# CHAPTER 26

# Designing A VME-To-SCSI Adapter

## By Donald Peterson

*The Small Computer Systems Interface (SCSI) has grown dramatically in popularity. For example, only a few years ago none of the 5.25 inch Winchester disk drive vendors were selling drives with SCSI interfaces built in, and now most of the models being sold include a SCSI interface. In fact, disk drives with SCSI interfaces are now available from all the major disk drive manufacturers. In this chapter, Don discusses features that should be on a VME-to-SCSI adapter board, to provide the best possible performance.*

As a system integrator, you're faced with a wide range of choices when it comes to putting together a disk subsystem. For example, you can purchase a disk drive with a standard *device-level interface*, such as SMD or ESDI, and interface it to your VMEbus system with a VME-to-SMD or VME-to-ESDI disk controller board, as shown in Figure 1.

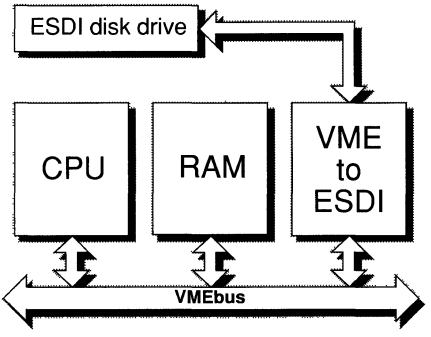


**Figure 1**

Another alternative is to buy a VME-to-SCSI adapter board. At one time this was not a very appealing approach, since interfacing the SCSI bus to most disk drives was awkward. You typically had to purchase a *bridge controller*, which provided a SCSI interface on one side, and an SMD or ESDI interface on the other, as shown in Figure 2. However, that situation has changed, and today most disk drive manufacturers offer SCSI-compatible drives. With these drives, you can simply daisy chain a cable from the VME-to-SCSI adapter to each of the disk drives in your system, as shown in Figure 3.

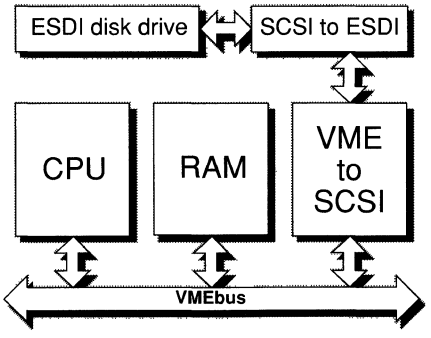


**Figure 2**

However, although the initial SCSI compatible products claimed conformance to the SCSI specification, they were often incompatible with each other. This was because the SCSI specification allowed for differing interpretations of many of the SCSI bus commands. In an effort to bring order out of chaos a group of companies (led by OMTI, a division of Scientific Micro Systems) formed a committee and selected a subset of the original SCSI commands, which they called the Common Command Set (CCS). Adherence to this standard provides a much higher level of compatibility between products.

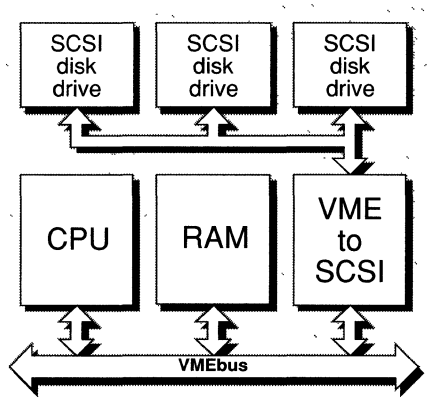


**Figure 3**

Early SCSI bus users expected to see disk performance comparable to what they had previously seen on *device level* subsystems, such as SMD or ESDI. However, they were disappointed for 2 basic reasons:

- Device-level disk controller vendors had years of hardware and firmware experience behind them, while SCSI vendors had not yet developed that level of expertise.
- The architecture of a SCSI-based subsystem has one additional step in the command flow.

The 1st problem has largely been solved, as SCSI compatible drive manufacturers have gained experience. The 2nd problem has not been so easy to overcome.

Figure 4 shows the flow of control in a typical ESDI-based disk subsystem. Each disk access begins with an operating system call. This activates the software driver for the disk being accessed. The driver builds an I/O parameter block and passes it to the disk controller board, which then translates the logical block number to a physical cylinder, track and sector number.

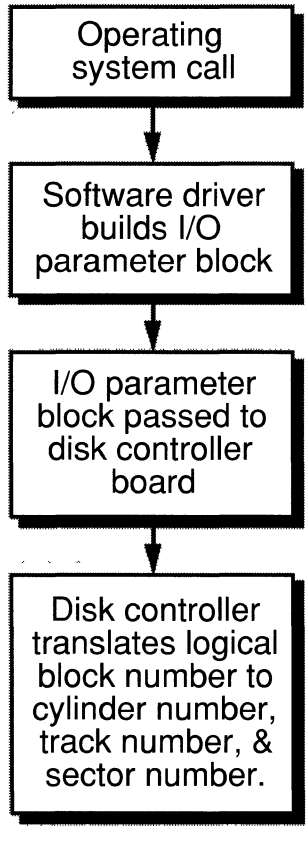


**Figure 4**

Contrast that with the flow of control in a SCSI-based subsystem, as shown in Figure 5. The operating system call activates the software driver, which builds an I/O parameter block and passes it to the VME-to-SCSI adapter board. The adapter then extracts the SCSI command from the parameter block and sends it over the SCSI bus, where the SCSI compatible disk controller in the drive translates the logical block number to a physical cylinder, track and sector number.

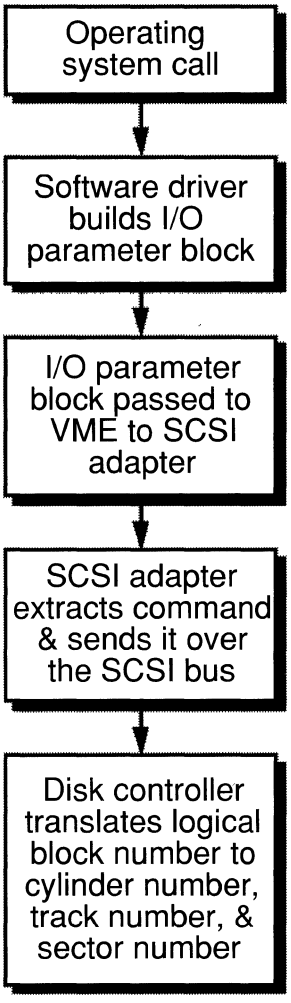


**Figure 5**

Obviously, there is an extra step in the flow of control in the SCSI-based system. However, with careful design, a SCSI-based system will be able to deliver performance approaching that of ESDI or SMD systems, while still providing much greater flexibility in the selection of disk drives.

There are 6 major factors that affect the performance of a SCSI subsystem:

- The VME-to-SCSI adapter firmware
- The SCSI controller chip used
- The VMEbus interface
- The adapter's buffer architecture
- The operating system's software driver.
- The disk controller that interfaces the disk drive to the SCSI bus

**VME-to-SCSI adapter firmware**

The adapter firmware should be written to minimize the overhead needed to handle SCSI commands. One technique that works well is to embed the SCSI command block into the I/O parameter block built by the operating system's software driver. The VME-to-SCSI adapter can then simply extract the SCSI command block from the parameter block, and send it directly over the SCSI bus, without any changes. In addition to the speed benefits, this allows the VME-to-SCSI adapter firmware to accept and transmit vendor-unique commands, in addition to the commands of the common command set.

If the adapter firmware supports command queueing, you can also use another technique called *command combining*. The firmware scans and sorts the queued commands, and then combines groups of them into single SCSI bus commands, reducing the SCSI command processing overhead. It also permits scatter/gather

operations, where the data from discontiguous memory areas is stored onto the disk (or retrieved from it) with a single SCSI command.

*Command sorting* also permits the VME-to-SCSI adapter to minimize the amount of head motion needed on each drive to access the required sectors. There are various algorithms you can use to sort the accesses. One popular algorithm is called *C-scan*. It accesses the lowest logical block number, and then the higher block numbers, in ascending order. Assuming that the physical disk blocks are arranged on the disk in the same sequence as the logical blocks (which they usually are) this will access the sectors closest to cylinder zero (near the hub of the disk) and then access the sectors further out toward the edge of the disk. Since the accesses are done in order, random head motion (and the resulting seek latencies) are minimized. When the disk has accessed the sector furthest from the hub, the head is repositioned to the center, and the process begins again.

If the firmware sorts all pending disk access requests by disk drive, it can also improve the performance of a multiple disk subsystem. It does this by sending concurrent commands to several disk drives on the SCSI bus. Note: In order to do this, the firmware must support the SCSI *disconnect/reconnect protocols*.

