TC2000™
Hardware Architecture

VERSION OF FEBRUARY 14, 1990

THIS DOCUMENT IS BBN ACI PROPRIETARY

Revision: 2/14/90

February 14, 1990

Part No.
Document Rev:

BBN Advanced Computers Inc.

TC2000™
Hardware Architecture

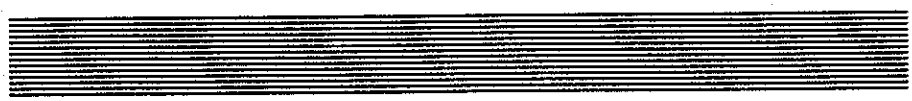
VERSION OF FEBRUARY 14, 1990

THIS DOCUMENT IS
BBN ACI PROPRIETARY

Revision: 2/14/90

February 14, 1990

Part No.
Document Rev:



BBN Advanced Computers Inc.

RESTRICTED RIGHTS LEGEND

BBN Advanced Computers Inc
10 Fawcett St.
Cambridge MA 02138

RELEASE LEVEL

This document conforms to the February 1990 TC2000™ multiprocessor hardware..

NOTICE

BBN ACI has prepared this manual for the exclusive use of BBN customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by BBN ACI. BBN ACI assumes no responsibility for any errors that appear in this document.

TRADEMARKS

Butterfly is a registered trademark of Bolt Beranek and Newman Inc.
Chrysalis, TC2000, nX, Uniform System, Xtra, Gist, and TotalView are trademarks of Bolt Beranek and Newman Inc.
UNIX is a registered trademark of AT&T Bell Laboratories.
DEC, VAX, and VT are registered trademarks of Digital Equipment Corporation.
VMS, VAX/VMS, MicroVAX, Ultrix, and DECnet are trademarks of Digital Equipment Corporation.
IBM and IBM PC are trademarks of International Business Machines.
Multibus and Intel are registered trademarks of Intel Corporation.
X Window System is a trademark of the Massachusetts Institute of Technology.
MC68000, MC68020, MC68881, MC68882, MC68851, MC88000, MC88100, MC88200, and VMEbus are trademarks of Motorola Semiconductor Products, Inc.
QTC and Math Advantage are registered trademarks of Quantitative Technology Corporation.
pSOS, pSOS^{+}, pSOS^{+m}, pRISM, pUCP, pREP/C, pROBE, and pHILE are trademarks of Software Components Group, Inc.
MS-DOS is a registered trademark of Microsoft Corporation.
TeleSoft and TeleGen2 are trademarks of Telesoft.
Sun Microsystems and Sun Workstation are registered trademarks of Sun Microsystems, Inc.
OSN, ONC, NeWS, and NFS are trademarks of Sun Microsystems, Inc.
4.2BSD and 4.3BSD are trademarks of the Trustees of the University of California.
Ethernet is a registered trademark of Xerox Corporation.

I am grateful to the engineers on this project for their support and encouragement, their writing of documents used as source material for this book, and their helpfulness and patience. John Goodhue has been instrumental in managing the Butterfly II hardware development, and in guiding the development of this book as well.

— Micahel Beeler, author, April 1989

Justin Aborn
Arthur Babitz
Tom Blackadar
Kent Cauble
Guy Fedorkow
Tom Feist
John Goodhue
Lise Graham
Ward Harriman
Phil Herman
Jack Hill
Kent Hoult
Brian Kalita
Allen King
Cinsy Krehbiel
Stan Lackey
Andy Latto
Phil Levin
Kevin McGrattan
Paul Moruzzi
Gary Muntz
Lou Piazza
Randy Rettberg
Mike Sollins
Julie Tiao
Jeff Wise

Also instrumental were the early developers of bootstrap, TCS, diagnostic and operating system software:

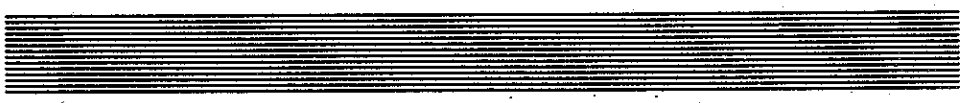
Tom Calderwood
Ken Chin
Bob Clements
Jim Gibson
Carl Howe
Mark Hutcheson
Nancy Keene
Jeff Kenton
Bill Mathews
Jim Mcleish
Roger Preston
Wolfgang Rupprecht

The development of the Butterfly II machine builds on earlier multiprocessor work at BBN, particularly the Pluribus and more recently the original Butterfly computer. The architects and developers of these machines have indirectly, and often directly, aided the development of the Butterfly II machine.

BBN ACI thanks the above contributors for their efforts in developing this manual.

Contents

List of Figures



Foreword to February 14, 1990 Version

This version represents the completion of the present phase of the planned document. Changes from the previous (April 14, 1989) version are scattered throughout, but the major ones are:

- The memory chapter (4) has been updated to reflect the new CPU interface on the function board.
- The core components (chapter 9 — midplane, switch cards, clock card) descriptions have been improved in many places.
- The TC/FPV chapter (11) has been very substantially revised, particularly to reflect the changes from the earlier B2VME function board.
- The T-bus Specification is now included, as Appendix B.
- The machine name *Butterfly II* is now properly *TC2000*, and the circuit card names have similarly changed from *B2—* to *TC/—*. The text of this document is updated only in the chapter on the TC/FPV function board.

The reader may find the outline below helpful in casting this version in the context of the planned document. The chapters that are not yet available are marked ☐.

1. Introduction ☐

— intent of machine
— outgrowth of BF I
— bullets on important features:
  high performance (speed, size, contention, latency (of switch, intrrupts))
  high availability (redundancy, TCS)
  uniform I/O system
— major components: switch, function cards
— bullets on switch features:
  bidirectional, locking, bounded latency (priority)
— bullets on function cards: processor + memory, b2vme, future I/O
— memory design: global address space, mapping, demand paging, interleaving
— support components: clock, TCS, power, packaging
— topics of other chapters, other (planned) books on the product

2. Design concepts ☐

This chapter describes the details of the major components to a level sufficient to support the complete discussion of components in following chapters. This material is more detailed than in the introduction, and the reader can stop reading after this chapter with a coherent picutre of the machine, if detailed knowledge is not needed. The support components will also be discussed here, in less depth than the major components. Discussion of design tradeoffs, and

motivation for the chosen design, belongs in this chapter. Hardware redundancy is discussed here fully (switch, clock, TCS, power).

3. The BF II switch

4. BF II memory system

5. BF II function card ☐

— concept:
  different function cards provide different mixes of processing and I/O
  e.g., A.P. card, VME + processor (B2VME), T1 I/O card,
  multi-88000 card, ...
— 88000 and CMMU
— transaction bus
— interrupt system and latency
— clocking
— power

6. Test and Control System

7. Power distribution ☐

— 48-volt bulk power
— local regulation

8. Signalling ☐

— single-ended signals
— signal quality issues and analysis
— controlled impedance
— connectors
— cabling
— timing constraints

9. Butterfly II switch components

10. Packaging ☐

— description of modules (physical size and layout, power, etc.)
— eight-slot module
— cabinet — dimensions, clearances, position of compoments
— I/O — cabling, connectors, etc.

11. B2VME function card

Appendix A: SIGA Specification

Appendix B: T-bus Specification

Appendix C: SGA Specification ☐ (available separately)

Appendix D: LCON Specification ☐ (available separately)

Appendix E: TCS Voltage Monitoring Analysis ☐ (available separately)

Appendix F: Data Sheets and References ☐ (available separately)

Glossary ☐

Index ☐

3

The Butterfly II Switch

The Butterfly II switch provides communication among function boards. It is central to the machine architecture and embodies many important and unique features. This chapter begins with a tutorial discussion of the switch, then examines the major operational aspects, and concludes with details of the switch protocol and the hardware that implements the switch.

Knowing the names of some switch components will help the reader in the sections below. The fundamental hardware of the switch is three types of custom gate array VLSI chips, the SGA, SIGA and LCON. The SGA (Switch Gate Array) performs the actual routing of messages through the switching network of many interconnected SGA chips. The SIGA (Switch Interface Gate Array) interfaces the switching network to devices on function boards. The SIGA chip contains a "requester" section that initiates connections through the switch and a "server" section that responds to connections. The LCON (Level CONverter) converts signals between the different electrical conventions used in the SGA and in the SIGA.

3.1 Overall Design Concept

The Butterfly switch distinguishes the Butterfly family of computers from many other parallel processor designs. Its importance is reflected in the fact that the custom VLSI chips in the Butterfly II implement and support the switch; other portions of the machine are assembled using commonly available parts. Because of significance of the switch, this section begins at a more elementary level than other parts of this book.

3.1.1 Butterfly Switch Basics

Computers of the Butterfly family share a general switch design that is specialized from one generation of Butterfly computer to another to match that computer's overall architecture. The Butterfly switch is a network of crossbar

switches. This design has several advantages compared to other common parallel processor interconnection schemes. For example, a bus design has a maximum bandwidth that limits performance when the machine is expanded beyond the point of bus saturation. A full crossbar interconnect expands as the square of the number of interconnected components, rapidly becoming prohibitive in size and cost. Cube connected machines scale up more gracefully than do bus or full crossbar designs, but each processor node must be designed with several interconnection ports, and the number of those ports places a design limit on the maximum machine size. Tree architectures often require very large cabling among switching nodes near the top of the tree to avoid a bandwidth bottleneck there. The cabling in a large Butterfly machine is distributed throughout the machine, not concentrated in one area.

The network of crossbar switches used in Butterfly computers supports scaling of the machine from small to very large configurations. As processor nodes are added, only a moderate growth in the size of the switch is needed; N nodes require a switch of size proportional to N log N. The bandwidth of the switch grows as the machine is scaled up, so the switch does not become a bottleneck at any size of machine. The switch is made of identical switch components, so expansion is easy and does not require redesign or mixtures of different hardware. The Butterfly switch has relatively small cables between switch components and between the switch and function boards (such as processor nodes). The design of the switch interface on function boards is not affected by the machine size, so the same interface applies to any size of Butterfly machine.

Figure 3–1 Two–by–two crossbars.

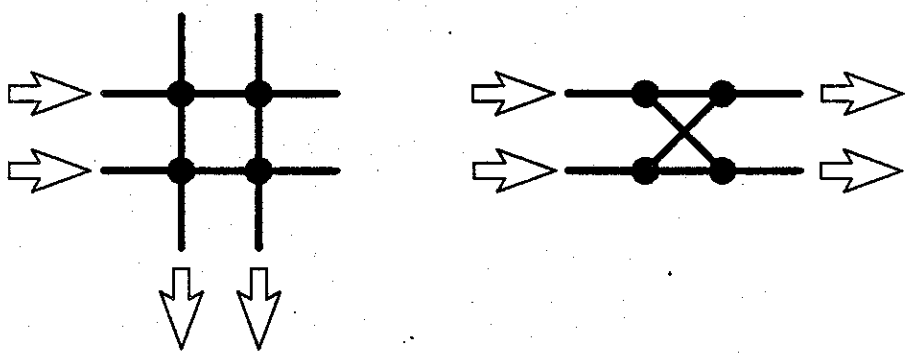


Figure 3–1 shows a two–by–two crossbar switch in two notations. The left diagram shows two horizontal wires and two vertical wires. Each of the four intersections of wires is a *crosspoint*. The wires at the crosspoint are normally insulated from each other, and closing the crosspoint connects them. This diagram resembles the physical construction of electromechanical crossbar switches once used in telephone exchanges. The diagram on the right shows

two wires on the left side, each of which may be passed straight through to the wire on the right, or may be switched over to the other wire on the right. This diagram resembles railroad tracks, and the data flow in some Fast Fourier Transform algorithms. Its resemblance to a butterfly is the origin of the name "butterfly transform" in signal processing, and of the Butterfly family of computers.

Figure 3–2 Small switch showing Butterfly concept.

