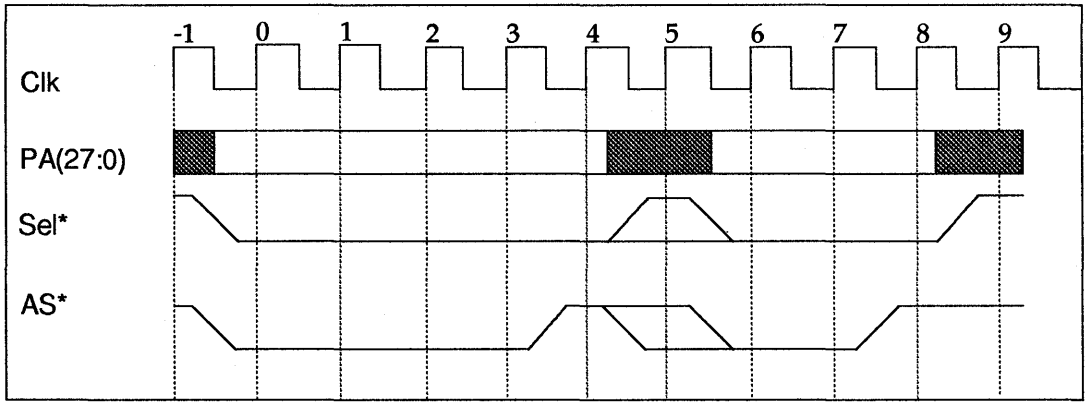
**Profile recommendation:** SBus devices that do not use Open Boot, or SBus devices installed in systems that do not implement Open Boot, should only require the use of the PA(24:00) signals to achieve maximum interoperability.

**Recommendation:** SBus controllers should unassert AS* during the clock cycle immediately following the final acknowledgment. boards should use the minimum amount of address space possible, and should not be extravagant with address space in general, since this might result in an inefficient use of mapping resources.

Also, addresses used should be concentrated in the low-end of the address space for maximum compatibility with systems which may not support the full 28-bit physical address space available.

**Observation:** The Sun SPARCstation 1 drives PA(27:25) to an unspecified value. Thus, devices requiring more than 25 bits of physical address may need additional selection hardware to work properly in the SPARCstation 1.

*Figure 3-2.* PA(27:0), Sel*, *and* AS*



## Request*, Grant*, and Arbitration

Each SBus master must have a signal, called Request* (BR*) which acquires mastership of the bus. For every SBus master, the SBus controller must provide a signal, called Grant* (BG*) which indicates to a requesting master that it has mastership of the bus. There is a unique BR* and BG* pair for each master in the system.

An SBus system must not have more than 8 masters.

Only one of the BG* lines may be asserted during any clock cycle. The SBus controller must arbitrate among requesting bus masters for use of the bus. Arbitration among DVMA masters must be fair, as defined by the following two rules:

- A DVMA master granted use of the bus during clock cycles $T_m$ through $T_n$ must not be allowed to use the bus again until all other masters which asserted their respective Request* during any clock $T <= T_n$ have been granted use of the bus (except during an atomic transaction).
- Within the above constraint, requests do not need to be processed in chronological order.

After asserting BR*, the master must leave it asserted until it receives BG*. During the clock cycle immediately following the assertion of BG*, the master must unassert BR* for at least one clock cycle (except if it is performing an atomic transaction), in which case the master must leave BR* asserted. For more information, see "Atomic Transactions" later in this chapter.

The SBus controller must keep BG* asserted until the end of the bus cycle.

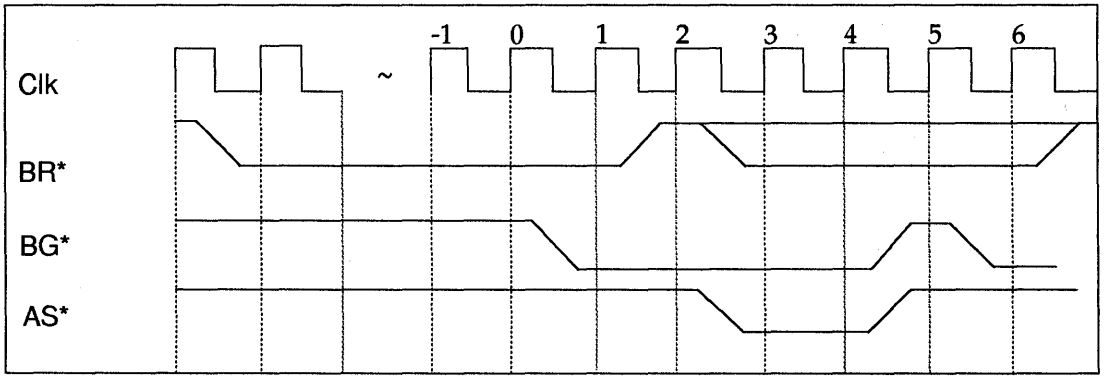
**Observation:** To reduce I/O latency, some systems may give higher priority to DVMA masters than to CPU masters. Thus, an SBus master attempting to use up all *extra* bus bandwidth by constantly requesting the bus may find that the CPU is never able to perform a bus cycle. In a system with only one such device, the device could wait until one or two clock cycles after its BG* became unasserted before reasserting its BR*. However, this technique will not work in a system containing multiple devices of this kind.

If a master (re-)asserts BR* while its BG* is asserted, the master needs to prepare to start the second request as soon as one clock cycle after BG* becomes unasserted.

The 8 master limitation is somewhat arbitrary. It is included as an aid to designers who must know absolute worst case conditions on the bus. In practice, most systems will have fewer masters. The number of slaves is not limited and does not affect performance *per se*. Nevertheless, the restrictions on maximum capacitive loading enforce a practical limit on the number of masters and slaves in most SBus systems.

**Note:** The number of clock cycles between BG* and AS* is a function of system translation time, and is therefore variable.

*Figure 3-3. Timing of* BR*, BG*, *and* AS*



## Bus Cycle

Every SBus bus cycle consists of a *translation cycle* and a *slave cycle*, with one exception — that in host-based systems, the CPU master may perform address translation without using the bus. In this case (as viewed on the SBus) it will appear as though the CPU master is performing slave cycles only.

**Note:** During Extended Transfers, certain sequencing details for Data(31:0), PA(27:0), Read, and Size(2:0) are changed. For more information, see Appendix B.

### Translation Cycle

The assertion of BG* begins a translation cycle on the bus. During the clock cycle immediately after it detects BG* has been asserted, the master must drive a virtual address onto Data(31:0) (D(31:0)), and drive Size(2:0) (Siz(2:0)) and Read (Rd) appropriately. The master must drive the virtual address on D(31:0) for exactly one clock cycle. If the master is performing a read, it must tristate the bus during the following clock cycle. If it is performing a write, it must drive the first datum onto D(31:0). Unless the master is performing an atomic transaction, it must unassert BR* for at least one clock cycle, beginning in the clock cycle after it receives BG*.

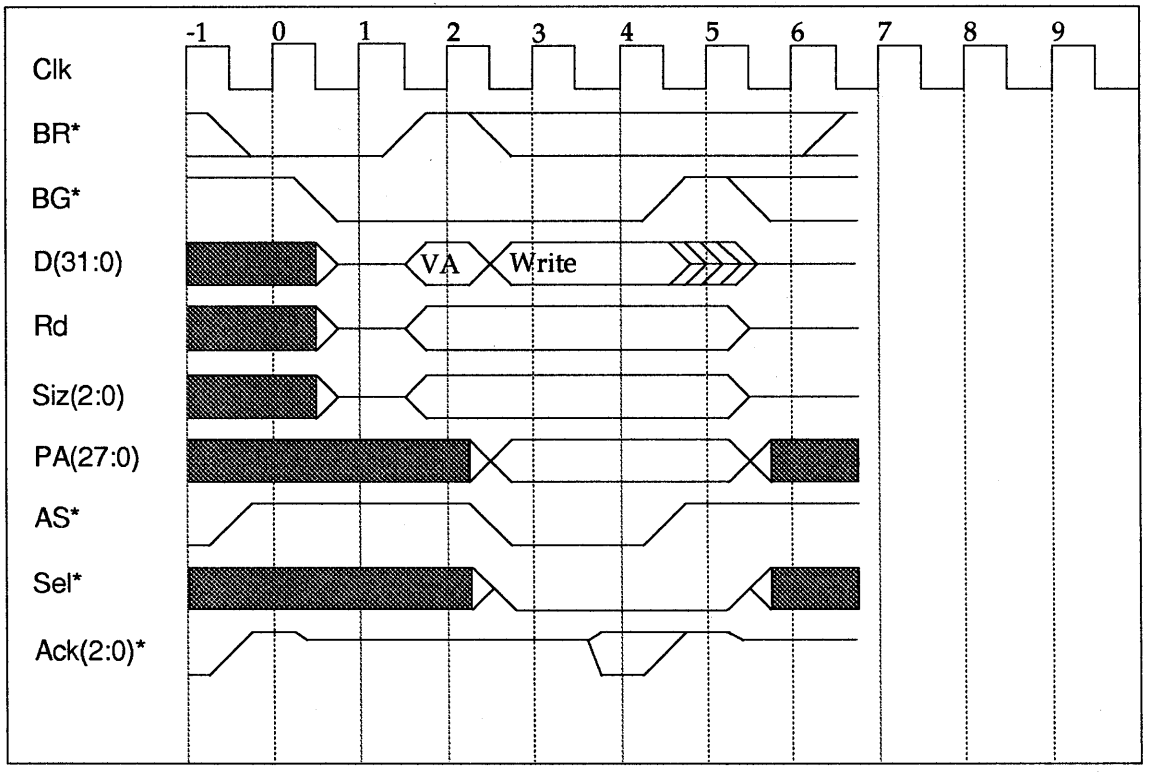
The SBus controller is responsible for translating the virtual address into a physical address. After translating the virtual address into a physical address, the controller must drive the physical address onto PA(27:0), assert the appropriate Sel*, and assert AS*. The SBus controller may drive the previous signals as soon as two clock cycles after asserting BG*, beginning in the clock cycle after receiving the virtual address. There is no predefined limit on the number of cycles the controller may take to translate the address.

If, as a result of a translation fault or access violation, the SBus controller needs to abort the bus cycle, it must not assert any Sel* and must signal this error to the current master with an Error Acknowledgment. The controller does not need to assert AS* when aborting the bus cycle in this manner, although it is free to do so. The SBus controller may assert LateError* (LErr*) as desired.

The method for translating a virtual address depends on the system. However, all SBus controllers must provide support for separate translation for blocks of addresses less than or equal to 64 Kbytes. This requirement allows designers of SBus cards to group registers to protect them through the VA to PA mappings: it does not prohibit support for page sizes larger than this limit; it simply requires support for at least one page size within this limit, in addition to any others that may be supported.

The bus controller may unassert BG* as early as the clock cycle in which it unasserts AS*. Under no circumstances may the controller unassert BG* or AS* until all words of data have been transferred, or the slave has issued an Error or Rerun Acknowledgment, or a timeout has occurred.

*Figure 3-4. Translation Cycle and Slave Cycle*



**Observation:** Because each master has its own BR* signal, the SBus controller can easily implement a separate translation table for each master, with context registers and all.

Sun systems use a variety of MMU structures. In the SPARCstation 1, the MMU is shared between the CPU and the SBus. However, only a single context is provided for all SBus DVMA masters.

If the master asserts Siz(2:0) and Rd at the same time it asserts the virtual address, the MMU can perform various checks on the transfer. Moreover, no other timing would work, since the assertion of AS* is controlled by the bus controller.
The minimum translation cycle requires two clock cycles: the first for asserting BG*; and the second for asserting the virtual address.

## Slave Cycle

A slave cycle begins when the SBus controller asserts AS*. The controller must keep AS* asserted until after the current slave gives its final Data Acknowledgment, or a Rerun or Error Acknowledgment. Thus, the SBus controller must monitor Siz(2:0) and Ack(2:0)* (Ack(2:0)*).

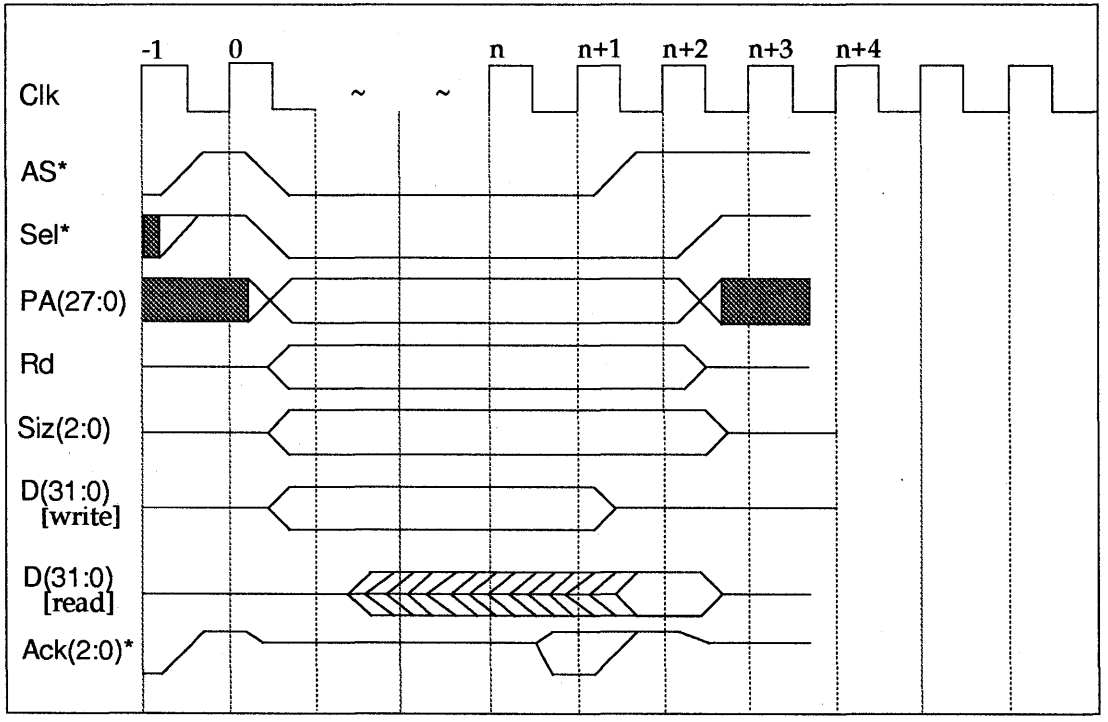
The SBus controller must be certain that PA(27:0) and Sel* are stable whenever AS* is asserted, beginning with the first clock edge at which a slave can sense AS* as asserted. Thus, PA(27:0), Sel*, Siz(2:0), and Rd must remain asserted until one clock cycle after AS* becomes unasserted.

The current master must keep Siz(2:0) and Rd stable until the earlier of BG* or AS* becoming unasserted. During the clock cycle following BG* becoming unasserted, the current master must stop driving Siz(2:0), and Rd. During a slave cycle, the SBus controller must keep BG* asserted at least as long as AS*.

When writing data to a slave, the master must drive the first datum onto the bus during the clock cycle following the virtual address (DVMA master), or in the same clock cycle in which AS* is asserted (CPU master). The master must keep the write data stable until acknowledged by the slave, and may tristate the data lines as late as the cycle after BG* has been unasserted (this makes the timing identical to the Rd and Siz signals).

A slave, however, may not depend on the data remaining valid after it acknowledges the data, which it may do as soon as one clock cycle following the assertion of AS*. During a burst transfer, the master must drive each successive word of data onto the data lines during the clock cycle immediately following the slave's acknowledgment of the previous word. Thus, when a slave generates an acknowledgment during a particular clock cycle of a write, it is acknowledging the data on the data lines during that clock cycle.

*Figure 3-5. Basic Slave Cycle Timing*



When data is read from a slave, the slave may generate an acknowledgment at any time (subject to timeout), beginning with the clock cycle following the assertion of AS*. D(31:0) *may* be driven by the slave (but need not be valid) as early as the clock cycle after that in which AS* and Sel* have been asserted. The slave *must* drive D(31:0), which must be valid during the clock cycle immediately following the corresponding acknowledgment. During a burst read transfer, the slave must drive the data lines with the appropriate word of data at each clock cycle in which a word is transferred.

A slave cycle ends only after the slave acknowledges the last word of data, or when the slave issues a Rerun or Error Acknowledgment. The types of SBus acknowledgments are described later in this chapter.

A DVMA master must use its BG* signal to perform all of its sequencing (except for the data sequencing, which is driven by Ack(2:0)*). The master needs to use the leading edge of BG* to know when to place a virtual address on the data lines. Similarly, the master must use the trailing edge of BG* to indicate that it should remove Rd, Siz(2:0), and D(31:0). The master must not use or make assumptions about AS* during a DVMA cycle, because the SBus controller may not assert AS* in some cases (for example, if the slave is a device that is logically but not physically connected to the SBus, or if the controller aborts a cycle due to a DVMA translation error).

As noted previously, DVMA masters must assert Siz(2:0) and Rd in the clock cycle following BG*. However, for CPU masters, Siz(2:0) and Rd are asserted at the same time as AS*.

**Observation:** The intent of the relative timing of AS*, Sel*, PA(27:0), Siz(2:0), and Rd is that a simple slave does not need to latch any of the above signals (with the exception of AS*). Hence, the reason for requiring the signals other than AS* to remain valid for one clock cycle after AS* becomes unasserted. However, AS* must always be sampled with Clk.

The minimum slave cycle requires three clock cycles:

1. The first for asserting AS*.
2. The second for asserting Ack(2:0)*.
3. The third for unasserting AS*.

For CPU masters in host-based systems, this timing means that *n* words could be transferred in *n*+2 clock cycles, assuming that a CPU read is not followed immediately by a CPU write. In this latter case, one additional clock cycle is required to prevent two different devices from simultaneously driving the bus (assuming that a CPU read is not immediately followed by a CPU write).

## Atomic Transactions

An SBus master may retain ownership of the SBus for multiple bus cycles to perform atomic transactions with a particular SBus slave. The intent is to provide an easy-to-use hardware mechanism for implementing semaphores.

To retain mastership of the bus, the SBus master must keep BR* asserted continually until it receives BG* for the last bus cycle. For each bus cycle in which the SBus controller detects that BR* has remained asserted (and, that it did not become unasserted during the clock cycle after BG* became asserted), the controller must give the same device mastership of the bus for the following bus cycle. If a master does not wish to perform an atomic transaction, it must unassert BR* during the clock cycle after BG* is asserted. A master which asserts its BR* must wait for the corresponding BG* before unasserting BR*.

Except for dummy reads (see later in this chapter), masters must perform, at most, two bus cycles during an atomic transaction. The first cycle must always be a read, and the second must always be a write. If a master cannot respond immediately with the write data (that is, by the time bus grant happens for the write cycle), the master may retain ownership of the bus by performing one or more *dummy* reads to the same address as the original read. To do so, the master must use the data from the first read.

Because of the way dummy read cycles are inserted to lengthen atomic transfers, adverse effects may occur on device registers if atomic operations are used to access them. Therefore, this practice is not recommended. Also, Rerun Acknowledgments must only be issued on dummy read cycles by bus couplers, because their meaning is otherwise unclear and results cannot be guaranteed.

If bus sizing occurs during an atomic transaction, there is no guarantee that the follow-on cycles will be performed atomically for an SBus DVMA master. Some systems may be implemented in which the CPU master accesses that are bus-sized remain atomic. Such atomicity is accomplished by designing the controller to give the CPU's follow-on accesses higher priority than all other accesses. However, this behavior depends on the system.

When a slave issues a Rerun Acknowledgment during an atomic transaction, the master must immediately unassert BR*, if it has not already done so. The SBus controller must unassert the corresponding BG*, as in the case of all Rerun Acknowledgments. The master must then reassert BR*.
An SBus master, after receiving a rerun on the read bus cycle or the write bus cycle of an atomic transaction, should restart the atomic transaction from the beginning. However, if the master receives a Rerun Acknowledgment on the dummy read bus cycle, the master should continue to issue dummy reads or proceed with the write bus cycle, since only a bus bridge should generate such a Rerun Acknowledgment.

Masters must not perform atomic transactions with a slave that issues Rerun Acknowledgments, unless the slave allows the transaction to be restarted beginning with the first bus cycle or the nature of the atomic transaction is such that the master can complete the transaction properly — even though the transaction starts at the current bus cycle instead of the first bus cycle.

However, an SBus slave must not assume that an atomic transaction which receives a Rerun Acknowledgment on the write phase will be restarted with the read. Slaves should avoid issuing reruns on the write portion of an atomic transaction, because it may be difficult for some masters to restart an atomic operation with the initial read.

A master that receives an Error Acknowledgment during any bus cycle of an atomic transaction should immediately deassert BR* for at least one bus cycle. The master may then reattempt the transfer by reasserting BR*, or it may take other appropriate action such as issuing an interrupt to the host.

There is no limit to the number of dummy reads which a master may perform to retain ownership of the bus. However, a master must not *hog* the bus (that is, use an undue portion of the bandwidth) with excessive use of dummy reads to itself. SBus controllers must support byte, half-word, and word atomic transfers, although they need not support bus sizing during atomic transfers. While controllers may support bursts during atomic transfers, they are not required to do so. SBus controllers that do not support burst transfers during atomic transfers must issue an Error Acknowledgment after the translation cycle instead of asserting AS*.

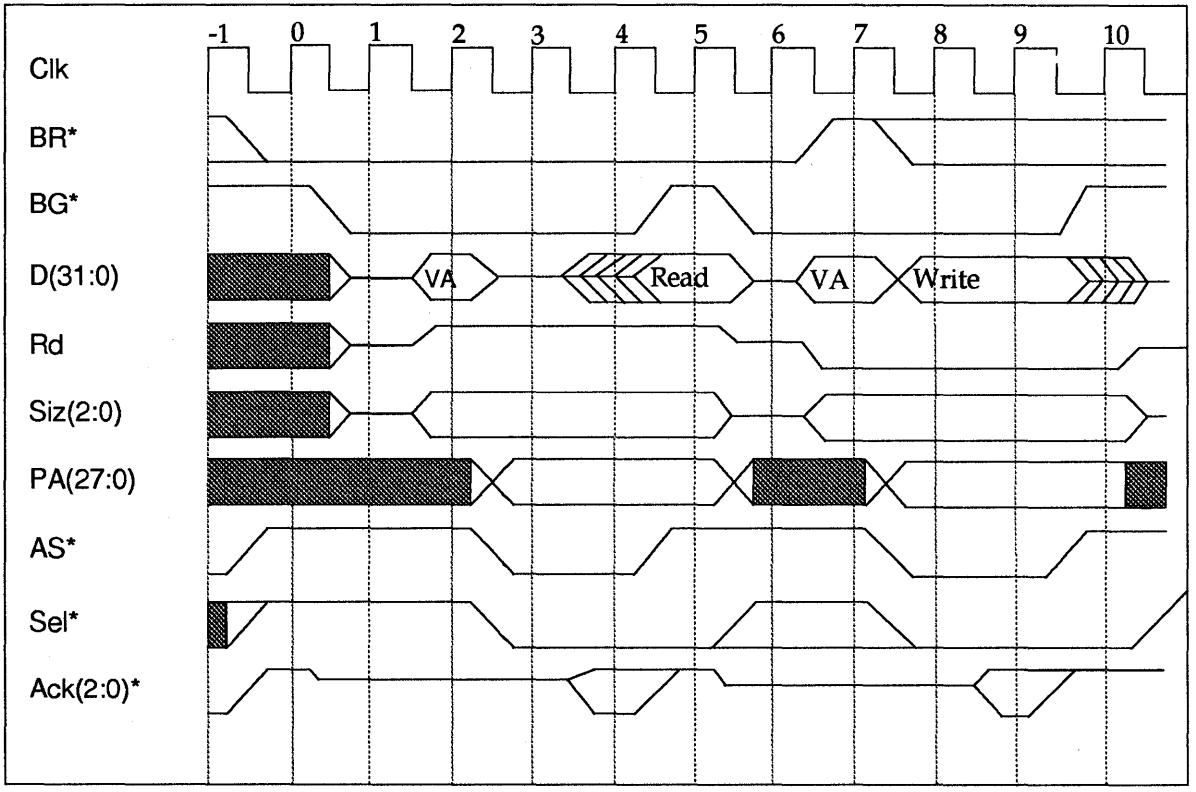
BG* may be unasserted for only a single clock cycle between the bus cycles of an atomic transaction. Hence, a master may have only two clock cycles (the first when BG* is unasserted, and the second when it is driving the virtual address) to modify data if it is attempting to perform an atomic read-modify-write transaction.

**Recommendation:** Avoid bus sizing during atomic transactions (or, conversely, avoid atomic transactions to slaves that bus size), since bus sizing works properly only in systems that support bus sizing for the CPU master, and then work only when the CPU master makes the access.

Avoid using bursts during atomic transactions, because there is no guarantee that a particular host will support them.

The specification for atomic cycles provides for flexibility in the handling of reruns during an atomic cycle. Unless there is a good reason to do otherwise, after a rerun masters should restart the atomic transaction beginning with the first bus cycle, and slaves should be designed to expect the transaction to restart with either bus cycle.

*Figure 3-6. Atomic Transaction Timing*




**Observation:** Except for the timing of BR* and the requirement that no other master is allowed to access the bus, the consecutive bus cycles in atomic transaction are just like any other SBus cycles.

In SPARC-based systems, the CPU master retains ownership of the bus for multiple bus cycles to implement the SPARC SWAP and LDSTUB instructions.

Do not attempt to build a master that keeps BR* asserted after receiving a Rerun Acknowledgment. Such behavior could easily cause a system to hang as a result of deadlock.

The SBus does not provide any indication to slaves that an atomic transaction is in progress. Slaves needing to know this information may wish to allocate part of their address space for this purpose.

An SBus controller may or may not check whether an atomic transaction consists of a read, followed by an optional dummy read, followed by a write. Thus, a master performing some other sequence of bus cycles during an atomic transaction may or may not receive an Error Acknowledgment. Furthermore, whether such an illegal sequence of bus cycles performs the intended behavior depends on the system. Even if no Error Acknowledgment is given, there is no guarantee that an illegal sequence of bus cycles will be performed correctly.

## Data(31:0)

Every SBus must have 32 signals, called Data(31:0) (D(31:0)), which transfers data and virtual addresses.

**Note:** The least significant bit of the data bus is D(0), and the most significant bit is D(31).

The SBus supports three primary data formats:

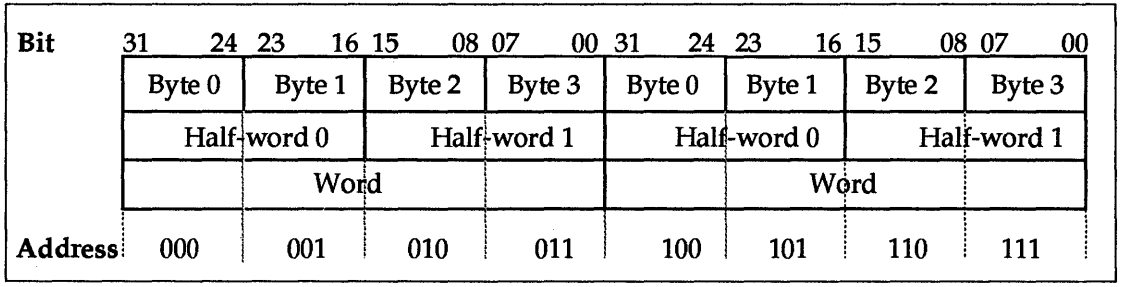
- Bytes, which consist of 8 bits of data.
- Half-words, which consist of 16 bits of data.
- Words, which consists of 32 bits of data.

The SBus also supports multi-word transfers, called burst transfers.

### Byte/Half-word Ordering and Addressing

The SBus uses what is commonly called *big-endian* addressing. As shown in the following figure, big-endian addressing means that the significance of bytes in a word or half-word decreases as the address of the bytes increase.

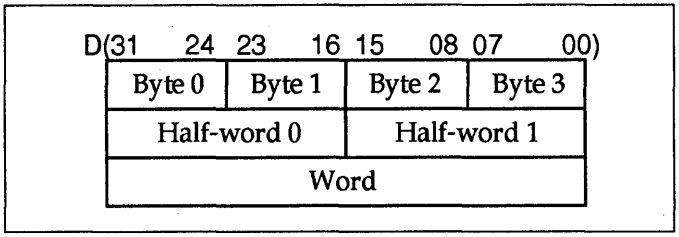
*Figure 3-7. Words, Half-words, and Bytes*



## Port Locations

The particular subset of D(31:0) over which data travels depends on the master's transfer size and the slave's Data Acknowledgment. To specify the subset, the SBus defines byte, half-word, and word ports on D(31:0) as shown in the following figure.

*Figure 3-8. Port Locations within a Word*




When a master performs a byte write, it must always place the byte data on D(31:24) which is byte port 0. In addition, the master must place a copy of the byte at the byte's natural location. If the byte's address ends in 01 (binary), a copy of the byte must be placed on D(23:16) which is byte port 1; if it ends in 10, a copy must be placed on D(15:8) which is byte port 2; and if it ends in 11, a copy must be placed on D(7:0) which is byte port 3. A master may place a copy of the byte at all four byte locations, if desired.

Similarly, when a master performs a half-word write, it must place the half-word data on D(31:16) which is half-word port 0. In addition, if the half-word address ends in 10, the master must place a copy of the data on D(15:0) which is half-word port 1. A master may place a copy of the half-word at both half-word locations, if desired.

When reading data from a slave, the location of the data depends on the slave's Data Acknowledgment. For more information, see information about "Data Acknowledgments" later in this section.

## Alignment, Wrapping, and Burst Transfers

All transfers on the SBus must be aligned to their proper address boundaries, subject to address wrapping in the case of burst transfers. An SBus master must never issue an unaligned word or half-word transfer. SBus slaves are not required to signal an error on an unaligned transfer, although they are free to do so, as long as the slave's Data Acknowledgment would have been for a size equal to or greater than the transfer size. For a further explanation of this constraint, see "Bus Sizing" later in this chapter.

In detail, data transfers must always be aligned to an address whose $\log_2$ (size of the transfer in bytes) least significant bit(s) is (are) 0. Thus:

- Bytes may be read and written at any address.
- Half-words may be read and written only from an address whose least significant bit is 0.
- A two/four/eight/sixteen word burst may be read and written only from a block whose starting address's three/four/five/six least significant bits are 0.

In the case of burst transfers, although the block itself must be properly aligned, the transfer may begin at any word within the block. The (starting) virtual address generated by the master need have only its two least significant bits be 0. This mode of addressing is called address wrapping.

During a burst transfer, it is the responsibility of the slave and master to transfer the proper word during each clock cycle. Since the slave receives only the starting address, after each word is transferred the slave must increment the address counter by 4, modulo the burst transfer size in bytes. If a slave supports burst transfers, it must implement this modulo counting.

If a slave supports a 32-byte burst, it must support a 16-byte burst as well. If it supports 64-byte bursts, it must support 32- and 16-byte bursts.

**Profile recommendation:** SBus master devices that do not use Open Boot, or SBus devices installed in systems that do not implement Open Boot, should only perform 16-byte bursts.

**Recommendation:** It is recommended that a slave supporting 16, 32, and/or 64-byte bursts also support 8-byte bursts. Moreover, the following recommendations are made:

- Use burst transfers whenever possible.

  They greatly improve overall use of the bus and bus performance.

- SBus controllers should support all burst sizes.

- A master supporting bursts to size *n* should support all bursts up to size *n*.

- Masters using bursts should be able to perform 16-byte bursts at minimum.