**The SCSI controller chip**
Command overhead is critical to the performance of SCSI controller chips. Initial SCSI controller chip designs (such as NCR's 1st generation chip, called the 5380, first sold in 1983) had command overheads in the multiple msec range. However, 2nd generation SCSI chips (such as the Western Digital 33C93A and Emulex ESP) have command overheads under 500 μsecs. The Western Digital chip also includes SCSI drivers and receivers, reducing the component cost and the real estate requirements.

**The VMEbus interface**
The VMEbus has a theoretical bandwidth of about 40 Mbytes per sec, while the SCSI bus (even in its synchronous mode) has an upper limit of around 5 Mbytes per second. Therefore, you don't want to just directly couple the VMEbus and the SCSI bus. The data would be transferred at too slow a rate over the VMEbus, and would waste much of its available bandwidth.

A better approach is to design the VME-to-SCSI adapter board to accumulate incoming data from the SCSI bus, and then periodically write it to VMEbus memory at a high burst rate. (In the case of data being sent out to the disk, the adapter would read data from the VMEbus in bursts, and then send it out at a slower rate over the SCSI bus.) In order to do this, the adapter must be equipped with either a *cache* or a FIFO (First-In-First-Out) buffer. By using a FIFO, a high performance adapter can transfer bursts of data over the VMEbus at rates over 30 Mbytes per second.

**Adapter buffer architecture**
The *physical architecture* of a typical SCSI disk subsystem includes 3 components:

- A VME-to-SCSI adapter board in the VMEbus chassis
- A SCSI-compatible disk drive
- A cable from the adapter board to the disk drive

This looks very similar to an ESDI or SMD disk subsystem. For example, in an ESDI subsystem you have:

- A VME-to-ESDI controller board in the VMEbus chassis
- An ESDI-compatible disk drive
- A cable from the controller board to the disk drive

However, the *functional architecture* of the 2 types of subsystems is very different, and should not be confused.

A VME-to-ESDI or VME-to-SMD disk controller board interfaces *directly* to the disk drive's control electronics. In order to do this, the controller must be able to format the drive, handle defects in the disk's media by reassigning sectors, and correct errors as it reads from the disk, based on an ECC code.

A VME-to-SCSI adapter does not handle any of these functions. They are handled by the disk controller circuitry, which is typically mounted *inside* the SCSI compatible disk's enclosure. (In cases where SCSI-to-ESDI or SCSI-to-SMD bridge controllers are used, the controller may be a free-standing board.) The VME-to-SCSI adapter just sends a command over the SCSI bus, requesting the read or write of a *logical block number*. The disk controller in the drive enclosure translates this logical block number into the appropriate cylinder, track and sector numbers, and initiates the control signal sequence needed to position the head over the appropriate track.

When an operating system reads a disk sector, it will quite likely read the next sector some time soon. One technique that improves disk performance is to provide a *multiple sector cache*. Whenever a sector is read from the disk, some of the subsequent sectors on the same track are also read, and stored in the sector cache. Then, if the operating system needs to read the next sector, the contents of that next sector can be made available immediately from the cache, without waiting for the head to position itself over the track again.

Sector caching in the *disk controller* works very well. However sector caching in the VME-to-SCSI *adapter* doesn't work nearly as well. The problem is that the adapter deals only with *logical* block numbers: it doesn't know where one disk track ends and the next begins. As a result, if it reads several subsequent blocks (to fill its local block cache) it may initiate head movements (to an adjacent track) that tie up the disk needlessly for several milliseconds. If another command is received by the adapter while the head movement is taking place, that command won't be processed until the head movement (and the subsequent sector reads) are completed. If the hit rate on the cached blocks is low, this logical block caching might even *decrease* disk performance.

There's another reason that you don't want to try to cache the blocks in the adapter: it requires you to first write each block into the adapter's local memory, and then to copy the block to VMEbus memory. It's much faster to just route an incoming block directly from the SCSI bus, through a FIFO, and onto the VMEbus.

If you use a FIFO in this manner, you must make sure that it's deep enough to continue accepting data from the SCSI bus, even when the VMEbus is temporarily busy. For example, suppose we have a 32-byte-deep FIFO. We first calculate the time required to empty the FIFO over the VMEbus. If we assume that the adapter can transfer data

across the VMEbus 32 bits at a time, and that those 32-bit data transfers take place every 400 nsecs, then it would take 3.2 μsecs to empty the 32-byte FIFO.

Next, we calculate the time required to fill the FIFO from the SCSI bus. If we assume that data is coming to the adapter from the SCSI bus at a rate of 5 Mbytes per second, then it would take 6.4 μsecs to fill the FIFO.

So, even with these conservative assumptions, the adapter will be able to keep a 32-byte FIFO from filling, as long as it is given access to the VMEbus at least once every 6.4 μsecs. (You can ensure this by giving the adapter a high bus arbitration priority, as is usually done for most mass storage devices.) In accomplishing the SCSI-to-VME data transfer, the adapter will use only 50% of the VMEbus bandwidth. In general, a 32-byte FIFO provides adequate buffering, because of the high speed of the VMEbus.

**The software driver**
The operating system's software driver should be designed to minimize overhead, and to take full advantage of the specific features of the VME-to-SCSI adapter, such as scatter/gather DMA transfers.

Writing drivers that work with all SCSI-based peripherals is not as simple as it seems. Disk, tape, and optical SCSI devices each require different driver code, and there are even subtle differences from model to model. Because of this you would be wise to test your device drivers on several different models of SCSI devices.

**The disk controller**
Any high-performance disk controller should have a fairly large built-in sector cache. This allows the controller to continue reading consecutive sectors after the requested sector has been read. Then, if the next sector access is sequential to the previous one, that sector will already be in the cache.

Some 2nd generation high performance SCSI compatible disks can also *sort* and *combine* sector accesses. This sorting allows the disk to access all of the requested sectors that lie along the same track, without repositioning the head. This feature can be made even more effective if the disk controller is equipped with a feature called *zero latency read.* If a disk controller has zero latency read, and if it's requested to read several sectors scattered along a single track, it can start looking for *any* of the requested sectors, as soon as the head arrives at the track.

Without zero latency read, the disk controller would first look for the *lowest numbered* sector, and then read the additional sectors in ascending order. If the lowest numbered sector just passed the head, none of the sectors would be read until the disk makes one complete revolution. Then, if the highest numbered sector is at the end of the track, the disk would have to make yet another complete revolution, before all the remaining sectors could be read.

With the zero latency read feature, the disk controller can always read all of the needed sectors from the track in only one rotation.

For the most part, 1st generation SCSI compatible disk controllers didn't implement these features. Fortunately, disk drive manufacturers saw the need to upgrade the performance of their SCSI compatible drives. Vendors such as Hewlett-Packard,

Quantum, and Micropolis introduced products that include track buffers, which capture *all* of the sectors of a track in a single revolution of the disk. This allows them to implement read-ahead caching and zero latency reads.

**The Rimfire 3510 VME-to-SCSI adapter**

Figure 6 is the block diagram of Ciprico's 2nd generation VME-to-SCSI adapter, called the Rimfire 3510. (See also Figure 7) The Rimfire 3510 is controlled by a 10 MHz 80186 microprocessor, which uses a private 16-bit-wide bus to access local RAM and EPROM. The *short burst FIFO* is a custom gate array that provides 32 bytes of buffering between the VMEbus and the SCSI bus. This FIFO has proven to be large enough to decouple the VMEbus and the SCSI bus, without incurring any performance degradation.

Ciprico has also designed another custom gate array called the *Pipelined System Interface* (PSI). It serves as a DMA controller, generating VMEbus addresses. However, it has one special feature that allows it to transfer data much faster than a typical DMA controller: a double-deep *starting address* register and a double-deep *transfer length* register. These double-deep registers allow the onboard 80186 to *preload* the starting address (and the transfer length) of a subsequent DMA operation, while the current

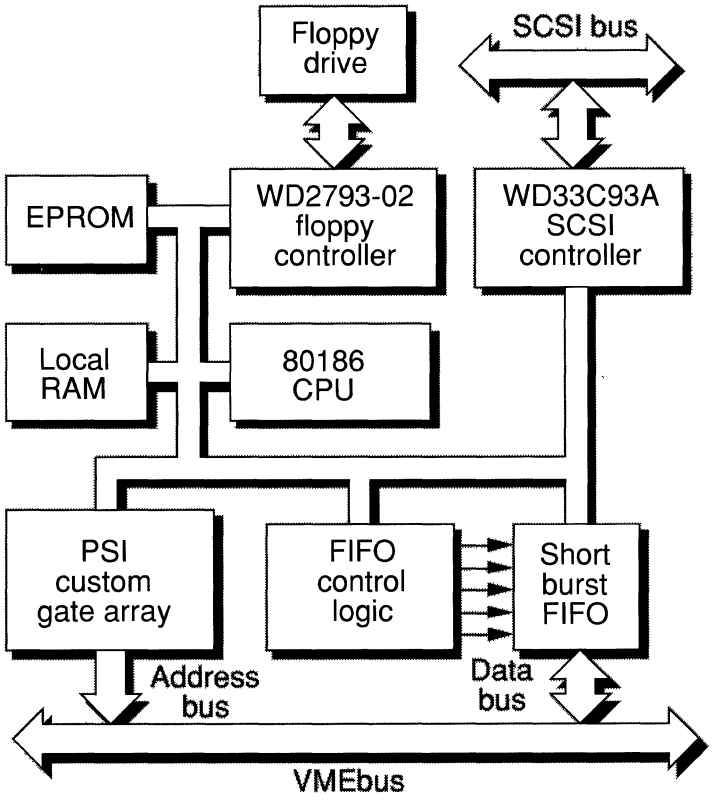


**Figure 6**

DMA transfer is still in progress. The use of *PSI* and the *short burst FIFO* gate arrays allows a DMA burst rate of greater than 30 Mbytes per second over the VMEbus.