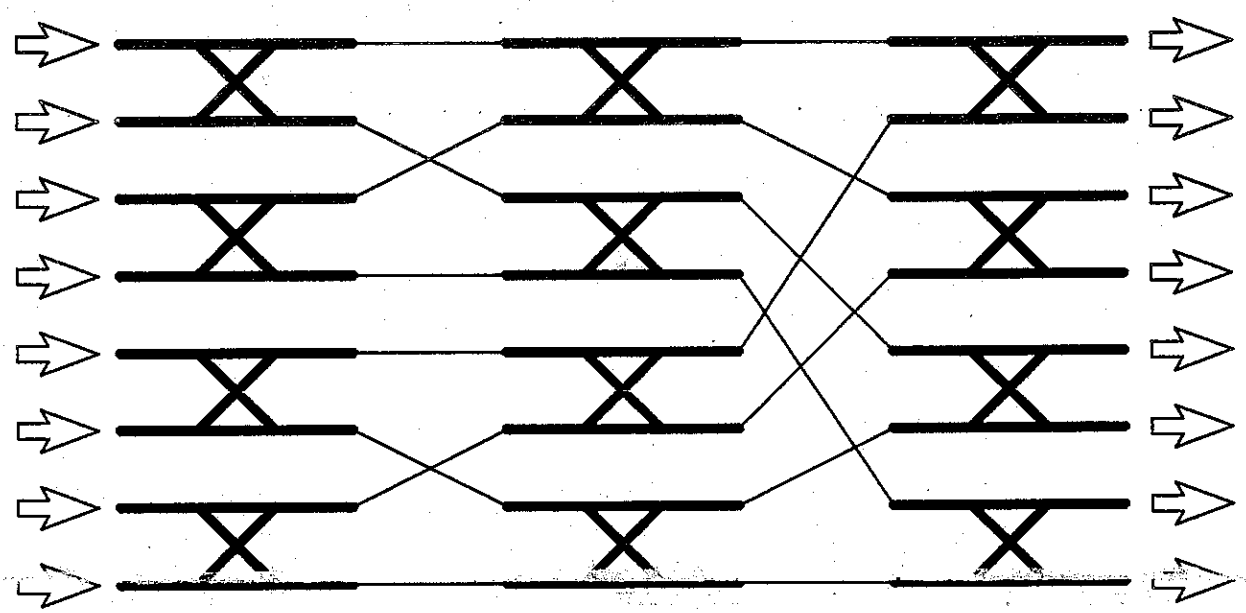


Figure 3–2 is a very small switch illustrating the Butterfly concept. Twelve two–by–two switches, as described above, are wired together to create a network that can connect any of the eight input port wires on the left side to any of the eight output port wires on the right side. The advantage of the Butterfly switch design over a full crossbar is evident even in this small switch; a full crossbar needs 8 squared, or 64, crosspoints to interconnect the same number of ports. The Butterfly switch shown here uses twelve identical units, each with four crosspoints, for a total of only 48 crosspoints. As the number of ports increases, the advantage of the Butterfly switch increases.

3.1.2 Routing in the Butterfly Switch

When a message enters the Butterfly switch, the part of the message that enters first specifies the routing through the switch. This routing header contains one piece of information for each element of the switch through which the message will pass. In Figure 3–2, a message encounters three switch elements before it emerges on one of the wires at the right side. In each element, the message must be routed on either the upper wire or the lower wire. This routing choice

is specified by one bit in the header. That bit is stripped off ("consumed") by the switch element, not passed on as the rest of the header and the rest of the message are. Therefore, following switch elements do not see routing information that was used earlier in the switch. The first information each element sees is the data it needs to route the message within itself. When the message emerges from the last switch element, the entire routing header has been removed.

A Butterfly switch element that switches between more than two wires will use, and consume, more bits of routing information. As we will see, each Butterfly II switch element makes an eight-way routing decision, so it consumes three bits of routing information.

An important consequence of this design is that the format of the routing header depends on the number of switch elements through which any message will pass. This is the number of *columns* in the switch. Therefore, the part of the machine that prepares the message for transmission into the switch must be configured for the number of columns in its switch. In the Butterfly II machine, the requester side of the SIGA does the message preparation, and is configured appropriately.

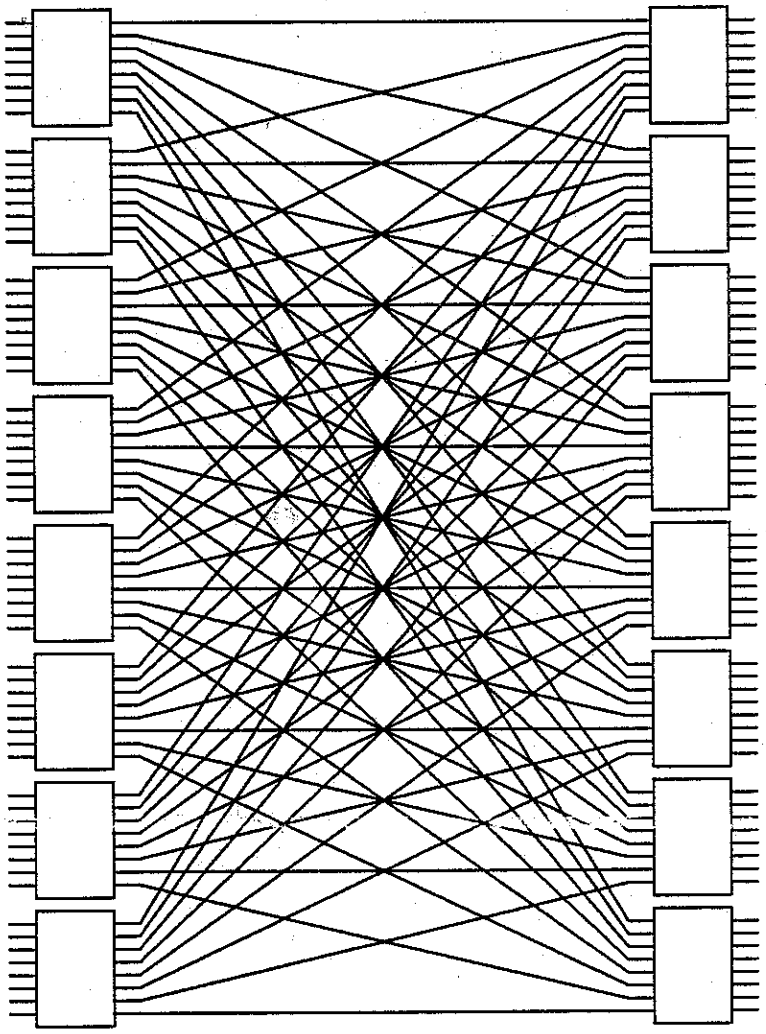
3.1.3 Organization of the Butterfly II Switch

Figure 3–2 above showed three columns of two-by-two crossbars assembled to create a network with eight inputs and eight outputs. Similarly, the Butterfly II switch is built of eight-by-eight crossbar units. Each unit is called a *switch node*. Each switch node makes an eight-way routing decision for incoming messages, so it consumes three bits of routing information. In fact, it consumes some additional information as well, for a total of one byte.

Since each switch node makes an eight-way choice in routing each message, a switch made of these nodes is sometimes called a *modulo-eight* or *base eight* Butterfly switch. Choice of the switch base is a design decision influenced by VLSI chip size, power and pinout restrictions, cabling constraints, failure mode considerations, switch performance, and cost. The initial Butterfly II is implemented with a modulo-eight switch, and has provision for using a modulo-sixteen switch in the future.

Columns of switch nodes are connected to create the switching network of the Butterfly II computer. Each port of the resulting network services one slot, or function board. Two columns of eight-by-eight switch nodes are used for a Butterfly II machine with up to 64 slots, and three columns are required for machines with 65 to 512 slots. For example, Figure 3–3 shows a 64-slot configuration. Two columns of eight eight-by-eight nodes comprise this switch, for a total of 16 nodes. A switch card holds one node, so such a machine has sixteen switch cards.

Figure 3-3 64-slot Butterfly II switch.



At the level described here, switch cards are all the same. In fact, there is a different kind of switch card for each column of a switch. The cards supply different auxiliary signals to function boards, and support different Test and Control System functions. The switch card is the smallest active component of the Butterfly II switch that is normally replaced in the field.

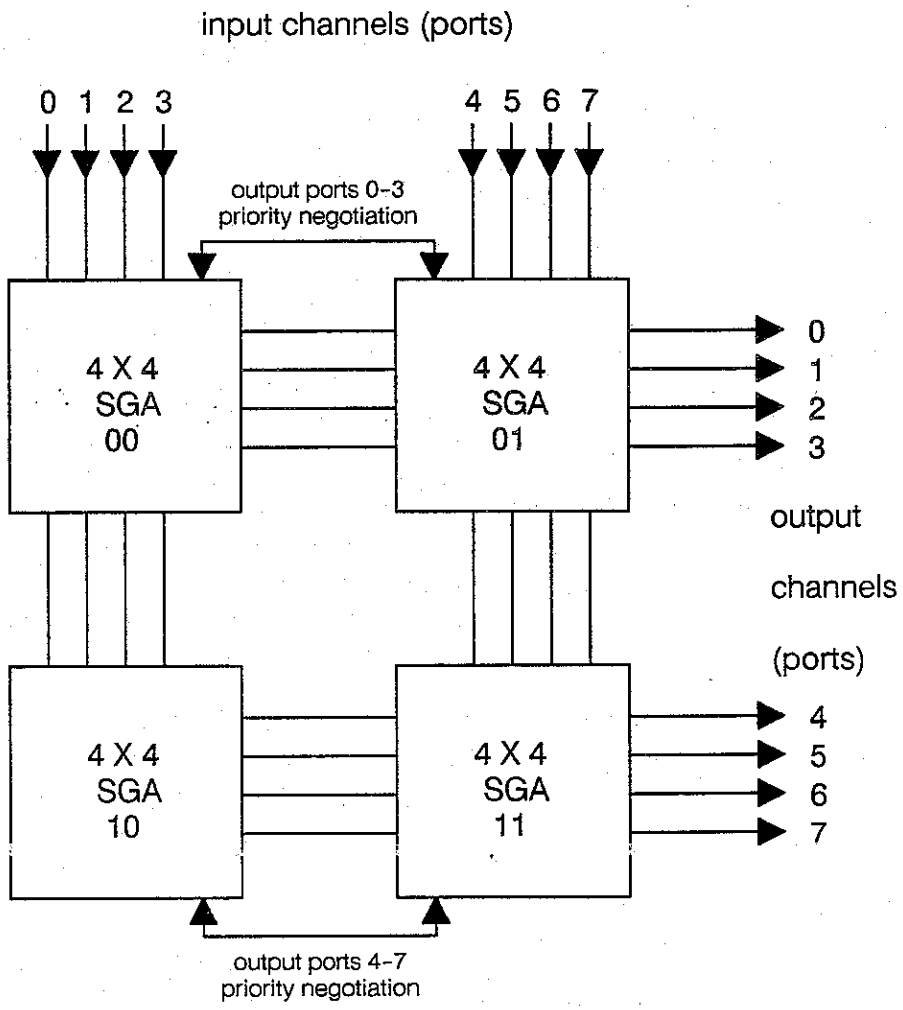
The switch node is implemented with four Switch Gate Array (SGA) chips. In the SGA, four groups of eight data wires and two control signals each are switched among four other groups of ten wires. Each group of ten wires is called a *port* or *channel*. The chip has four input channels and four output channels, and implements a full four-by-four crossbar switching function among these channels. In crossbar terminology, the chip functions as a four

(inputs) by four (outputs) by ten (depth, data and control bits in parallel) crossbar.

SGA chips implement a priority mechanism that is described later, and operate in pairs to negotiate the priority of data handled by each pair. There is close electrical and protocol coupling between the SGA chips in each pair to negotiate the priority of arriving data competing for the same output port. Consequently, the function of a pair of chips might have been implemented on one single chip if constraints such as circuit size and pin count permitted it. Conceptually, the pair of SGA chips form an eight input, four output crossbar.

Pairs of SGA chips are themselves paired, creating an array of four SGA chips that function as an eight by eight (by eight data bits deep) crossbar. It is helpful to think of this, the switch node, as a building block, shown in Figure 3–4. The internal design and behavior of this eight–by–eight block will be discussed in the remainder of this chapter, but the rest of the machine can easily be understood by considering the switch as an assembly of these eight–by–eight crossbar blocks. Within a switch node, each input channel can be connected to at most one output channel, and each output channel to at most one input channel. Consequently, at most eight connections can be in operation at any instant through any one switch node.

Figure 3–4 **Butterfly II eight-by-eight switch unit.**



3.1.4 How Data Flows

In the Butterfly II switch, data wires are not switched individually. Rather, groups of ten wires (eight data and two control) are switched together. The data paths through the switch are therefore eight bits wide. Use of wider data paths achieves higher throughput at the expense of a somewhat more complex hardware design.

Each group of eight data wires carries a byte of data in parallel. Several bytes of information are sent, one after another, in rapid succession. With each cycle of the switch clock, another bit of data appears on each wire, another byte on the group of eight wires.

These bytes sent close together are related and are called a message. A message has a particular structure. The first few bytes of a message specify the switch output port address and thus control the routing of the message through the switch to its destination. Other information needed by the SGA to properly handle the message, such as priority level described below, also appears in these bytes. Next is bytes of data conveyed transparently through the switch from a device on a function board to a (typically different) device on a (typically different) function board. Finally, there is a "checksum" byte to detect errors. The structure of a message is discussed more fully under switch protocol.