**Observation:** Address wrapping during burst transfers allows a CPU master to transfer the word that caused a cache miss, and then the rest of the words to fill up the cache line. Thus, the CPU can begin execution immediately without having to wait for the line to fill.

## Transfer Size: Size(2:0)

Every SBus must have three signals, called Size(2:0) (Siz(2:0)), which the current master transmits information describing the amount of data to be transferred during the bus cycle.

An SBus master must determine for each bus cycle how much data it wishes to transfer. Unless an error occurs during the middle of the transfer or the slave requests bus sizing, both the master and slave must transfer the amount of data indicated by Siz(2:0). The master must drive Siz(2:0) to its proper state during the clock cycle following the assertion of BG*. The master must keep Siz(2:0) stable until the clock cycle following BG* becoming unasserted.

CPU masters in host-based systems must assert Siz(2:0) no later than the clock cycle in which the controller asserts AS*, and keep it asserted until the clock cycle during which the controller unasserts AS*.

A master need implement only one transfer size; a slave need implement only one transfer size. However, a slave supporting 32-byte bursts must also support 16-byte bursts; a slave supporting 64-byte bursts must support 16- and 32-byte bursts.

Every SBus controller must support at least byte, half-word, word, and four word burst transfers. The SBus controller must issue an Error Acknowledgment in response to any transfer size it does not support. The controller must do this before initiating slave cycle to a particular slave. When issuing an Error Acknowledgment, it is not necessary for the controller to assert AS*. If, however, the controller does assert AS* before issuing Error Acknowledgment, it must not assert any Sel*.

All slaves must fully decode Siz(2:0), even though they may support only a subset of the transfer modes. A slave must issue an Error Acknowledgment in response to an unsupported transfer size.

The encodings in the following figure must be used for Siz(2:0).

**Note:** During ExtendedTransfers, Siz(2:0) is also used for D(62:60). For more information, see Appendix B.

*Figure 3-9.* Siz(2:0) *Encodings*

| Siz(2) | Siz(1) | Siz(0) | **Function** |
|---|---|---|---|
| 0 | 0 | 0 | Word (four byte) transfer |
| 0 | 0 | 1 | Byte transfer |
| 0 | 1 | 0 | Half-word (two byte) transfer |
| 0 | 1 | 1 | ExtendedTransfer* |
| 1 | 0 | 0 | Four Word Burst (16 bytes) |
| 1 | 0 | 1 | Eight Word Burst (32 bytes) |
| 1 | 1 | 0 | Sixteen Word Burst (64 bytes) |
| 1 | 1 | 1 | Two Word Burst (8 bytes) |

* The actual transfer size is encoded within the protocol. See Appendix B.

**Recommendation:** To work with Sun caches in SPARC-based systems, slaves designed to be cacheable must implement byte, half-word, word, four and eight word burst transfers. It is also recommended that two word bursts be implemented.

In particular, system memory is typically a cacheable device and should, therefore, implement these burst modes.

As explained in "Ack(2:0)*" later in this chapter, CPU masters which are compatible with SunOS must implement dynamic bus sizing. SBus controllers should support all sizes.

**Observation:** Many systems (for example, the Sun SPARCstation 1) do not implement all of the burst transfer modes. Thus, a master desiring to use other than four word burst transfers must be prepared to use single word transfers or four word burst transfers if the desired transfer size is not supported. Either the FCode driver or the device driver should enable the use of 8, 32, or 64 byte burst transfers.

Support for burst transfers is not only a function of the master and slave, but also of the SBus controller since it must know how many Data Acknowledgments are needed to negate BG* and AS* at the proper time.

In some SPARC-based systems, Load-double and Store-double are performed as two 4-byte transfers, instead of as a single 8-byte transfer. Thus, a slave should not depend on these SPARC instructions generating only a single bus transaction. This aspect of many implementations is an issue only for those slaves that depend on load and store double to be executed as a single atomic operation.

## Transfer Direction: Read

Every SBus must have a signal, called Read (Rd) which the current master must use to signal whether it will read data from the slave (Rd asserted) or write data to the slave (Rd unasserted). Rd must be stable, beginning with the clock cycle following the assertion of BG* (DVMA masters) or the clock cycle in which AS* is asserted (CPU masters). It must remain stable until the clock cycle following AS* or BG* becoming unasserted.

Slaves must gate their output drivers synchronously using Rd, Sel*, and AS*. In particular, a read-only slave must not drive the data lines when Rd is unasserted.

**Note:** During ExtendedTransfers, Rd is also used for D(63). For more information, see Appendix B.

## Ack(2:0)*

Every SBus must have three signals called Ack(2:0)* (Ack(2:0)*) which are driven by the currently selected slave or, if a bus timeout occurs, by the SBus controller. As a shared signal, the currently selected slave, after asserting Ack(2:0)*, must drive Ack(2:0)* to the idle (unasserted) state for one clock cycle before removing its drive. The SBus controller must terminate Ack(2:0)* with a 10 KΩ resistor to +5V.

SBus cycles are terminated by one or more acknowledgments, of which there are three general types:

- Data Acknowledgments.

  These indicate the successful transfer of data between a master and slave.

- Error Acknowledgment.

  This indicates that the attempted data transfer was unsuccessful.

- Rerun Acknowledgment.

  This indicates that the selected slave was unable to perform the requested transfer, and that the master must retry the operation.

A slave asserts an acknowledgment by driving Ack(2:0)* to the proper state for exactly one clock cycle. The slave is in complete control of when to generate an acknowledgment, subject to the constraints of bus timeouts.

When writing data to a slave, the master must drive the first datum onto the bus during the clock cycle following the virtual address (DVMA master), or in the same cycle during which AS* is asserted (CPU master). The slave may acknowledge the data as soon as one clock cycle following the assertion of AS*.

During a burst transfer, the master must drive each successive word of data onto the data lines during the clock cycle immediately following the slave's acknowledgment for the previous word. Thus, when a slave generates an acknowledgment during a particular clock cycle of a write, it is acknowledging the data on the data lines during that clock cycle.

**Note:** During ExtendedTransfers, data timing with respect to acknowledgments for writes is the same as the timing for reads. For more information, see Appendix B.

When data is read from a slave, the slave may generate an acknowledgment at any time (subject to timeout), beginning with the clock cycle following the assertion of AS*. The data corresponding to the acknowledgment must be driven onto D(31:0) for exactly one clock cycle during the clock cycle immediately following the Data Acknowledgment.

For all bus cycles (except burst transfers), after asserting Ack(2:0)* for one clock cycle, the slave must drive Ack(2:0)* to the idle (unasserted) state for exactly one clock cycle, after which the slave must stop driving (tristate) Ack(2:0)*.

During a burst transfer, the assertion of a Data Acknowledgment during a clock cycle indicates that a word of data has been accepted (writes) or will be transferred on the bus during the following clock cycle (reads). Thus, word acknowledgment can be asserted for up to 16 consecutive clock cycles in the case of a sixteen-word burst transfer. If the slave is unable to transfer data at the rate of a word per clock cycle, the slave must drive Ack(2:0)* to its idle state between each word of the transfer. In all cases, the entire burst transfer must be completed within 255 clock cycles.

A slave needs to be able to generate only one type of Data Acknowledgment. Slaves do not need to be able to generate Rerun Acknowledgment. A slave must generate an Error Acknowledgment for any transfer request it is unable to support.

Only the selected slave may drive Ack(2:0)*, except for those cases in which a bus timeout or address translation error occurs. As explained in "Bus Timeouts" later in this chapter, a slave issuing an acknowledgment of any kind must do so no later than the 255th clock cycle following the assertion of AS*.

All masters must operate properly, given any encoding of Ack(2:0)*. However, a master may be designed so that a slave may use only a particular subset of Data Acknowledgments in response to a data transfer. The master is free to treat all other Data Acknowledgments as though they were Error Acknowledgments. If a master does not support a particular type of Data Acknowledgment, it should not initiate a transfer to a slave that generates a Data Acknowledgment which the master does not support.

In the case where a slave requests bus-sizing, SBus masters must not expect to see a byte or half-word acknowledgment, since an intervening bus bridge may *hide* the bus sizing from the master. Even though a master may *know* that a particular slave will size an operation, a bus bridge may make it appear to the master as though no bus sizing occurred. For example, on a read, a bus bridge may reassemble the bytes or halfwords before passing them back to the master; on a write, the bus bridge may perform a single bus cycle with the master, even though the ultimate transfer to the slave uses bus sizing.

All masters must support Rerun Acknowledgment by retrying the bus cycle. All masters must support Error Acknowledgment. Masters must treat all reserved acknowledgments and any unsupported data acknowledgments as though they were Error Acknowledgments.

All masters must be able to handle an acknowledgment as early as the clock cycle immediately following the assertion of AS* and Sel*. A master requesting a burst transfer must be able to handle an acknowledgment every clock cycle for the length of the burst. Figure 3-11 later in this section shows the use of Ack(2:0)* for burst and non-burst transfers. The relationship of Ack(2:0)* and the data is shown, as well as the fact that Ack(2:0)* signals may be unasserted for a time during bursts, thereby allowing the slave control of the data transfer rate.

The SBus controller must monitor Siz(2:0), Rd, and Ack(2:0)* so that it is able to unassert AS* and BG* after the slave has issued the last possible acknowledgment for that bus cycle.

Ack(2:0)* must be encoded according to the following figure. Slaves must not generate the reserved encodings.

*Figure 3-10.* Ack(2:0)* *Encodings*

| Ack(2)* | Ack(1)* | Ack(0)* | **Function** |
|---|---|---|---|
| 1 | 1 | 1 | Idle/Wait |
| 1 | 1 | 0 | Error Acknowledgment |
| 1 | 0 | 1 | Byte (Data) Acknowledgment |
| 1 | 0 | 0 | Rerun Acknowledgment |
| 0 | 1 | 1 | Word (Data) Acknowledgment |
| 0 | 1 | 0 | Double-word (Data) Acknowledgment |
| 0 | 0 | 1 | Half-word (Data) Acknowledgment |
| 0 | 0 | 0 | Reserved |

**Profile recommendation:** SBus devices that do not use Open Boot, or SBus devices installed in systems that do not implement Open Boot, should check AS*, BG*, or both. If these signals are not negated following an Error Acknowledgment within a burst transfer, the device must continue to provide acknowledgments/errors until the end of the transfer. Cards that use and are installed in a host with Open Boot may optionally check to see if the host requires this type of behavior, and act accordingly.

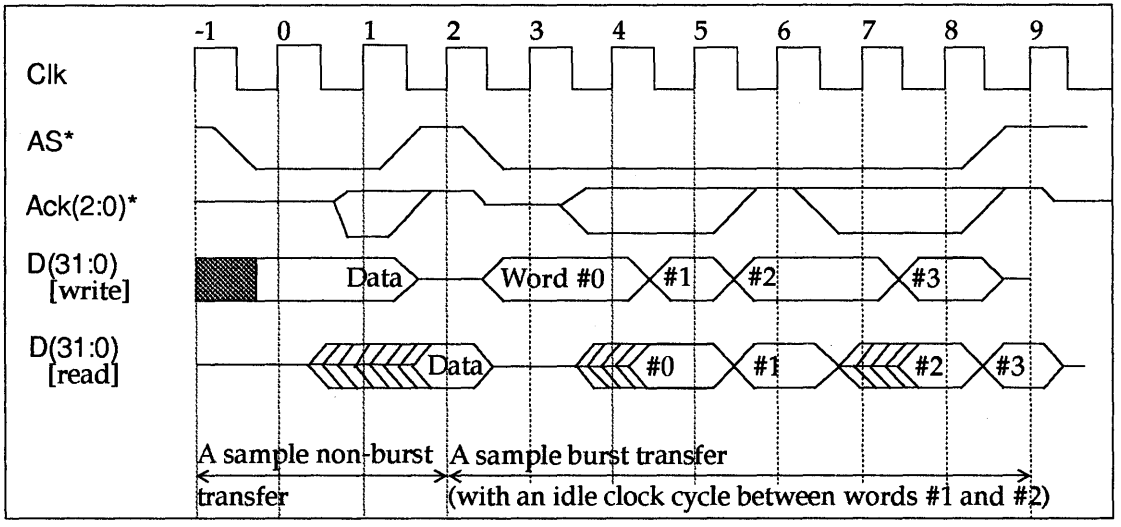
**Recommendation:** Masters should support as many transfer modes as possible to improve inter-operability. Masters must be designed to accept a word acknowledgment in response to a byte or halfword transfer, since many slaves issue only word acknowledgments.

A master could attempt to circumvent address translation protection checking by signaling a read to the SBus controller during the translation cycle, and then deassert read before the slave cycle. To detect such behavior, SBus controllers may wish to verify that the read signal from the master remains stable throughout the entire transfer (unless the transfer is an ExtendedTransfer. See Appendix B).

**Observation:** During a transfer, a slave need not continue to drive Ack(2:0)* after the slave has driven Ack(2:0)* to the idle state for a clock cycle, since bus terminators (pullups) will keep Ack(2:0)* in the idle state. However, a slave is free to actively drive Ack(2:0)* in the idle state any time it is selected during a slave cycle.

The encoding of Ack(2:0)* is such that a slave which never generates certain acknowledgments need not drive some of the Ack(2:0)* wires. For example, a slave which always responds with a word acknowledgment to all transfers need drive only Ack(2)*. This observation may be useful for saving pins on a gate array.

*Figure 3-11. Sample Burst Transfer*



## Data Acknowledgments

Three types of Data Acknowledgments are used on the SBus:

- Byte acknowledgment.
- Half-word acknowledgment.
- Word acknowledgment.

A slave need not be able to generate all three Data Acknowledgments.

A slave receiving a transfer request which it does not support must issue an Error Acknowledgment. In the case where a master requests a word or half-word transfer and the slave can perform only a half-word or byte transfer, the slave should respond with a half-word or byte acknowledgment. The master will then perform bus sizing, if it is able to do so.

A word acknowledgment is the only acceptable Data Acknowledgment for a burst transfer. Byte and half-word Data Acknowledgments must never be used.

The type of Data Acknowledgment is not allowed to change during bus sizing. Within a word, a slave must use the same acknowledgment for each port location within the word, independent of transfer size. For example, if a slave responds with byte acknowledgment to a word (or half-word) transfer at address 0, it must respond with a byte acknowledgment for each of the remaining three (or one) bytes.

A slave may issue any Data Acknowledgment in response to any transfer request, subject to the prohibition on issuing half-word or byte acknowledgments during burst transfers, or changing the acknowledgment as explained previously. The Data Acknowledgment indicates the port size of the slave, and determines where the slave must read and write data on the data lines as the following figure shows.

**Recommendation:** Although byte and half-word data is placed at the proper port location and the proper address aligned location on the data lines, it is recommended that slaves always read data from the proper port location when issuing an acknowledgment that implies use of that port location.

**Observation:** It is acceptable for a slave to respond with a word acknowledgment, even though a byte or half-word transfer was performed (similarly, half-word acknowledgment to byte transfers). In this case, the slave must read and write data to the proper address aligned location on the data lines. Also, the slave need not support byte or half-word transfers.

*Figure 3-12. Data Acknowledgment Semantics*

| | | |
|---|---|---|
| **Word acknowledgment** | | |
| | Word Transfer | Slave transfers data on all 32 bits of D(31:0) |
| | Half-word Transfer | Slave transfers data on half-word port indicated by PA(1) |
| | Byte Transfer | Slave transfers data on byte port indicated by PA(1:0) |
| **Half-word acknowledgment** | | |
| | Word Transfer | Dynamic Bus Sizing. Slave transfers data on most significant half-word D(31:16) |
| | Half-word Transfer | Slave transfers data on most significant half-word of data lines D(31:16) |
| | Byte Transfer | Slave transfers data on byte port of most significant half-word as indicated by PA(0) |
| **Byte acknowledgment** | | |
| | Word Transfer | Dynamic Bus Sizing. Slave transfers data on most significant byte D(31:24) |
| | Half-word Transfer | Dynamic Bus Sizing. Slave transfers data on most significant byte D(31:24) |
| | Byte Transfer | Slave transfers data on most significant byte D(31:24) |

## Rerun Acknowledgment

A slave may issue a Rerun Acknowledgment instead of an Error or Data Acknowledgment in response to any transfer request subject to one constraint — that, during a burst transfer, a slave may not issue a Rerun Acknowledgment if it has issued any Data Acknowledgments for the current bus cycle. The Rerun Acknowledgment must be the first and only acknowledgment for the bus cycle.

The timing for a Rerun Acknowledgment is the same as for Error and Data Acknowledgments: it must be issued no later than the 255th clock cycle following the assertion of AS*.

A slave may issue a Rerun Acknowledgment on any bus cycle that results from bus sizing. A slave is permitted to continue issuing Rerun Acknowledgments until it is able to complete the transfer. However, a slave is forbidden from issuing an infinite number of Rerun Acknowledgments. Neither the SBus controller nor SBus masters implement a mechanism for limiting the number of Rerun Acknowledgments a slave may issue. Slaves which might otherwise have the potential to issue Rerun Acknowledgments forever must implement some mechanism to avoid this possibility.

A slave cannot use rerun to control the order of accesses of various masters. The SBus controller has complete control over which master is granted access to the slave next. The controller may also insist that the master which received the original Rerun Acknowledgement from the slave be the only master granted access to that slave until the rerun cycle is completed.

For example, a slave must not issue a Rerun Acknowledgement to master A while waiting for an access from master B. This restriction prevents a deadlock resulting from conflicting Rerun Acknowledgements issued by the controller and slave.

After receiving a Rerun Acknowledgment, a master must relinquish the bus and request mastership of the bus. After obtaining mastership of the bus again, a master must perform the identical transfer which originally caused the slave to issue a Rerun Acknowledgment. During atomic transactions, this constraint may be impossible to achieve. For more information, see "Atomic Transactions" earlier in this chapter.

A master is not allowed to abandon (fail to retry) a transfer terminated by a Rerun Acknowledgment. This means that a slave is allowed to depend on the master retrying the transfer.

**Recommendation:** Slaves should limit their use of Rerun Acknowledgment. It may have a negative effect on system performance. Rerun Acknowledgment should be used only when no other hardware or software mechanism can accomplish the task.

It is preferable for a slave to issue Rerun Acknowledgment early in the bus cycle to allow other masters to access the bus. However, there is no absolute rule about when it is better to hold the bus in anticipation of completion versus issuing a Rerun Acknowledgment.

If a device has master and slave capabilities, it is recommended that they be designed so that the slave port can be accessed regardless of whether the master is enabled. This allows the master port to be disabled in software. The slave access may be delayed by some number of rerun cycles if necessary, but the designer should be careful to avoid livelock situations (dynamically hung on continuous deadlock-backoff cycles). In this way, a CPU can disable the master.

The use of Rerun Acknowledgment for busy-waiting is discouraged. It may cause undesirable system performance degradation if the slave stalls the CPU for more than a few microseconds. In almost all cases, it is preferable to have the CPU spin in an instruction loop to test the state of some bit, or have the slave issue an interrupt when it is available or needs to be serviced.

The use of buffering is encouraged as a method of avoiding Rerun Acknowledgments.

To implement a read/write atomic transaction properly, a master may need to violate the rule to retry the identical operation. A Rerun Acknowledgment during the write phase of the transaction will generally cause the master to restart the atomic transaction beginning with the read phase. Thus, slaves should be designed to work properly when a bus cycle, which was a write, is retried as a read.

Slaves using Rerun Acknowledgment to implement *split* or *disconnected* bus cycles should avoid saving state, or consider implementing some method for distinguishing between transfer requests so they can properly respond to multiple masters.

In systems where multiple masters may attempt to access the same slave concurrently and the slave can issue reruns, the SBus controller may wish to include hardware to prevent masters (other than the one being rerun) from accessing the slave. The controller would rerun these other masters (or delay AS*) while the first transfer is completed.

**Observation:** In some systems, deadlock can occur if an SBus master is in a mode preventing it from servicing a slave request as a result of its master mode requirements. An example is the situation in which a slave request requires use of an internal bus which is presently occupied as a result of an internal data transfer. The only time an SBus device can simultaneously be a master and slave is if, as master, it has been programmed to generate addresses that cause it to be accessed as a slave. In that case, the slave should issue an Error Acknowledgment or complete the transfer.

The requirement that a slave not issue an infinite number of Rerun Acknowledgments may place special requirements on devices to avoid poor system performance or deadly embrace. Deadly embrace is possible in a system in which two masters are concurrently attempting to perform slave access between themselves. If improperly designed, these two devices may issue Rerun Acknowledgments to each other forever.

The requirement that the master retry the identical transfer is designed to allow slaves to avoid saving state.

In multi-master systems including multiprocessors, if the controller lacks hardware interlocks, it may be necessary to have software prevent simultaneous access by different masters to slaves which can issue Rerun Acknowledgments. This is especially true if the slave saves state but is unable to distinguish requests from different masters.

In particular, the SBus does not support any notion of master identifier. For slaves which do not use the entire physical address space, one possibility is to have hardware and software use the convention that some number of high order address bits are the master identifier. The slave can then determine which request it is processing, and continue to issue a Rerun or Error Acknowledgment to the others until it is able to service them.

Rerun Acknowledgment can be used to implement a form of split or disconnected bus cycle. A slave with an access time greater than 10µs (40ns * 255) can use Rerun Acknowledgment to extend the transfer. The slave device starts the operation after receiving the first request, and gives Rerun Acknowledgments until it is able to supply the data. At that time, it issues a Data Acknowledgment.

In designing the SBus rerun capability, it was recognized that in some cases a master may not function properly if the slave issues a Rerun Acknowledgment where the master *needs* to read or write the data *now*. Although it seems to make sense to allow the master to abandon the transfer, it was deemed cleaner to require the master to finish the transfer, and use interrupts and status bits in internal registers to alert the CPU to the problem. Since the system has malfunctioned in either case, it was deemed desirable to keep latency issues and errors independent of Rerun Acknowledgment.

The Sun SPARCstation 1 violates this specification by implementing a Rerun Acknowledgment timeout for its CPU master bus cycles. Some slaves may need to be aware of this fact.

## Error Acknowledgment

A slave may issue an Error Acknowledgment during a bus cycle at any time an acknowledgment is allowed. In all cases, the Error Acknowledgment must be issued within the timeout period. An Error Acknowledgment aborts the transfer. Accordingly, the SBus controller must unassert AS* and BG* after receiving an Error Acknowledgment.

An Error Acknowledgment may be issued instead of Data Acknowledgment at any point in a burst transfer — that is, before any words have been transferred or after one or more words of data have been transferred. Such an Error Acknowledgment aborts the transfer. The master (slave) must not expect any more data, and the slave (master) must not send any more data.

An Error Acknowledgment may be issued during any bus cycle of a bus sizing operation. After receiving an Error Acknowledgment, the master should abort the remainder of the bus sizing operation. The slave which issued the Error Acknowledgment should not expect to receive any more transfers that are part of the bus sizing operation.

An SBus slave must issue an Error Acknowledgment in response to a transfer request it is unable to support, subject to the slave taking advantage of bus sizing when a master requests a (non-burst) transfer larger than the slave supports.

The SBus controller must issue an Error Acknowledgment whenever a bus timeout occurs. The SBus controller must issue an Error Acknowledgment in response to any transfer size it does not support. In these latter two cases, the SBus controller may or may not assert AS*.

**Recommendation:** Error Acknowledgment should not be used for *flow control.* In many high-performance systems, write-buffering may make it very difficult to signal errors in such a synchronous fashion. Thus, for example, a master receiving an Error Acknowledgment to indicate that a slave's input FIFO is full may not discover this until several additional writes have caused the FIFO to overrun. Alternative techniques, such as the use of high/low watermarks and interrupts, should be considered.

**Observation:** In general, the error mechanism on the SBus is designed on the assumption that the errors for which Error Acknowledgment will be used occur rarely. Thus, no attempt is made to make it *easy* to determine the source of an error. Individual masters and slaves are responsible for capturing whatever state is required to make error explanation and recovery feasible.

Error Acknowledgment indicates that the requested transfer cannot be performed correctly. Asynchronous errors not associated with a particular transfer should be reported using interrupts.

Even though this book specifies that in certain cases slaves should issue an Error Acknowledgment, such as unimplemented transfer modes, these cases should occur rarely in systems which are operating correctly.

In some systems, the SBus controller may generate an interrupt to the CPU if the controller receives an Error Acknowledgment on a non-CPU DVMA cycle. In these cases, the controller may wish to keep a copy of the original virtual address as an aid to error tracking.

## Bus Timeouts

By definition, every SBus cycle terminates as a result of one or more acknowledgments as explained previously. In the case where a particular Sel* is asserted and no slave responds (for whatever reason), it is the responsibility of the SBus controller to generate an Error Acknowledgment to terminate the bus cycle.

To prevent a bus timeout, the selected slave must generate its own acknowledgment no later than the 255th clock cycle following the assertion of AS*. If the SBus controller does not receive an acknowledgment by the 256th clock edge, it must generate an Error Acknowledgment within two clock cycles (although it may generate Error Acknowledgment within one clock cycle if it wishes).

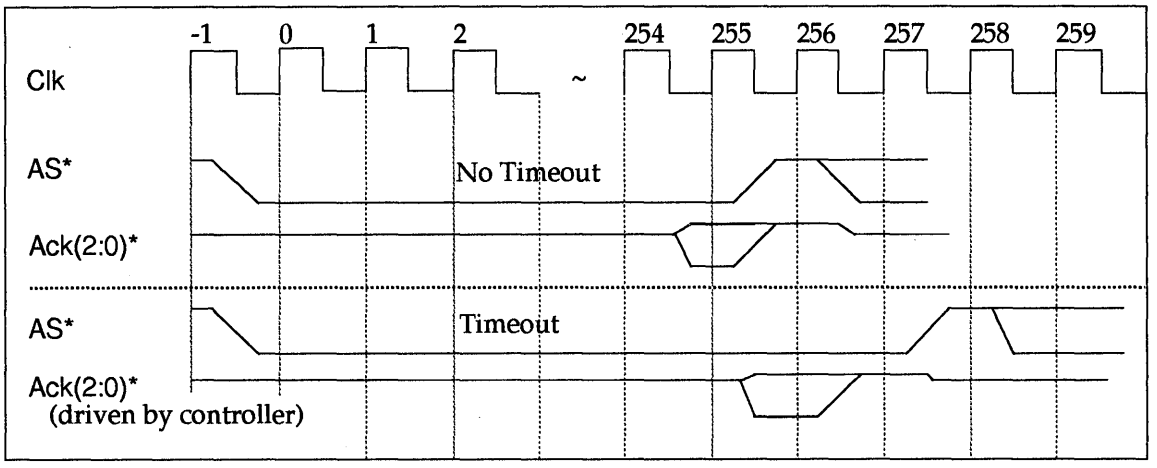
The SBus controller must do the following in sequence:

1. Drive Ack(2:0)* for exactly one clock cycle.
2. Drive Ack(2:0)* to the idle state for one clock cycle.
3. Remove its drive.

Slaves unable to respond within the timeout period, but which do not wish to abort the transfer, must use Rerun Acknowledgment as explained previously. A slave which does not respond with an acknowledgment within the timeout period is forbidden from responding to the current bus cycle.

A slave must issue an Error Acknowledgment and not depend on bus timeout if a master addresses an unused portion of the slave's address space.

*Figure 3-13. Bus Timeouts*




**Recommendation:** The purposeful use of timeout by a slave is strongly discouraged. Slaves unable to fulfill a transfer request should issue an Error Acknowledgment.

**Observation:** In a properly running system, bus timeout should never occur, except during system configuration. However, in systems allowing user mapping of the bus (for example, SunOS), it may be possible for user-level code to access nonexistent SBus devices.

The bus timeout frees masters from keeping track of clock cycles. Slaves which may occasionally timeout need to be certain that they do not accidentally drive Ack(2:0)* at the wrong time.

There are no predefined semantics associated with an Error Acknowledgment, except for aborting the transfer.
If appropriate, it is the master's responsibility to issue an interrupt to the CPU.

## LateError*

Every SBus must include a signal called LateError* (LErr*) which provides SBus slaves with a mechanism to pipeline error checking during a data transfer.

If LErr* is asserted by a slave, it must be asserted exactly two clock cycles after the corresponding acknowledgment, and for exactly one clock cycle. During the clock cycle after it asserts LErr*, the slave must drive LErr* to its unasserted (high) state.

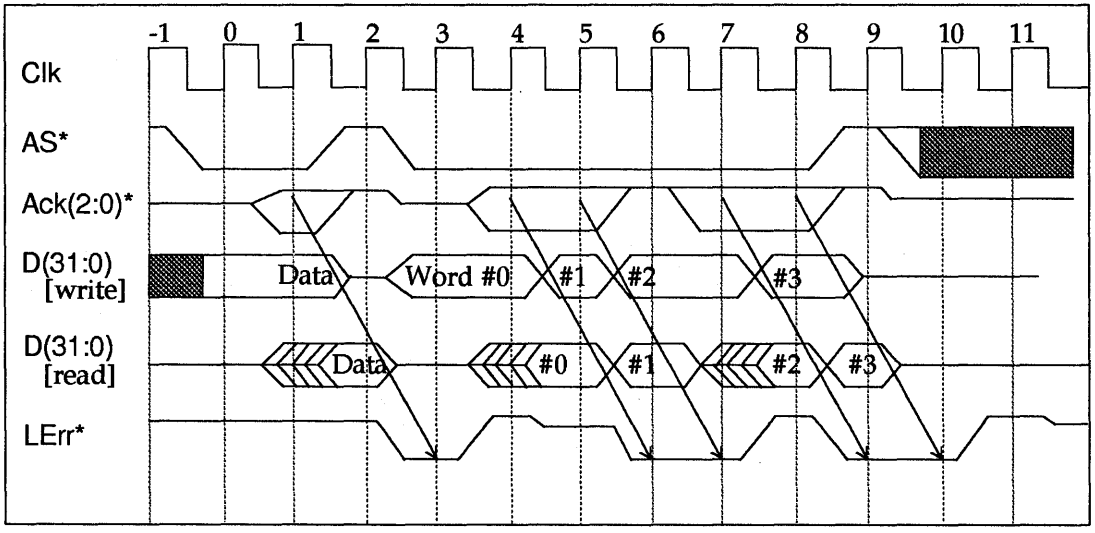
LErr* must not be asserted at any other time — that is, LErr* must be preceded by an acknowledgment. LErr* may be asserted at a time when the receiving master may have started a new bus cycle or is no longer owner of the bus. Thus, SBus devices must be careful to always associate LErr* with the preceding acknowledgment.

LErr* may be asserted only by the currently selected slave.

When asserted during a burst transfer, LErr* does not abort the bus cycle. The bus master must complete the entire transfer.

Like Error Acknowledgment, no specific interpretation of LErr* is implied by this specification; the action taken in response to the assertion of LErr* is specific to the current master.

*Figure 3-14.* LErr* *Timing*



**Recommendation:** SBus devices should use LErr* only in those cases where it is impossible to signal the error using Error Acknowledgment.

Although LErr* may follow a Rerun or Error Acknowledgment, this usage of LErr* is discouraged. It is unclear what LErr* means in such cases.

In many systems (for example, SPARCstation 1), LErr* is used to signal memory errors. Masters performing DVMA transfers to system memory should check for the assertion of LErr*. A master receiving a LErr* after a Data Acknowledgment should assume that the preceding data transfer occurred incorrectly, and that the master should discard the data in the case of a read, or try to re-write the data in the case of a write (avoid infinite retries). If the master reports an error to the CPU, the master should retain as much state as possible, particularly the offending virtual address, to facilitate error handling.

**Observation:** The LErr* mechanism follows from the general SBus philosophy of errors: because errors occur infrequently, detecting and reporting them should not interfere with the performance of the system.