The Western Digital WD33C93A SCSI controller chip is used on the adapter. This 2nd generation intelligent SCSI controller chip (preceded by the WD33C93) operates from a 16 MHz clock, and dramatically reduces SCSI command overhead. The overhead has been reduced from 1.2 milliseconds for the 1st generation chip to less than 500 microseconds.

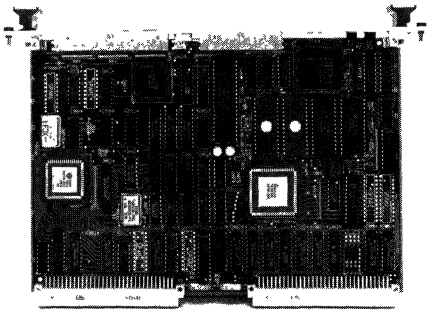

**Figure 7**

The SCSI controller chip also includes a 5-byte-deep FIFO. It can support synchronous data rates up to 5 Mbytes per second and asynchronous rates up to 2 Mbytes per second. (The 1st generation WD33C93 only provided 4 Mbytes per second synchronous transfers, and 1.5 Mbytes per second asynchronous transfers.)

The firmware on the Rimfire 3510 does *command sorting* and *command combining*, for both read commands and write commands. This improves performance by minimizing seek and rotational latency delays.

Many users still want to have a floppy disk drive on their system for software distribution and testing. However, there are few (if any) SCSI floppy disk drives available. For this reason, the Rimfire 3510 includes a floppy disk interface, which allows a choice of 3.5-inch or 5.25-inch floppy drives. A portion of the on-board RAM is used as a buffer for the floppy port, so SCSI performance is not degraded by floppy activity. (SCSI transfers are always given first priority by the onboard firmware.) Floppy drives are accessed from the VMEbus just as if they are normal SCSI devices. (The floppy drive is given an unused SCSI ID of FE (hexadecimal). Floppy commands are passed to the adapter board via a parameter block that has a format identical to normal SCSI commands.

**Summary**

SCSI provides a very flexible method for connecting multiple types of mass storage devices to VMEbus systems. With drive vendors improving the performance of their SCSI compatible drives, the VME-to-SCSI adapter design has become critical. A good adapter should have:

- Minimal overhead in the firmware
- A 2nd generation SCSI controller chip
- A 32-bit, FIFO-buffered VMEbus interface
- An efficient, well-debugged software driver

The resulting SCSI-based mass storage solution will give you the performance and flexibility you need at a reasonable cost.

**Donald C. Peterson** is the director of marketing at Ciprico, in Plymouth, Minnesota. He has been involved in the marketing of board level products for 12 years. Prior to joining Ciprico in 1986, he worked as product marketing manager at Qualogy in San Jose, California, and held various marketing positions at Intel in Hillsboro, Oregon. He holds a BSEE and MS in business from the University of Wisconsin, Madison.

# CHAPTER 28

# SCSI-2: SCSI's High Performance Offspring

## By Bill Moren

*On June 23rd, 1986 (after more than 4 years of work) the American National Standards Institute approved ANSI X3.131-1986. Slightly over 200 pages in length, the document contains 14 sections and 7 appendices. It defines an interface standard called the Small Computer System Interface (SCSI) bus for connecting peripherals to microcomputers. As the SCSI bus has become widely used in systems of all sizes and performance levels, it has been found to have some performance limitations. In this chapter Bill discusses those limitations, and describes the new version of SCSI (called SCSI-2) which has been designed to allow higher-performance operation.*

The Small Computer System Interface (SCSI) bus is a cable bus, used to connect peripherals to microcomputers. Each peripheral can request access to (and gain control of) the SCSI bus, in order to transfer data to (and from) the host system. The bulk of the ANSI standard specifies the bus's physical and logical characteristics, and defines command sets for each of 6 device types:

- Direct-Access (disk)
- Sequential-Access (tape)
- Printers
- Processors
- Write-Once-Read-Many (WORM) optical drive
- Read-Only-Direct-Access optical drive

Most of the commands defined in each command set are used exclusively on that type of device. However, there are some commands that are common to all device types.

**SCSI hosts and SCSI targets**

The SCSI bus is typically interfaced to a VMEbus-based system through a plug-in board called a VME-to-SCSI adapter, as shown in Figure 1. The VME-to-SCSI adapter provides an intelligent link between the VMEbus and the SCSI bus. Peripheral drives are interfaced to SCSI with a *peripheral controller.*

There are 2 basic types of SCSI devices:

- Initiators
- Targets

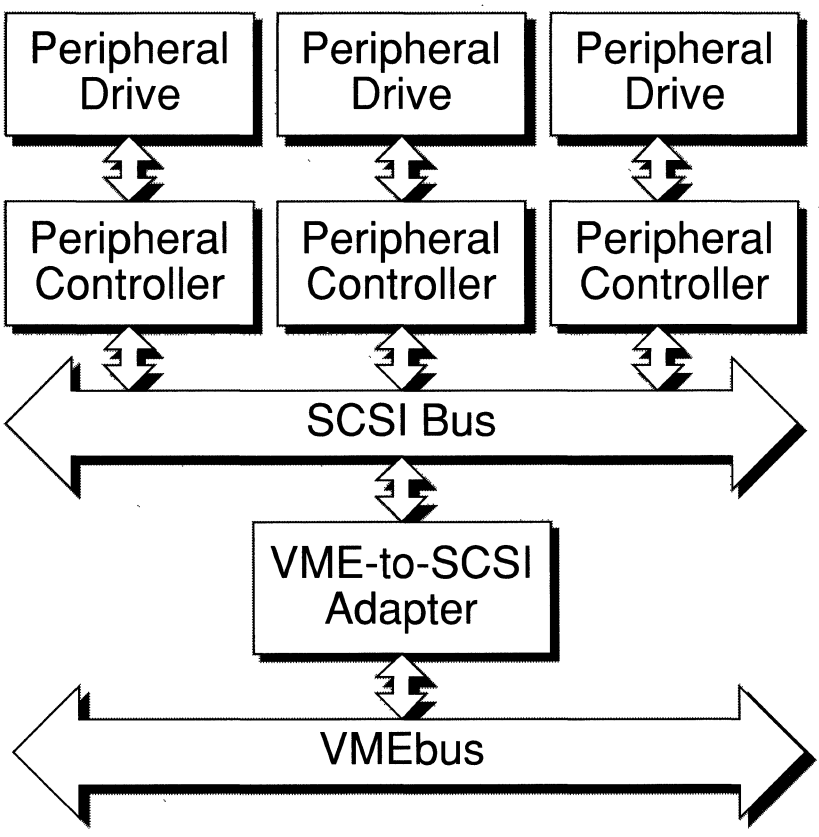


**Figure 1**

The VME-to-SCSI adapter is typically the *initiator*. It sends commands from the VMEbus system over the SCSI bus, initiating activity in the peripherals. Peripheral controllers are considered *target* devices, since they receive the commands sent by the VME-to-SCSI adapter. Each target can support up to 8 individual drives.

**SCSI signal lines and drivers**
SCSI provides an 8-bit-wide parallel data transfer bus. The SCSI signals include:

- 9 data signals (including 1 parity signal)
- 9 control signals

Two driver/receiver options are defined:

- Single-ended
- Differential

The *single-ended* drivers allow a maximum cable length of 6 meters, while the *differential* drivers allow a maximum cable length of 25 meters.