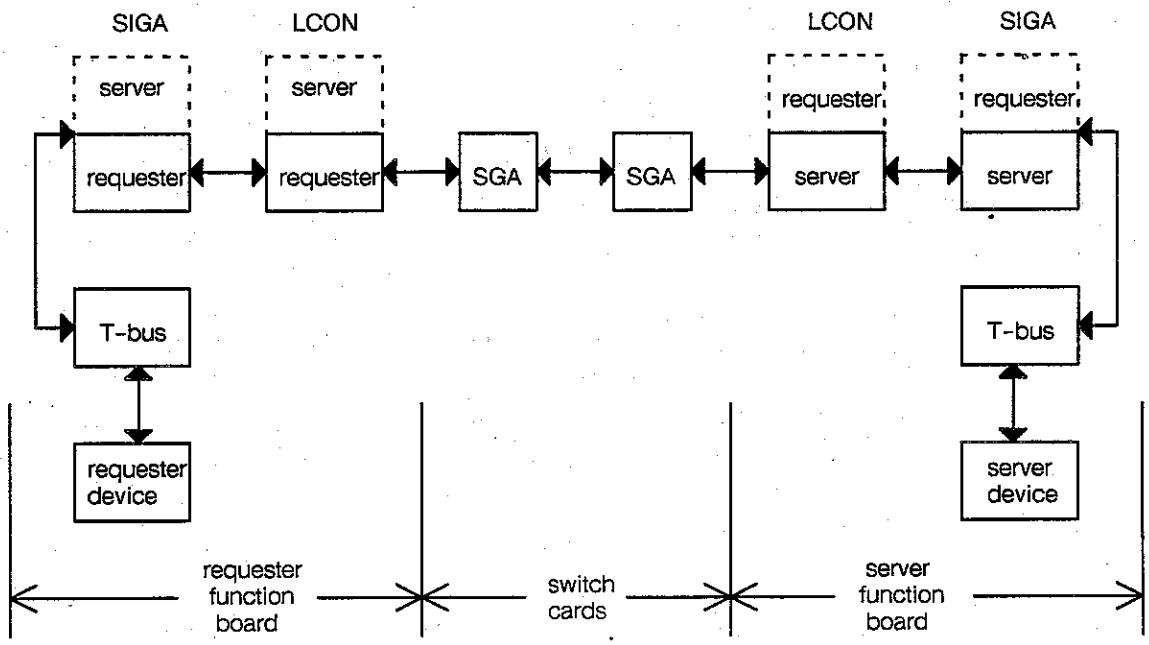
Along with the group of eight data bits are the control signals "frame" and "reverse". These ten signals, propagated through the switch and interpreted by the switch hardware as necessary, form a switch path. So long as that path is in use, we say a connection exists or "is open". The term, "switch path" emphasizes the physical circuitry that handles the signals, while "connection" emphasizes the ability to communicate data. As long as the connection is established, messages may flow on it; a connection may transport from one to several messages before it is closed. The switch hardware imposes no direct limit on the number of messages per connection, and only a loose limit on how long the connection may be kept open if messages are successfully flowing on it. Constraining the length of the connection is left to higher-level protocols, implemented in software and supported by timers in hardware.

The frame and reverse signals control the connection. They define when there is valid data on the data lines, and therefore determine the beginning and the end of the connection and of messages on the connection.

3.1.5 Resources Employed

What hardware is involved in supporting a connection through the switch and in transporting a message on it, and how many other connections and messages can be handled at the same time? This information appears in other sections in a distributed context, and is brought together here and shown in Figure 3–5 for clarity.

Figure 3–5 Resources employed in a connection.



An SGA chip, being a four–by–four crossbar, can support as many as four simultaneous connections. The eight–by–eight switch node, made of four SGA chips, can support as many as eight simultaneous connections.

The requester part of a SIGA can support only one connection at a time, and only one message at a time on that connection. The requester SIGA buffers one message. This is possible because there is a maximum message size (described in a later section), and that limit is small enough that the SIGA can buffer the entire message. The requester SIGA detects contention in the switch and retransmits the current message as described later. Any buffering or queueing of *multiple* messages or connections, however, is performed by the device(s) on the SIGA's T–bus. The server part of a SIGA similarly handles at most one connection and one message on that connection at a time.

Each LCON chip is associated with one SIGA, so the LCON supports at most one requester connection and one server connection at a time.

There can be more than one SIGA on a T–bus, which is typical in Butterfly II function boards. Therefore, if the LCON chips of those SIGAs are connected to switch ports, devices on the T–bus can have as many forward and as many reverse connections simultaneously open as there are SIGAs. Software may further constrain the number of connections. The section on redundancy further discusses multiple SIGAs per T–bus.

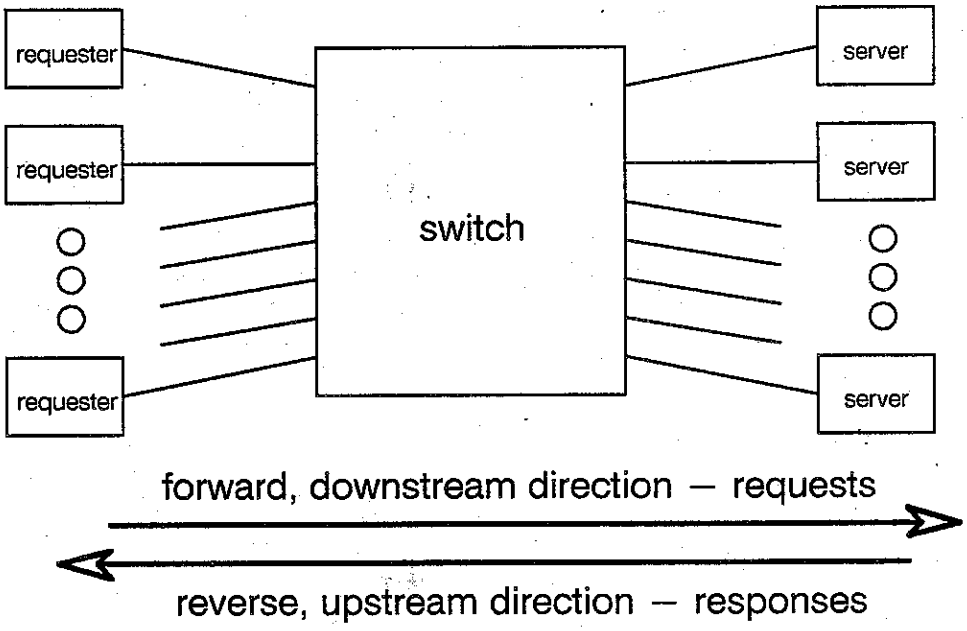
3.2 Bidirectionality

The communication paths provided by the Butterfly II switch can transmit data in either direction, although not at the same instant. At any instant, a given connection is sending data either in one direction or in the other. Thus, although the path is bidirectional, it is half duplex.

An important advantage of bidirectionality is support for atomic operations by locking the connection and holding the switch path. This is discussed further in the section below on locking. Another advantage is that any response(s) to a switch message can be returned along the same path, eliminating delay while a new switch path is set up. It also simplifies the buffering and coordination because the response may be sent immediately. Two disadvantages of bidirectionality are added complexity of the switch hardware and protocol, and possibly increased latency. Greater latency might arise because switch paths might be held longer per connection. The longer connection time is mitigated, however, by reduced contention for connection setup, an advantage in its own right as noted above.

The two ends of a connection through the switch are not identical. One is called the requester, the other the server. Only the requester end can initiate connections. Therefore, the flow of data from a requester through the switch to a server is called the forward direction. When the channel is turned around to send data from server to requester, we say data flows in the reverse direction. The conventional way to diagram the Butterfly II switch is with all the requesters on the left and all the servers on the right, so the forward direction of data flow is left to right as in Figure 3–6. The channels and pins on an SGA chip are named for the forward direction of data flow; an "input" pin is closer to the requester, and an "output" pin is closer to the server, on the data path. When the path is reversed, the chip will receive data on its "output" pins and transmit data on its "input" pins.

Figure 3–6 Bidirectional switch.



When a connection is first set up, data flows in the forward direction, from the requester to the server. The direction of data flow is reversed by a momentary drop in the "frame" control signal that accompanies data through the switch, for only one clock cycle. The frame signal is generated by the requester and always flows in the forward direction, so the requester controls the direction of data flow. When the requester has turned around the connection to enable upstream flow, the server controls whether data is actually flowing. More detail appears in the section on switch protocol.

3.3 Locking

Occasions arise when a sequence of operations should be performed without interference from other activity. One example of this is atomic addition. If multiple processors are attempting to read a location, increment the value and write back the results, an incorrect sum can arise if one processor reads the location between another processor's read and write. More complicated situations arise when complex data structures are involved, or when the updating is more complicated than simple addition. The general solution to this need is locking.

Locking a switch connection is a means of protecting a sequence of operations mediated by switch messages from interference arising from other switch messages. There are two aspects to locking, the device level and the switch level. Locking is primarily a device–to–device mechanism supported by features of the switch operation. The goal of locking is control of access to devices. The

Butterfly II switch supports that control by holding the connection open and by enforcing some rules about T-bus requests.

Two control signals, frame and reverse, accompany each eight-bit data path through the switch. Frame identifies the beginning and end of a connection, and controls the direction of data flow. The start of a message is defined as frame going high after being low for at least two consecutive clock cycles. Once the connection is established, the path through the switch is kept until the requester terminates it by dropping the frame signal for at least two clock cycles (and reverse is low). Thus, by holding the connection, the data path can be locked so that no other requester can connect to that server. While the path is locked, the requester similarly cannot connect to another server.

A device uses a T-bus transaction to ask its requester SIGA to send a message. This transaction contains two bits that specify one of four locking options, called *normal*, *open*, *maintain* and *bypass*. These bits are passed through the switch to the server device, where the majority of locking activity is implemented. The LCON and SGA chips pass the two locking bits through, taking no action on them or because of them. The server SIGA passes the bits through to its T-bus, and also detects an open so it knows the current connection is locked.

A normal request produces a normal switch message without any locking aspects. An open request causes the requester SIGA to hold the frame signal asserted after sending the message and receiving its acknowledgment, in preparation for further switch traffic. A maintain request sends an additional message on the locked connection that is already open. Bypass is used to access a device that ordinarily would not respond because of being locked. A bypass neither establishes a lock nor frees a lock if one exists.

The server SIGA detects the value of the lock bits in the open message, and remembers that a locked sequence is in progress. At the conclusion of the protected sequence of operations, the requester device releases the lock by issuing a special T-bus transaction (FREE-LOCKS). This causes the requester SIGA to drop frame, causing the switch path to drop out. The server SIGA detects the loss of frame during a sequence it knows is locked, and announces FREE-LOCKS on its T-bus. Server T-bus device(s) holding locks sense this and modify their state to reflect the release of the lock. Errors and timeouts also result in the frame signal dropping, and the locks being freed.

The requester SIGA imposes some constraints on T-bus protocol to do locking. In particular, if no lock is currently held, a normal, open or bypass request is accepted, and a maintain request is detected as an error. If a lock is held, a normal request is detected as an error, and open, maintain and bypass requests are accepted. The server SIGA imposes no further constraints. By design of the requester SIGA, the server SIGA will never see the illegal cases of a maintain when there is no lock, or a normal when there is a lock.

So long as the lock is in effect, the frame signal is asserted, so the connection is held open, so no other connections from the requester SIGA or to the server SIGA can be made. This supports the desired isolation of the sequence of operations from interference. The connection is between switch ports, however, and there may be more than one device on the T-bus of a SIGA. Therefore, the requester device may open multiple locks on server devices, if the server devices are all on the same T-bus. The server devices may be physically distinct, or they may be logically distinct components of the same physical device. This multiple lock capability is implemented by allowing open requests even when a locked connection is already established.

Conversely, during a locked connection another device might attempt to generate messages through the requester SIGA. This is not permitted; the requester SIGA detects this and refuses the attempt, regardless of the destination address or the message type.

It is primarily up to the server device to remember that locks are held and to enforce any restrictions on their use, as appropriate to the type of device and its current state. Those considerations are beyond the scope of the discussion here.

3.4 Priority

The Butterfly II switch implements a priority scheme to allow some switch traffic to take precedence over other switch traffic. There are two levels of priority, low and high. Low priority is the default mode, so low may also be viewed as "normal" priority. Priority is a state associated with an output port (channel) of a pair of SGA chips, and also is a property of each message. Each of the four output ports of every SGA chip has a priority independent of all other ports, except that the two SGA chips serving the same output port agree on the priority of that port.

The priority of a switch message is defined by a bit replicated in each byte of the routing header. The value of the bit is set by the requester SIGA as described below when the message is generated, and is not changed as the message travels through the switch. Each SGA through which the message passes strips off (consumes, deletes) one byte of routing header, so the priority bit appears in each such byte.

A signal called "hold" is distributed to every SGA in the machine and controls the operation of the priority mechanism. When hold is not asserted, all messages are treated the same regardless of their priority.

While hold is asserted, each output port remembers whether a message of high priority has attempted, since hold was asserted, to connect through that port. And if so, no low priority message will be allowed to connect through the port. We say that the port has become high priority. An incoming, high priority message makes the port high priority regardless of whether the port is free or the

port is busy and therefore the high priority message is rejected. The port's transition to high priority is caused by any high priority message *attempting* to connect to the port. If the port is busy at the time of the attempt, the existing connection is not aborted, even if it is of low priority. After the port has become high priority, subsequent attempts to connect to it by low priority messages will be rejected, even if the port is idle.

As a high priority message travels through the switch while hold is asserted, it sets every low priority output port it encounters to high priority. Therefore, it leaves a track of high priority ports behind it. Further high priority traffic on the same path will get better service from the switch, because it will contend only with other high priority messages. In particular, if the message itself is rejected, then retransmissions of the message are much more likely to succeed because of the track of high priority ports. It is intended that contention among high priority messages be very rare or non-existent, through careful choice of system configuration parameters by the designers.

Once a port is high priority, it remains so until the machine-wide hold signal is de-asserted twice. The first time hold is de-asserted while a port is high priority, the port circuitry remembers that a drop to low priority is pending. Hold becomes asserted again, and then when hold is de-asserted again, any port with a priority drop pending falls back to normal, low priority. The design requires two de-assertions of hold, instead of just one, to ensure that a blocked, high priority message has at least one hold period to retry and succeed in making its connection.