LErr* provides a mechanism for memory subsystems to perform error checking without impeding data transfer. For example, a word of data can be transferred at the same time parity checking begins. The results of the parity check can be reported using LErr* during the following clock cycle.

In some systems (for example, the Sun SPARCstation 1), the SBus controller, after determining that LErr* has been asserted, may independently generate an interrupt to the CPU, even though the CPU was not master of the bus. In this case, the controller may wish to keep a copy of the virtual address to facilitate error handling.

An SBus slave which never drives LErr* may leave it unconnected.

## Bus Sizing

Bus sizing allows a master to initiate a word or half-word transfer to a slave device without regard to whether the slave supports a transfer size that large. This design allows a master to treat the slave as though it were a word or half-word device, even though the slave may implement only half-word or byte transfers.

Bus sizing can occur only during word or half-word transfers. It cannot occur during any burst transfers. A slave unable to support a burst transfer must issue an Error Acknowledgment if a master attempts such a transfer. A slave must never issue a byte or half-word acknowledgment in response to a burst transfer.

Support for bus sizing is the responsibility of the master. Except as recommended later in this section, masters need not support bus sizing. However, if a master does not support bus sizing, it should not initiate a transfer that might require bus sizing — for example, a word transfer to a byte slave.

Unlike burst transfers, in which multiple words of data are transferred in a single bus cycle, during bus sizing each byte or half-word must be transferred using an independent bus cycle. The first element of data (the one that invoked bus sizing) must always be transferred as part of the original bus cycle.

Thus, a half-word must be transferred in a total of two bus cycles, whereas a word must be transferred in two bus cycles (slave responds with two half-word acknowledgments) or four bus cycles (slave responds with four byte acknowledgments). The master must generate the correct address for the data being transferred during each bus cycle of the transfer. Masters should change only the two least significant address bits in follow-on bus cycles.

A slave must respond with the same Data Acknowledgment for each bus cycle of a bus sizing operation. This restriction means that a slave that responds with a byte acknowledgment for the first byte of the transfer must respond with a byte acknowledgment for each of the remaining transfers. Within a word, a slave must use the same acknowledgment for each port location within the word, independent of transfer size. If a slave responds with byte acknowledgment to a word (or half-word) transfer at address 0, it must respond with a byte acknowledgment for each of the remaining three (or one) bytes. The important effect of this rule is that the type of Data Acknowledgment is not allowed to change during bus sizing.

A master may abort the bus sizing operation after any cycle. However, if the slave issues a Rerun Acknowledgment, the master must rerun the current bus cycle. The master must not restart the transfer at the original bus cycle (except as explained previously for atomic transactions).

The data port location is determined, as always, by the slave's Data Acknowledgment. The type of Data Acknowledgment returned by a slave may not depend on the transfer size; it must be a function of its own data path width. A slave must treat every cycle individually, with no retained state about whether previous bus sizing cycles have occurred.

During the follow-on bus cycles, the master may keep the Siz(2:0) signal set at the original size. This results in follow-on cycles that appear to be unaligned transfers. Since the Data Acknowledgment returned by the slave must be the same for each port location within the word, these cycles are completed with the same acknowledgment used by the slave for the first transfer.

Since a slave has no knowledge of atomic transactions, bus sizing may occur during atomic transactions. However, because atomicity is not guaranteed if bus sizing occurs during an atomic transaction, masters should avoid atomic transactions to slaves which bus size. For more information, see "Atomic Transactions" earlier in this chapter.

**Recommendation:** The use of bus sizing during atomic transactions is discouraged. It dramatically increases the time during which a master has exclusive use of the bus. However, in some cases, software may not have any way of knowing that bus sizing is occurring (other than prior knowledge about the slave) and, thus, cannot prevent it from occurring during atomic transactions.

CPU masters intended to be compatible with SunOS must support dynamic bus sizing for both byte and half-word devices.

Bus bridges should never initiate bus sizing, and should always acknowledge with the size requested by the master. Thus, bus bridges need to support all transfer sizes and acknowledgments.

**Observation:** Slaves can use bus sizing to reduce software complexity. For example, an 8-bit frame-buffer that is otherwise functionally identical to a 32-bit frame-buffer can use the 32-bit software without modification.

A slave requires no special hardware to take advantage of bus sizing.

In many systems, the CPU is the only master which implements bus sizing. However, other SBus masters may implement bus sizing if desired (this is entirely up to the master). The fact that the CPU implements bus sizing does not help any other master implement bus sizing.

## Interrupts

Every SBus must have seven open-drain interrupt lines, called IntReq(7:1)* (IntReq(7:1)*) which SBus slaves can use to asynchronously signal the CPU.

Any SBus slave may assert one or more of IntReq(7:1)* at any time, subject to system configuration considerations. A slave must drive the interrupt lines using open-drain output drivers. Unlike other shared signals, interrupt lines are not driven to their unasserted state by the slave. After an interrupt has been serviced, the asserting slave must stop driving the interrupt line (unassert its output). The SBus controller must pull up each of IntReq(7:1)* with 10 KΩ resistor to +5V.

The slave may assert and unassert interrupts without regard for setup and hold times with respect to Clk.

After asserting an interrupt, the slave must set a bit in an internal register (which is readable by the CPU) to indicate that the slave is generating an interrupt at this level. Either the act of reading this bit must cause the slave to stop asserting the interrupt, or the slave must include some other CPU-accessible mechanism to clear the interrupt. Slaves must not unassert an interrupt until polled by the CPU.

By convention, IntReq(7)* is the highest priority interrupt, and IntReq(1)* is the lowest priority interrupt.

**Recommendation:** When the event causing the slave to generate an interrupt is naturally synchronous with Clk, the slave should meet standard SBus setup and hold times with respect to Clk to avoid any chance of metastable behavior. SBus controllers should be designed so that meeting SBus setup and hold times removes any opportunity for metastable behavior. To create a synchronous interrupt signal, a slave device whose interrupts are naturally asynchronous to Clk should not include a synchronizer.

The interrupt synchronizer on the SBus controller should be carefully designed to keep the probability of metastable behavior low.

Slave devices should include the capability to disable interrupts. After reset, interrupts should be disabled.

**Observation:** Interrupts provide a mechanism for SBus devices to interrupt the CPU. However, it is up to the system designer to determine how this is accomplished. The assignment and processing of interrupts is system-specific.

Interrupts on the SBus are allowed to be asynchronous, since they are often generated in response to external unsynchronized events. System reliability is improved by having a single resource, the SBus controller, perform synchronization because the controller can be designed to have known failure probabilities that reflect overall system MTBF requirements.

## Other Timing Diagrams

This section contains additional timing diagrams. These diagrams follow directly from the specification, and are included to help illustrate various features of the SBus. They represent a small fraction of the possible bus sequences.

*Figure 3-15. DVMA Cycle with Wait States*

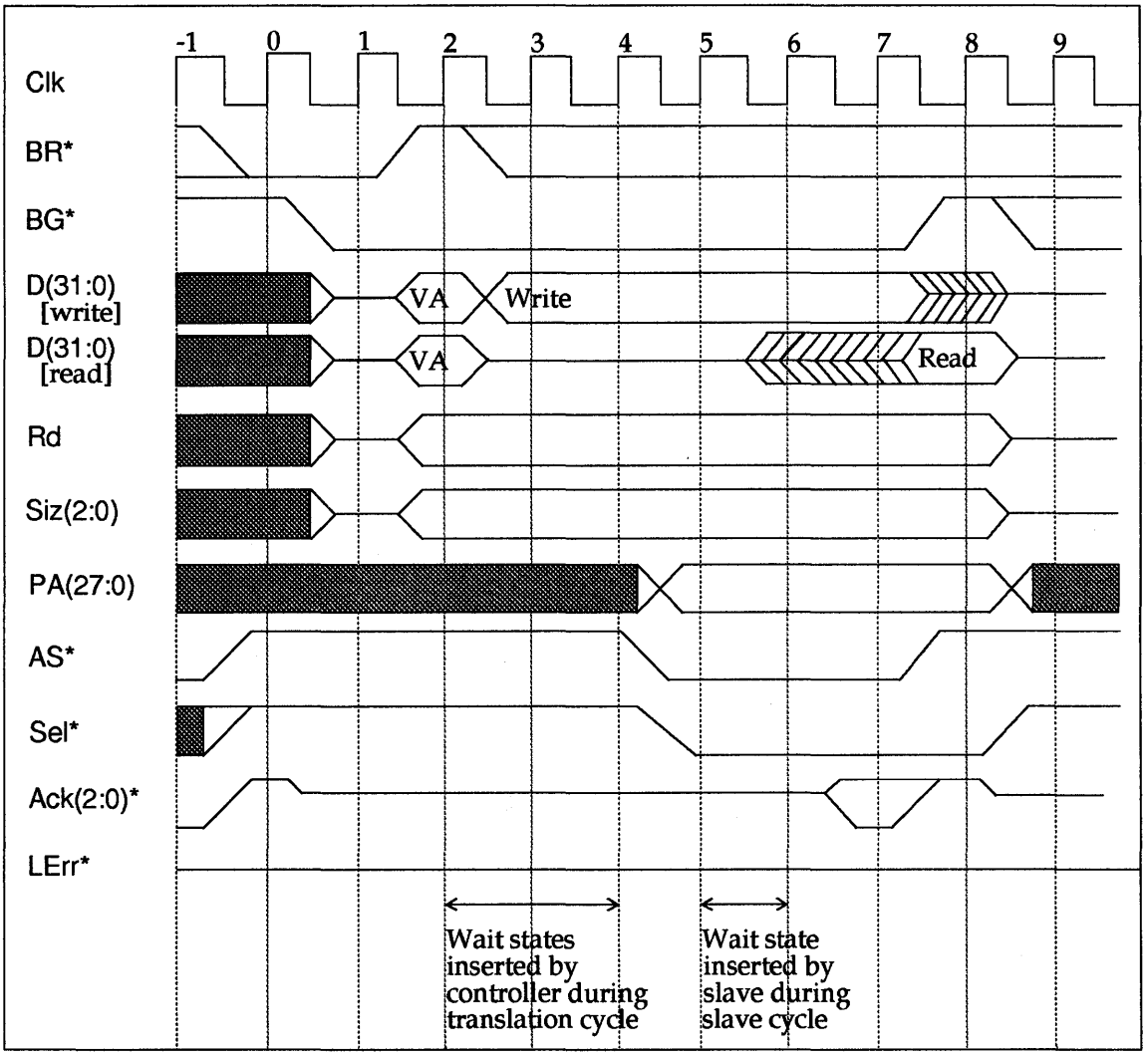
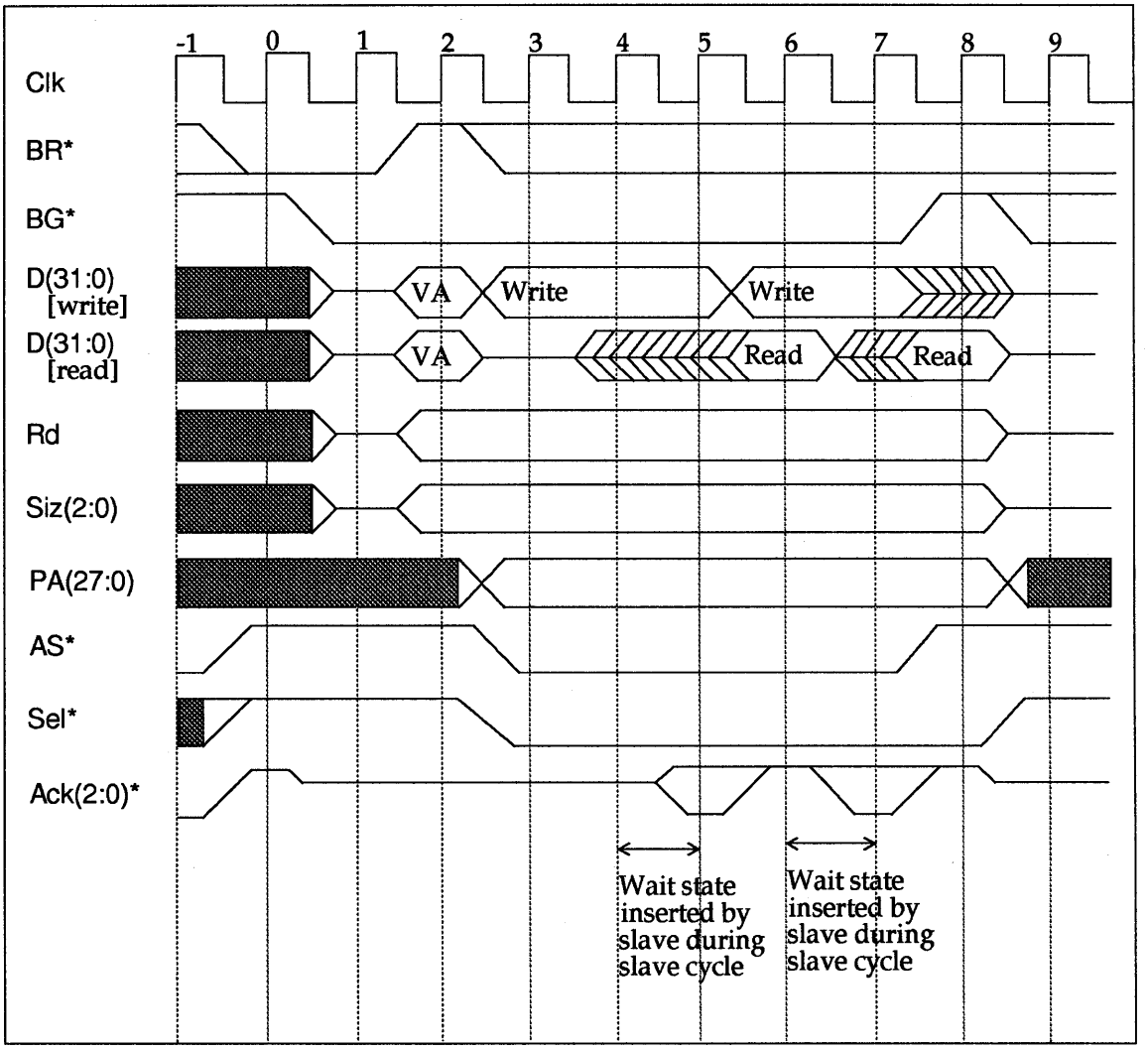


*Figure 3-16. DVMA Burst Cycle with Wait States*



# 4

# *Electrical and Mechanical Design*

This chapter describes the specification for the electrical and mechanical operation of SBus slaves, masters, and controllers. It also includes SBus profiles, which contain considerations which might affect the electrical and mechanical design of the SBus.

## SBus Profiles

SBus profiles define a minimum SBus card/controller feature set which guarantees plug compatibility between SBus cards and hosts. The features affected by SBus Profiles include:

- 25- versus 28-bit physical address widths.
- Burst sizes.
- Data parity.
- Error handling.
- Bus timeouts.
- 64-bit SBus.

The effect on specific SBus specification requirements by the use of SBus Profiles are pointed out in the affected sections of this book. The effect is explained in a special note called a *Profile recommendation.*

SBus Profiles coexist with, but do not require the use of, Open Boot on an SBus card or host. Whereas SBus Profiles specify a minimum feature set that must be implemented on both SBus hosts and cards, Open Boot provides the mechanism to expand SBus hosts and cards.

The following SBus Profile matrix shows the relationship between SBus cards and hosts which do/do not use Open Boot, in relation to their respective use of profiles and extension of their feature set beyond this.

*Figure 4-1. SBus Profile Matrix*

| Solution | Host with Open Boot | Host without Open Boot |
|---|---|---|
| **SBus card uses Open Boot** | SBus card may require any feature in the profile. Other features must be optional. | SBus card should default to the profile feature set. |
| **SBus card does not use Open Boot** | SBus card should be built along the profile guidelines. | |

## Electrical Design

The SBus is neither strictly a TTL bus nor a CMOS bus. It is designed to be compatible with several families of CMOS logic including, but not limited to, CMOS gate arrays with TTL-level pads.

### Operating Range

SBus systems and expansion cards must operate over an ambient temperature range, Ta, of 0 to +70 degrees centigrade. System designers must provide adequate ventilation or forced airflow to maintain this requirement.

### Power

SBus expansion devices and systems must conform to the following power supply conditions per slot. Expansion devices occupying multiple slots may consume the amount of power specified per expansion connector. Thus, a two-slot SBus device may consume 4A at +5V. SBus expansion devices must connect to all five +5V lines, and to all seven Gnd lines. Current should be drawn equally through all pins.

The average currents (Icont) specified for the +5V, +12V, and -12V supplies include any transient or peak currents (Ipeak). Cards drawing transient currents greater than the average must be designed to draw a quiescent current low enough to make the 500mS time-averaged value no more than the value given for Icont in the following figure.

No duty cycle is specified for the instantaneous peak currents. This is intentional: it is unnecessary as long as the time-averaged current is not exceeded, and adequate bypassing exists in the system and on the expansion card.

*Figure 4-2. Power Parameters*

| Parameter | Condition | Symbol | Min. | Max. | Unit |
|---|---|---|---|---|---|
| +5V Supply | I = 2A | +5V | 4.75 | 5.25 | V |
| +12V Supply | I = 30ma | +12V | 11.25 | 12.75 | V |
| -12V Supply | I = -30ma | -12V | -12.75 | -11.75 | V |
| Continuous Ripple, +5V | I = 2A | $V_{R5}$ | -0.1 | +0.1 | V |
| Continuous Ripple, +12V | I = +30ma | $V_{R+12}$ | -0.25 | +0.25 | V |
| Continuous Ripple, -12V | I = -30ma | $V_{R-12}$ | -0.25 | +0.25 | V |
| Continuous Current 5V | V = 5V nominal | $I_{cont5}$ | | 2.0* | A |
| Continuous Current +12V | V = +12V nominal | $I_{cont+12}$ | | .03* | A |
| Continuous Current -12V | V = -12V nominal | $I_{cont-12}$ | | -.03* | A |
| Peak Current 5V | $T_{peak}$ <= 1 mS | $I_{peak5}$ | | 3 | A |
| Peak Current +12V | $T_{peak}$ <= 1 mS | $I_{peak+12}$ | | .05 | A |
| Peak Current -12V | $T_{peak}$ <= 1 mS | $I_{peak-12}$ | | -.05 | A |

*Icont averaged over any 500mS interval.

**+5V**

An SBus expansion card must not draw more than 2 amperes average at +5 volts with respect to ground.

The SBus controller must guarantee that the +5V supply is within ±0.25V.

SBus expansion cards need to provide adequate power supply decoupling as a function of the current they draw.

An SBus expansion device must be able to tolerate negative or positive voltage spikes of 1 volt. The duration of such spikes must not exceed 1 µs.

**+/- 12 Volts**

An SBus expansion card must not draw more than 30 mA average at +12 volts with respect to ground. An SBus expansion device must not draw more than -30 mA average at -12 volts with respect to ground.

## Capacitive Loading

Because the SBus is designed to be compatible with CMOS devices (which are capable of only modest output drive), it is necessary to specify a maximum capacitive load per signal. In a properly designed SBus system, each expansion device must contribute no more than 20 pF per signal per expansion connector. This restriction includes the capacitive effects of any connectors and printed circuit board traces associated with the device.

An SBus device should not connect more than a single input to any SBus signal. Signals having a fan-out greater than 1 should be buffered by the device.

*Figure 4-3. Capacitive Loading*

| Parameter | Condition | Symbol | Max. |
|---|---|---|---|
| Loading per signal per SBus device | | $C_{ss}$ | 20 pF |
| Total loading per signal | $F_{Clock}$ <= 20 MHz | $C_{ts}$ | 160 pF |
| | 20 < $F_{Clock}$ <= 25 MHz | $C_{ts}$ | 100 pF |

## Stub Length

Traces for SBus signals on an SBus card should be as short as possible, and in all cases be less than 50.8 millimeters (approximately 2-inches in length).

## Signal Termination

The SBus is designed to work only over a small physical distance in which rise times are long compared with propagation delay. In such an environment, signals do not usually behave as transmission lines and, therefore, termination is not necessary except where noted specifically.

As long as SBus leakage current or drive requirements are not exceeded, SBus masters and slaves may have pullups, pulldowns, or other termination; SBus controllers may use holding amplifiers.

### Ack(2:0)* and LateError*

The SBus signals Ack(2:0)* (Ack(2:0)*) and LateError* (LErr*) must be driven to their unasserted state before being undriven. Bus termination for these signals need only maintain the state.

The SBus controller must terminate each of these signals with a 10 KΩ ±10% resistor connected to the +5V supply.

### IntReq(7:1)*

The shared SBus interrupt lines,IntReq(7:1)* (IntReq(7:1)*) do not conform to the SBus principle of being driven to their unasserted state before being undriven. The SBus controller must terminate each shared interrupt line with a 10 KΩ ±10% resistor connected to the +5V supply.

### Data(31:0), Size(2:0), and Read

To prevent excessive power dissipation as a result of floating outputs, the SBus controller must terminate each of the data lines with a 10 KΩ resistor connected to the +5V supply. As an alternative, holding amplifiers may be used.

SBus controllers supporting ExtendedTransfers must connect Read (Rd) and Size(2:0) (Siz(2:0)) to ground using a 2 KΩ resistor, instead of to the +5V supply; and Siz(1:0) must be terminated to the +5V supply using a 10 KΩ resistor. As an alternative, holding amplifiers may be used.

## DC Parameters

SBus signals are neither strictly TTL compatible nor CMOS compatible. As shown in the following figure, SBus signals are designed to use TTL-like voltage levels while consuming minimal static current. These parameters are compatible with CMOS gate arrays which have TTL compatible input and output pads, as well as other standard families of components.

SBus signals must *not* be driven or received using ordinary TTL circuitry as found in standard 7400, 74LS00, 74S00, and 74F00 families of devices, because leakage currents, pin capacitance, or other parameters may be incompatible with SBus.

SBus signals should not be received using CMOS/NMOS input thresholds. Inputs must be sensitive to what are commonly referred to as *TTL voltage levels.*

*Figure 4-4. DC Parameters*

| Parameter | Condition | Symbol | Min. | Max. | Unit |
|---|---|---|---|---|---|
| Input Low Voltage | | $V_{IL}$ | | 0.8 | V |
| Input High Voltage | | $V_{IH}$ | 2.0 | | V |
| Output Low Voltage | $I_{OL}$ = 4.0 mA | $V_{OL}$ | -0.4 | 0.4 | V |
| Output High Voltage | $I_{OH}$ = 2.5 mA | $V_{OH}$ | 2.4 | 5.5 | V |
| Input Leakage Current | $V_{In}$ = -.5V to 5.5V | $I_{IL}$ | -30 | 30 | mA |
| Output Leakage Current (Driver turned off) | $V_{I/O}$ = -.5V to 5.5V | $I_{IL}$ | -30 | 30 | mA |

## AC Parameters

To provide adequate design margin, all SBus signal drivers must be capable of meeting the timing specifications shown in the following figure when driving the maximum capacitive load.

All SBus devices must be capable of operating across the entire allowable clock range.

Rise and fall times are measured from the 10% to 90% points for worst case logic levels. Setup, hold, and delay times are measured from midpoint of the Clock (Clk) transition to the midpoint of the signal transition — that is, midway between .4V $V_{OL}$ and 2.4V $V_{OH}$.

All times are specified with respect to the SBus connector. Any additional times due to trace or logic delays in the expansion card or host must be added or subtracted by designers as appropriate. These additional times are not reflected in the figures.

*Figure 4-5. AC Parameters*

| Parameter | Condition | Symbol | Min. | Max. | Unit |
|---|---|---|---|---|---|
| Clk frequency | | $F_{Clk}$ | 16.67 | 25 | MHz |
| Clk period | | $T_{CP}$ | 40 | 60 | ns |
| Clk high time | $F_{Clk}$= 16.67 to 25 MHz | $T_{CH}$ | 17 | | ns |
| Clk low time | $F_{Clk}$ = 16.67 to 25 MHz | $T_{CL}$ | 17 | | ns |
| Clk skew | $C_L$ = 160 pF | $T_{CS}$ | 0 | 2.5 | ns |
| Clk rise and fall time | $C_L$ = 160 pF | $T_{CR}$, $T_{CF}$ | 1 | 3 | ns |
| IntReq(7:1)* fall time | $C_L$ = 160 pF, $R_L$ = 10 KW | $T_F$ | 5 | 20 | ns |
| IntReq(7:1)* rise time | $C_L$= 160 pF, $R_L$ = 10 KW | $T_{IR}$ | 5 | 1200 | ns |
| Other signals, rise/fall | $C_L$ = 160 pF, $R_L$ = 1 KW | $T_R$, $T_F$ | 5 | 20 | ns |
| Rising edge of Clk to output valid @ 20 MHz | $F_{Clk}$ <= 20 MHz $C_L$ = 160 pF | $T_{OD20}$ | 2.5 | 32.5 * | ns |
| Rising edge of Clk to output valid @ 25 MHz | 20 < $F_{Clk}$ <= 25 MHz, $C_L$ = 100 pF | $T_{OD25}$ | 2.5 | 22.5 | ns |
| Output hold time after rising edge of Clk | $C_L$ = 0 pF | $T_{OH}$ | 2.5 | | ns |
| Rising edge of Clk to Output Z | | $T_Z$ | | $T_{CP}$-5 | ns |
| Input setup time before rising edge of Clk | $C_L$ = 160 pF | $T_{IS}$ | 15 | | ns |
| Input hold time | $C_L$ = 160 pF | $T_{IH}$ | | 0 | ns |

*This number applies to systems only, which may be designed for operation at or below 20 MHz. Cards must be designed for 25 MHz operation to ensure maximum interoperability.

# Mechanical Design

A conforming SBus system need not have any expansion capabilities. SBus systems that have expansion capabilities must adhere to the mechanical specifications in this section.

An SBus expansion card consists of an expansion connector, a printed circuit board, and a backplate. An external I/O connector may be mounted on the backplate, as appropriate, provided it does not violate the mechanical specifications in this section.

All measurements are in millimeters unless otherwise indicated.

## Expansion Connector

The SBus uses a high-density 96-pin connector. Expansion cards use a male connector mounted on the *solder* side of the board. Motherboards use a female connector mounted to allow proper mechanical support and electrical shielding.

Double-width cards must have two expansion connectors.

Expansion connectors may be keyed or unkeyed. A keyed connector is identical to the unkeyed connector, except for the addition of a small plastic tab to prevent the connector from being incorrectly loaded into the board at manufacturing time

The following three figures provide information about the expansion connector and connector pinout. The subsequent two figures show the mechanical details of the keyed connector (the unkeyed connector is not shown because it is upward compatible with the keyed connector).

**Recommendation:** It is recommended that SBus expansion cards and motherboards be laid-out with the keyed connector PCB mounting hole pattern. This way, the keyed or unkeyed expansion connectors may be used.

**Observation:** The following vendors supply SBus expansion connectors. Connectors are also available from other vendors, and can be used as long as they meet the mechanical specifications described on the following pages.

*Figure 4-6. SBus Expansion Connectors*

| Manufacturer | Connector Gender | Mounting Hole Pattern | Part Number |
|---|---|---|---|
| Honda | Male | Unkeyed | PCS-96MD |
| | | Keyed | * |
| | Female | Unkeyed | PCS-96FD2 |
| | | Keyed | PCS-96FD2KP |
| Fujitsu | Male | Unkeyed | FCN-234P096-G0 |
| | | Keyed | FCN-234P096-G/Y |
| | Female | Unkeyed | FCN-234J096-G0 |
| | | Keyed | FCN-234J096-G/U |

*The part number for this connector has not been assigned as of the publication of this book. For information, please contact the SBus Technical Support Group at Sun Microsystems, Inc.

This specification makes no recommendation about the suitability of parts from these or any other vendors for a particular application.