Both single-ended and differential configurations use a 50-conductor cable. The signals for the single-ended configuration are shown in Figure 2. Single-ended signal conductors are interleaved with grounded return conductors. The signals for the differential configuration are shown in Figure 3.

| | |
|---|---|
| 1 | GND |
| 2 | –DB(0) |
| 3 | GND |
| 4 | –DB(1) |
| 5 | GND |
| 6 | –DB(2) |
| 7 | GND |
| 8 | –DB(3) |
| 9 | GND |
| 10 | –DB(4) |
| 11 | GND |
| 12 | –DB(5) |
| 13 | GND |
| 14 | –DB(6) |
| 15 | GND |
| 16 | –DB(7) |
| 17 | GND |
| 18 | –DB(P) |
| 19 | GND |
| 20 | GND |
| 21 | GND |
| 22 | GND |
| 23 | RESERVED |
| 24 | RESERVED |
| 25 | OPEN |
| 26 | TERMPWR |
| 27 | RESERVED |
| 28 | RESERVED |
| 29 | GND |
| 30 | GND |
| 31 | GND |
| 32 | –ATN |
| 33 | GND |
| 34 | GND |
| 35 | GND |
| 36 | –BSY |
| 37 | GND |
| 38 | –ACK |
| 39 | GND |
| 40 | –RST |
| 41 | GND |
| 42 | –MSG |
| 43 | GND |
| 44 | –SEL |
| 45 | GND |
| 46 | –C/D |
| 47 | GND |
| 48 | –REQ |
| 49 | GND |
| 50 | –I/O |

**Figure 2**

| | |
|---|---|
| 1 | GND |
| 2 | GND |
| 3 | +DB(0) |
| 4 | –DB(0) |
| 5 | +DB(1) |
| 6 | –DB(1) |
| 7 | +DB(2) |
| 8 | –DB(2) |
| 9 | +DB(3) |
| 10 | –DB(3) |
| 11 | +DB(4) |
| 12 | –DB(4) |
| 13 | +DB(5) |
| 14 | –DB(5) |
| 15 | +DB(6) |
| 16 | –DB(6) |
| 17 | +DB(7) |
| 18 | –DB(7) |
| 19 | +DB(P) |
| 20 | –DB(P) |
| 21 | DIFFSENS |
| 22 | GND |
| 23 | RESERVED |
| 24 | RESERVED |
| 25 | TERMPWR |
| 26 | TERMPWR |
| 27 | RESERVED |
| 28 | RESERVED |
| 29 | +ATN |
| 30 | –ATN |
| 31 | GND |
| 32 | GND |
| 33 | +BSY |
| 34 | –BSY |
| 35 | +ACK |
| 36 | –ACK |
| 37 | +RST |
| 38 | –RST |
| 39 | +MSG |
| 40 | –MSG |
| 41 | +SEL |
| 42 | –SEL |
| 43 | +C/D |
| 44 | –C/D |
| 45 | +REQ |
| 46 | –REQ |
| 47 | +I/O |
| 48 | –I/O |
| 49 | GND |
| 50 | GND |

**Figure 3**

The SCSI standard permits the use of either shielded or non-shielded cabling and connectors. For both options, the standard provides mechanical drawings of the connectors.

**SCSI bus operation**
The SCSI interface is always in 1 of 4 states, called *phases*:

- bus free
- arbitration (optional)
- selection/reselection
- information

The permissible state transitions are depicted in Figure 4.

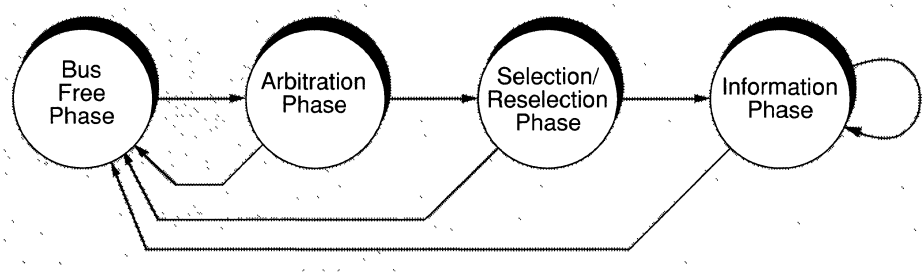


**Figure 4**

The *information* phase refers collectively to 4 other sub-phases:

- command
- data
- status
- message

During the information phase, information passes between initiators and targets. Each target peripheral controller can accept one command at a time for each of the peripheral drives attached it.

**SCSI data transfer modes:**
SCSI supports 2 modes of data transfer:

- asynchronous
- synchronous

During *asynchronous* data transfers the receiving device acknowledges receipt of each byte before the next is sent.

The *synchronous* mode allows the sender to send several data bytes over the SCSI bus at a previously established data transfer rate, without waiting for an acknowledge for each byte. To establish the data transfer rate, the initiator uses the asynchronous mode to send a *synchronous data transfer request message* to the target. Within that message is a parameter that specifies a data transfer rate.

Each time it sends a data byte over the SCSI bus synchronously, the sender generates a strobe on the REQ line. If it were sending the data in the asynchronous mode, the sender would then need to wait until the recipient responded with a transition on the ACK line. However, in the synchronous mode, the sender need not wait for the ACK. Instead, it continues to send data bytes at the specified data rate.

As the sender sends bytes to the receiver, the receiver stores them in a FIFO, to await later processing. However, this FIFO has a finite depth, so some provision must be made to avoid FIFO overruns. For this reason the sender is only permitted to send a specified number of bytes before it must stop and wait for an acknowledge on the ACK line (this number is known as the REQ/ACK offset). For example, the initiator specifies the number of bytes a target can send to it, before waiting for an ACK.

The minimum data transfer period permitted by the SCSI specification is 200 nsec. This allows a maximum synchronous data transfer rate of 5 Mbytes/sec. However, the actual synchronous data transfer rates are typically determined by the speed of the SCSI controller chips chosen by the engineer who designs the adapter board. These controller chips are available from:

- Western Digital
- NCR
- Emulex
- National
- Fujitsu

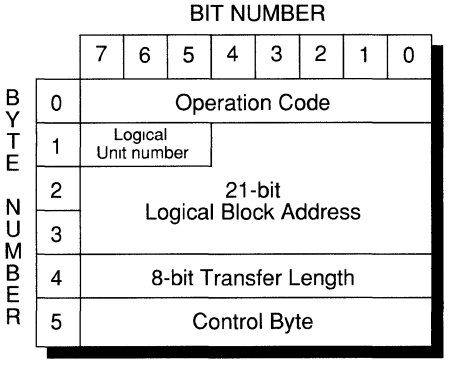


**Figure 5**

**SCSI command descriptor blocks**

Commands are sent to peripherals in the form of *command descriptor blocks*. A typical command descriptor block is shown in Figure 5. This block includes:

- an operation code (op code)
- a logical unit number
- a logical block address
- a data transfer length
- a control byte

There are also other types of command descriptor blocks, which permit access to disks with larger storage capacity. (For example, Figure 6 shows a 10-byte block.) In each case, the recipient can

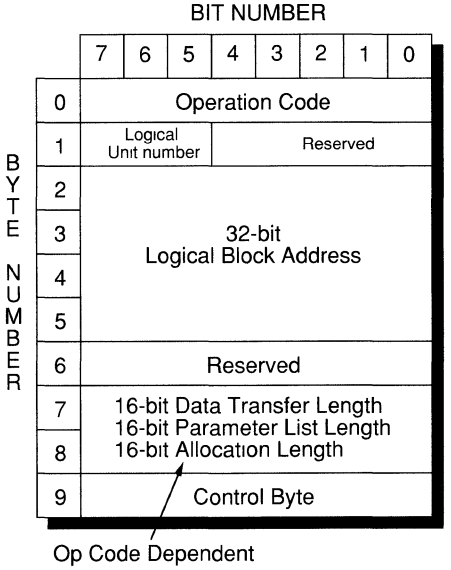


**Figure 6**

deduce the length of the block from the op code in the first byte of the block. The 256 op codes are divided into groups:

- Group 0 = 000x xxxx = 6-byte parameter block
- Group 1 = 001x xxxx = 10-byte parameter block
- Group 2 = 010x xxxx = Reserved
- Group 3 = 011x xxxx = Reserved
- Group 4 = 100x xxxx = Reserved
- Group 5a = 1010 xxxx = Vendor Unique
- Group 5b = 1011 xxxx = Reserved
- Group 6 = 110x xxxx = Vendor Unique
- Group 7 = 111x xxxx = Vendor Unique

**SCSI command classes**
Within each of these groups, individual op codes are designated as:

- Mandatory
- Extended
- Optional
- Vendor unique
- Reserved

The SCSI standard specifies commands and command descriptor block formats for the *mandatory*, *extended* and *optional* op codes. All SCSI devices must respond properly to *mandatory* commands. If a SCSI device allows system software to configure it in a device-independent fashion, it must do so through the *extended* commands. Devices need not be able to respond to *optional* commands to be considered SCSI compatible.