NOTE

TECHNICAL DETAIL
The de-assertion of hold does not set a pending drop to low priority if a message, of either high or low priority, is using the port. The port must be idle for the de-assertion of hold to set the pending drop. If a drop is pending, a de-assertion of hold drops the port to low priority whether or not a message is using the port.

The intent is that the hold signal normally will be asserted, and will be de-asserted for just one cycle of the switch clock on a periodic basis. This clears out high priority paths that are no longer needed. The next time a high priority path is needed, the next message sent will re-establish a high priority track. In fact, the high priority track is intended to be a very brief phenomenon.

NOTE

TECHNICAL DETAIL
A port's drop to low priority is precipitated by hold being de-asserted for two cycles of the switch clock, not by the de-assertion itself. A single de-assertion lasting for two cycles would serve the same purpose as two separate one-cycle de-assertions, but would not achieve the desired interaction with the priority time slot mechanism described later.

NOTE

TECHNICAL DETAIL
In the initial SGA implementation, if a high priority message arrives at an SGA at exactly the same clock cycle as hold is de-asserted, the port will not be set to high priority. This has subtle implications for switch behavior and is discussed further in the section on analysis of the switch.

There are two ways a message may become high priority. One is by request of the T-bus device that originates the message. This is intended only for testing and diagnostics. The other way is by promotion from low to high priority by the requester SIGA, and is discussed in the following section.

NOTE

HISTORICAL AND TECHNICAL DETAIL
The original design provided for three priority levels, not just two. The highest priority, "express", is like the high level implemented, and its purpose is to set an upper bound on how long a switch message may be delayed before successful transmission. The middle priority, "foreground", was designed to support circuit switching applications such as packetized voice communications. The lowest priority, "background", was all other traffic. A remnant of this early design is that the requester SIGA copies *two* bits (T_PRIORITY<1..0>) from the T-bus into each bid (routing) byte of each message, as bits 5 (P1) and 4 (P0) respectively. The SGA uses only P0, and ignores P1. P0 is zero for high priority and one for low priority. The SIGA forces both P0 and P1 to zero when it promotes the priority of a message to high.

3.5 Message Priority Promotion

When a new connection through the switch is attempted, the initial message will encounter contention at one of the switch nodes if the desired output port on the node is already in use. The partial switch path is released and the requester SIGA is informed of the contention. The message is retried later. This retransmission is repeated if contention persists. The rejection and retry strategy, described in detail in a later section, could lead to long delays before the

message is successfully transmitted through the switch. Once the initial message gets through, the connection is established and, if the connection is held open, further messages can flow without any switch contention delay. The possible delay in getting the first message through, called *switch latency*, could be very long, even theoretically unbounded, unless some mechanism is included to limit it. This is the purpose of high priority; it guarantees a maximum switch latency. Remote memory is accessed through the switch, so the guarantee of maximum switch latency in turn permits an upper bound on remote memory access time.

A message becomes high priority either by the requester device declaring it so in the T-bus transaction initiating the message, or by promotion from low priority. Declaration by the requester device is intended only for maintenance and diagnostic testing. During normal operation, all messages are intended to be low priority when submitted to the T-bus. This is by convention only, and is not enforced by the hardware.

Promotion to high priority is performed by the requester SIGA, and occurs whenever the retransmission of a message is pending during a particular interval called the *priority time slot*. Only the retries are promoted, not the initial transmission. After the header of an initial message is sent, that message is said to be awaiting retransmission. If there is a message awaiting retransmission at any time during the priority time slot, subsequent retries of that message will be at high priority. This is true whether the message was already awaiting retransmission when the priority time slot began, or was first transmitted during the slot and therefore started awaiting after the slot began. Also, promotion occurs whether the actual retry occurs within the slot or after the slot has ended.

Once the message is promoted to high priority, it remains at high priority through any retransmissions; priority is a "sticky" quality. The requester SIGA associates the high priority with that particular message. After the message is successfully transmitted or is discarded due to repeated failure, the promotion is forgotten and subsequent messages start off at low priority as usual. The SIGA ensures that all header bytes of a message contain the same priority value, even if the priority time slot begins or ends while the header is being sent.

Each requester SIGA gets a priority time slot on a periodic basis. Conceptually, the priority time slot is a virtual token passed among all the active SIGA chips in the machine. When a SIGA has the token, messages pending retransmission by that SIGA are promoted and therefore are very likely to get through the switch. The virtual token is not actually passed from SIGA to SIGA, but rather the SIGA chips are initialized so that each knows when its turn comes up. They take turns, round-robin fashion. In fact, the SIGAs can be set up so there are more than one virtual token. Also, each SIGA may be configured to disable its priority promotion.

The priority promotion mechanism does not itself ensure that the switch contains at most one high priority message at a time. Rather, there are ways of using the mechanism that ensure this, and other ways that do not.

The priority time slot in each SIGA is controlled by three values in registers on the chip. These are the slot and the mask fields of the priority time configuration register, and the real time clock. The slot and mask are 16 bits each, and their values are set via the T-bus. The real time clock is 32 bits, ticks each microsecond, and is synchronized among all SIGAs in the machine. Only the low 16 bits (RTC.Lo) of the real time clock are used in determining the priority time slot. The real time clock is discussed in detail in the sections on clocks.

NOTE

TECHNICAL DETAIL
The priority slot and mask, and all other SIGA Configuration Status Unit (CSU) registers, may be set by any master on the SIGA's T-bus. One such master is the Test and Control System (TCS). The TCS is responsible for initializing the machine, including SIGA registers such as the slot and mask.

A SIGA's priority time slot occurs while the RTC.Lo equals the slot value in all bit positions that are zero in the mask. For example, a mask value of zero in all sixteen bits means the priority time slot will be one microsecond long and will occur once every 65,536 microseconds, when the RTC.Lo exactly equals the slot value. Normally, the slot values of different SIGAs are configured to different values, so the priority time slots occur at different times. As another example, a mask value of f00f hexadecimal causes the high four bits and the low four bits to be ignored in comparing RTC.Lo to the slot value, so the slot would last 16 microseconds and would occur once every 4,096 microseconds. There are only 256 16-microsecond intervals in a period of 4,096 microseconds, so if a Butterfly II computer with 512 SIGAs were configured with these parameters, two SIGAs would share each priority time slot.

The duration of the priority time slot is set to any of a wide range of values, as seen in the paragraph above. The intent is that the slot duration be set the same for all SIGAs in the machine, although there is no hardware requirement that they be the same.

Not all instants of time need belong to the priority time slot of some SIGA. The parameters can be set up so there is an idle time after each SIGA's priority time slot. This could be a good way to configure the switch, because it allows a grace period for a promoted message to succeed through the switch before another SIGA might inject its own promoted message. Without a grace period, contention among high priority messages, though rare, could compromise the bound on switch latency.

Because the slot duration is how long a SIGA will hold the "virtual token" that lets it promote the priority of messages, its effect interacts with the period of

the machine-wide *hold* signal. As a very rough approximation, the two intervals should be about the same, because [THE FOLLOWING OVERSTRUCK STUFF IS NOT RIGHT — WHAT IS CORRECT??? PERHAPS THE LARGER GOAL OF BOUNDING SWITCH LATENCY.] ~~the intent of the priority promotion and hold mechanism is to make very likely the success of a message sent into the switch during the priority time slot. If the hold signal had a significantly longer period, subsequent messages on the same path would receive an unintended advantage, riding the track of high-priority SGA ports set up during the past priority time slot.~~ If the hold signal had a much shorter period, a message of promoted priority would have undue difficulty getting through a crowd of contending, low priority messages. Each time the promoted message sallied forth into the switch, it would get part way, set those ports it touched to high priority, hit a port already in use, back off and try again. But its retry would usually fare no better, because the work it had done in setting ports to high priority would be undone by the rapid period of the hold signal.

So the hold signal and the priority time slot will have roughly equal times, and further consideration gives more guidance on their relative values. If the SGA port priority were reset to low whenever the hold signal dropped for one clock cycle, then the behavior of promoted messages would vary a lot. Messages that got promoted just before the hold signal dipped would not get much benefit from being promoted, and messages promoted just after a hold signal dip would have an entire hold period to enjoy the benefits of high priority. This behavior is smoothed out by having SGA ports drop their priority to low after *two* dips in the hold signal. Now a message of promoted priority has at least one hold period, and at most two hold periods, to exercise its high priority. Therefore, the interval from the start of one priority time slot to the next is intended to be about the same as two periods of the hold signal.

NOTE

TECHNICAL DETAIL

The "benefits of high priority" gained by using two dips in hold arise when a high priority message is rejected because a required output port is already busy with another message. The port gets set to high priority, and the high priority message will be retransmitted soon. But, if a single dip in hold could drop the port to low priority, that dip might occur before the retransmission, wiping out the progress the message had made. Requiring two dips in hold protects that progress long enough for the retransmission to make use of the high priority port. If a single dip in hold does occur, so the port has a drop pending, the arrival of the retransmission will erase the pending drop, setting the port back to solid high priority.

3.6 Traffic Injection Pacing

The LCON and SGA chips can accept the start of a switch message at any cycle of the switch clock. Extremely heavy input of messages from the function boards, however, could cause congestion in the switch, degrading performance. It is useful to have a switch architecture that can be configured to pace, or throttle, the entry of traffic into the switch. The Butterfly II switch provides this capability through slotting.

The requester SIGA has three strategies for pacing the initial transmission of messages into the switch: immediate, random and slotted. If a message is rejected, the SIGA retries the message using either the slotted or the random strategy. Immediate pacing means no delay; transmission occurs as soon as the SIGA has sufficient data to begin transmission of the message into the switch. Random pacing invokes a delay of a random number of switch clock cycles before transmission or retry. Slotted pacing restricts transmission of messages into the switch to certain instants of time, called slots. The slots occur every ½, 1, 2 or 4 microseconds.

The Test and Control System initializes registers in each SIGA to define the parameters for random and slotted strategies. Each T-bus request for message transmission selects a strategy for that specific message from among those set up earlier (normally, by the TCS).

The operation of the random and slotted strategies is the same for retransmissions of a message as it is for the initial transmission, so the details of these strategies are described in the following section. The immediate strategy is available only for the initial transmission of a message.

3.7 Contention and Retry Strategies

This section examines the switch strategy for dealing with contention among messages in the switch. The basic strategy is to back up and try again. Contention arises when a needed port is already in use, or when two messages meet head-on at the same port.

3.7.1 Port Is Already In Use

When a message is working its way through the switch, it may encounter an output port that is already in use. When this happens, the SGA containing that port asserts the "reverse" signal going back upstream toward the requester, for just one cycle of the switch clock. This pulse of the reverse signal for one clock cycle is called a *reject*, and causes the switch path to be torn down. Each upstream SGA in turn detects a reject, drops the frame signal it had been asserting downstream on that port, sends a reject upstream, and frees up the input and output ports for other work. This reject mechanism is also invoked

by other conflicts detected by the switch hardware, such as the port being at high priority and therefore unable to service an arriving low priority message. The server SIGA also generates rejects, as described in the protocol section below.

The message rejection travels upstream to the requester SIGA. There, the message is retransmitted into the switch at a later time. Often, the contention the message suffered the first time has cleared, and the message succeeds on the second try. If contention arises again, the message is retransmitted again, and so forth. Normally, if contention persists, the requester SIGA's priority time slot will eventually arrive. Then the message will be promoted to high priority, and will get through. If priority promotion is disabled, or if the switch is broken, retransmission will continue until a timeout occurs, signaling a transmission failure, and the requester SIGA will notify the requesting device on its T-bus.

3.7.2 Head-on Collision at the Port

Two messages can arrive at (bid for) one output port on the same clock tick. When this happens, arbitration circuitry in the SGA selects one message to acquire the port and one message to be rejected. The message that succeeds proceeds as if there had been no contention at the port, and the rejected message is treated as if the port had already been in use as described above. In fact, up to eight messages may simultaneously contend for a single output port, since the Butterfly II switch unit is an eight-by-eight crossbar. Note that each output port has its own arbitration logic; the arbitration on one port is not affected by bids, priority, or use of another output port.

The SGA arbiter's algorithm is as follows. If the output port is at high priority, then any low-priority bids are rejected. If the output port is at low priority, then all bits compete, even if some are high and some low priority. Of the remaining bids, one is chosen at random to get the output port, and any others are rejected.

NOTE

TECHNICAL DETAIL
The random choice is driven by three bits supplied to the SGAs by a pseudo-random number generator on the switch card. A 7-bit maximum sequence generator ($x^7 + x^3 + 1$), updated on every switch clock cycle, is used. The Test and Control System can set the generator to all ones (the dead state) or all zeros for testing, and is responsible for ensuring that the generator is not in the dead state after system startup.

The original implementation produces a slight unfairness among competing bids. This is discussed further in the section on analysis, but is not expected to affect performance in normal use, and may not be detectable even by special purpose software.

3.7.3 Retransmission Pacing

Just as pacing the initial transmission of messages into the switch is important for reducing contention, so is pacing of retransmissions. In fact, it can be even more important, because whenever contention happens to occur, a bad retransmission strategy would exacerbate the contention by swamping the switch with traffic.