*Figure 4-7. Expansion Connector Pinout*

| | | |
|---|---|---|
| 01. Gnd | 33. PA(06) | 65. D(18) |
| 02. BR* | 34. PA(08) | 66. D(20) |
| 03. Sel* | 35. PA(10) | 67. D(22) |
| 04. IntReq(1)* | 36. Ack(0)* | 68. Gnd |
| 05. D(00) | 37. PA(12) | 69. D(24) |
| 06. D(02) | 38. PA(14) | 70. D(26) |
| 07. D(04) | 39. PA(16) | 71. D(28) |
| 08. IntReq(2)* | 40. Ack(1)* | 72. +5V |
| 09. D(06) | 41. PA(18) | 73. D(30) |
| 10. D(08) | 42. PA(20) | 74. Siz(1) |
| 11. D(10) | 43. PA(22) | 75. Rd |
| 12. IntReq(3)* | 44. Ack(2)* | 76. Gnd |
| 13. D(12) | 45. PA(24) | 77. PA(01) |
| 14. D(14) | 46. PA(26) | 78. PA(03) |
| 15. D(16) | 47. DtaPar | 79. PA(05) |
| 16. IntReq(4)* | 48. -12V | 80. +5V |
| 17. D(19) | 49. Clk | 81. PA(07) |
| 18. D(21) | 50. BG* | 82. PA(09) |
| 19. D(23) | 51. AS* | 83. PA(11) |
| 20. IntReq(5)* | 52. Gnd | 84. Gnd |
| 21. D(25) | 53. D(01) | 85. PA(13) |
| 22. D(27) | 54. D(03) | 86. PA(15) |
| 23. D(29) | 55. D(05) | 87. PA(17) |
| 24. IntReq(6)* | 56. +5V | 88. +5V |
| 25. D(31) | 57. D(07) | 89. PA(19) |
| 26. Siz(0) | 58. D(09) | 90. PA(21) |
| 27. Siz(2) | 59. D(11) | 91. PA(23) |
| 28. IntReq(7)* | 60. Gnd | 92. Gnd |
| 29. PA(00) | 61. D(13) | 93. PA(25) |
| 30. PA(02) | 62. D(15) | 94. PA(27) |
| 31. PA(04) | 63. D(17) | 95. Reset* |
| 32. LErr* | 64. +5V | 96. +12V |

*Figure 4-8. Signal Location*

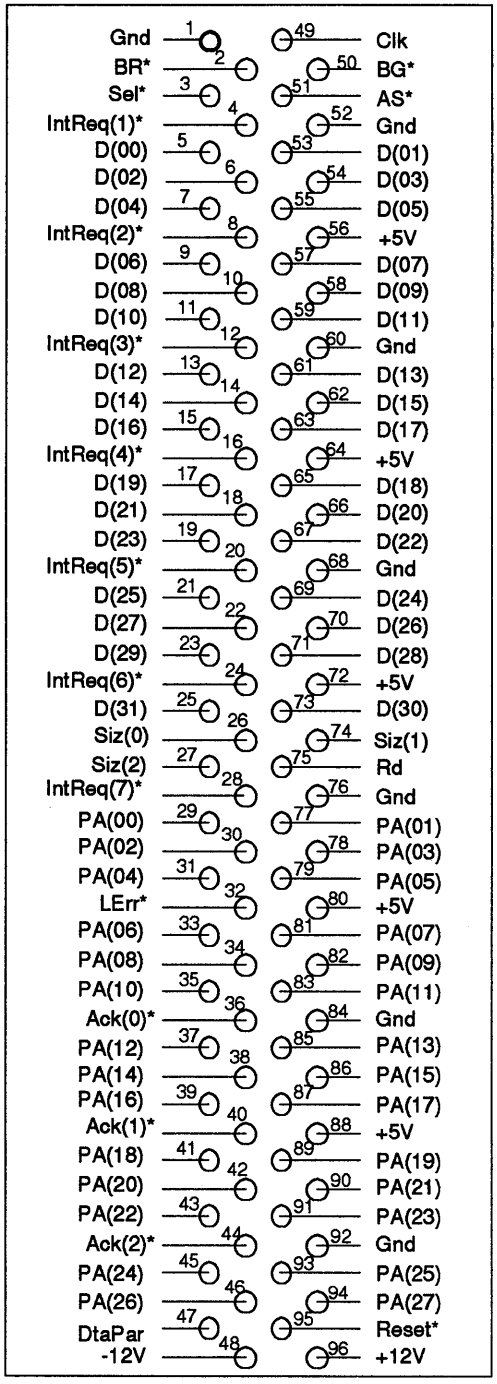


*Figure 4-9. Male Expansion Connector*

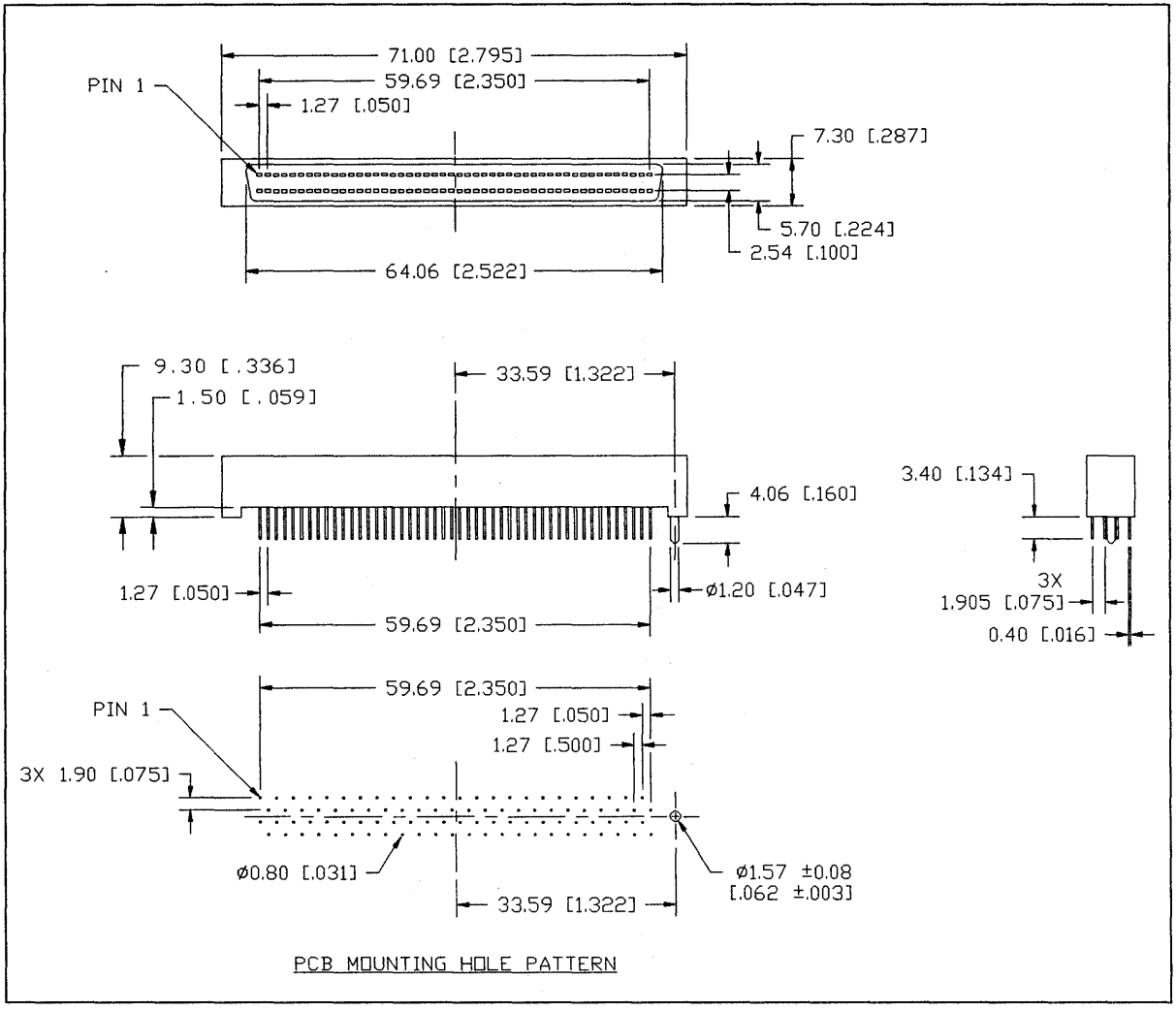
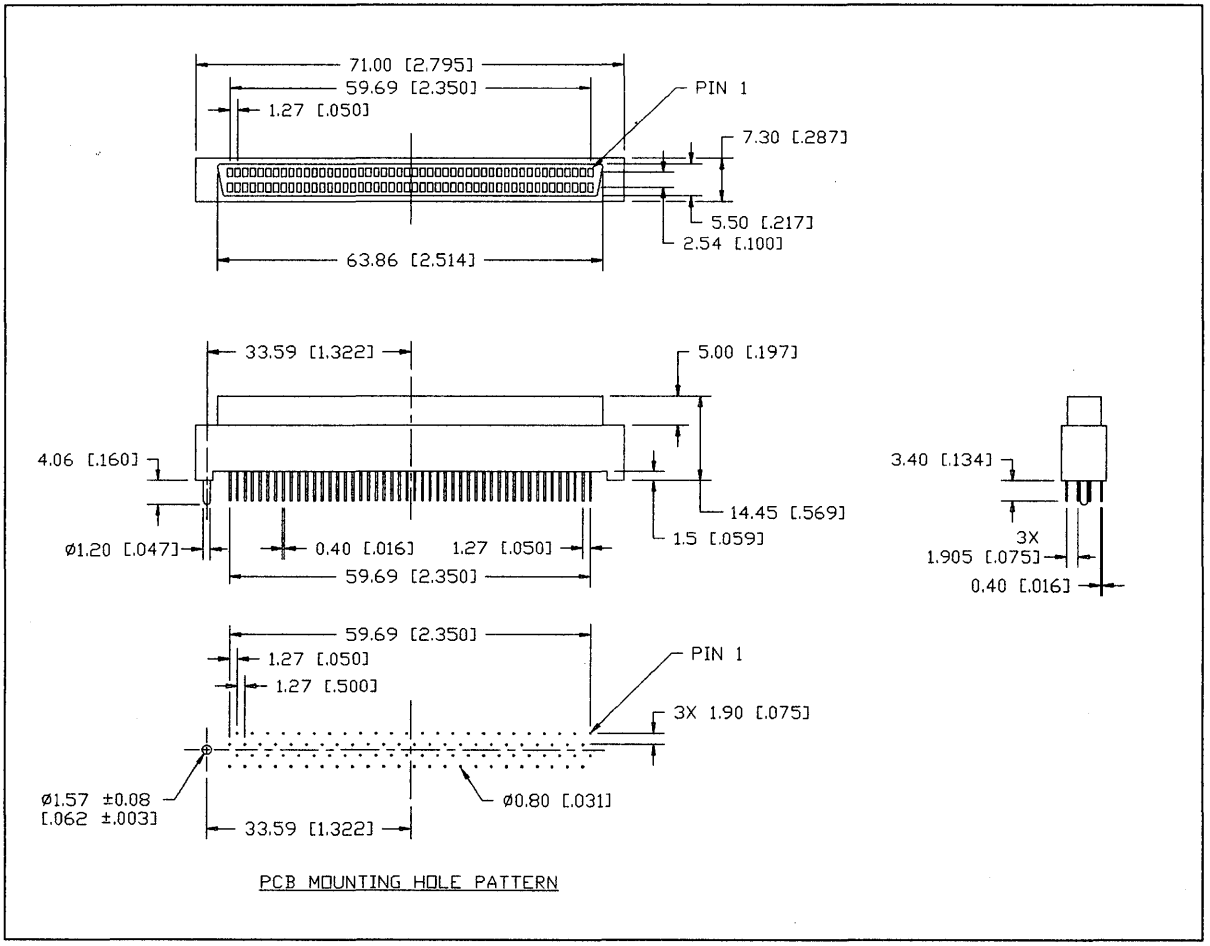


*Figure 4-10. Female Expansion Connector*



## Expansion Board Types and Sizes

Two types of SBus expansion boards are currently defined:

- Single-width.
- Double-width.

**Note:** Not all systems support all board types.

*Figure 4-11. Expansion Board Sizes*

| Type | Total Length | Total Width |
|---|---|---|
| Single-width | 146.70 mm | 83.82 mm |
| Double-width | 146.70 mm | 170.28 mm |

**Recommendation:** Triple-width SBus cards are not described in this book. Until Sun has more operational experience with triple-width cards, designers are discouraged from building them.

Also, even though the Sun SPARCstation 1 has three SBus expansion connectors, future systems may not accommodate a triple-width SBus card.

## Board Materials

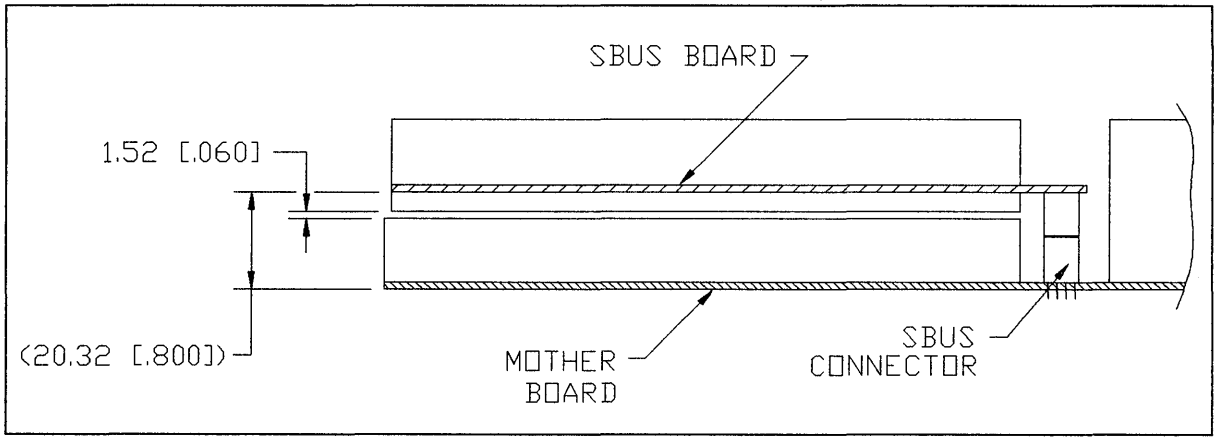
The board shall be 1.60 mm, ±0.20 mm thick. The combination of board warpage, component lead length, and component height shall not exceed the specified maximum allowable component or lead heigth limits as shown in figure 4-13.

## Component Clearance

The following figure shows the minimum acceptable gap between any solder-side components and any components on the motherboard. This gap guarantees that, under normal shock and vibration, there is no unintentional contact between the SBus card and components on the motherboard. Ultimately, this places a restriction on component heights allowable on the motherboard, without restricting the spacing of the SBus card above the motherboard.

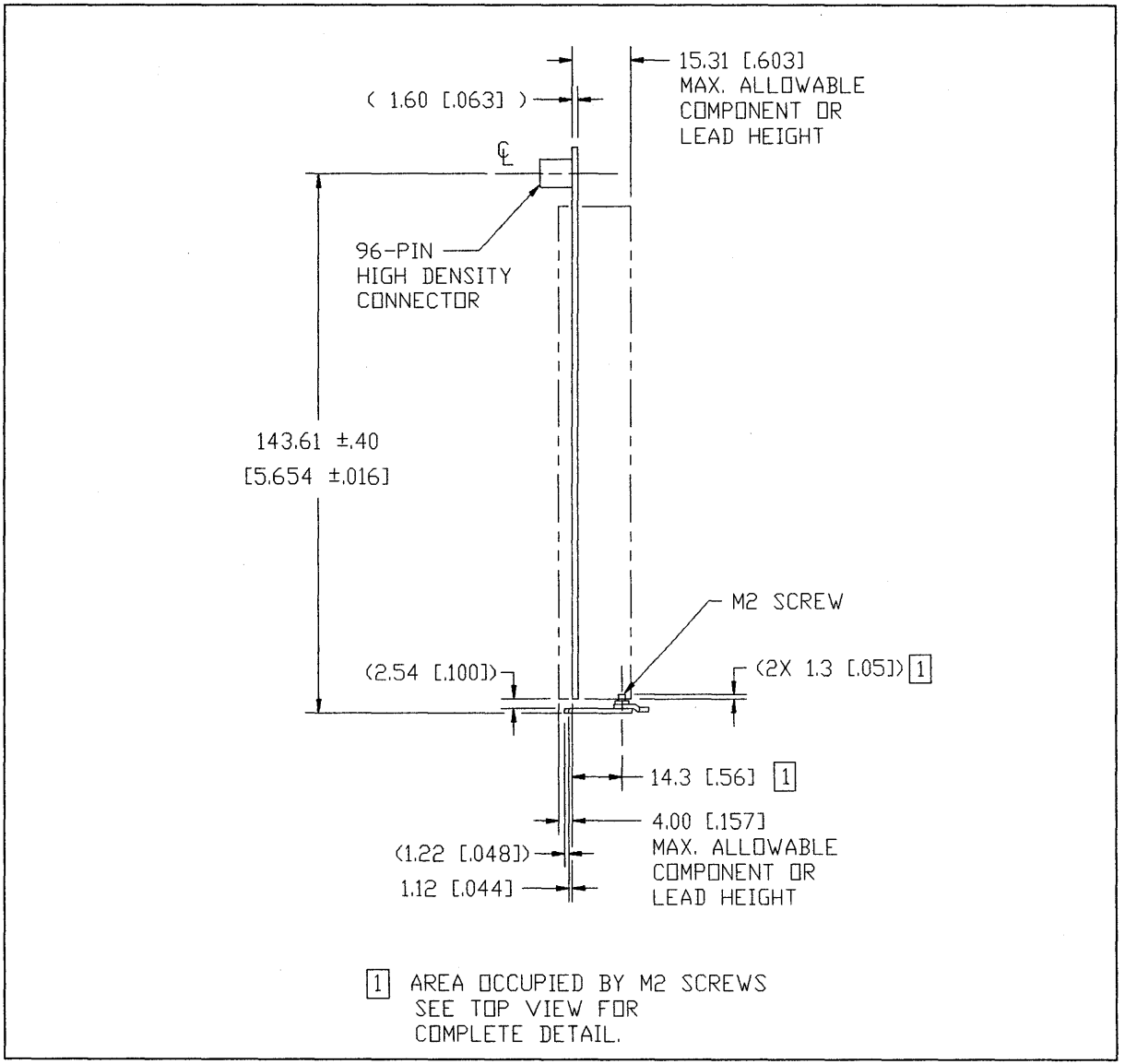
The maximum component height, including board thickness, is 15.31 mm (0.60 in). The maximum component or lead height below the board is 4 mm (0.16 in). This spacing is sufficient to allow the mounting of low profile SMT devices such as DRAMS. See the figure on the next page.

*Figure 4-12. Minimum Component Gap*



There is no longer any space reserved for a possible future connector. This space may be used for components.

*Figure 4-13. Component Clearance*



## Backplate

Every SBus card must have a metal backplate appropriate for its width, as shown in the following four figures for single-width, and in the subsequent four figures for double-width. These backplates are two piece assemblies to allow them to work in desktop environments or constricted-height environments such as laptops and VME-based applications.

It is permissible to replace double-width backplates with two single-width backplates appropriately spaced across the back of the SBus card.

The area available for connector openings on the backplate must be treated as a tunnel that extends perpendicular to the backplate. Any connector used must fit entirely within this tunnel, including the connector shell and any mechanical restraint mechanisms used.

After it is installed in the system, the backplate must be electrically connected to chassis ground by the system. It must not be connected to the logic ground on the SBus expansion card directly, via capacitive (including stray capacitance) or inductive means. It may be necessary to electrically isolate the connector or connectors of an expansion card from the backplate to meet this requirement. Also, it may be necessary in some cases to use differential, transformer, or opto-isolation techniques.

## Single-width SBus Card

The following figure shows a single-width SBus card. The subsequent three figures show a single-width backplate.

*Figure 4-14. Single-width SBus Card*

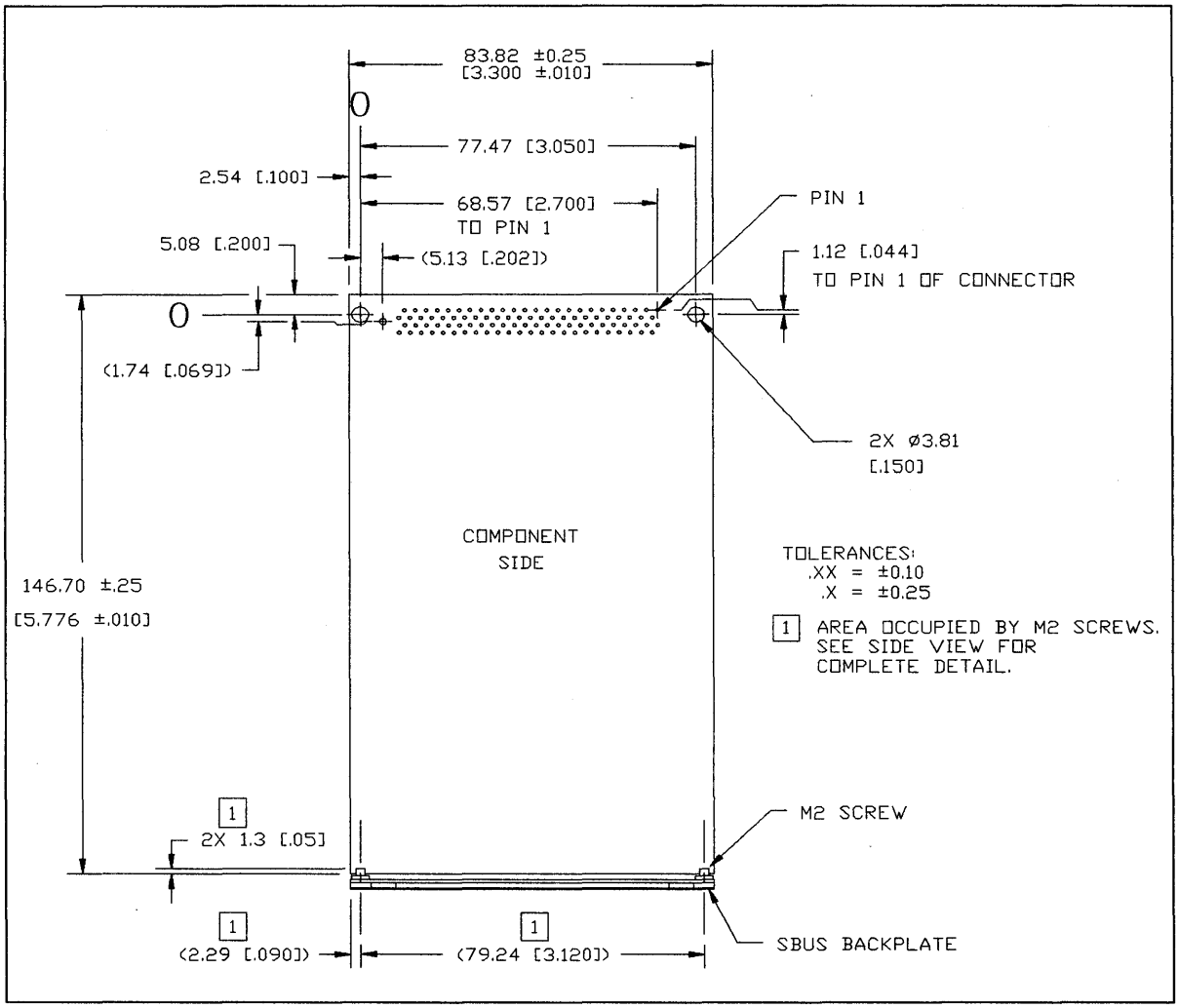


*Figure 4-15. Single-width Backplate*

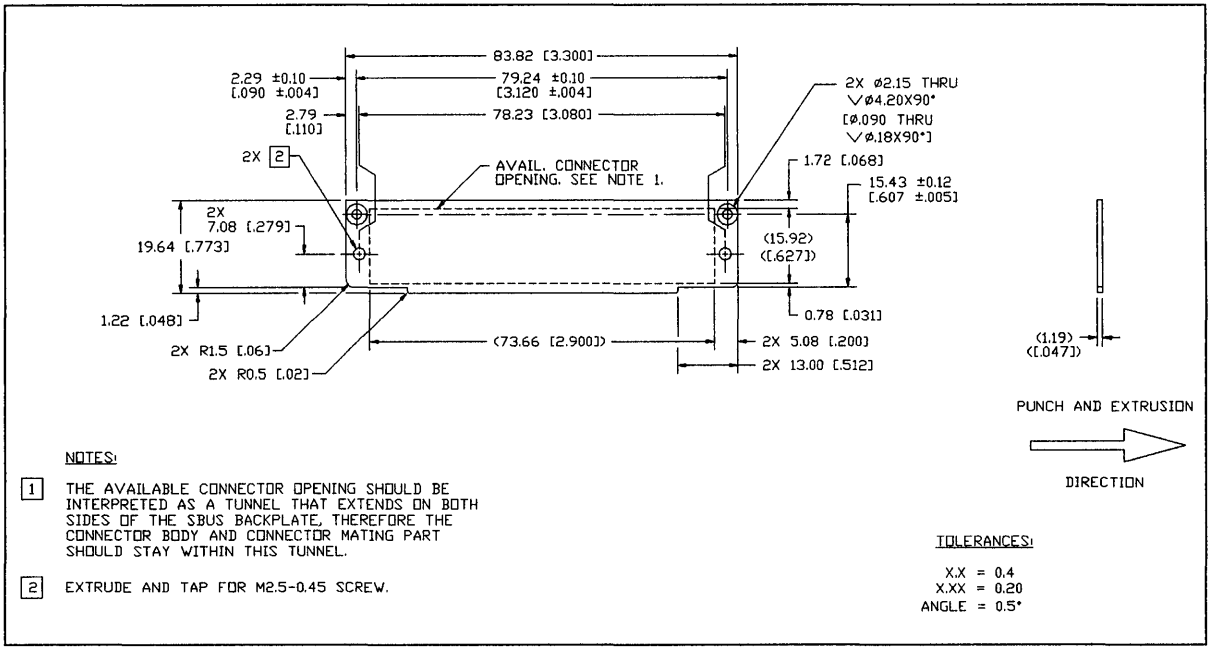



*Figure 4-16. Detail of Single-width Backplate Adaptor*

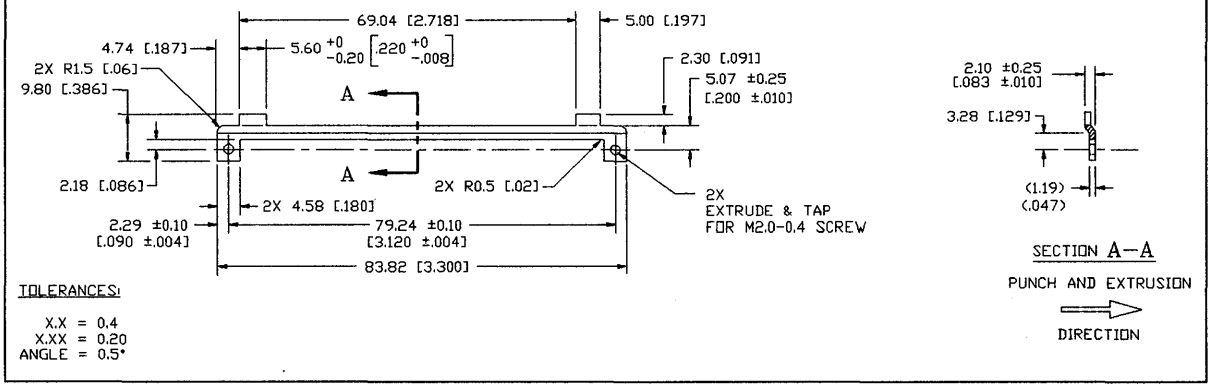
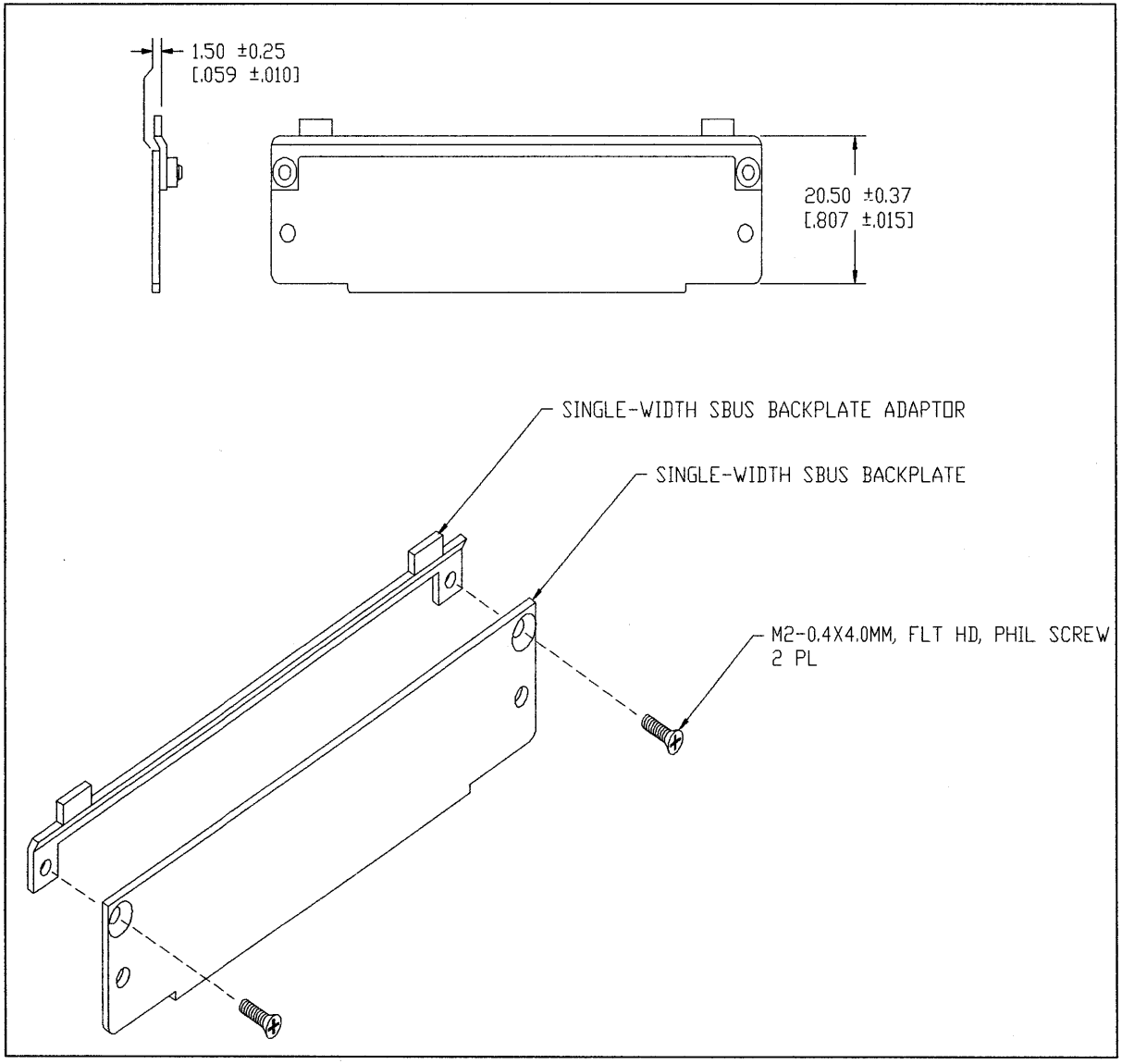


*Figure 4-17. Single-width Backplate Assembly*



## Double-width SBus Card

The following figure shows a double-width SBus card. The subsequent three figures show a double-width backplate.