*Vendor unique* command codes are available for use at the discretion of an implementer.

*Reserved* commands are reserved for future standardization.

Figure 7 lists the SCSI commands used to access disk drives. (Disk drives are called *direct access devices* in the SCSI specification.)

**The common command set**
As peripheral vendors gained experience with SCSI, they found that it didn't go far enough in some respects. For example, important commands, which should have been mandatory were only made optional. Another example was where the SCSI standard left important operations to be implemented with vendor unique commands.

Recognizing the potential for incompatibility, several SCSI disk vendors agreed among themselves to support an informal standard called the *common command set*.

Another problem with SCSI was that the intelligence within each SCSI *peripheral* was sometimes forced to make decisions over which the *host* system should have some influence. For example, when the host system issued a SCSI *format* command to a disk controller on the SCSI bus, it needed to specify how many *spare* sectors to allocate, for later use when media defects were detected. However, the SCSI specification made no provision for passing such a parameter.

| Code | Command | SCSI | SCSI-2 |
|---|---|---|---|
| 00 | Test unit ready | O | M |
| 01 | Rezero unit | O | O |
| 03 | Request sense | M | M |
| 04 | Format unit | M | M |
| 07 | Reassign blocks | O | O |
| 08 | Read | M | M |
| 0A | Write | M | M |
| 0B | Seek | O | O |
| 12 | Inquiry | E | M |
| 15 | Mode Select | O | O |
| 16 | Reserve | O | M |
| 17 | Release | O | M |
| 18 | Copy | O | O |
| 1A | Mode Sense | O | O |
| 1B | Start/Stop Unit | O | O |
| 1C | Receive diagnostic results | O | O |
| 1D | Send diagnostic | O | M |
| 1E | Prevent/Allow medium removal | O | O |
| 25 | Read capacity | E | M |
| 28 | Read | E | M |
| 2A | Write | E | M |
| 2B | Seek | O | O |
| 2E | Write and Verify | O | O |
| 2F | Verify | O | O |
| 30 | Search data high | O | O |
| 31 | Search data equal | O | O |
| 32 | Search data low | O | O |
| 33 | Set limits | O | O |
| 34 | Prefetch | R | O |
| 35 | Synchronize cache | R | O |
| 36 | Lock/Unlock cache | R | O |
| 37 | Read defect data | R | O |
| 39 | Compare | O | O |
| 3A | Copy and Verify | O | O |
| 3B | Write buffer | R | O |
| 3C | Read buffer | R | O |
| 3E | Read long | R | O |
| 3F | Write long | R | O |
| 40 | Change definition | R | O |
| 41 | Write same | R | O |
| 4C | Log select | R | O |
| 4D | Log sense | R | O |
| 55 | Mode select | R | O |
| 5A | Mode sense | R | O |

M = Mandatory, O = Optional, E = Extended, R = Reserved

**Figure 7**

To provide for this, the common command set defines a data structure called a *page*. This page (which contains a value for each device-specific parameter stored inside the peripheral) can be passed back and forth between the host and the target.

Even though the common command set was not officially endorsed by ANSI (or by any other recognized standard-setting organization) it was supported by SCSI disk vendors because it ensured compatibility between vendors, and within each vendor's product line.

**SCSI has been able to evolve**
Many interface standards (once they have been created) remain static until they're eventually obsoleted by a subsequent standard. However, it is possible to evolve an interface standard in a way that permits it to continue serving an industry. That only happens if the industry is involved not only in *developing* the standard, but also in its *modification*, as time passes. The SCSI standard is a good example of this. As the need for higher performance became apparent, peripheral manufacturers evolved SCSI into a new upward-compatible standard, called SCSI-2.

**The goals of SCSI-2 are compatibility and connectivity**
Many high performance disk users are drawn to SCSI-2 because of its high data transfer rate. However, although SCSI-2 provides high performance, the 3 objectives stated in the scope of the SCSI-2 draft emphasize *compatibility* and *connectivity*:

- "...to provide host computers with device independence within a class of devices."
- "...to provide compatibility with those SCSI-1 devices that support bus parity and that meet conformance level 2 of SCSI-1."
- "...to move device-dependent intelligence out to the SCSI-2 devices."

The last objective emphasizes SCSI's ability to hide the device-specific details of each peripheral drive from the host system. For example, when you interface *directly* to a disk drive, you must understand the physical organization of the disk drive media, including:

- how many heads the disk has.
- how many tracks the disk has.
- how many sectors per track the disk has.
- where the imperfections are in the media.

With a SCSI disk, you don't have to concern yourself with these physical details. A disk just looks like a string of *logical blocks*, starting with block 1, and ending with the size of the disk. The disk appears to have perfect media: no defects require you to remap sectors that lie on blemishes. If the host system reads logical block 100, the SCSI disk controller inside the drive enclosure translates logical block number 100 into the appropriate physical head number, track number, and sector number.

**SCSI-2 is upward compatible with SCSI**
Even though a considerable amount of functionality has been added to SCSI-2, compatibility with SCSI is still preserved. SCSI and SCSI-2 devices may reside on the same bus and may interact. The physical characteristics of SCSI-2 are the same as SCSI. For example, SCSI-2 uses the same cabling and electrical scheme as SCSI, unless wide data transfers (discussed later) are implemented, in which case a 2nd cable is added. The standard SCSI cable is referred to as the *A-cable* in SCSI-2 nomenclature, and the 2nd cable is called the *B-cable*.

SCSI targets can reject SCSI-2 commands that aren't defined in the SCSI standard. For example, suppose you have a VME-to-SCSI-2 adapter sending commands to a SCSI disk drive. The adapter may attempt to initiate a *wide* data transfer, by sending a message to the disk drive. However, since the SCSI disk drive has no notion of wide data transfers, it doesn't understand the message, and responds by *rejecting* the message, using the SCSI rejection protocol. The SCSI-2 adapter then detects the rejection, and initiates the data transfer using byte-wide SCSI-compatible data transfers.

Note: The actual strategy used by the SCSI-2 host adapter is not prescribed in the SCSI-2 standard. For example, the adapter might first respond to the rejection by initiating a *synchronous* SCSI data transfer, before finally defaulting to an *asynchronous* SCSI data transfer.

Since a SCSI-2 host bus adapter implements the entire mandatory SCSI protocol (and in many cases the entire common command set) it can choose to operate entirely within those protocols. That ensures compatibility with SCSI peripherals.

Since SCSI-2 peripherals respond to all mandatory SCSI commands, they can be directed by SCSI host adapters.

**SCSI-2 offers 3 extensions for increased performance**
SCSI-2's performance capabilities are the result of 3 extensions:

- *Fast* synchronous data transfers
- *Wide* data transfers
- *Tagged commands*

Fast synchronous data transfers and wide data transfers combine to increase SCSI-2's maximum data transfer rate to 40 Mbytes/sec. The other extension (tagged commands) allows target peripheral controllers to queue multiple commands, and to sequence the execution of those commands for optimum throughput.

**Fast synchronous data transfers**
The *fast* synchronous data transfer option (fast SCSI) is the best-known SCSI-2 feature. It doubles the 5 MHz maximum SCSI synchronous data transfer rate to 10 MHz. In doing this, the minimum period for synchronous data transfers has been reduced from SCSI's 200 nsec to a minimum of 100 nsec for SCSI-2.

The term *fast SCSI* refers only to the number of *transfers* per second. When fast SCSI transfers are done over a single cable (which permits only 8-bit transfers) data transfers can be done at 10 Mbytes/sec. When 2 cables are used, data can be transferred 32 bits at a time. This permits data transfer rates up to 40 Mbytes/sec.

Fast SCSI is an *optional* mode of operation, and can only be used if both the initiator and the target are equipped to handle it. To initiate a fast SCSI data transfer, the initiator sends a *synchronous data transfer request message* to the target. Embedded in the message are 2 parameters:

- a parameter that specifies the REQ/ACK offset
- a parameter which specifies the data transfer period

The *REQ/ACK offset* tells the target peripheral controller how many REQ pulses it can send during a data transfer before it must wait for an ACK pulse from the adapter.

The *data transfer period* tells the target peripheral controller the data rate to use when sending to the adapter. If the peripheral controller can't send at that rate, it *rejects* the message. Then the adapter sends a new message specifying a lower data rate.

**Wide data transfers**

The *wide* data transfer mode of operation (often called *wide SCSI*) increases the data path width from 8 bits (SCSI) to either 16 bits or 32 bits (SCSI-2), through the used of a 2nd cable, called the *B cable*. This 2nd cable is a 68-conductor cable that provides 24 additional data bus signals, 3 parity signals, a REQ signal, and an ACK signal. (See Figures 8 and 9) Note: The B cable is only required if wide data transfers are used.

The wider data path is used only for data transfers. Commands and status are still passed using 8-bit transfers. Widening the data path increases data transfer rates by transferring more data per transfer cycle.