The SIGA has two retransmission strategies, random and slotted. The parameters for each of these reside in registers in the SIGA, and are set via the T-bus (normally, by the Test and Control System during system startup). There are two sets of registers for each strategy, so actually four different strategies are available: two random strategies and two slotted strategies. A fifth strategy, immediate transmission, is available only to the initial transmission of messages, not to retransmissions. The random and slotted strategies are available to both the initial transmission (start) and retransmissions (retries), so they are discussed below without saying which transmission is happening.

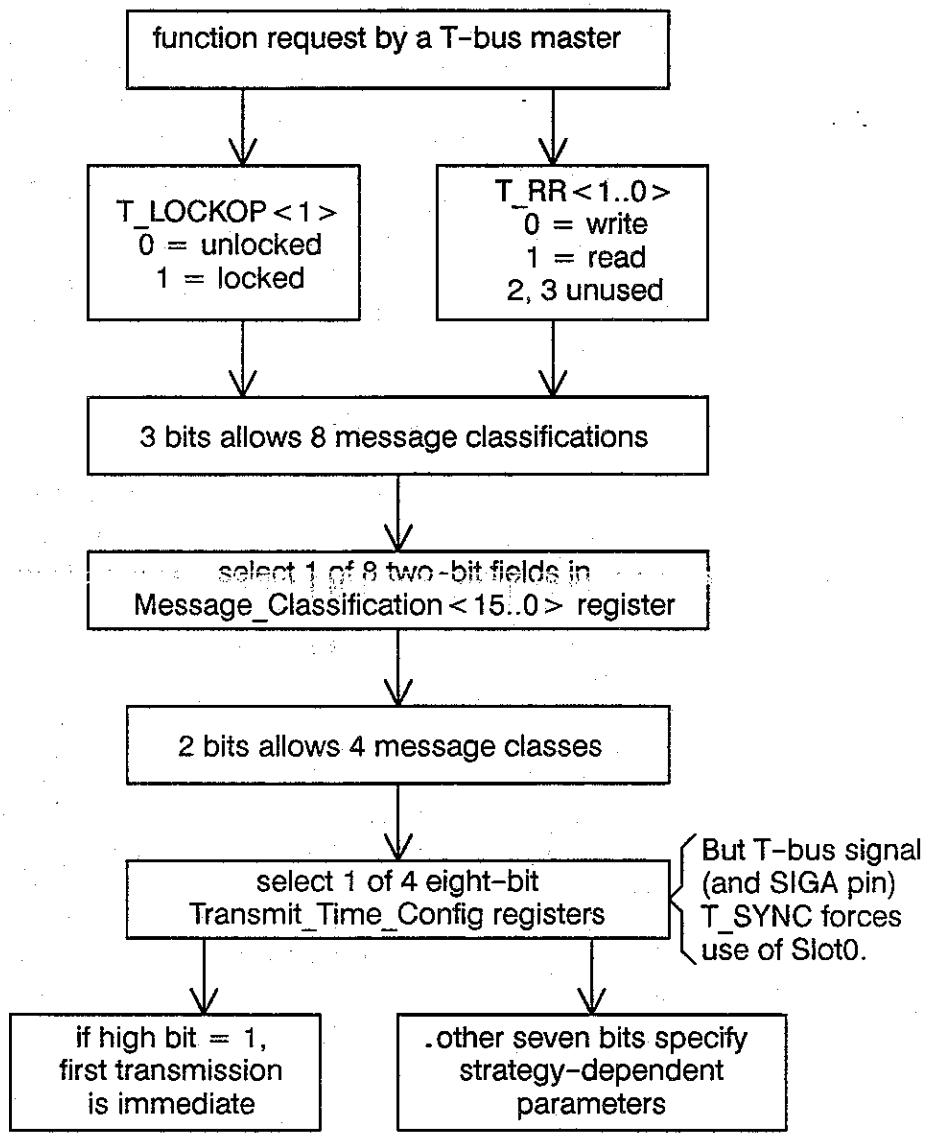
3.7.4 Strategy Selection

When a T-bus device requests transmission of a message, the request contains information about the nature of the message. Three of the distinctions are write or read, unlocked or locked, and normal or auxiliary. ("Auxiliary" is not used in the first released version of the hardware, but is available for possible future expansion.) Three bits in the T-bus request specify these distinctions, and together they define eight possible kinds of message. The SIGA uses these three bits to extract one of eight two-bit fields from a 16-bit message classification register that is initialized by the Test and Control System. Thus, the SIGA maps eight kinds of T-bus message request into a 2-bit value, coding for four *message classes*. Based on the message class, the SIGA selects one of four start/retry strategies to govern the initial transmission and any retransmissions of the message. The first two strategies are random, and differ only in their parameters of randomness; the other two strategies are slotted, and differ only in their slotting parameters. Each strategy has an 8-bit register, initialized by the Test and Control System, holding its parameters. The SIGA locates the appropriate register and applies the chosen strategy with the parameters given.

In each of the four registers, the leftmost bit, if a one, forces the initial transmission to be immediate rather than random or slotted. This is how the immediate transmission strategy is invoked. If transmission is immediate, the remaining parameter bits are ignored for the initial transmission. On retransmissions, the immediate bit is ignored, the retransmission strategy is either random or slotted, and the seven remaining parameter bits are used. The meaning of the remaining parameter bits is different for a random strategy

than for a slotted strategy. Each of these is described below. Figure 3–7 illustrates the overall selection of a strategy and parameters. The selection of strategy can be overridden by assertion of the signal T_SYNC, which forces use of the "Slot0" strategy. This signal is asserted only by the CPU interface, and therefore only when the access arises from the CPU, under control of the Process Configuration register.

Figure 3–7 Strategy selection for message transmission.



3.7.5 Random Strategy

To understand the details of the random strategy, it helps to keep in mind the general intent. The random strategy implements a version of "binary exponen-

tial backoff" message transmission algorithm. This algorithm says that before each (re)transmission of a message, a delay will be imposed. The amount of this delay is doubled after each (re)transmission. For example, suppose you are calling a friend on the telephone. You get a busy signal, so you wait one minute and try again. You get a busy signal again, so you wait twice as long — two minutes — and call again. Still busy! Wait four minutes and try again. Your retries are getting less and less frequent (backing off), at an exponential rate by doubling (base two; binary) each try. Backoff is a common mechanism in communication networks, where it alleviates congestion. Binary exponential backoff is a frequently used backoff algorithm, because it behaves well and is easy to implement. The Butterfly II switch implements a modified form of this algorithm. In the telephone analogy, the modification is to wait a *random* amount of time: up to one minute before the first retry, up to two minutes after the first and before the second retry, up to four minutes between the second and third, and so on.

The binary exponential backoff example above has no random component. The random strategy in the Butterfly II switch employs randomness by selecting a uniformly distributed, random delay between zero and the strict binary exponential value. The random component helps to stagger retransmissions that otherwise might continue to collide with retransmissions of other traffic. Traffic that is coincidentally clumped gets de-synchronized, so the average level of usage describes the switch statistics well.

Before (re)transmitting a message, the random strategy performs a delay by decrementing a 12-bit counter at the switch clock frequency. When this backoff counter underflows to -1, the message is sent into the switch. When a random strategy delay is begun, the counter is loaded with the bitwise AND of a 12-bit pseudo-random number and a 12-bit backoff mask.

NOTE

TECHNICAL DETAIL
The 12-bit pseudo-random number is twelve bits of an on-chip, 15-bit maximum sequence generator ($x15+x4+1$), continuously updated at the switch clock frequency, and reset via the T-bus. The Test and Control System is responsible for resetting the generator during system startup, which ensures the generator is not in its dead state (all ones). The two random strategies share a single generator.

The backoff mask is computed by a process controlled by the remaining seven bits of the random strategy control register. The mask is derived from the value in a 6-bit Johnson counter (see note below). Before the first transmission of the message, the Johnson counter is initialized from five bits of the strategy control register with a zero prefixed. The Johnson counter is advanced each time a 3-bit accumulator overflows. This accumulator is cleared to zero when the requester SIGA is idle, that is, before the requester SIGA services a T-bus request to open a connection. The accumulator is incremented by the value

of the remaining two bits of the strategy control register, after each switch reject.

In operation, each time the current message is rejected, the accumulator is incremented. If the accumulator does not overflow, then the backoff mask will be unchanged for the next retransmission. If the accumulator overflows, the Johnson counter is advanced, and a new backoff mask is derived by decoding the Johnson counter. The values this decoding can produce, and therefore the possible values of the backoff mask, are zero or more "0" bits followed by one or more "1" bits. The successive Johnson counter values produce successive backoff masks that are in effect shifted left, allowing more and more bits of the 12-bit random number to be transferred into the backoff counter.

NOTE

TECHNICAL DETAIL
A *Johnson counter* can take on only values that are zero bits followed by one bits, or vice versa. It is easily implemented in hardware, because stepping the counter is a one-bit shift, using the complement of the bit shifted out as the bit to shift into the other end. An N-bit Johnson counter has 2N states, so the 6-bit Johnson counter used in the random strategy has 12 states. These are decoded to give 12 different backoff mask values, ranging from a single one bit to twelve one bits. Note that the backoff mask can never be all zeros, so the random number generator always has some effect.

If the Johnson counter is advanced several times, it will "wrap around" from 100000 to 000000, and continue from there. The effect on the behavior of the strategy is that if there have been many rejects, the average backoff gets longer and longer, and finally jumps back to the minimum, from which it again grows.

WARNING TO SYSTEM PROGRAMMERS
The Johnson counter must be initialized to a legal value. For example, legal values of the 5-bit field are 00011 or 01111. Values 00100 or 11110, for example, are illegal. The latter becomes 011110 in the counter.

Figure 3-8 illustrates the random strategy implementation, and Figure 3-9 shows the encoding of the Johnson counter and backoff mask.

Figure 3–8 Random strategy for message transmission.

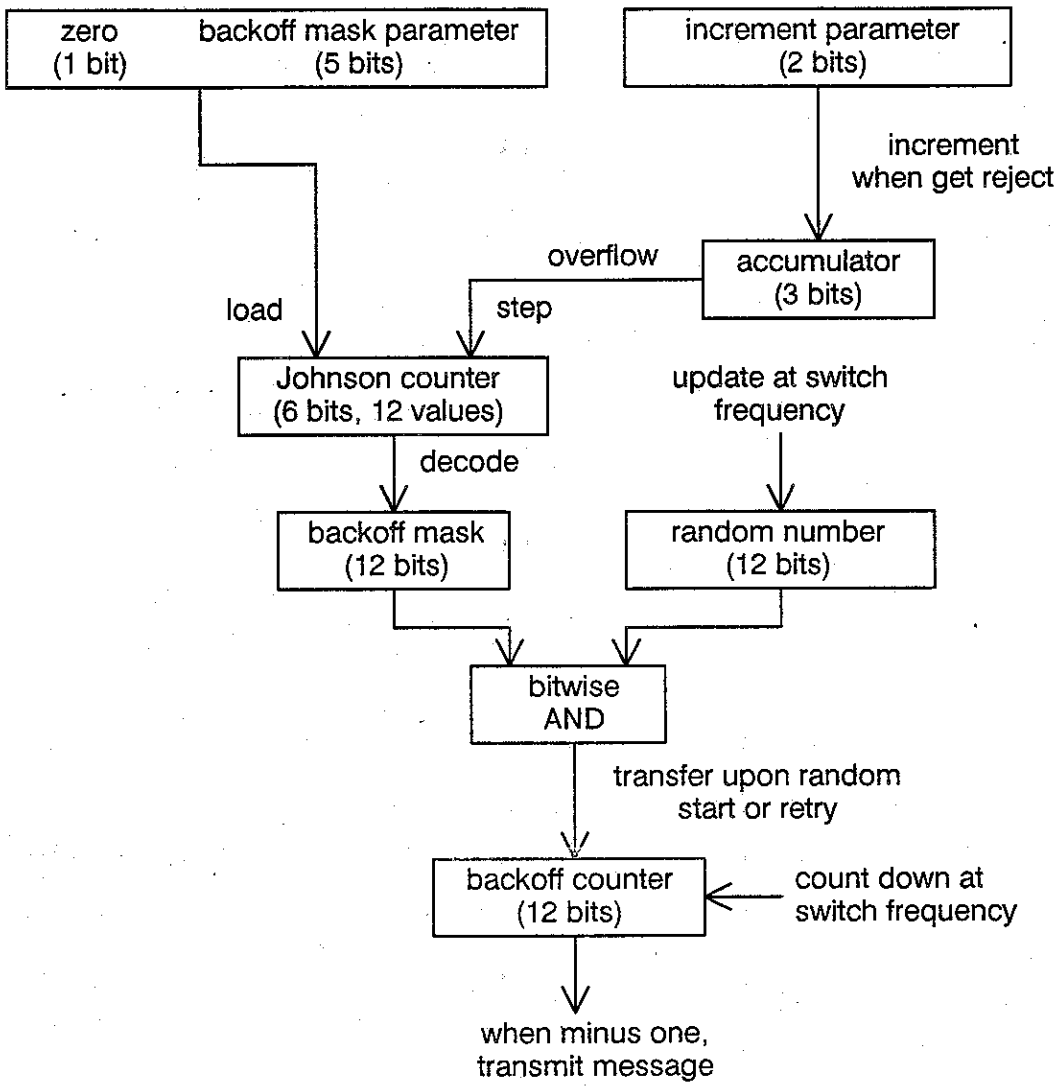
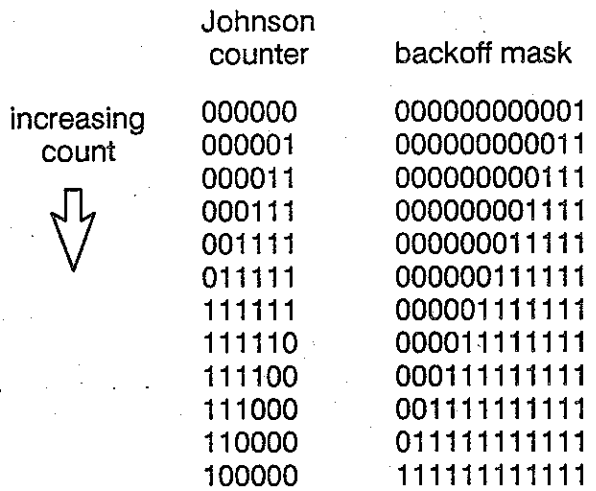


Figure 3–9 **Random start and retry mask encoding.**



3.7.6 Slotted Strategy

The idea behind the slotted strategy is a sequence of periodic time slots. The next time a slot arrives, the message is transmitted or retransmitted. In this strategy, there is no dependence on which transmission is being made — initial, first retry, second retry, or whatever — except for the immediate mode on the initial transmission.