*Figure 4-18. Double-width SBus Card*

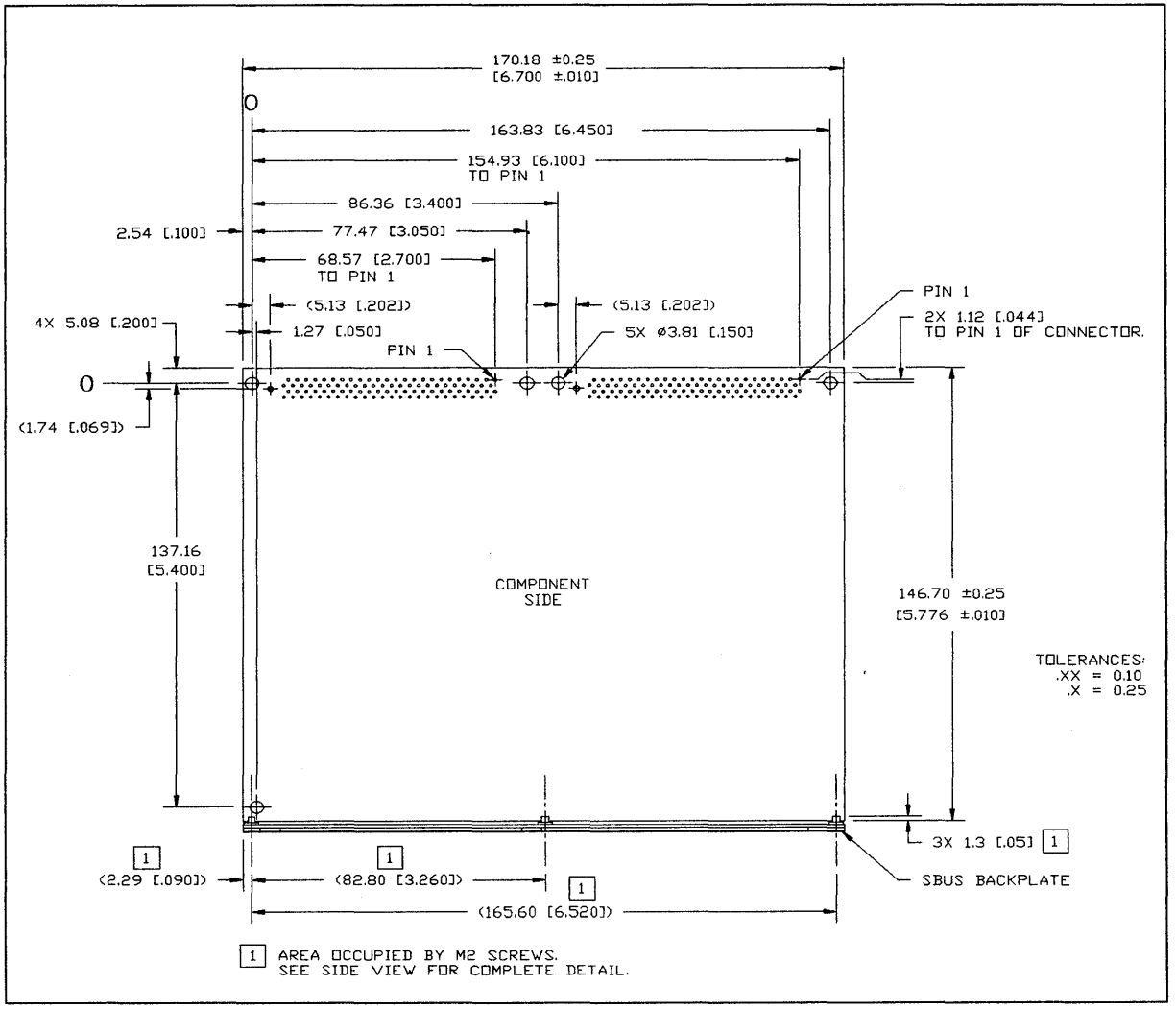


*Figure 4-19. Double-width Backplate*

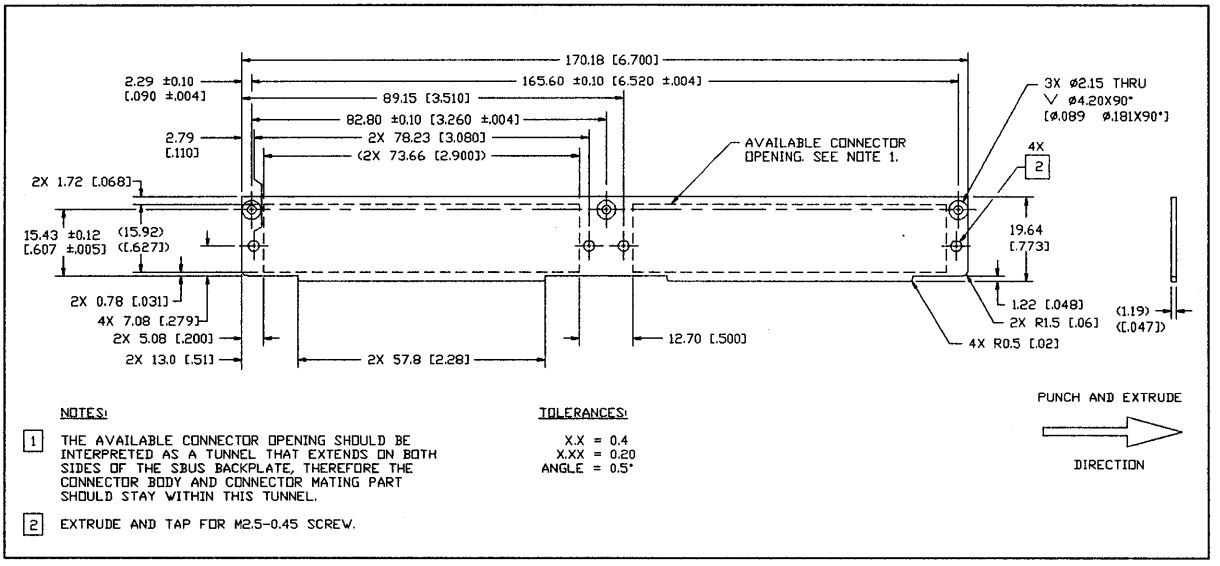



*Figure 4-20. Detail of Double-width Backplate Adaptor*

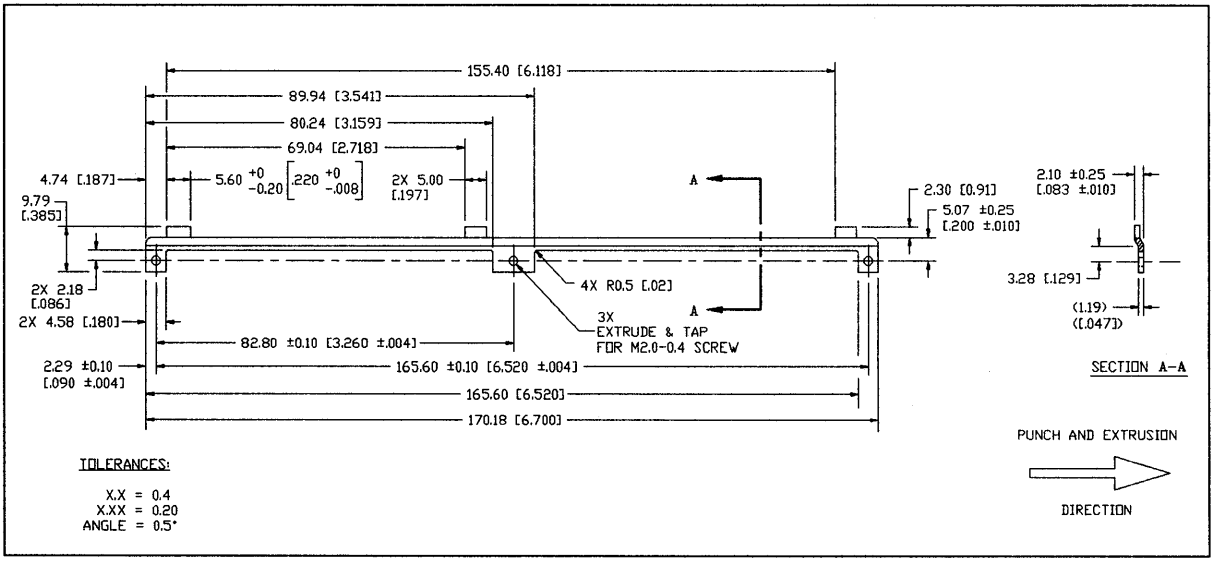
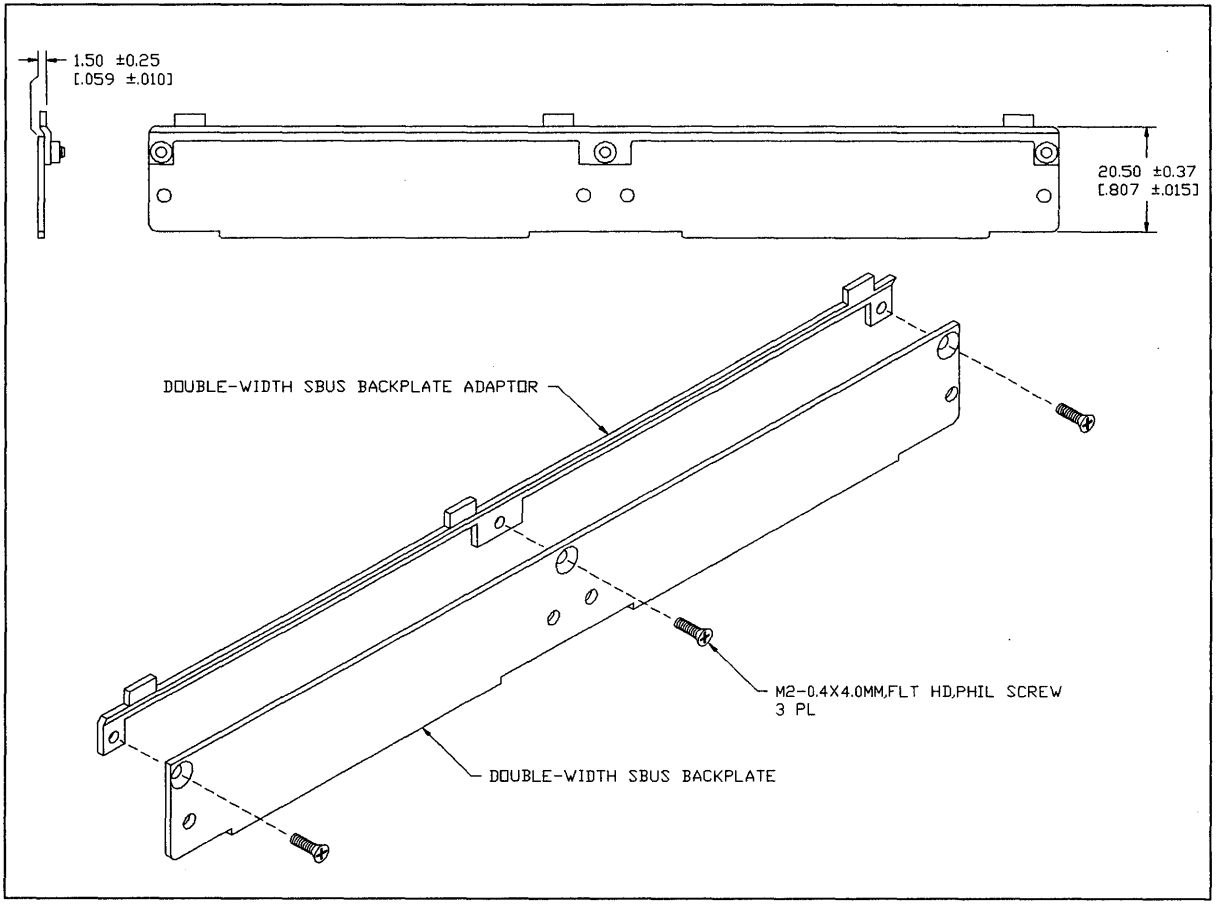


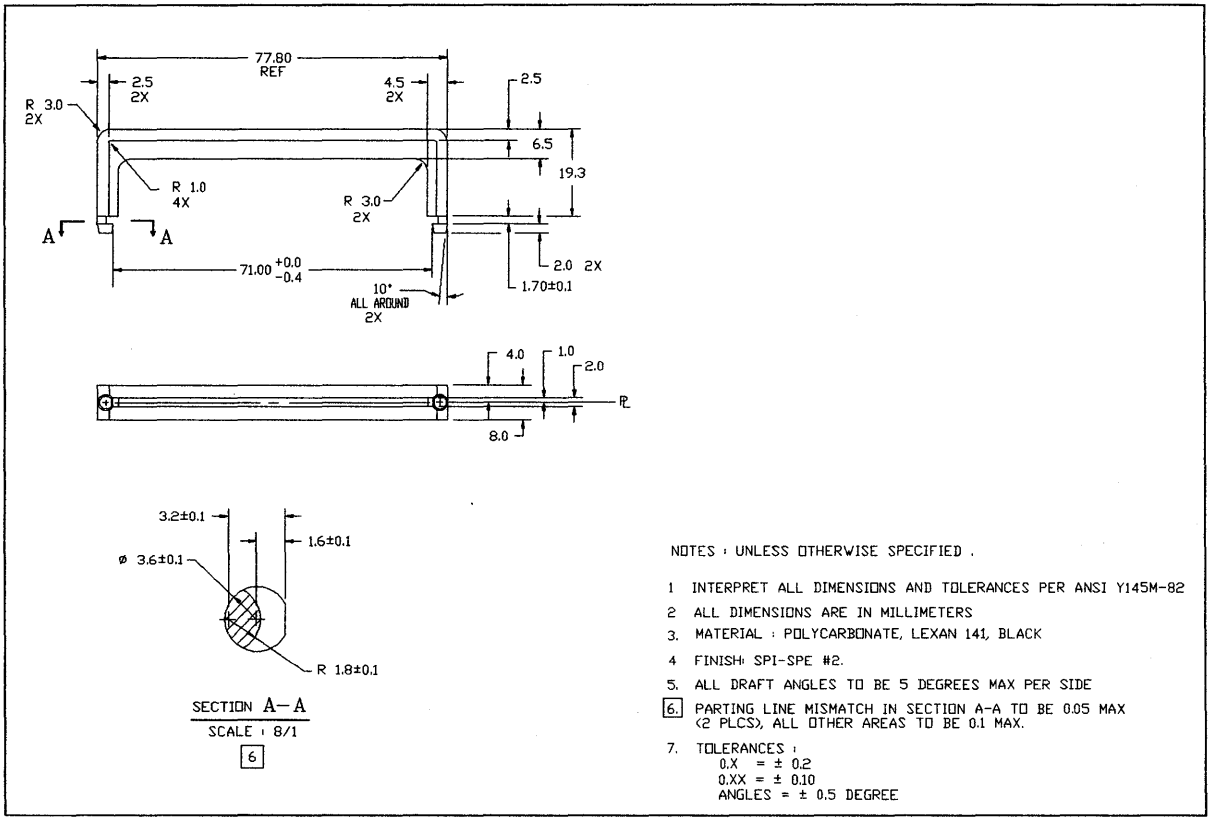
*Figure 4-21. Double-width Backplate Assembly*



## SBus Retainer and Stand-off

The following figure details the SBus retainer. The retainer assists the insertion or removal of the SBus card, and provides mechanical restraint for it against shock and vibration in a desktop environment.

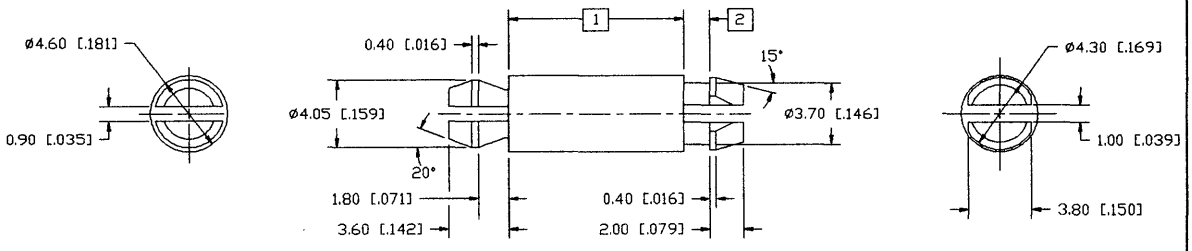
*Figure 4-22. SBus Retainer*



In other non-desktop applications such as laptops and VME-based applications, the retainer may not be necessary and can be easily removed. In these applications, a pair of stand-offs such as those in the following figure may be used.

These stand-offs mount to the SBus card using the same holes to which the retainer otherwise mounts. Some mechanical means must be provided to retain the SBus card in its slot.

*Figure 4-23. SBus Stand-off*




1 EQUAL TO ENGAGED HEIGHT OF SBUS CONNECTORS.

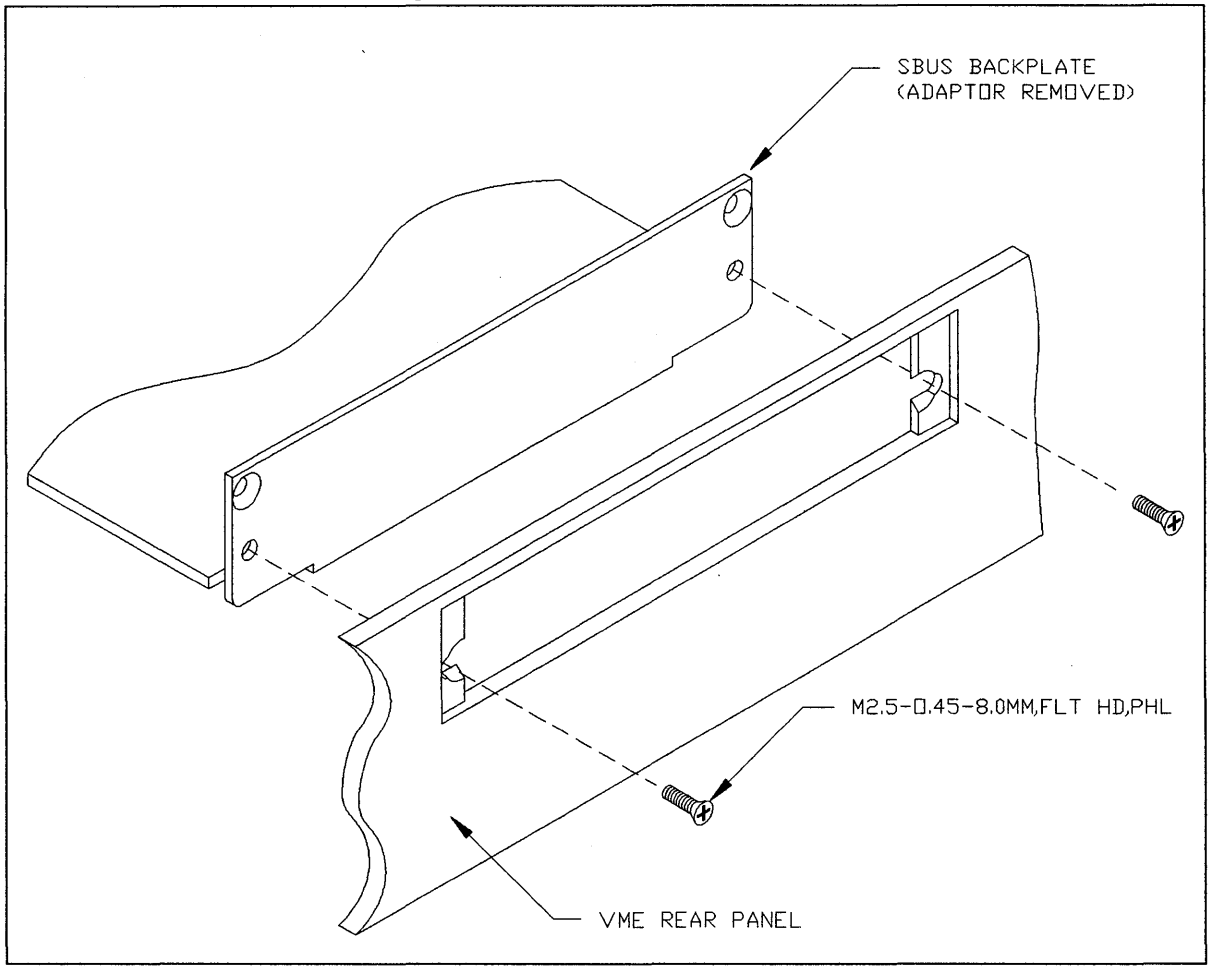
2 EQUAL TO THICKNESS OF MOTHER BOARD.

3. MATERIAL: NYLON 6/6.

## VME/FUTUREBUS Installation

The following figure shows how an SBus backplate is installed on a VME rear panel.

*Figure 4-24. SBus Card Installed on VME Card*



# 5

# *FCode Drivers for SBus Cards*

This chapter describes the programmatic operation of SBus slaves, masters, and controllers. Each SBus expansion card must have a Programmable Read-Only Memory (PROM) which identifies the device and contains an optional software driver to allow the card to be used as a boot device or as a display device during booting. This driver may also contain diagnostic, self-run routines.

## FCode PROMs

SBus device PROMs must be written in the FCode programming language. FCode has the following advantages:

- Source format is machine and system independent.
- Binary format is machine, system, and position independent.
- Binary format is compact.
- Binary format may be interpreted easily and efficiently.
- Programs are easy to develop and debug.
- Source format can be easily translated into binary format.
- Binary format can be *untranslated* back to a source format.

## Program Format

The FCode PROM must begin at address 0 within the SBus card's physical address space. Its size may range from 30 bytes up to the entire physical address space of the SBus card. Typical sizes are 60 bytes (for a simple card that identifies itself but does not need a driver) and 1-4 Kbytes (for a card with a boot driver or on-board diagnostics, or both).

The FCode PROM must be organized as follows:

- *Header* (8 bytes) containing the following information:
  - Magic number.
  - Version number.
  - Length.
  - Checksum.
- *Body* (0 or more bytes) containing the FCode program.
- *End Token* (a zero byte).

## Program Interpretation

For each SBus slot, the FCode program is interpreted as follows:

- Location 0 of the SBus card is read with a 32-bit access.

  The card must return the first 4 bytes of the PROM, or return the first byte and respond with a byte acknowledgment so that the CPU can perform bus sizing for the remaining 3 bytes.

- If there is no response (meaning there is no card in that slot), the slot is subsequently ignored.

- If the high-order byte of the value returned from that access is not the FCode *magic number* 0xfd or other values 0xf0 - 0xf3, the slot is subsequently ignored.

- Otherwise, the PROM is assumed to contain a valid FCode program.

  The FCode PROM is then interpreted by starting at location 0. Reading one byte at a time, the procedure associated with the value of that byte is executed.

- When a byte containing 00 is interpreted, interpretation ceases.

**Note:** Configuration parameters stored in non-volatile memory on the CPU board control the order in which the various SBus slots are interpreted.

## Device Identification

An FCode PROM must identify its device. The identification information must at least include the driver name of the card. Identification information may describe additional characteristics of the device for the benefit of the operating system and the CPU boot PROM.

Each property must have a name and a value. The name is a string; the value is an array of bytes which may encode strings, numbers, various other data types, and combinations thereof.

Properties may be created arbitrarily by FCode PROMs. The CPU boot PROM understands certain property names which inform it about such things as the type of the device (for example, whether it is a disk, tape, network, display). The CPU boot PROM may use this information to determine how to use the device (if at all) during the booting process.

**Observation:** In most systems, the CPU's FCode interpreter stores each device's identification information in a *device tree* that contains a node for each device. Each *device node* contains a *property list* to identify and describe the device. The property list is created as a result of interpreting the program in the FCode PROM.

The UNIX® operating system understands other property names that provide information for configuring the operating system automatically. These properties include the driver name (which is treated as a *hint*), the addresses and sizes of the device's registers, and the interrupt levels and interrupt vectors used by the device.

Other properties may be used by individual UNIX device drivers. The names of such properties and the interpretation of their values are subject to agreement between the writers of the FCode PROM and the UNIX driver, but may otherwise be arbitrarily chosen. For example, a display device might declare *width*, *height*, and *depth* properties to allow a single UNIX driver to automatically configure itself for one of several similar but different devices.

## FCode Language

The FCode programming language is closely related to the FORTH-83 programming language. FCode is essentially FORTH-83 with extensions appropriate to its use for device identification and boot drivers. Additionally, FCode has a well-specified binary format, whereas FORTH-83 specifies only the source format. In contrast to FORTH-83, FCode is based on a 32-bit stack width and 32-bit arithmetic.

**Observation:** FCode may be thought of as *byte-coded FORTH.* FCode PROMs are developed by writing FORTH source code; then a simple *tokenizer* program is used to convert the source code to the binary (byte code) format. The binary version is then loaded into a PROM and installed on the SBus card.

In most cases, each FORTH source code corresponds to a single FCode binary code. For some FORTH commands, the tokenizer provides *macros* to convert a single FORTH source code command into a sequence of several FCode binary codes. A version of the tokenizer program (for Sun-4 workstations) is available through the Sun SBus Technical Support Group.

## FCodes and FORTH

For information about FCode primitives currently supported by the Open Boot PROM, see Appendix C. For information about FCodes and the FORTH programming language, see one of the following books:

- *Writing FCode Programs for SBus Cards*, a Sun Microsystems, Inc. publication, part number 800-5673-10.
- *Starting FORTH*, 2d edition
  by Leo Brodie
  (Prentice-Hall).
- *Mastering FORTH*
  by Anita Anderson and MartinTracy
  (Brady Publishing).
- *FORTH: A Text and Reference*
  by Nicholas Spies and Mahlon Kelley
  (Prentice-Hall).

# A

# *Specification Compliance*

This appendix describes the minimal requirements for SBus cards to be compliant with the SBus specification. SBus slaves, masters, and systems can be compliant with the specification without supporting all SBus features, as described in this appendix.

## Slave

To be a compliant SBus slave, an SBus device must at least:

- ❑ Use Clock (Clk) to determine signal validity.
- ❑ Meet all setup, hold, and delay times.
- ❑ Meet all AC and DC electrical specifications.
- ❑ Use at least AddressStrobe* (AS*) and SlaveSelect* (Sel*) to determine whether it should participate in the current bus cycle.
- ❑ Drive Data(31:0) (D(31:0)) in accordance with the Read (Rd) signal.
- ❑ Sense all Size(2:0) (Siz(2:0)) signals, and support at least one transfer size.

  At least one non-burst transfer size must be supported, a byte, half-word or word.

- ❑ Drive the Ack(2:0)* (Ack(2:0)*) signals, and be able to respond with at least one kind of Data Acknowledgment.
- ❑ Respond with a valid acknowledgment within the bus timeout period.

  Otherwise, no response should be given.
- ❑ Have an FCode PROM beginning at physical address 0.

## DVMA Master

To be a compliant SBus master, an SBus device must at least:

- ❑ Use the SBus Clk to determine signal validity.
- ❑ Meet all setup, hold, and delay times.
- ❑ Meet all AC and DC electrical specifications.
- ❑ Assert Request* (BR*) to access the bus.

  However, never keep BR* asserted for more than two consecutive bus cycles, except during an atomic transaction which causes bus sizing or uses dummy reads.
- ❑ Sense Grant* (BG*), and place virtual addresses on D(31:0) at the appropriate time.
- ❑ Drive all Siz(2:0) signals, and support at least one transfer size.
- ❑ Drive Rd to indicate the transfer direction.
- ❑ Drive D(31:0) in accordance with the Rd signal.
- ❑ Sense all of the Ack(2:0)* signals, and terminate the bus cycle after receiving appropriate acknowledgment.
- ❑ Support Error Acknowledgment by aborting the bus cycle, and Rerun Acknowledgment by reissuing the bus cycle.

  To be compliant, a master may not terminate a bus cycle until the slave (or, in the case of a timeout, the SBus controller) has terminated the bus cycle.

## SBus Controller

To be a compliant SBus controller, the controller must:

- ❑ Provide a Clk in the range of 16.67 MHz to 25 MHz.
- ❑ Drive AS*, Sel*, PhysAddr(27:0) (PA(27:0)), and Reset*.
- ❑ Support at least one bus master.

  In host-based systems, this master may be just the CPU. In a system with DVMA masters, the controller must arbitrate fairly among the master's BR*, and issue the appropriate BG*.
- ❑ Provide address translation facilities for all DVMA masters.
- ❑ Issue an Error Acknowledgment in case of a bus timeout.
- ❑ Support 1-, 2-, 4- and 16-byte transfers.
- ❑ Support rerun.

# B

# *SBus Extensions*

This appendix describes extensions to the SBus. These extensions provide recommendations about implementing certain features which are not a standard part of the SBus specification.

## Parity Checking

In some systems, it may be desirable to be able to check whether data is transferred properly between master and slave, and that no data corruption has occurred while traversing chip pins, connectors, and wiring.

When the SBus is reset, all extended parity checking must be disabled. An SBus device using extended parity checking must use its FCode program to determine whether the system supports extended parity checking. Extended parity checking must remain disabled, unless supported by the system.

**Note:** The following information applies primarily to 32-bit per clock cycle transfers. For information about 64-bit per clock cycle transfers, see the next section in this appendix.

Devices that compute extended parity checking must do so only over the bits of Data(31:0) (D(31:0)), and for all values placed onto D(31:0), including virtual addresses during translation cycles. Odd parity is computed. The SBus device currently driving D(31:0) must drive DtaPar so that XORing the DtaPar signal with D(31:0) results in a logic 1. The DtaPar signal must be driven with the same timing as D(31:0). In the case of byte or half-word transfers, devices supporting extended parity must drive the high-order bits of the data lines to compute parity correctly.

The receiving SBus device is responsible for detecting parity errors. During translation cycles, the SBus controller must check the parity of the virtual address, and issue an Error Acknowledgment. The master must then abort the bus cycle, or retry it as desired.

When data is written by a master, the master must generate parity on D(31:0) for each datum transferred, and the slave must check the parity of D(31:0). If the slave detects a parity error, it should generate an Error Acknowledgment or LateError* (LErr*), or both. When data is read by a master, the slave must generate parity on D(31:0) for each datum transferred, and the receiving master must check the parity of D(31:0). If the master detects parity error, it should generate an interrupt (but the master must not use LErr* or Error Acknowledgment).

**Observation:** When a device is driving data, data parity may always be generated or driven. Data parity must not be checked, unless it is supported by the system.

# SBus 64-bit Transfer Protocols

This section describes a set of extensions to the basic SBus protocols to transfer 64 bits of data each clock cycle instead of only 32 bits each clock cycle.

## Scope and Compatibility

The 64-bit SBus extensions apply to all SBus systems and environments. SBus systems and devices conforming to the specifications in this section will operate both in existing 32-bit SBus environments, as well as in the 64-bit SBus environment.

SBus masters and slaves using only the 32-bit protocols will work correctly in systems that implement the 64-bit protocols; and, conversely, SBus masters and slaves using the 64-bit protocols will operate correctly in systems that implement only 32-bit protocols. Furthermore, 32-bit SBus masters and slaves can transfer data to and from 64-bit slaves and masters.

## Overview

The 64-bit SBus protocols provide a means to improve SBus bandwidth and reduce SBus latency while maintaining forward and backward compatibility. To achieve these goals, the 64-bit SBus protocols use the same signals as the existing 32-bit protocols.

However, they take advantage of time-multiplexing the PhysAddr(27:0) (PA(27:0)), Size(2:0) (Siz(2:0)), Read (Rd), and Data(31:0) (D(31:0)) signals to create a 64-bit wide path capable of transferring a double-word of data every clock cycle. The 64-bit transfer SBus extension also provides for 128 byte transfers.

Figure B-2 shows the nature of the 64-bit protocol. The master asserts its Request* (BR*) signal the same as for all SBus requests. After receiving Grant* (BG*), the master begins a translation cycle. As for all transfers, the master drives a virtual address onto D(31:0), drives Rd to the proper state, and sets Siz(2:0) to ExtendedTransfer. In the fastest possible case, the SBus controller then drives a physical address onto PA(27:0), and asserts AddressStrobe* (AS*).

At the same time, the requesting master drives Rd to 0 (indicating a write), independent of the actual transfer direction. It also drives D(31:0) with the following Extended Transfer Information:

*Figure B-1. Extended Transfer Information*

| | |
|---|---|
| D(31) | ExtendedType. |
| D(30:28) | Extended Size |
| D(27) | Read/write. |
| D(26:25) | Atomic transaction |
| D(24:0) | Reserved |

If an SBus slave supports ExtendedTransfers, it must latch the physical address on PA(27:0) and the cycle information on D(31:21) on the clock edge following the assertion of AS*. Unlike 32-bit transfers, where this information remains valid throughout the bus cycle, in an ExtendedTransfer this information is valid only at the clock edge following the assertion of AS*.

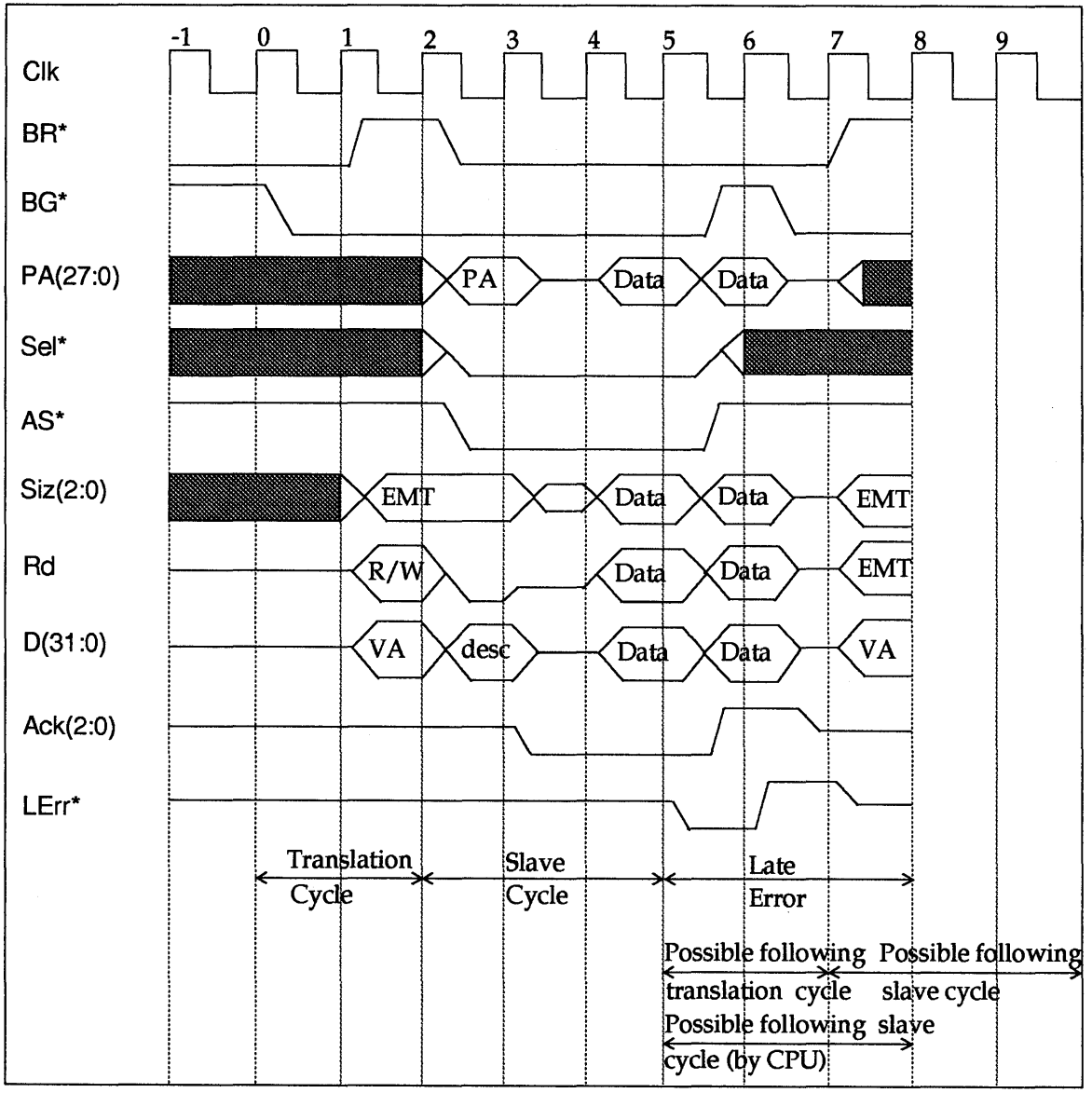
During the clock cycle after the assertion of AS*, the SBus controller must tristate PA(27:0), while the SBus master must tristate Siz(2:0), Rd, and D(31:0). The SBus controller must have pulldown resistors on Rd and Siz(2), and pullup resistors on Siz(1:0) to maintain them in the proper state.