Like fast SCSI, wide SCSI is an option. Wide data transfers are initiated with a *wide data transfer request message*. This message includes a parameter that specifies the data transfer width.

**SCSI-2 allows commands to be queued in the target**

SCSI only permits the VME-to-SCSI adapter to send one command at a time to each peripheral drive. The adapter must then wait for a *status response* from that drive, before sending another command to it.

| | |
|---|---|
| 1 | GND |
| 2 | GND |
| 3 | GND |
| 4 | -DB(8) |
| 5 | GND |
| 6 | -DB(9) |
| 7 | GND |
| 8 | -DB(10) |
| 9 | GND |
| 10 | -DB(11) |
| 11 | GND |
| 12 | -DB(12) |
| 13 | GND |
| 14 | -DB(13) |
| 15 | GND |
| 16 | -DB(14) |
| 17 | GND |
| 18 | -DB(15) |
| 19 | GND |
| 20 | -DB(P1) |
| 21 | GND |
| 22 | -ACKB |
| 23 | GND |
| 24 | GND |
| 25 | GND |
| 26 | -REQB |
| 27 | GND |
| 28 | -DB(16) |
| 29 | GND |
| 30 | -DB(17) |
| 31 | GND |
| 32 | -DB(18) |
| 33 | TERMPWRB |
| 34 | TERMPWRB |
| 35 | TERMPWRB |
| 36 | TERMPWRB |
| 37 | GND |
| 38 | -DB(19) |
| 39 | GND |
| 40 | -DB(20) |
| 41 | GND |
| 42 | -DB(21) |
| 43 | GND |
| 44 | -DB(22) |
| 45 | GND |
| 46 | -DB(23) |
| 47 | GND |
| 48 | -DB(P2) |
| 49 | GND |
| 50 | -DB(24) |
| 51 | GND |
| 52 | -DB(25) |
| 53 | GND |
| 54 | -DB(26) |
| 55 | GND |
| 56 | -DB(27) |
| 57 | GND |
| 58 | -DB(28) |
| 59 | GND |
| 60 | -DB(29) |
| 61 | GND |
| 62 | -DB(30) |
| 63 | GND |
| 64 | -DB(31) |
| 65 | GND |
| 66 | -DB(P3) |
| 67 | GND |
| 68 | GND |

**Figure 8**

SCSI-2 has *tag messages*, which allow an adapter to send multiple commands to a peripheral drive, without waiting for a status response for previously sent commands. As the SCSI-2 peripheral controller receives these commands, it queues them for later execution.

*Tagged commands* are commands that are preceded by a tag message. The tag message contains a 1-byte unsigned integer, called the *tag value*. When the target receives a tagged command, it puts the tag value into the queue, along with the command.

There are 2 types of tagged commands:

- *Ordered* tagged commands
- *Unordered* tagged commands

*Ordered tagged commands* can be used to ensure that commands are executed in a first-come-first-served manner.

*Unordered tagged commands* are used when the command execution order is not important. The target takes these *unordered* tagged commands, and queues them, just as it would queue *ordered* tagged commands. However, it is then allowed to *sort* the unordered tagged commands, to speed up overall throughput. There are 2 techniques used to improve throughput:

- Command combining
- Seek optimization

*Command combining* is when the disk controller searches for queued commands that access adjacent sectors, and combines them into a single, multiple-sector disk access. This reduces the number of disk revolutions needed to execute a series of commands.

*Seek optimization* reduces seek latencies by reordering the sequence of the

| | |
|---|---|
| 1 | GND |
| 2 | GND |
| 3 | +DB(8) |
| 4 | –DB(8) |
| 5 | +DB(9) |
| 6 | –DB(9) |
| 7 | +DB(10) |
| 8 | –DB(10) |
| 9 | +DB(11) |
| 10 | –DB(11) |
| 11 | +DB(12) |
| 12 | –DB(12) |
| 13 | +DB(13) |
| 14 | –DB(13) |
| 15 | +DB(14) |
| 16 | –DB(14) |
| 17 | +DB(15) |
| 18 | –DB(15) |
| 19 | +DB(P1) |
| 20 | –DB(P1) |
| 21 | +ACKB |
| 22 | –ACKB |
| 23 | GND |
| 24 | DIFFSENS |
| 25 | +REQB |
| 26 | –REQB |
| 27 | +DB(16) |
| 28 | –DB(16) |
| 29 | +DB(17) |
| 30 | –DB(17) |
| 31 | +DB(18) |
| 32 | –DB(18) |
| 33 | TERMPWRB |
| 34 | TERMPWRB |
| 35 | TERMPWRB |
| 36 | TERMPWRB |
| 37 | +DB(19) |
| 38 | –DB(19) |
| 39 | +DB(20) |
| 40 | –DB(20) |
| 41 | +DB(21) |
| 42 | –DB(21) |
| 43 | +DB(22) |
| 44 | –DB(22) |
| 45 | +DB(23) |
| 46 | –DB(23) |
| 47 | +DB(P2) |
| 48 | –DB(P2) |
| 49 | +DB(24) |
| 50 | –DB(24) |
| 51 | +DB(25) |
| 52 | –DB(25) |
| 53 | +DB(26) |
| 54 | –DB(26) |
| 55 | +DB(27) |
| 56 | –DB(27) |
| 57 | +DB(28) |
| 58 | –DB(28) |
| 59 | +DB(29) |
| 60 | –DB(29) |
| 61 | +DB(30) |
| 62 | –DB(30) |
| 63 | +DB(31) |
| 64 | –DB(31) |
| 65 | +DB(P3) |
| 66 | –DB(P3) |
| 67 | GND |
| 68 | GND |

**Figure 9**

queued commands, so that the drive's read heads will never pass over a cylinder for which there is a pending request.

**SCSI-2 offers 3 upgrades in the logical interface**
Even though most of the fanfare is directed at the *physical* changes that allow higher SCSI-2 data transfer rates, much of the effort put into SCSI-2 involves upgrading the *logical* interface with 3 broad types of enhancements:

- Incorporation of the *common command set*
- Addition of new *logical functions*
- Addition of new *device types*

**Incorporation of the common command set**
Much of the SCSI common command set has been made mandatory in SCSI-2. An example of this is the use of *pages* to pass parameter values between the host system and SCSI-2 peripherals. For example, several page types have been defined for use with disk drives:

- Caching
- Floppy disks
- Formatting
- Medium types
- Notching
- Partitioning
- Read/Write error recovery
- Rigid disk geometry
- Verify error recovery

SCSI-2 also defines page formats for accessing and modifying the parameters stored in other types of peripherals.

**New logical functions**
SCSI protocols require a peripheral to return a *status* to the host adapter, upon receiving each command. However, depending upon the peripheral, this might take considerable time. For example, suppose a host system initiates a REWIND operation on one of its SCSI tape drives. The host would not get a *completion status* until the rewind operation is finished.

It would be convenient if the tape drive could initiate the rewind operation, and then return a status to the host, indicating that it had *accepted* the command. Then, at some later time, the drive could notify the host when the rewind was complete.

Unfortunately the SCSI standard has no provision for doing this. However, SCSI-2 defines an *asynchronous event notification* protocol. This protocol allows a peripheral to notify an adapter of some event that has occurred inside the drive. This event need not be the result of a previous command from the host. For example, if someone takes the tape drive off-line to change tapes, the drive could notify the host system that it has been taken off-line.

**New device types in SCSI-2**
SCSI-2 also defines 3 new device types:

- Scanner
- Medium Changer
- Communications

In addition, SCSI-2 has *redefined* the optical device types previously defined by SCSI...

- write-once-read-many (WORM) direct access
- read-only direct access

...to give 3 new types of optical devices:

- write-once-read-many direct access
- CD-ROM serial access
- read-only or write-many direct access

Each of these SCSI-2 device types have command sets that have been designed to support currently available products.