The slotted strategy has two parameters, the slot period and the phase. The phase parameter permits staggering of message injection among different SIGAs in the machine. The period and phase are specified by a 2-bit field and a 5-bit field, respectively, in the slotted strategy control register. As described above, the eighth bit of this register specifies immediate transmission the first time the message is sent, and overrides waiting for the slot.

The arrival of the slot is detected by comparing time to a given value. The slot period parameter controls how many bits are compared, and the slot phase parameter specifies what value time must have in those bits. The time used in this comparison is an 8-bit value described below. The 2-bit slot period parameter specifies that all eight, or the lower seven, six or five bits will be compared, corresponding to periods of four, two, one, and one half microsecond. The 5-bit slot phase parameter is the value that the low five bits of time are compared against. If the slot period parameter specifies more than five bits of comparison, the high bits of time are compared against zero.

The eight bits of time used in the comparison come from two registers in the SIGA. The high two bits of the eight are the lowest two bits of the real time clock, which ticks every microsecond. The low six bits used in the comparison come from the six bits of the real time prescaler, a counter used to maintain

the real time clock. Both the clock and the prescaler are described in the section on clock distribution.

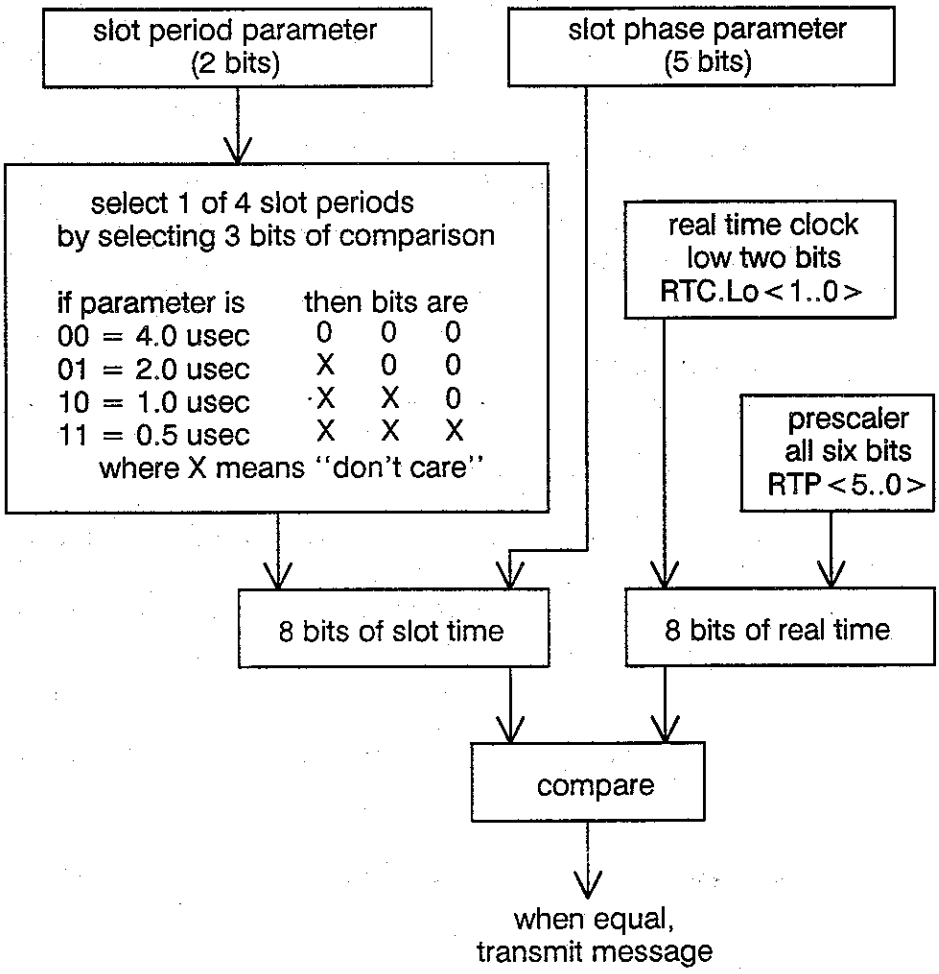
NOTE

WARNING TO SYSTEM PROGRAMMERS
The real time prescaler does not necessarily take on all possible values. The range of values it takes on are configuration dependent, and are controlled by a register in the SIGA (normally initialized by the Test and Control System at system startup). *The comparison value specified by the phase parameter for the slotted strategy must be a value that the prescaler will take on.* If the comparison value specified is one the prescaler does *not* encounter, a message using the slotted strategy might *never* be transmitted. The prescaler is described in the Real Time Clock section below.

TECHNICAL DETAIL
The real time clock is synchronized across all SIGAs (on the same switch). If a slot period greater than 0.5 microsecond is used, periods during which slots arrive for all SIGAs will alternate with periods during which no slots arrive, because the high bits of time must be zero. For example, a two microsecond period results in a 25% duty cycle.

Figure 3–10 illustrates the slotted strategy implementation.

Figure 3–10 Slotted strategy for message transmission.



▶ *Merge in this text from the TC/FPV chapter:*
The synchronized access mechanism can be used when accessing shared data for which the programmer expects there may be contention, such as a mutual exclusion lock. Attempts to obtain the lock can be made with the synchronized access *bit of the PCR asserted, signaling the SIGA that the access should be made at a reduced rate. This prevents the switch and destination memory module from being flooded with access attempts when the lock is not free and multiple processors are waiting for it to be freed. [Say something about why it's called "synchronized access", and about Slot0 (and slotted in general) being sort of a low–priority class. As Guy says, the synchronized access thing is a way for the processor to communicate its desire for slotted access tuned for spin–locking. Called "synchronized" because all contenders jump in at once, at slot time.]*

3.8 Redundancy

The Butterfly II switch design provides redundancy through duplication. Any T-bus may have multiple SIGAs wired to it. The Butterfly II design takes advantage of this capability by placing two SIGAs on the T-bus of typical function boards. Each SIGA is attached to a separate LCON, and each of the two LCON chips attaches to a different switching network. Thus, two entire switches are available, each with its own switch cards, and with its own LCON and SIGA on every T-bus. The request on a T-bus selects which SIGA is used, and thus which switch is used for a message. The intent is that if one switch fails, the second remains available.

Normally, the TCS will enable all the switch-related hardware, in one or both switches. The hardware places no constraint on whether one or both switch systems are enabled. If both are enabled, the operating system and user software select which switch to use, by addressing the corresponding SIGA. If some components fail, the TCS will disable them and any other components whose proper operation depends on the disabled ones. Also, in a Butterfly II without the second switch implemented, the SIGAs and LCONs associated with the absent switch would be disabled or omitted entirely.

Separate clock hardware supplies separate clock signals to the switch cards in the two separate switches. If the clock distribution system in one fails, the other continues to work. The two systems of switch and clock are named "A" and "B". Each of the two systems connect to *both* TCS masters, so any combination of switch systems and TCS masters suffices to support operation of the machine.

NOTE

TECHNICAL DETAIL
There is no hardware constraint preventing the connection of SIGA-LCONs on the same T-bus to the same switch network. It is merely convention and design intent that they be connected to separate switches.

3.9 Clock Distribution

This section describes the distribution of the switch clock signal. The Butterfly II has other clock signals, and the following section on independent clocks describes those.

As described in the previous section on redundancy, a Butterfly II may have two completely independent switches. Each switch has its own clock signal, and the two signals are NOT synchronized. The comments in the rest of this section apply to a single switch. The reader should keep in mind that, if the

machine is configured with a redundant switch, there is an entirely separate clock distribution system operating in parallel with the one described.

The entire switch operates synchronously from a single clock source, within the limits of distribution skew. Skew is kept low by design, but the propagation delays over wires between sections of the machine can be substantial. Therefore, data and connection control signals are reclocked (resynchronized to the switch clock) where necessary, and different phases of clock can be used in different columns of the switch.

3.9.1 Clock Master Card

The clock signal is generated by a *clock master card*. The distribution of the clock signals is from the clock master card, through clock slave cards if necessary, to switch cards where it is used and also passed on to function boards.

The clock master card generates different clock signals for the different columns in the switch. The phasing of these separate clocks can be selected to match the clock phasing to the cable lengths in the particular machine. For example, signal delay in 12 feet of cable between two switch columns is about half a bit period at a clock frequency of 40 megahertz, so a 180 degree phase difference in clock signals will compensate for that propagation delay.

NOTE

TECHNICAL DETAIL
In the initial Butterfly II production run, there are only two switch columns, not three. These are called the requester column and the server column. The clock master card for this configuration generates two clock signals, R_CLK and S_CLK, for the two columns respectively. R_CLK and S_CLK may be in phase or may be 180 degrees out of phase, as selected by a jumper on the clock master card.

The clock master card has a fanout of eight cards per level. The master card alone suffices for a machine with 64 or fewer function boards. For larger configurations, the clock signal is fanned out by a set of clock slave cards.

3.9.2 Switch Card

In the switch card, the clock signal is buffered by three drivers. One driver supplies clock to two SGAs and to four additional drivers, each of which send clock to four function boards, if the card is in the first or last column of the switch. The second of the three drivers supplies clock to the other two SGAs and to four more drivers for the other four function boards. The third driver supplies support circuits on the switch card. If the switch card is in an internal

column of the switch, it connects only to other switch cards, not to any function boards.

Each of the eight function boards served by the switch card is served by a separate clock driver on the switch card. This isolates the function board clock signals from each other and from the switch card's internal circuits, so failure of a function board will not corrupt the clock signal elsewhere. Similar isolation of switch cards from each other is provided by separate drivers on the clock master (and slave) card.

3.9.3 Function Boards

In the function board, the clock signal is received in duplicate by the LCON. One section of the LCON connects to the requester side of the switch, and another section connects to the server side. The LCON receives a clock signal from each side of the switch, and the phase relationship of these two is unspecified. The two sections of the LCON operate largely independently.

Each section of the LCON redrives its clock signal out to the SIGA, and also uses it internally to reclock frame, reverse and eight data lines. Reclocking removes skew that may have accumulated as these signals traveled to the LCON. The frame and reverse are unidirectional signals, but the data lines are bidirectional, requiring a more complicated reclocking circuit that uses different edges of the clock depending on the direction of the data. The LCON also reclocks the 65-millisecond pulse received from the requester side of the switch.

The LCON also supplies the clock from the requester side of the switch on a tri-statable output pin. This pin can be wire-OR'ed with the corresponding pin on other LCONs, and frequency divided to provide a main clock for the function board. The Test and Control System would enable this pin on one of the LCONs, leaving the others tri-stated. The enabling of this pin is independent of enabling the LCON's data passing functions, so the two switches of a machine can be tested without disrupting the function board's clock. Alternatively, the function board may supply its own clock, as discussed later.

From the generation of the clock signal on the clock master card, through the input to the LCON, the clock signal is transmitted as a differential signal on two wires. This increases noise immunity and isolates the clock signal from ground noise. The actual signal names reflect this, such as "clock plus" and "clock minus". When the clock signal is used in logic, and on the SIGA side of the LCON, a single wire is used.

The SIGA receives both the requester clock and the server clock; as in the LCON, the SIGA has relatively separate sections to deal with the requester and the server ends of connections. Each section of the SIGA uses its clock signal to process the transmission and receipt of messages. Also, the SIGA uses the requester clock signal to maintain an on-chip real time clock (RTC).