As early as this same clock cycle but no later than 255 clock cycles after the assertion of AS*, an SBus slave supporting ExtendedTransfers should assert double-word acknowledgmentAn SBus slave which does not support ExtendedTransfers should issue an Error Acknowledgment. Until the acknowledgment is received, the master must continue to keep D(31:0), Rd, and Siz(2:0) tristated, while the controller must continue to keep PA(27:0) tristated.

During a write, in the cycle following the acknowledgment, the master must drive the first double-word of data onto D(63:0) — that is, Rd, Siz(2:0), PA(27:0), and D(31:0). During a read, in the cycle following the acknowledgment, the slave must drive D(63:0).

In both cases, the data lines are driven for exactly one clock cycle. In the case of a burst transfer, the next double-word acknowledgment can come as soon as the cycle during which the double-word is driven. Thus, ExtendedTransfer timing for both reads and writes is the same as 32-bit timing for reads.

*Figure B-2. 64-bit Protocol*



The remainder of this appendix describes the specifications for performing ExtendedTransfers.

### Clock

The specification for the SBus Clk signal remains unchanged for ExtendedTransfers.

### Reset*

The specification for the SBus Reset* (Reset*) signal remains unchanged for ExtendedTransfers.

### AddressStrobe* and SlaveSelect*

The specification for the SBus AS* remains unchanged for ExtendedTransfers. In ExtendedTransfers, AS* is asserted in the clock cycle during which physical address is driven onto PA(27:0) by the SBus controller. AS* remains asserted throughout the entire transfer.

However, during a 32-bit transfer, a master may use BG* for all of its timing, and need not look at AS*. During an ExtendedTransfer, a master must look for the assertion of AS* to determine when to stop drivingSiz(2:0), Rd, and D(31:0).

The specification for SlaveSelect* (Sel*) remains unchanged for ExtendedTransfers.

### PhysAddr(27:0)

During an ExtendedTransfer, the SBus controller must drive a physical address onto PA(27:0) for exactly one clock cycle, no sooner than two clock cycles following the assertion of BG* to some master, and no later than the clock cycle in which it asserts AS*. In this respect , 64-bit transfers and 32-bit transfers use of , and timing for, PA(27:0) are identical.

In the clock cycle following the assertion of AS*, the SBus controller must tristate (stop driving) PA(27:0), and leave this signal undriven until at least two clock cycles after it deasserts the latter of BG* or AS*. For each slave, the physical address space for ExtendedTransfers must be the same as for 32-bit transfers. It must be possible to access data using the same physical address with 32- or 64-bit transfers.

## Request*, Grant*, and Atomic Transactions

The specification for BR* and BG* remain unchanged for ExtendedTransfers. The sequence of events for atomic transactions also remains unchanged. However, if the first bus cycle of an atomic transaction uses an ExtendedTransfer, all bus cycles in the atomic transaction (including any dummy reads) must also use ExtendedTransfers.

For 32-bit transfers, a master may use BG* for all of its timing and need not look at AS*. However, for ExtendedTransfers, a master must look for the assertion of AS* to determine when to stop driving Siz(2:0), Rd, and D(31:0).

**Observation:** Restricting all bus cycles of an atomic transaction to the same transfer size simplifies the design of bus couplers.

## 64-bit Transfer Bus Cycle

An ExtendedTransfer bus cycle consists of two phases: a *translation cycle* and a *slave cycle.*

### Translation Cycle

Like 32-bit transfers, an ExtendedTransfer (64-bit) translation cycle begins when the SBus controller asserts BG*. During the clock cycle immediately following BG*, the master must drive a virtual address onto D(31:0), drive Rd appropriately for the direction of data transfer and, unlike 32-bit transfers, drive Siz(2:0) to ExtendedTransfer. In the second clock cycle following the assertion of BG*, the SBus master must drive Rd to logic level 0, independent of the direction of transfer.

During this same clock cycle, the master must drive D(31:0) with Extended Transfer Information. The master must continue to drive these signals until the clock cycle after which the SBus controller asserts AS* (which it may do as early as the second clock cycle after the assertion of BG*). However, the SBus controller is allowed to take more than one clock cycle to translate the virtual address. After translating the virtual address, the SBus controller must drive the corresponding physical address onto PA(27:0), and assert AS* and the appropriate Sel*.

If an SBus controller does not support ExtendedTransfers, it should issue an Error Acknowledgment instead of asserting AS*. Nevertheless, a master should be enabled to issue ExtendedTransfers only in systems supporting ExtendedTransfers and to slaves supporting ExtendedTransfers. Thus, an Error Acknowledgment occurs only if a master is misprogrammed.

**Slave Cycle**

Like 32-bit transfers, an ExtendedTransfer slave cycle begins with the assertion of AS*. At the clock edge following the assertion of AS*, the selected 64-bit slave must latch the physical address, the Extended Transfer Information on D(31:0), and the fact that an ExtendedTransfer is being performed as indicated by Siz(2:0).

Thus, unlike 32-bit transfers, PA(27:0) is valid during this first clock cycle only. During the clock cycle following the assertion of AS*, the master must tristate Rd, Siz(2:0), and D(31:0), while the SBus controller must tristate PA(27:0). After tristating these signals, the controller must not drive PA(27:0) again until the latter of two clock cycles following the last acknowledgment, or one clock cycle after it unasserts BG* and AS*.

As soon as one clock cycle after the assertion of AS*, the selected slave may drive Ack(2:0)*. Slaves requiring additional time may wait to drive Ack(2:0)*, as long as the entire transfer is completed within 255 clock cycles after AS* is asserted — that is, the SBus timout period remains unchanged for ExtendedTransfers.

If the selected slave does not support ExtendedTransfers, it must issue an Error Acknowledgment. If the selected slave does support ExtendedTransfers, it should generate a double-word acknowledgment. In the case of a read (as indicated by ExtendedTransferRead) the slave must drive D(63:0) with the first 8 bytes of data during the clock cycle following its acknowledgment, for exactly one clock cycle.

In the case of a write (as indicated by ExtendedTransferRead, not the Rd signal), the master must drive the first double-word of data during the clock cycle following the slave's acknowledgment. Thus, whereas 64-bit read transfers follow the same timing as 32-bit read transfers, 64-bit write transfers do not. Instead, 64-bit write transfers use the same timing as 64-bit read transfers.

If more than 8 bytes of data are transferred (as indicated by ExtendedTransferSiz(2:0)), the slave must continue to generate double-word acknowledgments.

The controller may unassert AS* and BG* as early as one clock cycle following the last double-word acknowledgment for the bus cycle. As for 32-bit transfers, following the last acknowledgment, the slave must drive Ack(2:0)* to the logic 1 state for one clock cycle, after which the slave must stop driving Ack(2:0)*. By the latter of two clock cycles after the last acknowledgment or the clock cycle following BG* being unasserted, the slave (in the case of a read) or the master ( in the case of a write) must tristate D(63:0).

As for 32-bit transfers, LErr* may be used to indicate errors. If it is used, its timing is the same as for 32-bit read transfers: it must be asserted two clock cycles after the corresponding double-word acknowledgment, for exactly one clock cycle, after which it must be driven to the logic 1 state of one clock cycle. In the case of burst transfers, it may remain asserted for each double-word in error.

## Data(63:0)

For ExtendedTransfers, 8 bytes of data are transferred on the SBus signals D(63:0). SBus masters, slaves, and controllers must use the following mapping between D(63:0) and the standard SBus signals.

*Figure B-3. Using* D(63:0) *for ExtendedTransfers*

| ExtendedTransfer | Standard SBus Signal |
|---|---|
| D(63) | Rd |
| D(62:60) | Siz(2:0) |
| D(59:32) | PA(27:0) |
| D(31:0) | D(31:0) |

**Note:** D(63) is the most significant bit of the double-word; D(0) is the least significant bit.

When used to transfer data during a slave cycle, D(63:0) must be driven no sooner than the clock cycle following the assertion of the first non-idle acknowledgment; and it must be tristated no later than the latter of the second clock cycle following the last acknowledgment, or one clock cycle after BG* is unasserted. For ExtendedTransfers, each double-word of data is driven for one clock cycle beginning with the clock cycle following the associated acknowledgment. Unlike 32-bit transfers, this timing is the same for both reads and writes.

Since addressing is big-endian, D(63:56) is the most significant byte and is located at address A mod 8 = 0. Byte 7 (D(7:0)) is located at address A mod 8 = 7. Similarly, the words at address A mod 8 = 0 and A + 4 are placed on D(63:32) and D(31:0), respectively.

**Observation:** Port locations for bytes, half-words, and words are not defined for ExtendedTransfers, since ExtendedTransfers always take place in multiples of 8 bytes.

## Extended Transfer Information

Every SBus master supporting ExtendedTransfers must drive Extended Transfer Information onto D(31:0), beginning with the second clock cycle following the assertion of BG*, until the clock cycle following the assertion of AS*, at which time the master must tristate D(31:0).

Every SBus slave supporting ExtendedTransfers must latch the Extended Transfer Information driven on D(31:0) at the clock edge following the assertion of AS*.

The following mapping must be used between the Extended Transfer Information signals and D(31:0).

*Figure B-4. Using* D(31:0) *for ExtendedTransfers*

| ExtendedTransfer | Standard SBus Signal |
|---|---|
| ExtendedTransferType | D(31) |
| ExtendedTransferSize(2:0) | D(30:28) |
| ExtendedTransferRead | D(27) |
| ExtendedTransferAtomic(1:0) | D(26:25) |
| ExtendedTransferReserved(22:0) | D(24:0) |

**ExtendedTransferType**

During an ExtendedTransfer, the ExtendedType signal must be set to *64-bit Transfer* as the following figure shows. Masters performing ExtendedTransfers must not use the reserved encoding of ExtendedType. Slaves that support ExtendedTransfers and detect ExtendedType set to the reserved value must generate an Error Acknowledgment.

*Figure B-5.* ExtendedType *functions*

| ExtendedTransferType | **Function** |
|---|---|
| 0 | 64-bit Transfer |
| 1 | Reserved |

**Observation:** The ExtendedType signal provides a mechanism for future extensions to the SBus.

**ExtendedTransferSize(2:0)**

For ExtendedTransfers, ExtendedTransferSize(2:0), not Siz(2:0), determines how many bytes of data are transferred during a bus cycle. The following encodings for ExtendedTransferSize must be used.

*Figure B-6.* ExtendedType

| **ETSize (2)** | **ETSize (1)** | **ETSize (0)** | **Function** |
|---|---|---|---|
| 0 | 0 | 0 | Reserved |
| 0 | 0 | 1 | Reserved |
| 0 | 1 | 0 | Reserved |
| 0 | 1 | 1 | 8 bytes |
| 1 | 0 | 0 | 16 bytes |
| 1 | 0 | 1 | 32 bytes |
| 1 | 1 | 0 | 64 bytes |
| 1 | 1 | 1 | 128 bytes |

Masters must not generate the reserved values of ExtendedTransferSize. Slaves should issue an Error Acknowledgment if they detect a reserved value of ExtendedTransferSize.

A master may implement only a subset of the ExtendedTransferSizes. A slave may support only a subset of the ExtendedTransferSizes. A master or slave supporting an ExtendedTransferSize of size *n*, must support all extended transfer sizes up to size *n*.

**Observation:** 1, 2, and 4 byte transfers must be performed using normal 32-bit SBus protocols.

**ExtendedTransferRead**

During a ExtendedTransfer, the master must drive ExtendedTransferRead to 0 to perform a write, and to 1 to perform a read.

The value of ExtendedTransferRead must be identical to the value the master drives onto the SBus signal Rd during the clock cycle following the assertion of BG*.

**ExtendedTransferAtomic (1:0)**

ExtendedTransferAtomic should be set to Normal (0b00) except for bus cycles that are part of an atomic transaction. Masters performing atomic transactions should drive ExtendedTransferAtomic(1:0) as the following figure shows.

*Figure B-7.* ExtendedTransferAtomic(1:0)

| ETAtomic (1) | ETAtomic (0) | Function |
|---|---|---|
| 0 | 0 | Normal bus cycle (non-atomic bus cycle) |
| 0 | 1 | First bus cycle of an atomic transaction |
| 1 | 0 | Intermediate bus cycle of an atomic transaction |
| 1 | 1 | Last bus cycle of an atomic transaction |

### Size(2:0)

The SBus master must drive Siz(2:0) to ExtendedTransfer (encoding 0b011), beginning with the clock cycle following the assertion of BG* until the clock cycle following the assertion of AS*. Beginning with the clock cycle following the assertion of AS* until the slave asserts a non-idle acknowledgment, the master must tristate Siz(2:0). Beginning with the clock cycle following the acknowledgment, Siz(2:0) signals are used as D(62:60) and must follow D(63:0) timing described previously.

SBus controllers supporting ExtendedTransfers must terminate Siz(2) using a 2 KΩ resistor to ground or a holding amplifer. The SBus controller must terminate Siz(1:0) using a 10 KΩ resistor to the +5V supply or a holding amplifer.

**Observation:** The termination requirement for Siz(2:0) keeps this signal in the proper state of ExtendedTransfer during the time in which the bus is turned around.

### Read

The SBus master must drive Rd to a logic 0 state (in the case of a write) or a logic 1 state (in the case of a read), beginning with the clock cycle immediately following the assertion of BG*, for exactly one clock cycle. Beginning with the second clock cycle following the assertion of BG*, until the clock cycle following the assertion of AS*, the master must drive Rd to a logic 0 state, even if the master is performing a read. Beginning with the clock cycle following the assertion of AS* until the slave asserts a non-idle acknowledgment, the master must tristate Rd. Beginning with the clock cycle following the acknowledgement, Rd is used as D(63) and must follow D(63:0) timing described previously.

ExtendedTransferRead must be set to the same value that Rd has during the clock cycle following the assertion of BG*.

SBus controllers supporting ExtendedTransfers must terminate Rd with a 2 KΩ resistor to ground or a holding amplifer.

**Observation:** In the event a 32-bit slave is selected accidently, Rd is driven to logic 0 state after the first clock cycle of the translation cycle to ensure that the drivers of the slave and master do not *fight* during the clock cycle following the assertion of AS*.

The termination requirement for Rd keeps it in the proper state of logic 0 during the time in which the bus is turned around.

Ack(2:0)*

The timing and use of Ack(2:0)* is largely the same for 64-bit and 32-bit transfers. However, the meaning of byte, half-word, and word acknowledgments is undefined during an ExtendedTransfer and, therefore, should not be used.

As explained earlier, the timing for writing D(63:0) with respect to Ack(2:0)* is not the same for 32-bit and 64-bit transfers. For ExtendedTransfers, D(63:0) signals are driven with data one clock cycle after Ack(2:0)* is asserted for both reads and writes.

*Bus sizing* is not supported for ExtendedTransfers. Only multiples of 8 bytes can be transferred.

The following acknowledgments apply to ExtendedTransfers:

- Word Acknowledgment.

  A slave must not generate a word acknowledgment during an ExtendedTransfer. A master or controller receiving a word acknowledgment during an ExtendedTransfer must abort the bus cycle.

- Double-word Acknowledgment.

  A slave must generate a double-word acknowledgment (encoding 0b010) to indicate it is ready to transfer a double-word of data during the next clock cycle.

- Error Acknowledgment.

  The use of Error Acknowledgment remains unchanged for ExtendedTransfers.

  **Note:** Because write data comes after the corresponding acknowledgment, it may be necessary for a slave to use LErr* instead of Error Acknowledgment to signal data errors. Error Acknowledgment can still be used to signal addressing or Extended Transfer Information errors.

- Rerun Acknowledgment.

  The use of Rerun Acknowledgment remains unchanged for ExtendedTransfers.

A slave must not generate the following acknowledgments during an ExtendedTransfer:

- Byte Acknowledgment.

  An SBus master or controller receiving a byte acknowledgment during an ExtendedTransfer must abort the bus cycle.

- Half-word Acknowledgment.

  A master or controller receiving a half-word acknowledgment during an ExtendedTransfer must abort the bus cycle.

## Timeouts

The timeout rules for the SBus remain unchanged for ExtendedTransfers.

## LateError*

During ExtendedTransfers, the timing for LErr* is the same as for 32-bit transfers: LErr* must be asserted for one clock cycle, beginning two clock cycles after the associated acknowledgment LErr* must then be driven to its unasserted state for one clock cycle, after which it must be tristated.

## DataParity

The use of DtaPar during ExtendedTransfers is similar to its use during 32-bit transfers. SBus devices should not check DtaPar unless they are enabled to do so.

If enabled, DtaPar must be generated as follows. During the translation cycle, DtaPar must be a check only on D(31:0). Beginning with the clock cycle after the first double-word acknowledgment, whenever D(63:0) has valid data, DtaPar must be a check on D(63:0). During any clock cycle in which D(31:0) must be tristated, DtaPar must also be tristated.

## Compatibility Considerations

Except for 128 byte transfers, an SBus device supporting ExtendedTransfers must support 32-bit SBus transfers of the same size (in number of bytes). SBus devices must also support 1 word 32-bit transfers.

It must be possible to program a master to perform 32-bit transfers only.

An SBus master should not initiate ExtendedTransfers unless the intended slave is known via Open Boot to support ExtendedTransfers.

## Signal Termination

| | |
|---|---|
| Size(2:0) | SBus controllers supporting ExtendedTransfers must pull Siz(2) using a 2 KΩ resistor to ground and Siz(1:0) using a 10 KΩ resistor to the +5V supply, or use holding amplifers. |
| Read | SBus controllers supporting ExtendedTransfers must pull Rd using a 2 KΩ resistor to ground or a holding amplifer. |

# C

# *FCode Reference*

This appendix contains two lists: FCode primitives and FCode byte values. FCode primitives are grouped according to function, while FCode byte values appear in hexadecimal order.

## FCode Primitives

The following figures describe FCodes currently supported by the Open Boot PROM. New 2.0 FCodes are indicated by the comment, "valid only in 2.0 or greater systems." Both the FCode token values and Forth names are included. A token value entry of CR indicates a cross-compiler-generated sequence, while - indicates that no FCode is generated.

*Figure C-1. Stack Manipulation*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 51 | depth | ( -- +n ) | How many items on stack? |
| 46 | drop | ( n -- ) | Removes n from the stack |
| 52 | 2drop | ( n1 n2 -- ) | Removes 2 items from stack |
| 47 | dup | ( n -- n n ) | Duplicates n |
| 53 | 2dup | ( n1 n2 -- n1 n2 n1 n2 ) | Duplicates 2 stack items |
| 50 | ?dup | ( n -- n n \| 0 ) | Duplicates n if it is non-zero |
| CR | 3dup | ( n1 n2 n3 -- n1 n2 n3 n1 n2 n3 ) | Copies top 3 stack items |
| 4d | nip | ( n1 n2 -- n2 ) | Discards the second stack item |
| 48 | over | ( n1 n2 -- n1 n2 n1 ) | Copies second stack item to top of stack |
| 54 | 2over | ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 ) | Copies 2 stack items |
| 4e | pick | ( +n -- n2 ) | Copies +n-th stack item |
| 30 | >r | ( n -- ) ( rs: -- n) | Moves a stack item to the return stack * |
| 31 | r> | ( -- n ) ( rs: n -- ) | Moves item from return stack to data stack * |
| * Use these FCodes cautiously. | | | |
| 32 | r@ | ( -- n ) ( rs: -- ) | Copies the top of the return stack to the data stack |
| 4f | roll | ( +n -- ) | Rotates +n stack items |

| | | | |
|---|---|---|---|
| 4a | rot | ( n1 n2 n3 -- n2 n3 n1 ) | Rotates 3 stack items |
| 4b | -rot | ( n1 n2 n3 -- n3 n1 n2 ) | Shuffles top 3 stack items |
| 56 | 2rot | ( n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2 ) | Rotates 3 pairs of stack items |
| 49 | swap | ( n1 n2 -- n2 n1 ) | Exchanges the top 2 stack items |
| 55 | 2swap | ( n1 n2 n3 n4 -- n3 n4 n1 n2 ) | Exchanges 2 pairs of stack items |
| 4c | tuck | ( n1 n2 -- n2 n1 n2 ) | Copies the top stack item below the second item |

*Figure C-2. Arithmetic Operations*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 20 | * | ( n1 n2 -- n3 ) | Multiplies n1 times n2 |
| 1e | + | ( n1 n2 -- n3 ) | Adds n1+n2 |
| 1f | - | ( n1 n2 -- n3 ) | Subtracts n1-n2 |
| 21 | / | ( n1 n2 -- quot ) | Divides n1/n2 |
| CR | 1+ | ( n1 -- n2 ) | Adds one |
| CR | 1- | ( n1 -- n2 ) | Subtracts one |
| 59 | 2* | ( n1 -- n2 ) | Multiplies by 2 |
| 57 | 2/ | ( n1 -- n2 ) | Divides by 2 |
| 27 | << | ( n1 +n -- n2 ) | Left shifts n1 by +n places |
| 28 | >> | ( n1 +n -- n2 ) | Right shifts n1 by +n places |
| CR | <<a | ( n1 +n -- n2 ) | Arithmetic left shifts (same as <<) |
| 29 | >>a | ( n1 +n -- n2 ) | Arithmetic right shifts n1 by +n places |
| 2d | abs | ( n -- u ) | Absolute value |
| ae | aligned | ( adr1 -- adr2 ) | Adjusts an address to a machine word boundary |
| 23 | and | ( n1 n2 -- n3 ) | Logical and |
| ac | bounds | ( startadr len -- endadr startadr ) | Converts start,len to end,start for DO loop |
| 2f | max | ( n1 n2 -- n3 ) | n3 is maximum of n1 and n2 |
| 2e | min | ( n1 n2 -- n3 ) | n3 is minimum of n1 and n2 |
| 22 | mod | ( n1 n2 -- rem ) | Remainder of n1/n2 |
| CR | */mod | ( n1 n2 n3 -- rem quot ) | Rem, quotient of n1*n2/n3 |
| 2a | /mod | ( n1 n2 -- rem quot ) | Remainder, quotient of n1/n2 |
| 2c | negate | ( n1 -- n2 ) | Changes the sign of n1 |
| 26 | not | ( n1 -- n2 ) | One's complement |
| 24 | or | ( n1 n2 -- n3 ) | Logical or |
| 2b | u/mod | ( ul un -- un.rem un.quot ) | Unsigned 32-bit divide of ul/un |
| 58 | u2/ | ( u1 -- u2 ) | Logical right shifts 1 bit |
| 25 | xor | ( n1 n2 -- n3 ) | Exclusive or |
| * The following four FCodes are valid only in 2.0 or greater systems. | | | |
| d4 | u*x | ( u1[32] u2[32] -- product[64] ) | Multiplies two unsigned 32-bit numbers, yields an unsigned 64-bit product |
| d5 | xu/mod | ( u1[64] u2[32] -- remainder[32] quot[32] ) | Divides an unsigned 64-bit number by an unsigned 32-bit number, yields a 32-bit remainder and quotient |
| d8 | x+ | ( x1 x2 -- x3 ) | Adds two 64-bit numbers |
| d9 | x- | ( x1 x2 -- x3 ) | Subtracts two 64-bit numbers |

*Figure C-3. Memory Operations*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 72 | ! | ( n adr -- ) | Stores a 32-bit number into the variable at adr |
| 6c | +! | ( n adr -- ) | Adds n to the 32-bit number stored in the variable at adr |
| 77 | 2! | ( n1 n2 adr -- ) | Stores 2 numbers at adr; n2 at lower address |
| 76 | 2@ | ( adr -- n1 n2 ) | Fetches 2 numbers from adr; n2 from lower address |
| 6d | @ | ( adr -- n ) | Fetches a number from the variable at adr |
| CR | ? | ( adr16 -- ) | Displays the 32-bit number at adr |
| 75 | c! | ( n adr -- ) | Stores low byte of n at adr |
| 71 | c@ | ( adr -- byte ) | Fetches a byte from adr |
| CR | blank | ( adr len -- ) | Sets len bytes of memory to ASCII space, starting at adr |
| CR | cmove | ( adr1 adr2 u -- ) | Same as MOVE |
| CR | cmove> | ( adr1 adr2 u -- ) | Same as MOVE |
| 7a | comp | ( adr1 adr2 len -- n ) | Compares two byte arrays including case. n=0 if same |
| CR | erase | ( adr len -- ) | Sets len bytes of memory to zero, starting at adr |
| 79 | fill | ( adr u byte -- ) | Sets u bytes of memory to byte |
| 73 | l! | ( l adr -- ) | Stores the 32-bit number at adr, must be 32-bit aligned |
| 6e | l@ | ( adr -- l ) | Fetches the 32-bit longword at adr, must be 32-bit aligned |
| 78 | move | ( adr1 adr2 u -- ) | Copies u bytes from adr1 to adr2, handles overlap correctly. |
| 6b | off | ( adr -- ) | Stores false (32-bit 0) at adr |
| 6a | on | ( adr -- ) | Stores true (32-bit -1) at adr |
| 74 | w! | ( w adr -- ) | Stores a 16-bit word at adr, must be 16-bit aligned |
| 6f | w@ | ( adr -- w ) | Fetches the unsigned 16-bit word at adr, must be 16-bit aligned |
| 70 | <w@ | ( adr -- n ) | Fetches the signed 16-bit word at adr, must be 16-bit aligned |

*Figure C-4. Comparison Operations*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 36 | 0< | ( n -- flag ) | True if n < 0 |
| 37 | 0<= | ( n -- flag ) | True if n <= 0 |
| 35 | 0<> | ( n -- flag ) | True if n <> 0 |
| 34 | 0= | ( n -- flag ) | True if n = 0, also inverts any flag |
| 38 | 0> | ( n -- flag ) | True if n > 0 |
| 39 | 0>= | ( n -- flag ) | True if n >= 0 |
| 3a | < | ( n1 n2 -- flag ) | True if n1 < n2 |
| 43 | <= | ( n1 n2 -- flag ) | True if n1 <= n2 |
| 3d | <> | ( n1 n2 -- flag ) | True if n1 <> n2 |
| 3c | = | ( n1 n2 -- flag ) | True if n1 = n2 |
| 3b | > | ( n1 n2 -- flag ) | True if n1 > n2 |
| 42 | >= | ( n1 n2 -- flag ) | True if n1 >= n2 |
| 44 | between | ( n min max -- flag ) | True if min <= n <= max |
| CR | false | ( -- 0 ) | The value FALSE |
| CR | true | ( -- -1 ) | The value TRUE |
| 40 | u< | ( u1 u2 -- flag ) | True if u1 < u2, unsigned |
| 3f | u<= | ( u1 n2 -- flag ) | True if u1 <= u2, unsigned |
| 3e | u> | ( u1 n2 -- flag ) | True if u1 > u2, unsigned |
| 41 | u>= | ( u1 n2 -- flag ) | True if u1 >= u2, unsigned |
| 45 | within | ( n min max -- flag ) | True if min <= n < max |

*Figure C-5. Text Input*

| Byte | Function | Stack | Description |
|---|---|---|---|
| - | ( *text*) | ( -- ) | Begins a comment (ignored) |
| - | \ | ( -- ) | Ignore rest of line (comment) |
| CR | ascii *x* | ( -- char ) | ASCII value of next character |
| CR | control *x* | ( -- char ) | Interprets next character as ASCII CONTROL character |
| 8e | key | ( -- char ) | Reads a character from the keyboard |
| 8d | key? | ( -- flag ) | True if a key has been typed on the keyboard |
| 8a | expect | ( adr +n -- ) | Gets a line of edited input from the keyboard; store at adr |
| 88 | span | ( -- adr ) | Variable containing the number of characters read by EXPECT |
| - | (s *text*) | ( -- ) | Begins a comment (ignored) |

*Figure C-6. ASCII Constants*

| Byte | Function | Stack | Description |
|---|---|---|---|
| ab | bell | ( -- n ) | The ASCII code for the bell character; decimal 7 |
| a9 | bl | ( -- n ) | The ASCII code for the space character; decimal 32 |
| aa | bs | ( -- n ) | The ASCII code for the backspace character; decimal 8 |
| CR | carret | ( -- n ) | The ASCII code for the carriage return character; decimal 13 |
| CR | linefeed | ( -- n ) | The ASCII code for the linefeed character; decimal 10 |
| CR | newline | ( -- n ) | The ASCII code for the newline character; decimal 10 |

*Figure C-7. Numeric Input*

| Byte | Function | Stack | Description |
|---|---|---|---|
| a4 | -1 | ( -- n ) | Constant -1 |
| a5 | 0 | ( -- n ) | Constant 0 |
| a6 | 1 | ( -- n ) | Constant 1 |
| a7 | 2 | ( -- n ) | Constant 2 |
| a8 | 3 | ( -- n ) | Constant 3 |
| CR | b# *number* | ( -- n ) | Interprets next number in binary |
| - | binary | ( -- ) | If outside definition, input text in binary |
| CR | d# *number* | ( -- n ) | Interprets next number in decimal |
| - | decimal | ( -- ) | If outside definition, input text in decimal |
| CR | h# *number* | ( -- n ) | Interprets next number in hexadecimal |
| - | hex | ( -- ) | If outside definition, input text in hexadecimal |
| CR | o# *number* | ( -- n ) | Interprets next number in octal |
| - | octal | ( -- ) | If outside definition, input text in octal |

*Figure C-8. Numeric Primitives*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 99 | # | ( +l1 -- +l2 ) | Converts a digit in pictured numeric output |
| 97 | #> | ( l -- adr +n ) | Ends pictured numeric output |
| 96 | <# | ( -- ) | Initializes pictured numeric output |
| a0 | base | ( -- adr ) | USER variable containing number base |
| a3 | digit | ( char base -- digit true \| char false ) | Converts a character to a digit |
| 95 | hold | ( char -- ) | Inserts the char in the pictured numeric output string |
| 9a | #s | ( +l -- 0 ) | Converts the rest of the digits in pictured numeric output |
| 98 | sign | ( n -- ) | Sets sign of pictured output |
| The following FCode is valid only in 2.0 or greater systems. | | | |
| a2 | $number | ( adr len -- true \| n false ) | Converts a string to a number |