**Migrating from SCSI to SCSI-2**
Many current SCSI users want to migrate from SCSI to SCSI-2, as long as that migration doesn't *prevent* them from using existing (proven) SCSI peripherals. To allow for this, the SCSI-2 designers provided a migration path that can be followed in an incremental fashion. (See Figure 10)

**Migration step 1: Replace the VME-to-SCSI adapter**
A VME-to-SCSI-2 adapter should provide an operational *superset* of the VME-to-SCSI adapter it replaces. To avoid redesign of the existing adapter's software device driver,

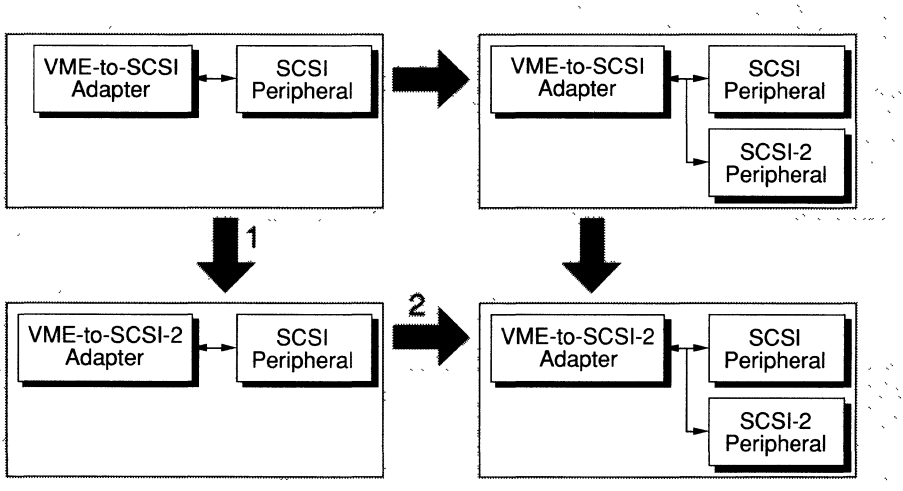


**Figure 10**

the VME-to-SCSI-2 adapter should also be designed to preserve the existing adapter's register interface. This allows you to substitute the VME-to-SCSI-2 adapter for the VME-to-SCSI adapter, while maintaining the full function of the existing system, with no changes in the SCSI peripherals.

**Migration step 2: Add SCSI-2 peripherals**
Once the VME-to-SCSI adapter has been replaced with a VME-to-SCSI-2 adapter, SCSI-2 peripheral devices can be added to the system. The SCSI-2 peripherals are simply plugged into the bus. Since the old VME-to-SCSI *software device driver* may not fully use the features of the VME-to-SCSI-2 adapter, you can't exercise all of the SCSI-2 features in these new peripherals. However, you can still test and verify them as SCSI devices. Even though you only use a portion of their full functionality, they can still be useful in the system.

**Migration step 3: Update the adapter device drivers**
The final step to an incremental SCSI-2 migration is to replace the VME-to-SCSI adapter's *device drivers* with drivers that fully support the extended features of the new VME-to-SCSI-2 adapter. The VME-to-SCSI-2 adapter can then send SCSI-2 messages and commands, when interacting with the peripherals.

Note: If the SCSI-2 system will be using wide data transfers, an additional step is required. You must install a 2nd cable between the VME-to-SCSI-2 adapter and the SCSI-2 peripherals. This cable provides the additional data lines that will be needed to move data 16 bits (or 32 bits) at a time.

An alternative migration path is also shown in Figure 10. SCSI-2 peripherals can be incorporated into the SCSI system before the VME-to-SCSI adapter is upgraded to a VME-to-SCSI-2 adapter. Since the SCSI-2 peripherals recognize all SCSI commands, they can be controlled by the VME-to-SCSI adapter.

**PTDs and PDAs: 2 ways to build fast disk systems**
There are 2 popular techniques for building high performance disk subsystems:

- Parallel Transfer Disks (PTDs)
- Parallel Disk Arrays (PDAs)

**What are PTDs?**
PTDs allow very fast data transfer rates by performing parallel data transfers to multiple disk surfaces, which all share a common spindle. The actual data transfer rate is the *product* of the data transfer rate for each surface and the number of surfaces. PTDs are often special versions of standard disks. For example, Fujitsu makes a PTD with an 18 Mbyte/sec data transfer rate. It reads from 6 heads in parallel, at data transfer rates of 3 Mbytes/sec, and routes the data to the host through an SMD-like interface. Its interface differs from SMD in that it has 6 serial data channels, instead of just one. (One channel for each disk head.)

**What are PDAs?**
A Parallel Drive Array (PDA) is a disk subsystem consisting of several independent disk drives, which are all accessed through a single disk controller. This PDA controller provides fast data transfer rates by using a technique called *striping*. As the controller

writes data to the disks, it divides the data up on a byte-by-byte basis. It then writes the 1st byte to drive 1, the 2nd byte to drive 2, etc.

Data transfers to the various drives in the drive array occur in *parallel*, so the resulting data transfer rate is the product of the number of drives in the array and each individual drive's data transfer rate. Because all of the striping details are handled by the intelligent PDA controller, the host sees the PDA subsystem as a single, fast, large-capacity drive.

**Using PTDs and PDAs in microcomputers**

PTDs and PDAs have traditionally been used in very expensive, extremely high performance systems, such as supercomputers or special imaging systems. However, extremely fast CISC microprocessors, RISC processors, and multiple-processor VMEbus systems have created a demand for disk I/O performance levels that can only be provided with PTDs and PDAs.

Historically, PTDs and PDAs have been built using proprietary interfaces. However, SCSI-2 now provides a viable interface for PTDs and PDAs. In addition to SCSI-2's performance, it also provides the benefits of the huge base of SCSI compatible disk drives.

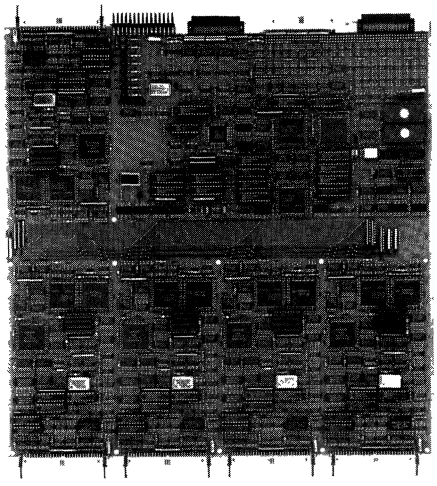

**Figure 11**

**The Rimfire 6600**

The Rimfire 6600 PDA SCSI-2 disk controller (shown in Figure 11) interfaces 5 ESDI compatible disks to the SCSI-2 bus. (See Figure 12) It contains 6 processors. One is a *supervisory* processor, which controls overall board operation. The other 5 processors each serve as an ESDI controller for one disk drive.

The onboard intelligence provided by these processors allows striping of the data bytes to 4 of the drives, as well as a parity byte to the 5th drive. The parity byte is generated from the data bytes stored on the 4 data drives. (See Figure 13.) Since this striping is hidden from the host computer, the disk array looks (to the host) like a single, fast, large capacity drive.

The actual data transfer rate of this disk array depends on the drives used. With 20 MHz ESDI drives (available from Hitachi and from Hewlett Packard) the

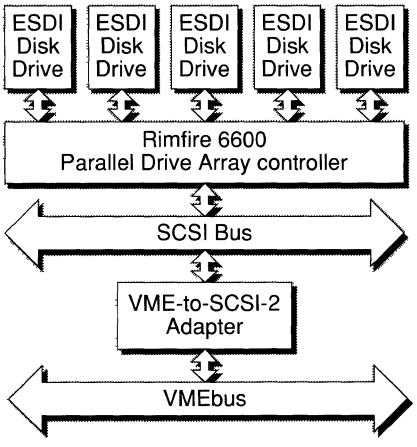


**Figure 12**

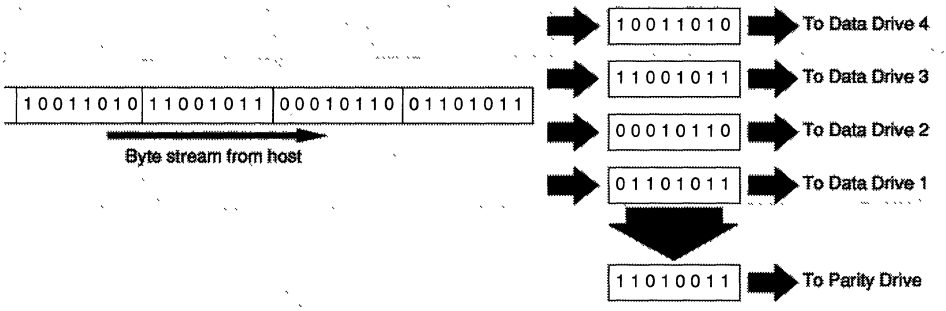


**Figure 13**

burst data transfer rate is 10 Mbytes/sec, as shown by the following computations:

(20 Mbits/sec)/(8 bits/byte) = 2.5 Mbytes/sec

(2.5 Mbytes/sec) x (4 drives) = 10 Mbytes/sec

The Rimfire 6600 controller board can sustain this 10 Mbyte/sec SCSI-2 data transfer rate indefinitely. However the actual data transfer rate will be lower because, even though the data in each sector is read from the disk at 20 Mbits/sec, the *gaps* between the sectors lower the *average* data transfer rate. In addition, when the head reaches the end of a track, it must be repositioned. This takes time.

In fact, head repositioning times can be quite large. As a result, the average sustained data rate is usually defined to be the rate at which you can read a *track* of data. That depends on how the disk is formatted.