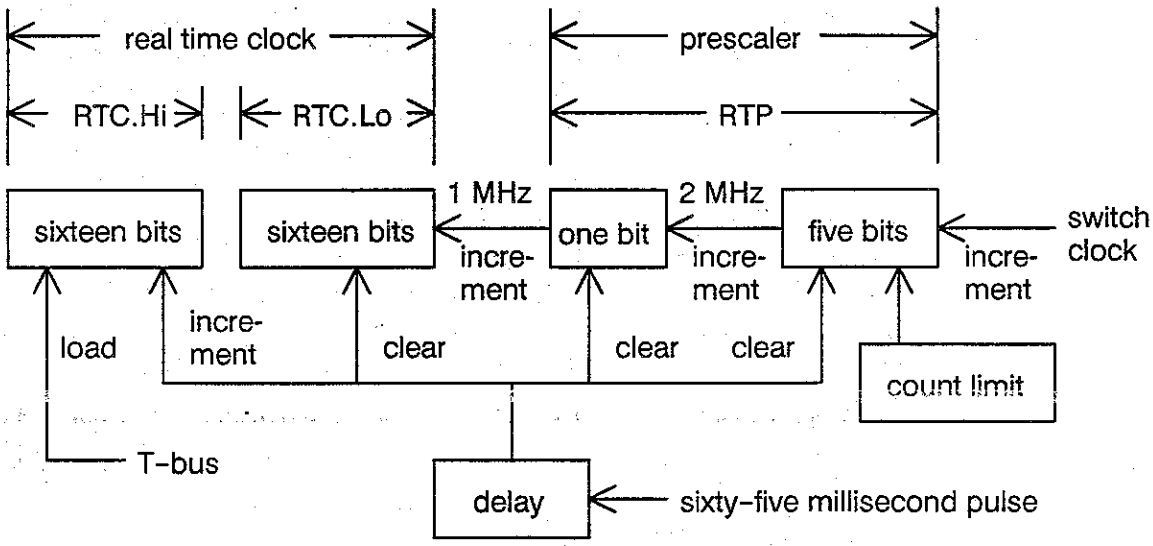
The RTC is available to devices on the T-bus to indicate time, and also is used internally by the SIGA in the slotted message transmission strategy, in an interrupt timer, and to drive switch protocol timers.

3.9.4 Real Time Clock

RTC Implementation

Figure 3-11 shows the implementation of the Butterfly II real time clock.

Figure 3-11 Real time clock implementation.



The real time clock (RTC) is implemented in the SIGA chip and has a number of components. To the T-bus, the RTC is a 32-bit register with a one-microsecond tick rate. To increment this register, the SIGA derives a one megahertz signal by dividing down the requester switch clock signal. The switch clock is nominally 40 megahertz, but the entire switch design accommodates a clock rate somewhat faster or slower than this. The switch clock frequency may vary from one application to another, so the RTC *prescaler* that divides the switch clock down to one megahertz is programmable. The Test and Control System initializes the RTC prescaler control register in each SIGA when the system is brought up.

The RTC prescaler consists a 5-bit programmable counter that feeds a 1-bit divide-by-two circuit, for a total of six bits. These six bits, and two bits of the RTC, are used within the SIGA in the slotted strategy for message transmission, as well as to produce the one-megahertz clock signal.

NOTE

TECHNICAL DETAIL

The programmable counter that is the low five bits of the RTC prescaler starts at zero and counts up at the switch clock frequency to a value given by a control register set via the T-bus (normally by the Test and Control System). When the counter reaches this value, the counter is reset to zero and the divide-by-two is incremented. The output of the divide-by-two is the one-microsecond signal. The switch clock frequency must be a rate that the prescaler can divide down to get exactly one megahertz — namely, *the switch clock frequency must be an even number of megahertz from 2 to 64.*

A small Butterfly II machine may have very short switch cables and thus little signal delay and skew, permitting a faster switch clock rate and higher performance. The initial SIGA implementation can run up to about 45 megahertz, and the initial SGA up to about 60 megahertz.

Sixty-five Millisecond Pulse

The *sixty-five millisecond pulse* is a signal generated by the clock master card and distributed to each SIGA in the switch. It is asserted for one switch clock period once every 65,536 microseconds, and is used to keep the RTCs synchronized. When the pulse occurs, it clears to zero all six bits of the RTC prescaler, clears to zero the low sixteen bits of the RTC, and increments by one the value in the high sixteen bits of the RTC. Thus, the low and high halves of the RTC are not directly connected by propagation of a carry signal, but depend on the sixty-five millisecond pulse to couple their values. Each SIGA contains a small, programmable delay between the actual receipt of the pulse and the actions it causes on the RTC. Any T-bus master (typically the Test and Control System) can set this delay to zero (no delay), one, two, or three switch clock periods.

NOTE

TECHNICAL DETAIL
The programmable delay in use of the sixty-five millisecond pulse is intended for compatibility with future versions of the SIGA and LCON. In particular, when the slotted strategy is used for message transmission, the arrival of a slot begins the process of transmitting a message into the switch. The number of switch clock cycles required before the message is actually presented to the first column of the switch depends on the details of the processing in the SIGA and LCON, and is not specified as a design parameter of the overall switch. Future SIGA or LCON implementations might take a different number of switch clock cycles to process the message through this "pipeline". If so, a Butterfly II machine with some early style SIGAs and LCONs and some of later design would give unfair advantage to messages from whichever function board had the shorter pipeline. The programmable delay permits equalization of pipelines in that situation.

Reading the RTC

Devices on the T-bus may read the RTC as a 32-bit register by using a word read. The low or high half of the RTC may be read using a half-word read. If the entire 32 bits are needed, a word read must be used, because two half-word reads will yield incorrect results if the RTC happens to tick between the two reads. The hardware ensures that a single read, either word or half-word, will not sample the RTC while it is changing. This is important, because the T-bus clock is not necessarily synchronized with the switch clock that drives the RTC, so a race hazard would exist otherwise. The configuration status unit, a part of the SIGA, provides this guarantee. Due to the way this guarantee is achieved, explained below, there is a one microsecond uncertainty in the RTC value read by a T-bus device.

NOTE

WARNING TO T-BUS MASTER DEVICE HARDWARE DESIGNERS
The configuration status unit (CSU) achieves correct RTC reading by handshaking with the RTC controller. This handshake waits for the next one-microsecond increment pulse from the prescaler. When the pulse occurs, the RTC must then be read before the next increment pulse, in one microsecond. This time includes synchronizer delay, CSU response time, and time for any pauses that the T-bus master may cause by asserting T_MASTER_PAUSE. Therefore, the T-bus master should use extreme caution when causing assertion of T_MASTER_PAUSE. Otherwise, the RTC value read cannot be guaranteed accurate!

Writing the RTC

Only the high sixteen bits of the RTC may be written, and a half-word write must be used. To avoid a race condition between the write and the update performed by the sixty-five millisecond pulse, the low half of the RTC should be examined first to determine that it will not overflow until the write has finished.

Time Of Next Interrupt

The SIGA supports a real-time interrupt feature for use by processors on the function board. The interrupt timer register is the Time Of Next Interrupt, or TONI. The SIGA contains two functionally identical TONI registers (A and B), and their associated circuitry. Each TONI register is 32 bits. The difference between each TONI and the RTC is continually computed, and used to generate an interrupt signal.

The value of the RTC, treated as a 32-bit unsigned number, is subtracted from the value of the TONI, also treated as a 32-bit unsigned number. Whenever the result, treated as a twos complement number, is negative, two effects happen. One effect is that the TONI_INT_PENDING bit in the status register (TONI[A,B]_Config, readable via the T-bus) is set to one. This bit is cleared only by writing into the TONI register a value larger than the current RTC value. The second effect is to assert the external pin TONI_INT, if enabled by the Enable_Toni_Interrupt bit in the Requestor_Config register. This pin follows the state of TONI minus RTC, rather than staying asserted as the status register bit does.

T-bus devices may read or write each TONI register at any time. Word operations must be used to write it. The CSU in the SIGA synchronizes writes with the next switch clock period to avoid spurious interrupts. If a value less than the current RTC is written, an interrupt will be pending immediately. In reading each TONI register, as with any SIGA internal register, all reads function as word operations.

Two RTCs in Redundant Machines

Remember that in a machine with two switches, each switch has its own clock. These two clocks are *not* synchronized, so the RTCs in SIGAs on the two switches will have different values.

3.9.5 Other Functions of the Clock Distribution System

The *hold* signal, used to control the priority of SGA ports as described above, is generated by the clock master card and distributed in parallel with the switch clock signal to each SGA. The hold signal is synchronized with the switch clock. Potentially different phases of the switch clock are sent to the

different switch columns, so different phases of hold are generated and distributed accordingly.

The clock distribution system supports a clock signal called *net time*. This signal is discussed in a separate section below.

The *sixty–five millisecond pulse*, used to keep the real time clocks synchronized as described above, is generated on the clock master card and distributed along with the switch clock to each SIGA. It is fanned out by separate drivers on the clock master card (and slave cards, if any), and by separate drivers on the switch card to drive each function board. This isolates failures, preventing corruption of the pulse elsewhere in the system. The pulse is distributed only via the column of switch cards on the requester side of the switch. This places a minor design constraint on function boards. Namely, if a function board were designed as a server only, with no connection to the requester side of the switch, its RTC would not work. No such boards are currently planned.

3.10 Independent Clocks

Besides the switch clock, or switch *clocks* in a machine configured for redundancy, a Butterfly II has other clock systems. These systems are independent, and are not synchronized. In a machine with two switches, the two switch clocks are independent of each other as well. The sections below discuss the additional clock systems.

3.10.1 Test and Control System Clocks

The Test and Control System (TCS) operates on its own clocks. The TCS master and each TCS slave have their own, independent processor clocks. Communication between the TCS master and TCS slaves is asynchronous, but the master and slaves must use the same baud rate, nominally 125 kilohertz. The communication between a TCS slave and the devices it controls is synchronous with a control clock signal generated by the slave. The TCS slave program sets the frequency of the control clock signal, nominally one megahertz. Each controlled device is responsible for appropriate handling of the difference between the control clock signal and data and functions in the device.

3.10.2 T–bus (Function Board) Clocks

The clock used to transfer data within a function board may be synchronized to the switch clock or may be asynchronous. Because the T–bus is considered a part of the function board, the SIGA does not depend on the T–bus being synchronous with the SIGA's requester or server switch clock, and performs synchronizing as necessary to communicate correctly with both its T–bus and the switch. There can be more than one T–bus on a function board, or addi-

tional asynchronous clocks on a function board. Net time and the VMEbus system clock, described below, are examples of the latter.

The tri-statable clock output of the LCON chip is intended for use when a synchronous T-bus clock is desired.

An example of a non-synchronized design is the B2VME function board, which has a nominally 20 megahertz clock. The B2VME processor and T-bus operate synchronously with respect to this clock, asynchronously with respect to the switch clock.

3.10.3 Net Time Support

The Butterfly II switch hardware includes support for collection and distribution of a clock signal used in certain packet switching applications. Specifically, the communication standard known as "T1" requires a knowledge of system time. If multiple Butterfly II function boards perform I/O to a T1 system, they need to have a consistent notion of the T1 system time, called *net time*.

Each Butterfly II T1 I/O function board may detect the net time signal. Each server-column switch card receives the net time signal from its attached function boards, selects one of them as source, and sends this signal on to the clock (slave or) master card, where again one of the incoming net time signals is selected. The net time signal selected by the clock master card becomes the one redistributed throughout the switch. It is sent to clock slave cards (if any) and thence to requester-column switch cards, from which it is sent to function boards.

Net time is collected and distributed with differential signals, except single ended in the first step from function boards to switch cards. Distribution fanout is performed by bussing rather than by multiple drivers.

NOTE

TECHNICAL DETAIL
T1 is a synchronous, framed, 1.544 megabits per second communication standard originated by the AT&T corporation. The initial implementation of Butterfly II technology does not include any function boards for T1 I/O.

3.10.4 VMEbus System Clock Support

The B2VME function board contains a general purpose interface that couples the T-bus and an external VMEbus. This interface includes the capability of being VMEbus system controller. As such, the B2VME must, among other things, drive a 16 megahertz clock signal on the VMEbus. This signal is not

synchronous with the switch clock, nor with the B2VME processor (T-bus) clock.

3.11 Switch Protocol and Message Formats

This section discusses the format and meaning of messages sent on switch connections. Locking connections and the priority of messages are described in sections above. Frame and reverse signals are summarized below, and then message content is discussed at length.

3.11.1 Frame and Reverse

The signals *frame* and *reverse* control the connection through the switch. Frame always travels from requester to server, the forward (downstream) direction through the switch. Reverse always travels the reverse (upstream) direction, toward the requester. They are important to the protocol because they define not only the connection but also the beginning and end of messages on the connection.

Frame is generated by the requester SIGA and travels through the requester LCON, SGAs and server LCON to the server SIGA. The requester and server LCONs do not use frame to determine direction of data flow, but rather are controlled by direction signals from their respective SIGAs. The SGAs use frame only to control enabling their forward drivers. The server SIGA uses frame to detect the beginning and end of downstream messages and the end of the connection. Data flow is reversed by a drop in frame for just one switch clock cycle. A drop in frame for two or more cycles indicates the end of the connection. Because the requester SIGA generates frame, it alone controls the direction and termination of the connection.

Reverse is generated either by SGAs or by the server SIGA and travels back to the requester SIGA. Reverse asserted for only one switch clock cycle is a "reject". A reject is understood by the upstream SGAs and the requester SIGA as a command to tear down the switch path and abort the attempt. An SGA generates a reject when it is does not accept a bid for an output port. Once an incoming message is granted a port, the SGA will not generate a reject for the lifetime of the connection, although a reject received from downstream is relayed upstream and tears down the connection, SGA by SGA, as it travels upstream.