*Figure C-9. Numeric Output*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 9d | . | ( n -- ) | Displays a number |
| CR | binary | ( -- ) | If inside definition, output in binary |
| CR | .d | ( n -- ) | Displays number in decimal |
| CR | decimal | ( -- ) | If inside definition, output in decimal |
| CR | .h | ( n -- ) | Displays number in hexadecimal |
| CR | hex | ( -- ) | If inside definition, output in hexadecimal |
| CR | octal | ( -- ) | If inside definition, output in octal |
| 9e | .r | ( n +n -- ) | Displays a number in a fixed width field |
| 9f | .s | ( -- ) | Displays the contents of the data stack |
| CR | s. | ( n -- ) | Displays n as a signed number |
| 9b | u. | ( u -- ) | Displays an unsigned number |
| 9c | u.r | ( u +n -- ) | Prints an unsigned number in a fixed width field |

*Figure C-10. General-purpose Output*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | ." *text"* | ( -- ) | Compiles string for later output |
| CR | .( *text*) | ( -- ) | Displays a string now |
| 91 | (cr | ( -- ) | Outputs ASCII CR character; decimal 13 |
| 92 | cr | ( -- ) | Starts a new line of display output |
| 8f | emit | ( char -- ) | Displays the character |
| CR | space | ( -- ) | Outputs a single space character |
| CR | spaces | ( +n -- ) | Outputs +n spaces |
| 90 | type | ( adr +n -- ) | Displays n characters |

*Figure C-11. Formatted Output*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 94 | #line | ( -- adr ) | Variable holding the line number on the output device |
| 93 | #out | ( -- adr ) | Variable holding the column number on the output device |

*Figure C-12. BEGIN Loops*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | again | ( -- ) | Ends BEGIN..AGAIN (infinite) loop |
| CR | begin | ( -- ) | Starts conditional loop |
| CR | repeat | ( -- ) | Returns to loop start |
| CR | until | ( flag -- ) | If true, exits BEGIN..UNTIL loop |
| CR | while | ( flag -- ) | If true, continues BEGIN..WHILE..REPEAT loop, else exits loop |

*Figure C-13. Conditionals*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | if | ( flag -- ) | If true, executes next FCode(s) |
| CR | else | ( -- ) | (optional) Executes next FCode(s) if IF failed |
| CR | then | ( -- ) | Terminates IF..ELSE..THEN |

*Figure C-14. DO Loops*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | do | ( end start -- ) | Loops, index *start* to *end-1* inclusive |
| CR | ?do | ( end start -- ) | Like DO, but skips loop if *end = start* |
| 19 | i | ( -- n ) | Returns current loop index value |
| 1a | j | ( -- n ) | Returns value of next outer loop index |
| CR | leave | ( -- ) | Exits DO loop immediately |
| CR | ?leave | ( flag -- ) | If flag is true, exits DO loop |
| CR | loop | ( -- ) | Increments index, returns to DO |
| CR | +loop | ( n -- ) | Increments by n, returns to DO. If n<0, index *start* to *end* |

*Figure C-15. Control Words*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 1d | execute | ( acf -- ) | Executes the word whose compilation address is on the stack |
| 33 | exit | ( -- ) | Returns from the current word |

*Figure C-16. Strings*

| Byte 1 | Byte 2 | Function | Stack | Description |
|---|---|---|---|---|
| CR | | *" text"* | ( -- adr len ) | Collects a string |
| 84 | | count | ( pstr -- adr +n ) | Unpacks a packed string |
| 82 | | lcc | ( char -- lower-case-char ) | Converts char to lower case |
| 83 | | pack | ( adr len pstr -- pstr ) | Makes a packed string from adr len, placing it at pstr |
| 81 | | upc | ( char -- upper-case-char ) | Converts char to upper case |
| The following FCode is valid only in 2.0 or greater systems. | | | | |
| 2 | 40 | left-parse-string | ( adr len char -- adrR lenR adrL lenL ) | Splits a string at the given delimiter (which is discarded) |

*Figure C-17. Defining Words*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | : (colon) *name* | ( -- ) | Begins colon definition |
| CR | ; (semicolon) | ( -- ) | Ends colon definition |
| - | alias *newname oldname* | ( -- ) | Creates *newname* with behavior of *oldname* |
| CR | buffer: *name* | ( size -- ) | Creates data array of *size* bytes |
| CR | constant *name* | ( n -- ) | Creates a constant |
| CR | create *name* | ( -- ) | Generic defining word |
| CR | defer *name* | ( -- ) | Execution vector (change with IS) |
| CR | field *name* | ( offset size -- offset+size ) | Creates a named offset pointer |
| CR | struct | ( -- 0 ) | Initializes for FIELD creation |
| CR | variable *name* | ( -- ) | Creates a data variable |
| CR | value *name* | ( n -- ) | Creates named VALUE-type variable (change with IS) |

*Figure C-18. Dictionary Compilation*

| Byte | Function | Stack | Description |
|---|---|---|---|
| d3 | , | ( n -- ) | Places a number in the dictionary |
| d0 | c, | ( n -- ) | Places a byte in the dictionary |
| ad | here | ( -- adr ) | Address of top of dictionary |
| d2 | l, | ( l -- ) | Places a 32-bit longword in the dictionary |
| d1 | w, | ( w -- ) | Places a 16-bit word in the dictionary |
| CR | is *name* | ( n -- ) | Changes value in a *defer* word or a *value* |

*Figure C-19. Dictionary Search*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | ' *name* | ( -- acf ) | Finds the word (while executing) |
| CR | ['] *name* | ( -- acf ) | Finds word (while compiling) |
| cb | $find | ( adr len -- adr len false \| acf +-1 ) | Finds a name in the Open PROM |
| The following FCode is valid only in 2.0 or greater systems. | | | |
| cd | eval | ( adr len -- ) | Executes FORTH commands within a string |

*Figure C-20. Conversions Operators*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 7f | bljoin | ( b.low b2 b3 b.hi -- l ) | Joins four bytes to form a longword |
| b0 | bwjoin | ( b.low b.hi -- w ) | Joins two bytes to form a 16-bit word |
| 5a | /c | ( -- n ) | Address increment for a byte; 1 |
| 66 | /c* | ( n1 -- n2 ) | Multiplies by /C |
| 5e | ca+ | ( adr1 index -- adr2 ) | Increments adr1 by index times /C |
| 62 | ca1+ | ( adr1 -- adr2 ) | Increments adr1 by /C |
| 80 | flip | ( w1 -- w2 ) | Swaps the bytes within a 16-bit word |
| 5c | /l | ( -- n ) | Address increment for a 32-bit longword; 4 |
| 68 | /l* | ( n1 -- n2 ) | Multiplies by /L |
| 60 | la+ | ( adr1 index -- adr2 ) | Increments adr1 by index times /L |
| 64 | la1+ | ( adr1 -- adr2 ) | Increments adr1 by /L |
| 7e | lbsplit | ( l -- b.low b2 b3 b.high ) | Splits a longword into four bytes |
| 7c | lwsplit | ( l -- w.low w.high ) | Splits a longword into two words |
| 5d | /n | ( -- n ) | Address increment for a normal; 4 |
| 69 | /n* | ( n1 -- n2 ) | Multiplies by /N |
| 61 | na+ | ( adr1 index -- adr2 ) | Increments adr1 by index times /N |
| 65 | na1+ | ( adr1 -- adr2 ) | Increments adr1 by /N |
| 5b | /w | ( -- n ) | Address increment for a 16-bit word; 2 |
| 67 | /w* | ( n1 -- n2 ) | Multiplies by /W |
| 5f | wa+ | ( adr1 index -- adr2 ) | Increments adr1 by index times /W |
| 63 | wa1+ | ( adr1 -- adr2 ) | Increments adr1 by /W |
| af | wbsplit | ( w -- b.low b.high ) | Splits a 16-bit word into two bytes |
| CR | wflip | ( l1 -- l2 ) | Swaps halves of 32-bit longword |
| 7d | wljoin | ( w.low w.high -- l ) | Joins two words to form a longword |

*Figure C-21. Memory Buffers Allocation*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 8b | alloc-mem | ( nbytes -- adr ) | Allocates nbytes of memory and returns its address |
| 8c | free-mem | ( adr nbytes -- ) | Frees memory allocated by ALLOC-MEM |

*Figure C-22. Miscellaneous Operators*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 86 | >body | ( acf -- apf ) | Finds parameter field address from compilation address |
| 85 | body> | ( apf -- acf ) | Finds compilation address from parameter field address |
| CR | emit-byte | ( n -- ) | Outputs FCode byte (use with TOKENIZER[) |
| 00 | end0 | ( -- ) | Marks the end of Fcode |
| ff | end1 | ( -- ) | Alternates form for END0 (not recommended) |
| CR | fcode-version1 | ( -- ) | Begins FCode program |
| - | fload *filename* | ( -- ) | Begins tokenizing *filename* |
| - | headerless | ( -- ) | Creates new names with NEW-TOKEN (no name fields) |
| - | headers | ( -- ) | Creates new names with NAMED-TOKEN (default) |
| 7b | noop | ( -- ) | Does nothing |
| cc | offset16 | ( -- ) | All further branches use 16-bit offsets (instead of 8-bit) |
| - | tokenizer[ | ( -- ) | Begins tokenizer program commands |
| - | ]tokenizer | ( -- ) | Ends tokenizer program commands |
| 87 | version | ( -- n ) | Returns the version# of the Fcode interpreter |
| The following two FCodes are valid only in 2.0 or greater systems. | | | |
| CR | fcode-version2 | ( -- ) | Begins 2.0 FCode program, compiles START1 |
| - | external | ( -- ) | Creates new names with EXTERNAL-TOKEN |

*Figure C-23. Internal Operators (invalid for program text)*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 1-f | table#1-15 | | Reserved byte codes, used for 2-byte entries |
| 10 | b(lit) | ( -- n ) | Followed by 32-bit#. Compiled by numeric data |
| 11 | b(') | ( -- acf ) | Followed by a token (1 or 2-byte code) . Compiled by ['] or ' |
| 12 | b(") | ( -- adr len ) | Followed by count byte, text. Compiled by " or ." |
| c3 | b(is) | ( n -- ) | Compiled by IS |
| fd | version1 | ( -- ) | Followed by null byte, checksum (2 bytes) , length (4 bytes). Compiled by FCODE-VERSION1, as the first Fcode bytes |
| fe | 4-byte-id | ( -- ) | Followed by 3 identifier bytes. First Fcode byte. Not supported. Used if no Fcode available, to uniquely identify device |
| 13 | bbranch | ( -- ) | Followed by 8-bit offset. Compiled by ELSE or AGAIN |
| 14 | b?branch | ( -- ) | Followed by 8-bit offset. Compiled by IF or UNTIL |
| 15 | b(loop) | ( -- ) | Followed by 8-bit offset. Compiled by LOOP |
| 16 | b(+loop) | ( n -- ) | Followed by 8-bit offset. Compiled by +LOOP |
| 17 | b(do) | ( end start -- ) | Followed by 8-bit offset. Compiled by DO |
| 18 | b(?do) | ( end start -- ) | Followed by 8-bit offset. Compiled by ?DO |
| 1b | b(leave) | ( -- ) | Compiled by LEAVE or ?LEAVE |
| b1 | b(<mark) | ( -- ) | Compiled by BEGIN |
| b2 | b(>resolve) | ( -- ) | Compiled by ELSE or THEN |
| c4 | b(case) | ( -- ) | Compiled by CASE |
| c5 | b(endcase) | ( -- ) | Compiled by ENDCASE |
| c6 | b(endof) | ( -- ) | Compiled by ENDOF |
| 1c | b(of) | ( sel testval -- sel \| none ) | Followed by 8-bit offset. Compiled by OF |
| b5 | new-token | ( -- ) | Followed by table#, code#, token-type. Compiled by any defining word. Headerless, not used normally. |
| b6 | named-token | ( -- ) | Followed by packed string (count,text), table#, code#, token-type. Compiled by any defining word (: VALUE CONSTANT etc.) |
| b7 | b(:) | | Token-type compiled by : |

| | | | |
|---|---|---|---|
| b8 | b(value) | | Token-type compiled by VALUE |
| b9 | b(variable) | | Token-type compiled by VARIABLE |
| ba | b(constant) | | Token-type compiled by CONSTANT |
| bb | b(create) | | Token-type compiled by CREATE |
| bc | b(defer) | | Token-type compiled by DEFER |
| bd | b(buffer:) | | Token-type compiled by BUFFER: |
| be | b(field) | | Token-type compiled by FIELD |
| c2 | b(;) | ( -- ) | End a colon definition. Compiled by ; |

The following five FCodes are valid only in 2.0 or greater systems.

| | | | |
|---|---|---|---|
| ca | external-token | ( -- ) | Like NAMED-TOKEN, but name header is *always* created at probe time |
| f0 | start0 | ( -- ) | Like VERSION1, but for version 2.0 FCodes. Uses 16-bit branches. Fetches successive tokens from same address |
| f1 | start1 | ( -- ) | Like VERSION1, but for version 2.0 FCodes. Uses 16-bit branches. Fetches successive tokens from consecutive addresses. Compiled by FCODE-VERSION2 |
| f2 | start2 | ( -- ) | Like VERSION1, but for version 2.0 FCodes. Uses 16-bit branches. Fetches successive tokens from consecutive 16-bit addresses |
| f3 | start4 | ( -- ) | Like VERSION1, but for version 2.0 FCodes. Uses 16-bit branches. Fetches successive tokens from consecutive 32-bit addresses |

*Figure C-24. Memory Allocation*

| Byte 1 | Byte 2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 01 | dma-alloc | ( nbytes -- virt ) | Maps in nbytes of DMA space, return virt. adr |
| 1 | 02 | my-address | ( -- phys ) | Returns the physical adr of this plug-in device. "phys" is a "magic" number, usable by other routines |
| 1 | 03 | my-space | ( -- space ) | Returns address space of plug-in device. "space" is a "magic" number, usable by other routines |
| 1 | 04 | memmap | ( phys space nbytes -- virt ) | Maps in a region, return virtual adrress |
| 1 | 05 | free-virtual | ( virt nbytes -- ) | Frees virtual memory from MEMMAP, DMA-ALLOC,or MAP-SBUS |
| 1 | 06 | >physical | ( virt -- phys space ) | Returns physical adr and space for virt. adr |

*Figure C-25. Non-volatile Parameters*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 0f | my-params | ( -- adr len ) | Returns a data array for this plug-in device. The data format is defined specifically for each plug-in device, in order to customize the device. Params for each device, as needed, will be stored in the system NVRAM |

*Figure C-26. Device Information*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 10 | attribute | ( xdr-adr xdr-len name-adr name-len -- ) | Declares an attribute with the given value structure, for the given name string. |
| 1 | 11 | xdrint | ( n -- xdr-adr xdr-len ) | Converts a number into a numeric attribute structure |
| 1 | 12 | xdr+ | ( xdr-adr1 xdr-len1 xdr-adr2 xdr-len2 -- xdr-adr1 xdr-len1+2 ) | Merges two attribute structures. They must have been created sequentially |
| 1 | 13 | xdrphys | ( phys space -- xdr-adr xdr-len ) | Converts physical address and space into an attribute structure |
| 1 | 14 | xdrstring | ( adr len -- xdr-adr xdr-len ) | Converts a string into a value structure |
| The following FCode is valid only for 2.1 or greater systems. | | | | |
| 1 | 15 | xdrbytes | ( adr len -- xdr-adr xdr-len ) | Converts a byte array into a value structure |

*Figure C-27. Commmonly-used Attributes*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 16 | reg | ( phys space size -- ) | Declares location and size of device registers |
| 1 | 17 | intr | ( intr-level vector -- ) | Declares interrupt level and vector for this device |
| 1 | 18 | driver | ( adr len -- ) | Declares driver for this device, not supported |
| 1 | 19 | model | ( adr len -- ) | Declares model# for this device, such as " SUNW,501-1415-01" |
| 1 | 1a | device-type | ( adr len -- ) | Declares type of device, e.g. " display", " disk", " network", or " byte" |
| CR | | name | ( adr len -- ) | Declares SunOS driver name, as in " SUNW,zebra" |
| The following four FCodes are valid in 2.0 or greater systems. | | | | |
| 2 | 01 | device-name | ( adr len -- ) | Creates the "name" attribute with the given value |
| 2 | 10 | processor-type | ( -- processor-type ) | Returns a code value for the type of CPU. Defined values: 1-MC68000, 2-MC68010, 3-MC68020, 4-MC68030, 5-SPARC, 6-i80386, 7-i80486, 8-MIPS, 9-MC88000, A-AMD29000 |
| 2 | 11 | firmware-version | ( -- n ) | Returns major/minor CPU firmware version, that is, 0x00020001 = firmware version 2.1 |
| 2 | 12 | fcode-version | ( -- n ) | Returns major/minor FCode version supported, that is, 0x00020000 = FCode version 2.0 |

*Figure C-28. Device Activation Vector Setup*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 1c | is-install | ( acf -- ) | Identifies "install" routine to allocate a boot device |
| 1 | 1d | is-remove | ( acf -- ) | Identifies "remove" routine, to deallocate a device |
| 1 | 1e | is-selftest | ( acf -- ) | Identifies "selftest" routine for this device |
| 1 | 1f | new-device | ( -- ) | Opens an additional device, using this driver package |
| 1 | 27 | finish-device | ( -- ) | Closes out current device, ready for NEW-DEVICE |

*Figure C-29. Self-test utility Routines*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 20 | diagnostic-mode? | ( -- flag ) | Returns "true" if extended diagnostics are desired |
| 1 | 21 | display-status | ( n -- ) | Outputs a selftest status message, with given status# |
| 1 | 22 | memory-test-suite | ( adr len -- status ) | Calls memory tester for given region |
| 1 | 23 | group-code | ( -- adr ) | Variable, used by MEMORY-TEST-SUITE (obsolete) |
| 1 | 24 | mask | ( -- adr ) | Variable, holds "mask" used by MEMORY-TEST-SUITE |

*Figure C-30. Time Utilities*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 25 | get-msecs | ( -- ms ) | Returns the current time, in milliseconds, approx. |
| 1 | 26 | ms | ( n -- ) | Delays for n milliseconds. Resolution is 1 millisecond |

*Figure C-31. Machine-specific Support*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 30 | map-sbus | ( phys size -- virt ) | Maps a region of memory in 'sbus' address space |
| 1 | 31 | sbus-intr>cpu | ( sbus-intr# -- cpu-intr# ) | Translates SBus interrupt# into CPU interrupt# |

**Note:** Figures C-32 thru C-38 apply only to *display* device-types.

*Figure C-32. User-set terminal Emulation Values*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 50 | #lines | ( -- n ) | # of lines of text being used for display. This word MUST be initialized (using IS). FBx-INSTALL does this automatically, and also properly incorporates the NVRAM parameter "screen-#rows" |
| 1 | 51 | #columns | ( -- n ) | # of columns (chars/line) used for display. This word MUST be initialized (using IS). FBx-INSTALL does this automatically, and also properly incorporates the NVRAM parameter "screen-#columns" |

*Figure C-33. Terminal Emulator-set Terminal Emulation Values*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 52 | line# | ( -- n ) | Current cursor position (line#). 0 is top line |
| 1 | 53 | column# | ( -- n ) | Current cursor position (column#). 0 is left char. |
| 1 | 54 | inverse? | ( -- flag ) | True if output is inverted (white-on-black) |
| 1 | 55 | inverse-screen? | ( -- flag ) | True if screen has been inverted (black background) |

*Figure C-34. Terminal Emulation Routines**

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 57 | draw-character | ( char -- ) | Paints the given character and advance the cursor |
| 1 | 58 | reset-screen | ( -- ) | Initializes the display device |
| 1 | 59 | toggle-cursor | ( -- ) | Draws or erase the cursor |
| 1 | 5a | erase-screen | ( -- ) | Clears all pixels on the display |
| 1 | 5b | blink-screen | ( -- ) | Flashes the display momentarily |
| 1 | 5c | invert-screen | ( -- ) | Changes all pixels to the opposite color |
| 1 | 5d | insert-characters | ( n -- ) | Inserts n blanks just before the cursor |
| 1 | 5e | delete-characters | ( n -- ) | Deletes n characters starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 5f | insert-lines | ( n -- ) | Inserts n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 60 | delete-lines | ( n -- ) | Deletes n lines starting with the current line, lower lines are scrolled upward |
| 1 | 61 | draw-logo | ( line# logoaddr logowidth logoheight -- ) | Draws the logo |

* DEFER-type loadable routines.

*Figure C-35. Frame Buffer Parameter Values**

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 6c | char-height | ( -- n ) | Height (in pixels) of a character (usually 22) |
| 1 | 6d | char-width | ( -- n ) | Width (in pixels) of a character (usually 12) |
| 1 | 6f | fontbytes | ( -- n ) | # of bytes/scan line for font entries (usually 2) |
| 1 | 62 | frame-buffer-adr | ( -- adr ) | Address of frame buffer memory |
| 1 | 63 | screen-height | ( -- n ) | Total height of the display (in pixels) |
| 1 | 64 | screen-width | ( -- n ) | Total width of the display (in pixels) |
| 1 | 65 | window-top | ( -- n ) | Distance (in pixels) between display top and text window |
| 1 | 66 | window-left | ( -- n ) | Distance (in pixels) between display left edge and text window left edge |

* These must all be initialized before using any FBx- routines.

*Figure C-36. Font Operators*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 6a | default-font | ( -- fontbase charwidth charheight fontbytes #firstchar #chars ) | Returns default font values, plugs directly into SET-FONT |
| 1 | 6b | set-font | ( fontbase charwidth charheight fontbytes #firstchar #chars -- ) | Sets the character font for text output |
| 1 | 6e | >font | ( char -- adr ) | Returns font address for given ASCII character |

*Figure C-37. One-bit Framebuffer Utilities*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 70 | fb1-draw-character | ( char -- ) | Paints the character and advance the cursor |
| 1 | 71 | fb1-reset-screen | ( -- ) | Initializes the display device (noop) |
| 1 | 72 | fb1-toggle-cursor | ( -- ) | Draws or erases the cursor |
| 1 | 73 | fb1-erase-screen | ( -- ) | Clears all pixels on the display |
| 1 | 74 | fb1-blink-screen | ( -- ) | Inverts the screen, twice (slow) |
| 1 | 75 | fb1-invert-screen | ( -- ) | Changes all pixels to the opposite color |
| 1 | 76 | fb1-insert-characters | ( n -- ) | Inserts n blanks just before the cursor |
| 1 | 77 | fb1-delete-characters | ( n -- ) | Deletes n characters, starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 78 | fb1-insert-lines | ( n -- ) | Inserts n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 79 | fb1-delete-lines | ( n -- ) | Deletes n lines starting with the current line,lower lines are scrolled upward |
| 1 | 7a | fb1-draw-logo | ( line# logoaddr logowidth logoheight -- ) | Draws the logo |
| 1 | 7b | fb1-install | ( width height #columns #lines -- ) | Installs the one-bit built-in routines |
| 1 | 7c | fb1-slide-up | ( n -- ) | Like FB1-DELETE-LINES, but doesn't clear lines at bottom |

*Figure C-38. Eight-bit Framebuffer Utilities*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 80 | fb8-draw-character | ( char -- ) | Paints the character and advance the cursor |
| 1 | 81 | fb8-reset-screen | ( -- ) | Initializes the display device (noop) |
| 1 | 82 | fb8-toggle-cursor | ( -- ) | Draws or erases the cursor |
| 1 | 83 | fb8-erase-screen | ( -- ) | Clears all pixels on the display |
| 1 | 84 | fb8-blink-screen | ( -- ) | Inverts the screen, twice (slow) |
| 1 | 85 | fb8-invert-screen | ( -- ) | Changes all pixels to the opposite color |
| 1 | 86 | fb8-insert-characters | ( n -- ) | Inserts n blanks just before the cursor |
| 1 | 87 | fb8-delete-characters | ( n -- ) | Deletes n characters starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 88 | fb8-insert-lines | ( n -- ) | Inserts n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 89 | fb8-delete-lines | ( n -- ) | Deletes n lines starting with the current line, lower lines are scrolled upward |
| 1 | 8a | fb8-draw-logo | ( line# logoaddr logowidth logoheight -- ) | Draws the logo |
| 1 | 8b | fb8-install | ( width height #columns #lines -- ) | Installs the eight-bit built-in routines. |

**Note:** The FCodes in figures C-39 thru C-46 are valid only in 2.0 or greater systems.

*Figure C-39. Package Support*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 02 | my-args | ( -- adr len ) | Returns the argument string "adr len" passed when this package was opened |
| 2 | 03 | my-self | ( -- ihandle ) | Returns the instance handle of currently-executing package instance |
| 2 | 04 | find-package | ( adr len -- false \| phandle true ) | Finds a package named "adr len" |
| 2 | 05 | open-package | ( adr len phandle -- ihandle \| 0 ) | Opens an instance of the package "phandle," passes arguments "adr len" |
| 2 | 06 | close-package | ( ihandle -- ) | Closes an instance of a package |
| 2 | 07 | find-method | ( adr len phandle -- false \| acf true ) | Finds the method (command) named "adr len" within the package "phandle" |
| 2 | 08 | call-package | ( [...] acf ihandle -- [...] ) | Executes the method (command) "acf" within the instance "ihandle" |
| 2 | 09 | $call-parent | ( adr len -- ) | Executes the method (command) "adr len" within the parent's package |
| 2 | 0a | my-parent | ( -- ihandle ) | Returns the instance handle of the parent of the current package instance |
| 2 | 0b | ihandle>phandle | ( ihandle -- phandle ) | Converts an instance handle to a package handle |
| 2 | 0d | my-unit | ( -- offset space ) | Returns the physical unit number pair for this package |
| 2 | 0e | $call-method | ( adr len ihandle -- ) | Executes the method (command) named "adr len" within the instance "ihandle" |
| 2 | 0f | $open-package | ( arg-adr arg-len adr len -- ihandle \| 0 ) | Finds a package "adr len," then opens it with arguments "arg-adr arg-len" |

*Figure C-40. Asyncronous Support*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 13 | alarm | ( acf n -- ) | Executes the method (command) indicated by "acf" every "n" milliseconds |

*Figure C-41. Miscellaneous Operations*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 14 | (is-user-word) | ( adr len acf -- ) | Creates a new word called "adr len" which executes "acf" |
| 2 | 36 | wflips | ( adr len -- ) | Exchanges bytes within 16-bit words in the specified region |
| 2 | 37 | lflips | ( adr len -- ) | Exchanges 16-bit words within 32-bit longwords in the specified region |
| 1 | a4 | mac-address | ( -- adr len ) | Returns the MAC address |

*Figure C-42. Interpretation*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 15 | suspend-fcode | ( -- ) | Suspends execution of FCode, resumes later if an undefined command is required |

*Figure C-43. Error Handling*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 16 | abort | ( -- ) | Aborts FCode execution, returns to the "ok" prompt |
| 2 | 17 | catch | ( [...] acf -- [...] error-code ) | Executes "acf," returns THROW error code or 0 if THROW not encountered |
| 2 | 18 | throw | ( error-code -- ) | Returns given error code to CATCH |

*Figure C-44. Package Attributes*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 1a | get-my-attribute | ( nam-adr nam-len -- true \| xdr-adr xdr-len false ) | Returns the value string for the given attribute name |
| 2 | 1b | xdrtoint | ( xdr-adr xdr-len -- n ) | Converts an xdr-encoded string to an integer |
| 2 | 1c | xdrtostring | ( xdr-adr xdr-len -- adr len ) | Converts an xdr-encoded string to a normal string |
| 2 | 1d | get-inherited-attribute | ( nam-adr nam-len -- true \| xdr-adr xdr-len false ) | Returns the value string for the given attribute, searches parents' attributes if not found |
| 2 | 1e | delete-attribute | ( nam-adr nam-len -- ) | Deletes the attribute with the given name |
| 2 | 1f | get-package-attribute | ( adr len phandle -- true \| xdr-adr xdr-len false ) | Returns the value string for the given attribute name in the package "phandle" |
| 1 | 1b | decode-2int | ( xdr-adr xdr-len -- phys space ) | Converts an xdr-coded string into a physical address and space |

*Figure C-45. Atomic Access*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 30 | rb@ | ( adr -- byte ) | Reads the 8-bit value at the given address, atomically |
| 2 | 31 | rb! | ( byte adr -- ) | Writes the 8-bit value at the given address, atomically |
| 2 | 32 | rw@ | ( adr -- word ) | Reads the 16-bit value at the given address, atomically |
| 2 | 33 | rw! | ( word adr -- ) | Writes the 16-bit value at the given address, atomically |
| 2 | 34 | rl@ | ( adr -- long ) | Reads the 32-bit value at the given address, atomically |
| 2 | 35 | rl! | ( long adr -- ) | Writes the 32-bit value at the given address, atomically |

*Figure C-46. Data Exception Tests*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 2 | 20 | cpeek | ( adr -- false \| byte true ) | Reads the 8-bit value at the given address, returns false if unsuccessful |
| 2 | 21 | wpeek | ( adr -- false \| word true ) | Reads the 16-bit value at the given address, returns false if unsuccessful |
| 2 | 22 | lpeek | ( adr -- false \| long true ) | Reads the 32-bit value at the given address, returns false if unsuccessful |
| 2 | 23 | cpoke | ( byte adr -- flag ) | Writes the 8-bit value at the given address, returns false if unsuccessful |
| 2 | 24 | wpoke | ( word adr -- flag ) | Writes the 16-bit value at the given address, returns false if unsuccessful |
| 2 | 25 | lpoke | ( long adr -- flag ) | Writes the 32-bit value at the given address, returns false if unsuccessful |

## FCode Byte Values

The following figure lists, in hexadecimal order, currently-assigned FCode byte values.