In general, a disk is formatted with a fixed number of bytes per sector, plus a few bytes of overhead, associated with each sector. If you have 512-byte sectors, and 80 bytes of overhead per sector, you will not be able to put as much data on each track as you could with 1024-byte sectors, and the same 80 bytes of overhead per sector.

For example, suppose you have a total unformatted track capacity of 20,000 bytes. If you format it with 512-byte sectors, you might get about 16.5 Kbytes of storage per track. However, if you use 2-Kbyte sectors, you might get 18 Kbytes of storage on the track.

In general, with an optimal sector size, a Rimfire 6600-based PDA can provide sustained rates of about 85% to 90% of the combined burst data transfer rates of the drives.

**Surviving a disk failure**

The parity drive allows the array to continue operating in the event of a single drive failure. If a single drive failure occurs, the failed drive can be removed, and a replacement drive installed. Then the data from the failed drive is *regenerated* on the replacement drive.

The removal, replacement, and data regeneration process is performed while the Rimfire 6600 continues to execute SCSI-2 commands. The ability to tolerate single drive failures improves the Mean Time Data Availability (MTDA) of the disk subsystem. The MTDA of an array of disk drives predicts the mean time between *simultaneous* failures of two disk drives, and can be calculated as follows...

$$MTDA = \frac{(MTBF)^2}{(N_d+N_p)(N_d+N_p-1)(MTTR)}$$

Where...

$N_d$ = Number of data drives
$N_p$ = Number of parity drives

For a 5-drive array, with individual drive MTBFs of 40,000 hours, and a Mean Time To Repair (MTTR) for a single drive failure of 72 hours, the calculated MTDA is approximately 1.1 million hours (127 years).

**Spindle synchronization**
One problem with using a simple parallel drive array is that the various disks may be at different rotational positions when an access is started. Since striping stores data on all the disks, the access cannot be completed until the last drive rotates far enough to bring the needed sector under its read head.

If there were only a single disk drive, the average time would be half a rotation. However, with 5 drives, it's quite likely that at least one drive will need more than half a rotation, so the average time is greater.

To reduce this average time, a technique called *spindle synchronization* is used. One drive serves as a *master*, and the other drives are *slaves*. The master drive provides a *reference* signal, and the other drives synchronize their spindle motion to that reference. Once synchronized, the corresponding sectors on each of the disk drives pass under their respective read heads at the same time. This reduces the average access time to what it would be for a single drive: half a rotation.

The advantages of spindle synchronization were great enough to convince ESDI vendors to *add* the needed signals, as an appendix to the ESDI specification. The Rimfire 6600 controller board uses these signals to tell the various drives in the array who is the master, and who are the slaves. If a drive fails, its dedicated processor (on the 6600 board) detects the failure, and reports it to the supervisory processor. If the supervisory processor detects a failure of the master drive, it selects another drive to be the master drive, allowing the failed drive to be pulled out and replaced.

Note: The information on the parity drive is *not* used to detect *drive* failures. A drive failure is detected by the dedicated processor, based upon CRC errors, etc. The parity information is used, *after* a failed drive has been replaced, to *regenerate the data* .

**Cache Memory**
To further improve throughput, the Rimfire 6600 controller board has 512-Kbytes of *cache memory*. This cache is physically organized as 128 Kbytes of cache per data

drive. If the controller is commanded to read a block of data, it can *pre-read* an entire track of data from each of the drives into the cache. With 128 Kbytes per drive, the cache can hold 3 to 5 tracks per drive at any given time. Sequential requests (typical of applications requiring PDA technology) can then be satisfied by simply accessing the cache, eliminating the need for additional disk accesses.

**The Rimfire 6600's SCSI-2 interface**
The Rimfire 6600's SCSI-2 interface supports burst transfers across the SCSI-2 bus at rates up to 20 Mbytes/sec. This is achieved through 32-bit-wide data transfers, and a 5 MHz synchronous data transfer rate. The board also implements tagged commands, allowing it to receive multiple commands from the host. The controller optimizes command execution through command combining and seek optimization.

**The Rimfire 3550**
Ciprico has also designed a VME-to-SCSI-2 adapter board called the Rimfire 3550. (See Figure 14) This VME-to-SCSI adapter supports both the A cable and the B cable of the SCSI-2 bus, allowing wide (32-bit) data transfers. It also allows the VMEbus system to send tagged commands over SCSI-2. Like the Rimfire 6600, the 3550 supports synchronous SCSI-2 data transfers at 20 Mbytes/sec.

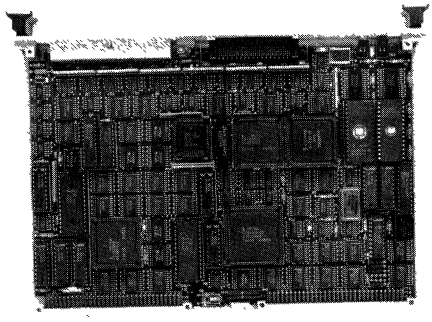

**Figure 14**

The 3550 board uses a *short-burst FIFO* to buffer the data flow between the SCSI-2 bus and the onboard 2-Kbyte FIFO. (See Figure 15.) This 1st Short-Burst FIFO, and the 2-Kbyte FIFO, accept data from the SCSI-2 bus, while the board waits for permission to use the VMEbus. Once the board gains access to the VMEbus, a 2nd Short-Burst FIFO empties the 2-Kbyte FIFO at VMEbus rates. This 2nd Short-Burst FIFO allows the board to transfer data over the VMEbus in excess of 30 Mbytes/sec.

The adapter also uses a *pipelined system interface* to optimize VMEbus DMA transfers. Its address and counter registers are *pipelined*. This allows the adapter's onboard processor to *preload* the starting address (and the data transfer count) for each DMA transfer while the previous DMA transfer is still in progress.

This pipelined register architecture is especially useful for *scatter/gather* data transfers, often seen in UNIX-based systems. Suppose an application program requests the UNIX operating system to open and read several contiguous sectors from a disk file. UNIX then selects memory buffers from a buffer pool, directing one sector into each buffer. Since the available buffers in the pool aren't usually contiguous in memory, sequential sectors from the disk may not be contiguous in memory. Because of this you need some way to DMA each sector to a different buffer location in memory. The pipelined address and counter registers allow the board to DMA sequential sectors into different areas of memory, without pausing at the end of each DMA transfer to reconfigure the DMA controller.

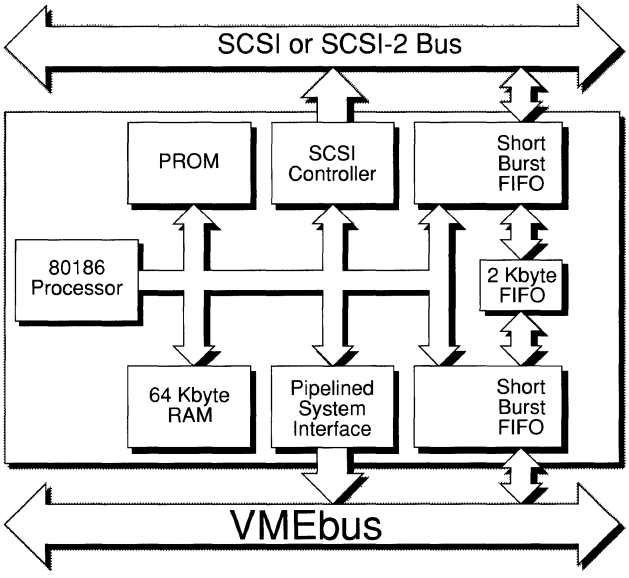


**Figure 15**

**The Rimfire 3550 software interface**
The software interface to the Rimfire 3550 VME-to-SCSI-2 adapter consists of parameter blocks, which include SCSI-2 command descriptor blocks (like the ones shown earlier in Figures 5 and 6) plus some additional parameters that are needed to execute the command. For example, in order to transfer data between VMEbus memory and the disk drive, you need to provide a VMEbus address.

A VMEbus processor board builds a *circular queue* of these parameter blocks in VMEbus memory, and then writes a pointer to the 3550 adapter board, to indicate the memory location of the head of the queue. The 3550 board then extracts and executes these parameter blocks, updating the pointer each time.

Since this method of operation is the same as that used on the earlier Rimfire 3510 Series of VME-to-SCSI adapters, the Rimfire 3550 board operates as a VME-to-SCSI adapter when used with 3510 device drivers. If minor coding changes are incorporated into the 3510 drivers, the wide data transfer option, and the tagged command facility of the 3550 can also be used.

**Bill Moren** has worked for Ciprico since June of 1978, and has served in various engineering and marketing positions prior to his current position as product manager. His current responsibilities include the marketing management of the company's products from concept to obsolescence.