*Figure C-47. FCode Byte Values*

| Byte | Name | Stack Comment | |
|---|---|---|---|
| 0 | end0 | ( -- ) | |
| 1 | table1 | | |
| 2 | table2 | | |
| 3 | table3 | | |
| 4 | table4 | | |
| 5 | table5 | | |
| 6 | table6 | | |
| 7 | table7 | | |
| 8 | table8 | | |
| 9 | table9 | | |
| a | table10 | | |
| b | table11 | | |
| c | table12 | | |
| d | table13 | | |
| e | table14 | | |
| f | table15 | | |
| | | | |
| 10 | b(lit) | \ then 32-bit#. | ( -- n ) |
| 11 | b(') | \ then token. | ( -- acf ) |
| 12 | b(") | \ then cnt,letters. | ( -- adr len ) |
| 13 | bbranch | \ then offset. | ( -- ) |
| 14 | b?branch | \ then offset. | ( -- ) |
| 15 | b(loop) | \ then offset. | ( -- ) |
| 16 | b(+loop) | \ then offset. | ( n -- ) |
| 17 | b(do) | \ then offset. | ( end start -- ) |
| 18 | b(?do) | \ then offset. | ( end start -- ) |
| 19 | i | ( -- index ) | |
| 1a | j | ( -- outerindex ) | |
| 1b | b(leave) | ( -- ) | |
| 1c | b(of) | \ then offset. | ( selector testval -- sel \| none ) |
| 1d | execute | ( acf -- ) | |
| 1e | + | ( n1 n2 -- n3 ) | |
| 1f | - | ( n1 n2 -- n3 ) | |
| | | | |
| 20 | * | ( n1 n2 -- n3 ) | |
| 21 | / | ( n1 n2 -- n3 ) | |
| 22 | mod | ( n1 n2 -- n3 ) | |
| 23 | and | ( n1 n2 -- n3 ) | |
| 24 | or | ( n1 n2 -- n3 ) | |
| 25 | xor | ( n1 n2 -- n3 ) | |
| 26 | not | ( n1 -- n2 ) | |
| 27 | << | ( n1 cnt -- n2 ) | |
| 28 | >> | ( n1 cnt -- n2 ) | |

| Byte | Name | Stack Comment |
|---|---|---|
| 29 | >>a | ( n1 cnt -- n2 ) |
| 2a | /mod | ( n1 n2 -- rem quot ) |
| 2b | u/mod | ( n1 n2 -- rem quot ) |
| 2c | negate | ( n1 -- n2 ) |
| 2d | abs | ( n1 -- n2 ) |
| 2e | min | ( n1 n2 -- n3 ) |
| 2f | max | ( n1 n2 -- n3 ) |
| | | |
| 30 | >r | ( n -- ) ( rs: -- n ) |
| 31 | r> | ( -- n ) ( rs: n -- ) |
| 32 | r@ | ( -- n ) ( rs: -- ) |
| 33 | exit | ( -- ) |
| 34 | 0= | ( n -- flag ) |
| 35 | 0<> | ( n -- flag ) |
| 36 | 0< | ( n -- flag ) |
| 37 | 0<= | ( n -- flag ) |
| 38 | 0> | ( n -- flag ) |
| 39 | 0>= | ( n -- flag ) |
| 3a | < | ( n1 n2 -- flag ) |
| 3b | > | ( n1 n2 -- flag ) |
| 3c | = | ( n1 n2 -- flag ) |
| 3d | <> | ( n1 n2 -- flag ) |
| 3e | u> | ( n1 n2 -- flag ) |
| 3f | u<= | ( n1 n2 -- flag ) |
| | | |
| 40 | u< | ( n1 n2 -- flag ) |
| 41 | u>= | ( n1 n2 -- flag ) |
| 42 | >= | ( n1 n2 -- flag ) |
| 43 | <= | ( n1 n2 -- flag ) |
| 44 | between | ( n min max -- flag ) |
| 45 | within | ( n min max -- flag ) |
| 46 | drop | ( n -- ) |
| 47 | dup | ( n -- n n ) |
| 48 | over | ( n1 n2 -- n1 n2 n1 ) |
| 49 | swap | ( n1 n2 -- n2 n1 ) |
| 4a | rot | ( n1 n2 n3 -- n2 n3 n1 ) |
| 4b | -rot | ( n1 n2 n3 -- n3 n1 n2 ) |
| 4c | tuck | ( n1 n2 -- n2 n1 n2 ) |
| 4d | nip | ( n1 n2 -- n2 ) |
| 4e | pick | ( +n -- n2 ) |
| 4f | roll | ( +n -- ) |

| Byte | Name | Stack Comment |
|---|---|---|
| 50 | ?dup | ( n -- 0 \| n n ) |
| 51 | depth | ( -- +n ) |
| 52 | 2drop | ( n1 n2 -- ) |
| 53 | 2dup | ( n1 n2 -- n1 n2 n1 n2 ) |
| 54 | 2over | ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 ) |
| 55 | 2swap | ( n1 n2 n3 n4 -- n3 n4 n1 n2 ) |
| 56 | 2rot | ( n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2 ) |
| 57 | 2/ | ( n1 -- n2 ) |
| 58 | u2/ | ( n1 -- n2 ) |
| 59 | 2* | ( n1 -- n2 ) |
| 5a | /c | ( -- n ) |
| 5b | /w | ( -- n ) |
| 5c | /l | ( -- n ) |
| 5d | /n | ( -- n ) |
| 5e | ca+ | ( n1 index -- n2 ) |
| 5f | wa+ | ( n1 index -- n2 ) |
| 60 | la+ | ( n1 index -- n2 ) |
| 61 | na+ | ( n1 index -- n2 ) |
| 62 | ca1+ | ( n1 -- n2 ) |
| 63 | wa1+ | ( n1 -- n2 ) |
| 64 | la1+ | ( n1 -- n2 ) |
| 65 | na1+ | ( n1 -- n2 ) |
| 66 | /c* | ( n1 -- n2 ) |
| 67 | /w* | ( n1 -- n2 ) |
| 68 | /l* | ( n1 -- n2 ) |
| 69 | /n* | ( n1 -- n2 ) |
| 6a | on | ( adr -- ) |
| 6b | off | ( adr -- ) |
| 6c | +! | ( n adr -- ) |
| 6d | @ | ( adr -- n ) |
| 6e | l@ | ( adr -- L ) |
| 6f | w@ | ( adr -- w ) |
| 70 | <w@ | ( adr -- n ) |
| 71 | c@ | ( adr -- b ) |
| 72 | ! | ( n adr -- ) |
| 73 | l! | ( n adr -- ) |
| 74 | w! | ( n adr -- ) |
| 75 | c! | ( n adr -- ) |
| 76 | 2@ | ( adr -- n1 n2 ) |
| 77 | 2! | ( n1 n2 adr -- ) |
| 78 | move | ( adr1 adr2 cnt -- ) |
| 79 | fill | ( adr cnt byte -- ) |
| 7a | comp | ( adr1 adr2 cnt -- n ) |
| 7b | noop | ( -- ) |
| 7c | lwsplit | ( L -- w.lo w.hi ) |
| 7d | wljoin | ( w.lo w.hi -- L ) |
| 7e | lbsplit | ( L -- b.lo b b b.hi ) |
| 7f | bljoin | ( b.lo b b b.hi -- L ) |

| Byte | Name | Stack Comment |
|---|---|---|
| 80 | flip | ( w1 -- w2 ) |
| 81 | upc | ( char -- upper-case-char ) |
| 82 | lcc | ( char -- lower-case-char ) |
| 83 | pack | ( adr len pstr -- pstr ) |
| 84 | count | ( pstr -- adr len ) |
| 85 | body> | ( apf -- acf ) |
| 86 | >body | ( acf -- apf ) |
| 87 | version | ( -- n ) |
| 88 | span | ( -- adr ) |
| 89 | (reserved) | |
| 8a | expect | ( adr +n -- ) |
| 8b | alloc-mem | ( cnt -- adr ) |
| 8c | free-mem | ( adr cnt -- ) |
| 8d | key? | ( -- flag ) |
| 8e | key | ( -- char ) |
| 8f | emit | ( char -- ) |
| | | |
| 90 | type | ( adr +n -- ) |
| 91 | (cr | ( -- ) |
| 92 | cr | ( -- ) |
| 93 | #out | ( -- adr ) |
| 94 | #line | ( -- adr ) |
| 95 | hold | ( char -- ) |
| 96 | <# | ( -- ) |
| 97 | #> | ( L -- adr +n ) |
| 98 | sign | ( n -- ) |
| 99 | # | ( +L1 -- +L2 ) |
| 9a | #s | ( +L -- 0 ) |
| 9b | u. | ( u -- ) |
| 9c | u.r | ( u cnt -- ) |
| 9d | . | ( n -- ) |
| 9e | .r | ( n cnt -- ) |
| 9f | .s | ( -- ) |
| | | |
| a0 | base | ( -- adr ) |
| a1 | (reserved) | |
| a2 | $number | ( adr len -- true \| n false ), valid only in 2.0 or greater systems |
| a3 | digit | ( char base -- digit true \| char false ) |
| a4 | -1 | ( -- -1 ) |
| a5 | 0 | ( -- 0 ) |
| a6 | 1 | ( -- 1 ) |
| a7 | 2 | ( -- 2 ) |
| a8 | 3 | ( -- 3 ) |
| a9 | bl | ( -- n ) |
| aa | bs | ( -- n ) |
| ab | bell | ( -- n ) |
| ac | bounds | ( n cnt -- n+cnt n ) |
| ad | here | ( -- adr ) |
| ae | aligned | ( adr1 -- adr2 ) |
| af | wbsplit | ( w -- b.lo b.hi ) |

| Byte | Name | Stack Comment |
|---|---|---|
| b0 | bwjoin | ( b.lo b.hi -- w ) |
| b1 | b(<mark) | ( -- ) |
| b2 | b(>resolve) | ( -- ) |
| b3 | (reserved) | |
| b4 | (reserved) | |
| b5 | new-token | \ then table#, code#, token-type |
| b6 | named-token | \ then string, table#, code#, token-type |
| b7 | b(:) | \ token-type |
| b8 | b(value) | \ token-type |
| b9 | b(variable) | \ token-type |
| ba | b(constant) | \ token-type |
| bb | b(create) | \ token-type |
| bc | b(defer) | \ token-type |
| bd | b(buffer:) | \ token-type |
| be | b(field) | \ token-type |
| bf | (reserved) | |
| | | |
| c0 | (reserved) | |
| c1 | (reserved) | |
| c2 | b(;) | ( -- ) |
| c3 | b(is) | ( n -- ) \ then token |
| c4 | b(case) | ( -- ) |
| c5 | b(endcase) | ( -- ) |
| c6 | b(endof) | ( -- ) |
| c7- c9 | (reserved) | |
| ca | external-token | ( -- ), valid only in 2.0 or greater systems |
| cb | $find | ( adr len -- adr len false \| acf +-1 ) |
| cc | offset16 | ( -- ) |
| cd | eval | ( adr len -- ), valid only in 2.0 or greater systems |
| ce - cf | (reserved) | |
| | | |
| d0 | c, | ( n -- ) |
| d1 | w, | ( n -- ) |
| d2 | l, | ( n -- ) |
| d3 | , | ( n -- ) |
| d4 | u*x | ( u1[32] u2[32] -- product [64] ), valid only in 2.0 or greater systems |
| d5 | xu/mod | ( u1[64] u2[32] -- remainder [32] quot [32] ), valid only in 2.0 or greater systems |
| d6 - d7 | (reserved) | |
| d8 | x+ | ( x1 x2 -- x3 ), valid only in 2.0 or greater systems |
| d9 | x- | ( x1 x2 -- x3 ), valid only in 2.0 or greater systems |
| da - df | (reserved) | |
| | | |
| f0 | start0 | ( -- ), valid only in 2.0 or greater systems |
| f1 | start1 | ( -- ), valid only in 2.0 or greater systems |
| f2 | start2 | ( -- ), valid only in 2.0 or greater systems |
| f3 | start4 | ( -- ), valid only in 2.0 or greater systems |
| f4 - fc | (reserved) | |
| fd | version1 | \ then 0byte, chksum(2bytes), length(4bytes) |
| fe | 4-byte-id | \ then 3 more bytes, not supported |
| ff | end1 | ( -- ) |

| Byte1 | Byte2 | Name | Stack Comment |
|---|---|---|---|
| 1 | 01 | dma-alloc | ( #bytes -- virtual ) |
| 1 | 02 | my-address | ( -- physical ) |
| 1 | 03 | my-space | ( -- space ) |
| 1 | 04 | memmap | ( physical space size -- virtual ) |
| 1 | 05 | free-virtual | ( virtual len -- ) |
| 1 | 06 | >physical | ( virtual -- physical space ) |
| 1 | 0f | my-params | ( -- adr len ) |
| | | | |
| 1 | 10 | attribute | ( xdr-adr xdr-len name-adr name-len -- ) |
| 1 | 11 | xdrint | ( n1 -- xdr-adr xdr-len ) |
| 1 | 12 | xdr+ | ( xdr-adr1 xdr-len1 xdr-adr2 xdr-len2 -- xdr-adr1 xdr-len1+2 ) |
| 1 | 13 | xdrphys | ( physical space -- xdr-adr xdr-len ) |
| 1 | 14 | xdrstring | ( adr len -- xdr-adr xdr-len ) |
| 1 | 15 | xdrbytes | ( adr len -- xdr-adr xdr-len ), valid only in 2.1 or greater systems |
| 1 | 16 | reg | ( physical space size -- ) |
| 1 | 17 | intr | ( int-level vector -- ) |
| 1 | 18 | driver | ( adr len -- ), not supported |
| 1 | 19 | model | ( adr len -- ) |
| 1 | 1a | device-type | ( adr len -- ) |
| 1 | 1b | decode-2int | ( xdr-adr xdr-len -- physical space ), valid only in 2.0 or greater systems |
| 1 | 1c | is-install | ( acf -- ) |
| 1 | 1d | is-remove | ( acf -- ) |
| 1 | 1e | is-selftest | ( acf -- ) |
| 1 | 1f | new-device | ( -- ) |
| | | | |
| 1 | 20 | diagnostic-mode? | ( -- flag ) |
| 1 | 21 | display-status | ( n -- ) |
| 1 | 22 | memory-test-suite | ( adr len -- status ) |
| 1 | 23 | group-code | ( -- adr ) |
| 1 | 24 | mask | ( -- adr ) |
| 1 | 25 | get-msecs | ( -- ms ) |
| 1 | 26 | ms | ( n -- ) |
| 1 | 27 | finish-device | ( -- ) |
| | | | |
| 1 | 30 | map-sbus | ( phys size -- virt ) |
| 1 | 31 | sbus-intr>cpu | ( sbus-intr# -- cpu-intr# ) |
| | | | |
| 1 | 50 | #lines | ( -- n ) |
| 1 | 51 | #columns | ( -- n ) |
| 1 | 52 | line# | ( -- n ) |
| 1 | 53 | column# | ( -- n ) |
| 1 | 54 | inverse? | ( -- flag ) |
| 1 | 55 | inverse-screen? | ( -- flag ) |
| 1 | 57 | draw-character | ( char -- ) |
| 1 | 58 | reset-screen | ( -- ) |
| 1 | 59 | toggle-cursor | ( -- ) |
| 1 | 5a | erase-screen | ( -- ) |
| 1 | 5b | blink-screen | ( -- ) |
| 1 | 5c | invert-screen | ( -- ) |
| 1 | 5d | insert-characters | ( n -- ) |
| 1 | 5e | delete-characters | ( n -- ) |
| 1 | 5f | insert-lines | ( n -- ) |

| Byte1 | Byte2 | Name | Stack Comment |
|---|---|---|---|
| 1 | 60 | delete-lines | ( n -- ) |
| 1 | 61 | draw-logo | ( line# laddr lwidth lheight -- ) |
| 1 | 62 | frame-buffer-adr | ( -- addr ) |
| 1 | 63 | screen-height | ( -- n ) |
| 1 | 64 | screen-width | ( -- n ) |
| 1 | 65 | window-top | ( -- n ) |
| 1 | 66 | window-left | ( -- n ) |
| 1 | 6a | default-font | ( -- fontbase charwidth charheight fontbytes #firstchar #chars ) |
| 1 | 6b | set-font | ( fontbase charwidth charheight fontbytes #firstchar #chars -- ) |
| 1 | 6c | char-height | ( -- n ) |
| 1 | 6d | char-width | ( -- n ) |
| 1 | 6e | >font | ( char -- adr ) |
| 1 | 6f | fontbytes | ( -- n ) |
| | | | |
| 1 | 70 | fb1-draw-character | ( char -- ) |
| 1 | 71 | fb1-reset-screen | ( -- ) |
| 1 | 72 | fb1-toggle-cursor | ( -- ) |
| 1 | 73 | fb1-erase-screen | ( -- ) |
| 1 | 74 | fb1-blink-screen | ( -- ) |
| 1 | 75 | fb1-invert-screen | ( -- ) |
| 1 | 76 | fb1-insert-characters | ( #chars -- ) |
| 1 | 77 | fb1-delete-characters | ( #chars -- ) |
| 1 | 78 | fb1-insert-lines | ( #lines -- ) |
| 1 | 79 | fb1-delete-lines | ( #lines -- ) |
| 1 | 7a | fb1-draw-logo | ( line# logoadr lwidth lheight -- ) |
| 1 | 7b | fb1-install | ( width height #cols #lines -- ) |
| 1 | 7c | fb1-slide-up | ( #lines -- ) |
| | | | |
| 1 | 80 | fb8-draw-character | ( char -- ) |
| 1 | 81 | fb8-reset-screen | ( -- ) |
| 1 | 82 | fb8-toggle-cursor | ( -- ) |
| 1 | 83 | fb8-erase-screen | ( -- ) |
| 1 | 84 | fb8-blink-screen | ( -- ) |
| 1 | 85 | fb8-invert-screen | ( -- ) |
| 1 | 86 | fb8-insert-characters | ( #chars -- ) |
| 1 | 87 | fb8-delete-characters | ( #chars -- ) |
| 1 | 88 | fb8-insert-lines | ( #lines -- ) |
| 1 | 89 | fb8-delete-lines | ( #lines -- ) |
| 1 | 8a | fb8-draw-logo | ( line# ladr lwidth lheight -- ) |
| 1 | 8b | fb8-install | ( width height #cols #lines -- ) |
| | | | |
| 1 | a4 | mac-address | ( -- adr len ), valid only in 2.0 or greater systems |

**Note:** All FCodes beginning with 02 are valid only in 2.0 or greater systems.

| Byte1 | Byte2 | Name | Stack Comment |
|---|---|---|---|
| 2 | 01 | device-name | ( adr len -- ) |
| 2 | 02 | my-args | ( -- adr len ) |
| 2 | 03 | my-self | ( -- ihandle ) |
| 2 | 04 | find-package | ( adr len -- false \| phandle true ) |
| 2 | 05 | open-package | ( adr len phandle -- ihandle \| 0 ) |
| 2 | 06 | close-package | ( ihandle -- ) |
| 2 | 07 | find-method | ( adr len phandle -- false \| acf true ) |
| 2 | 08 | call-package | ( [...] acf ihandle -- [...] ) |
| 2 | 09 | $call-parent | ( adr len -- ) |
| 2 | 0a | my-parent | ( -- ihandle ) |
| 2 | 0b | ihandle>phandle | ( ihandle -- phandle ) |
| 2 | 0d | my-unit | ( -- offset space ) |
| 2 | 0e | $call-method | ( adr len ihandle -- ) |
| 2 | 0f | $open-package | ( arg-adr arg-len adr len -- ihandle \| 0 ) |
| | | | |
| 2 | 10 | processor-type | ( -- processor-type ) |
| 2 | 11 | firmware-version | ( -- n ) |
| 2 | 12 | fcode-version | ( -- n ) |
| 2 | 13 | alarm | ( acf n -- ) |
| 2 | 14 | (is-user-word) | ( adr len acf -- ) |
| 2 | 15 | suspend-fcode | ( -- ) |
| 2 | 16 | abort | ( -- ) |
| 2 | 17 | catch | ( [...] acf -- [...] error-code ) |
| 2 | 18 | throw | ( error-code -- ) |
| 2 | 1a | get-my-attribute | ( nam-adr nam-len -- true \| xdr-adr xdr-len false ) |
| 2 | 1b | xdrtoint | ( xdr-adr xdr-len -- n ) |
| 2 | 1c | xdrtostring | ( xdr-adr xdr-len -- adr len ) |
| 2 | 1d | get-inherited-attribute | ( nam-adr nam-len -- true \| xdr-adr xdr-len false ) |
| 2 | 1e | delete-attribute | ( nam-adr nam-len -- ) |
| 2 | 1f | get-package-attribute | ( adr len phandle -- true \| xdr-adr xdr-len false ) |
| | | | |
| 2 | 20 | cpeek | ( adr -- false \| byte true ) |
| 2 | 21 | wpeek | ( adr -- false \| word true ) |
| 2 | 22 | lpeek | ( adr -- false \| long true ) |
| 2 | 23 | cpoke | ( byte adr -- flag ) |
| 2 | 24 | wpoke | ( word adr -- flag ) |
| 2 | 25 | lpoke | ( long adr -- flag ) |
| | | | |
| 2 | 30 | rb@ | ( adr -- byte ) |
| 2 | 31 | rb! | ( byte adr -- ) |
| 2 | 32 | rw@ | ( adr -- word ) |
| 2 | 33 | rw! | ( word adr -- ) |
| 2 | 34 | rl@ | ( adr -- long ) |
| 2 | 35 | rl! | ( long adr -- ) |
| 2 | 36 | wflips | ( adr len -- ) |
| 2 | 37 | lflips | ( adr len -- ) |
| 2 | 40 | left-parse-string | ( adr len char -- adrR lenR adrL lenL ) |

# *Glossary*

**32-bit master/slave/ controller/device** An SBus master/slave/controller/device supporting only 32-bit per clock cycle transfers.

**32-bit transfers** The basic SBus bus cycle in which 32-bits of data can be transferred each clock cycle.

**64-bit master/slave/ controller/device** An SBus master/slave/controller/device supporting 32- and 64-bit per clock cycle transfers.

**64-bit transfers** See *ExtendedTransfer*.

**acknowledgment**
Ack(2:0)* Any encoding of Ack(2:0)* to indicate that data has been transferred, or that the current bus cycle should be terminated, or both. Valid acknowledgments are:

- Double-word acknowledgment (ExtendedTransfers only).
- Word acknowledgment (32-bit transfers only).
- Half-word acknowledgment (32-bit transfers only).

- ❑ Byte acknowledgment (32-bit transfers only).
- ❑ Error Acknowledgment.
- ❑ Rerun Acknowledgment.

**address lines**
PhysAddr(27:0) (PA(27:0))
The SBus signals used by the SBus controller to send a physical address to a slave. During ExtendedTransfers, the signals are used for Data(59:32).

**address strobe**
AddressStrobe* AS*
The SBus signal used by the SBus controller to indicate that a slave cycle is in progress.

**asserted**
The state of a signal used to initiate an action.

**atomic transaction**
A sequence of bus cycles in which an SBus master retains control of the bus to prevent any other master from accessing the bus. Atomic transactions are used to implement semaphores.

**autoconfiguration**
The process by which the host fetches *SBus IDs* and *FCodes*, beginning at location 0 of each SBus slave used to identify the device.

**big-endian**
An ordering of bytes within a *word* where the most significant *byte* is at the lowest address, and the least significant byte is at the highest address.

**board**
See *SBus expansion card.*

**burst transfer**
A single bus cycle in which multiple words of data are transferred.

**bus cycle** A series of clock cycles beginning (in the case of a *DVMA master*) with a particular master receiving a grant and, in all cases, concluding with address strobe being unasserted by the SBus controller. For DVMA masters, a bus cycle is divided into two phases: a *translation cycle* and a *slave cycle*. However, in the case of a *CPU master*, the translation cycle does not occur as part of the bus cycle.

**bus sizing** A transfer mode in which a slave requests the master to turn a *word* (*half-word*) transfer into two half-words, or four (two) *byte* transfers. Each transfer is performed using a separate bus cycle. The first bus cycle is called the original bus cycle; remaining bus cycles are called follow-on bus cycles.

**byte** A set of 8 signals or bits taken as a unit.

**byte acknowledgment** An acknowledgment to indicate that the slave has read or written a *byte* from the most significant byte of the *data lines*. If the transfer size is greater than a byte, the master initiating the transfer may perform *bus sizing*.

**byte-addressing** A determination that the smallest addressable unit of information is a byte.

**card** See *SBus expansion card.*

**clock**
Clock (Clk) An SBus signal generated by the *SBus controller* which synchronizes all activity on the SBus.

**clock cycle** One period of the SBus clock (Clock). Each bus cycle consists of several clock cycles.

**CPU master** An SBus master that includes a central processing unit with a private means to perform virtual address translation (in contrast to a *DVMA master* which uses the SBus controller to perform virtual address translation). A bus cycle initiated by a CPU master consists only of a slave cycle. Typical SBus systems have one CPU master.

**Data Acknowledgment** An acknowledgment to indicate the slave has read or written a *byte, half-word, word,* or *double-word.* There are four types of Data Acknowledgment:

- Byte acknowledgment (32-bit transfers only).
- Half-word acknowledgment (32-bit transfers only).
- Word acknowledgment (32-bit transfers only).
- Double-word acknowledgment (ExtendedTransfers only).

**data lines** Data(31:0) D(31:0) The SBus signals used to transfer data between masters and slaves, and virtual addresses between masters and the *SBus controller.* For ExtendedTransfers, there are 64 data lines, called Data(63:0).

**device** See *SBus device.*

**Direct Virtual Memory Access (DVMA)** A mechanism to allow a device on the SBus to initiate data transfers between it and other SBus devices, such as system memory. To simplify overall system design, SBus DVMA transfers are performed using virtual addressing. The SBus controller contains a *Memory Management Unit* (MMU) responsible for performing virtual to physical address translation.

**double-word** A group of 64 signals or bits taken as a unit (8 bytes of data).

**double-word acknowledgment** An acknowledgment to indicate that the slave is ready to read or write a *double-word* of data. A double-word acknowledgment is used only for ExtendedTransfers. It is the only valid Data Acknowledgment during an ExtendedTransfer.

**driver overlap** A situation in which two different drivers are simultaneously sourcing or sinking current.

**dummy read** A bus cycle used by a master during an atomic transaction to hold the bus, so that it can process data before performing a write bus cycle. Dummy reads are performed to the same address as the original read.

**DVMA cycle** A bus cycle initiated by a *DVMA master*. A DVMA cycle consists of a *translation cycle* and a *slave cycle.*

**DVMA master** An SBus master able to initiate a bus cycle that uses the *SBus controller* to perform virtual address translation (in contrast to a CPU master which has a private means for virtual address translation). A bus cycle initiated by a DVMA master consists of a *translation cycle* and a *slave cycle.*

**Error Acknowledgment** An acknowledgment to indicate that the bus cycle is terminated as a result of an abnormal condition.

**expansion card** See *SBus expansion card.*

**expansion connector** A 96-pin connector to allow a user to insert an SBus card.

**ExtendedTransfer** An extended bus cycle protocol (also called a 64-bit transfer) in which 64-bits of data are transferred per clock cycle during the slave cycle. The upper 32-bits of data are multiplexed onto the Size(2:0), Read, and PhysAddr(27:0) lines.

**Extended Transfer Information** During an ExtendedTransfer, the Extended Transfer Information is driven onto Data(31:0) during the translation cycle, and the first clock cycle of the slave cycle. The Extended Transfer Information is the detailed description of the ExtendedTransfer, and consists of the following information:

- ExtendedTransferType.
- ExtendedTransferSize(2:0).
- ExtendedTransferRead.
- ExtendedTransferAtomic(1:0).
- ExtendedTransferReserved(24:0).

**FCodes** FORTH byte codes.

**follow-on bus cycle** One of up to three bus cycles during a bus sizing operation that follows the original bus cycle.

**geographical addressing** A mechanism by which a part of the physical address is presented to each SBus slave as an individual select signal, so that only one slave is selected at any given time.

**grant lines** Grant* BG* The set of SBus signals (one per master) generated by the *SBus controller* to inform masters when they may access the bus.

**half-word** A group of 16 signals or bits taken as a unit.

**half-word acknowledgment** An acknowledgment to indicate that the slave has read or written a *half-word* of data from the most significant half-word of the *data lines*. If the transfer size is greater than a half-word, the master initiating the transfer may perform *bus sizing*.

**high** Driven to a voltage greater than or equal to $V_{OH}$.

**late error** LateError* LErr* A special SBus signal to indicate that an error occurred during a preceding data transfer, even though the slave issued a *byte, half-word, word,* or *double-word* acknowledgment.

**latency** The time between when a master requests the bus and when its transfer is complete.

**logic 0** The logic state of a signal driven to $V_{OL}$ (or $V_{OH}$ if low asserted).

**logic 1** The logic state of a signal driven to $V_{OH}$ (or $V_{OL}$ if low asserted).

**low** Driven to a voltage less than or equal to $V_{OL}$.

**low asserted** The property of a signal to indicate that its logical polarity is the opposite of its physical polarity.

**master** An SBus device capable of initiating an SBus transaction. The term *CPU master* is used when a host CPU must be distinguished from a more generic SBus master. The term *DVMA master* is used when it is desired to explicitly exclude CPU masters. Any SBus master may communicate with any other *slave* on the same bus, regardless of system configuration. For more information, see "Configuration" in Chapter 1.

**motherboard** A circuit board containing the central processor, *SBus controller*, and any SBus expansion connectors.

**Open Boot** With regard to *SBus Profiles*, Open Boot is the facility by which the *FCode* program may interrogate the host and determine the state of various parameters it addresses. For information, see the Sun Microsystems, Inc. publication *Open Boot PROM Toolkit User's Guide.*

**open-drain** A bus driver or signal driven only low (sometimes referred to as *open collector*).

**original bus cycle** In a *bus sizing* operation, the first bus cycle of the transfer causing the master to perform bus sizing. Every bus sizing operation consists of an original bus cycle, plus one to three follow-on bus cycles, depending on the size of the original transfer and the type of Data Acknowledgment issued by the slave.

**real-time** An event or system that must receive a response to some stimulus within a narrow, predictable, deterministic, and repeatable time frame. Usually, this requires that the response is not strongly dependent on system performance parameters which are highly variable, such as processor load or interface latency.

**request**
Request* BR* The set of SBus signals (one per master) used by the master to request the *SBus controller* to grant access to the bus.

**Rerun Acknowledgment** An acknowledgment to indicate that the current master should abort the current transfer and re-request access to the bus to retry the transfer.

**sample** To determine the state of a signal at the rising edge Clock.

**SBus Bridge SBus Coupler** A device providing additional SBus slots by connecting two SBusses. In general, a bus bridge is functionally transparent to devices on the SBus. However, there are cases (for example, *bus sizing*) in which bus bridges may change the exact way a series of *bus cycles* are performed.

**SBus controller** The hardware responsible for performing arbitration, addressing translation and decoding, driving slave selects and address strobe, and generating timeouts.

**SBus device** A logical device attached to the SBus. This device may be on the *motherboard*, or on an *SBus expansion card.*

**SBus expansion card** A physical printed circuit assembly that conforms to the single- or double-width mechanical specifications, and that contains one or more *SBus devices.*

**SBus ID** A special series of bytes at address 0 of each SBus slave used to identify the *SBus device.*

**slave** An *SBus device* that responds with an acknowledgment to a slave select and address strobe. Any SBus *master* may communicate with any other slave on the same bus, regardless of system configuration. For more information, see "Configuration" in Chapter 1.

**slave cycle** That portion of a *bus cycle* that begins with placing an address on the physical *address lines*, and ends with AddressStrobe* being unasserted.

**slave select** SlaveSelect* Sel* A collection of SBus signals (one per slave) used to select which slave should be active during the current slave cycle.

**slot** An SBus entity for which there is an independent slave select wire. *Slot* is also used as an abbreviation for *SBus expansion slot.*

**timeout** A situation in which the *SBus controller* terminates a *bus cycle* which a slave has failed to acknowledge. In a correctly designed and operating system, timeouts should happen only during system configuration.

**transfer direction**
Read, Rd The SBus signal to indicate whether data is being read from or written to the selected slave. During ExtendedTransfers, the signal is used as Data(63).

**transfer size**
Size(2:0) Siz(2:0) The SBus signals used to indicate the number of bytes to be transferred during this bus cycle, assuming that no error occurs. During ExtendedTransfers, the se signals are used for Data(62:60).

**translation cycle** That portion of a *bus cycle* between the assertion of grant and the placing of an address on the physical address lines by the *SBus controller*. After receiving the grant, the designated master places a virtual address on the SBus *data lines*.

**tristate** An output able to remove its drive from a wire.

**TTL voltage levels** The voltage levels that determine whether a signal is a *logic 0* or a *logic 1* state, with respect to TTL or TTL-compatible logic families.

**unasserted** The state of a signal used to terminate an action.

**word** A group of 32 signals or bits taken as a unit.

**wrapping** The process, during *burst transfers*, by which the burst may begin at an arbitrary word boundary within the block, with the address incremented by 4, modulo the size of the burst in bytes.

# *Index*

## D



## E

## V



## W

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

*Part Number: 800-5922-10*
*Revision A of December 1990*