A server SIGA generates a reject for any of three reasons. First, if the initial message attempts to access a T-bus device that is already locked, the server SIGA returns a reject. (This can arise because some other device on the T-bus may have it locked.) Only the initial message can cause a reject. In a locked sequence, messages after the initial message may attempt access to additional devices, but if these are already locked the response is an error message, not a reject.

Second, a server SIGA that is not in the idle state will reject connection attempts. An example of this is when a previous, locked connection has ended, so the server SIGA is issuing a FREE-LOCK on its T-bus. Until it finishes this T-bus transaction, the server SIGA rejects connection attempts.

Third, a server SIGA will reject connection attempts under control of a bit in its configuration register, accessible via the T-bus. The server SIGA will still complete any transactions in progress. This provides a way to gracefully remove the server SIGA from service, such as when testing or when changing to the other switch of a redundant machine. The SIGA synchronizes this bit with its internal operation, so a T-bus device may assert or negate the bit at any time.

Other than the single-cycle reject pulse, reverse is generated only by the server SIGA while it supplies upstream bytes of message and checksum. This assertion of reverse for at least two switch clock cycles serves as a message acknowledgment to the requester SIGA. The SGAs use reverse to enable their upstream drivers. While reverse is thus asserted, the SGAs along the path will not tear down the connection even if frame drops. Therefore, the requester SIGA may drop frame as soon as the first byte of the upstream response to the final downstream request has made it through the switch. This tears down the connection faster, making resources available for other use sooner. This early dropping of frame is performed only on unlocked sequences, because on a locked sequence the requester SIGA does not know that there will be no further requests. Also, a SIGA configuration register bit must be set to enable the quick drop.

3.11.2 Message and Connection Boundaries

Frame and reverse define message boundaries. Frame indicates the boundaries of downstream messages, reverse those of upstream messages.

When there is no connection, frame is low; frame being low for at least two switch clock cycles is the definition of there being no connection. The requester SIGA asserts frame with the first (header) byte of the initial message, and holds it asserted until the last (checksum) byte. During transmission of the checksum byte frame is de-asserted, and immediately thereafter frame is asserted again. Frame is held asserted until the requester SIGA is either done with the connection or is ready to send another downstream message on it. If another downstream message is sent on the connection (after receiving a response to the previous downstream message), the requester SIGA de-asserts frame for one cycle immediately before that message begins, asserts frame during the message except for a one-cycle de-assertion during the last byte, and re-asserts frame after the message is sent. Thus, low-to-high transitions of frame mark the start and end of downstream messages, but whether a downstream message is in progress can be determined only by tracking the history of frame during the connection.

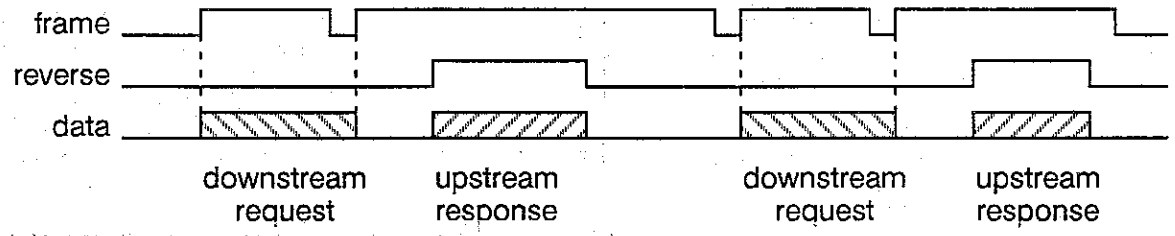
The server SIGA asserts reverse during every byte of every message it sends. Thus an upstream message is bounded by the assertion and negation of reverse.

Intervals of an unspecified length occur after a message in either direction. During these intervals, the data lines are driven in the upstream direction. The server SIGA presents constant padding data (all zeros) during this time.

The connection itself is also bounded by frame and reverse. The connection (or attempt at connection) begins when frame is asserted after being low for at least two switch clock cycles. The connection ends when frame stays low for two switch clock cycles. For an unlocked sequence, frame may be dropped during the response message, as described above.

Figure 3–12 shows an example of a connection, illustrating the use of frame and reverse to bound messages.

Figure 3–12 Frame and reverse during example connection.



3.11.3 Basics of Message Functions

Simply put, the Butterfly II switch message supports two functions: read and write. Requests originate exclusively at the requester end of a connection, and responses exclusively at the server end. Each message constitutes one complete request or response. The request contains a single address at which consecutive bytes of data are to be read or written. Data to be written is contained in the message requesting a write. Data read is returned in a message responding to a read request, on the same connection as the request. The amount of data read or written can be any of seven sizes from one to sixteen bytes.

A switch transaction may be modified by either of two properties: locked and stolen. Locking is a property of a connection; it maintains a switch path open so multiple messages may be exchanged on the connection, and excludes interfering actions at the server device. Stolen is a property of a location accessed in a server device; it provides a kind of mutual exclusion lock useful in parallel processing. Both locking and stealing are supported by bits in the message format.

Every request presented to the switch is actively acknowledged in some way. This is an important feature of the Butterfly II switch protocol. For example, it is never assumed that a write request was correctly received and executed. After every request, the requester SIGA is in a waiting state, from which one of two resolutions will follow. Either an acknowledgment will be received (with data, if the request was a read), or an error will be detected and reported.

3.11.4 Message Components

Messages are made of four major components. Not all components are present in every message, but when present they appear in the order described here.

- HEADER — contains routing and priority; also called BID BYTES. Present only in initial message on a connection, thus only in downstream messages. Consumed by SGAs as the message traverses the switch. The rest of the message is called the BODY.
- COMMAND — contains fields specifying locking, function (read or write), size, and address. Present in every downstream message, never in an upstream message.
- DATA — contains information needed to perform the function of the message. Content varies depending on the purpose of the message.
    - in a downstream write request, the DATA component contains the data to be written.
    - in an upstream response to a successful write request, the DATA component contains one byte of unspecified value.
    - in an upstream response to an unsuccessful write request, the DATA component contains one error code byte.
    - in a downstream read, the DATA component is absent.
    - in an upstream response to a successful read request, the DATA component contains the data read.
    - in an upstream response to an unsuccessful read request, the DATA component contains all data that was read before the error occurred, or through the word with a stolen bit set, whichever occurs first, except that the least significant byte of the last word is replaced by an error code. Only this last word is presented to the requester device.
- CHECKSUM BYTE — contains bits for error indication and stolen indication, and a "checksum" field. Present in every message.
    - ERROR BIT — indicates an error, in upstream messages only. Unused (zero) in downstream messages. Determines whether data bytes contain valid data or an error code.

- STOLEN BIT — supports memory access protocol. Present in all messages. Modifies interpretation of data bytes in an upstream message.
- CHECKSUM — detects transmission errors. Present in all messages.

The following sections describe these message components in detail.

3.11.5 Message Header

The purpose of the message header is to set up the path through the switch. Consequently, it contains the priority and routing information used to bid for an output port at each switch column, and is consumed byte-by-byte as the message establishes the connection. When the message arrives at the server LCON, no header remains. Only an initial message contains a header, thus only downstream messages can have headers, and therefore only a requester SIGA is able to generate a header. The bytes of header are sometimes called *bid bytes*.

Figure 3–13 Message header for a three–column, base–8 switch.

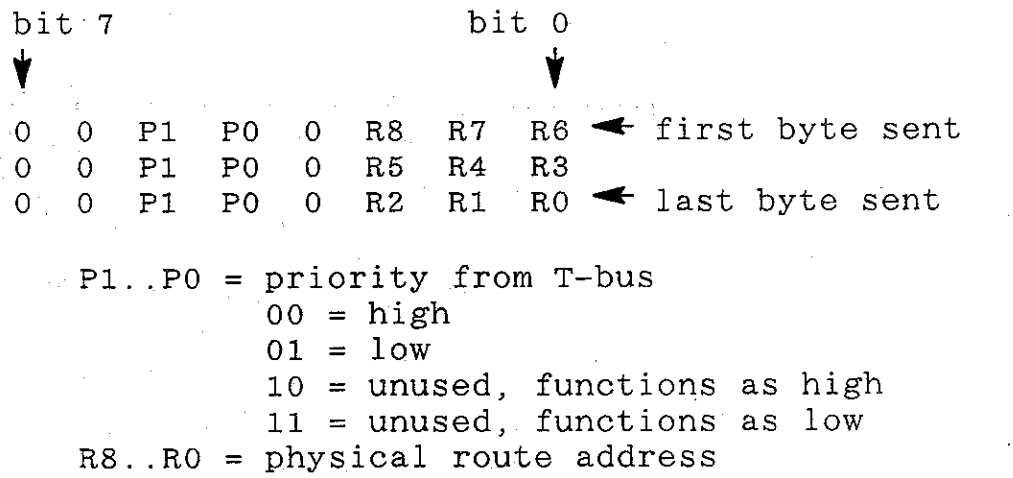
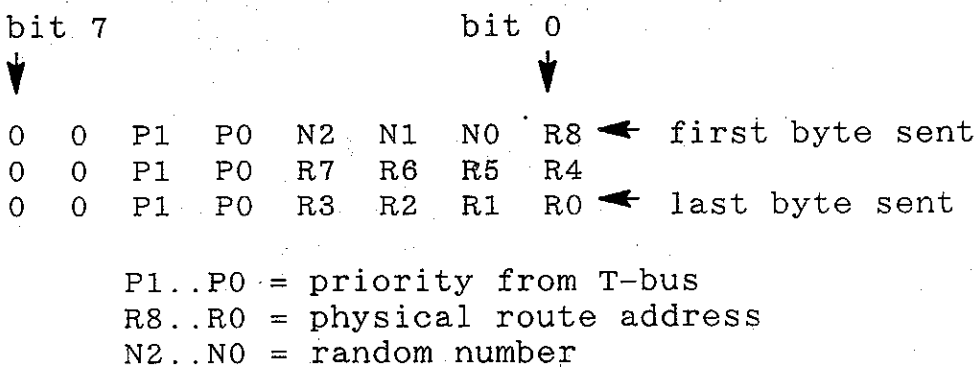


Figure 3–13 shows the header format for a three–column, base–8 switch. The priority bits appear in each byte so they are readily available to each SGA as it processes the first byte it receives. The physical route address derivation is explained below. For a two–column switch, the first byte is not sent, only the middle and last bytes. For a one–column switch, only the first or only the middle byte is sent (see note below).

NOTE

TECHNICAL DETAIL
A set of four chips of the original SGA design implements an eight–by–eight crossbar. The SIGA supports both this design and a sixteen–by–sixteen switch node. The message header for the latter, a base–16, three–column switch, is shown in Figure 3–14. In a two–column switch, only the middle and last bytes are sent. In a one–column switch, only the first or only the middle byte is sent The three bits of random number are obtained from the random route generator described below.

Figure 3–14 Base–16 switch message header format.



A configuration register in the SIGA selects whether base–8 or base–16 message headers are generated. Two additional bits control whether one, two or three bytes of header are sent. Two bits provide four values, but only three are apparently necessary; the fourth value selects an alternate form for one–byte headers, as shown in Figure 3–15. For a base–16, one–column switch, the alternate form is not useful because of its randomized bits.

Figure 3–15 Switch size specification for message header.

for switch	col1 bit	col2 bit	bytes sent in header
3 column	0	0	first, middle, last
2 column	0	1	middle, last
1 column (std)	1	0	middle
1 column (alt)	1	1	first

Whatever the switch base and the number of columns, the header contains a number of bytes equal to the number of switch columns, and two raised to the

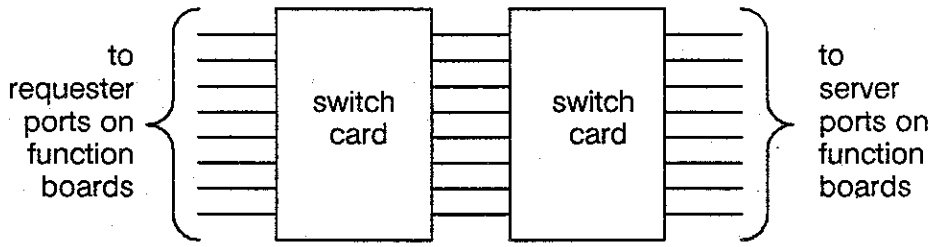
power of the number of routing bits per byte is the switch base (modulus). For example, a base–eight (23) switch requires three routing bits per byte.

Physical Route Address

The switch route address on the T–bus undergoes two transformations to generate the physical route address used in the message header. The first transformation supports optional interleaving. The switch route address on the T–bus (T_AD<33..25>) can be used directly (non–interleaved) or as transformed by the interleaver (bits MOD<8..0>). The interleaver output is used if both the SIGA's INTERLEAVED pin is asserted during the T–bus request cycle, and the Enable_Interleave bit in the requester SIGA's configuration register is asserted. Otherwise, the address straight from the T–bus is chosen. The result of this choice is the *logical route address*, and is used both in the second transformation and as input to the checksum calculation.

The second transformation takes the logical route address into the physical route address by randomizing some of its nine bits. If a Butterfly II machine has enough function boards that all available switch paths are essential, then all switch addresses are distinct and none of the bits should be randomized. Some smaller configurations, however, have alternate paths terminating at the same server. For example, a two–column machine with only eight function boards may be configured to have eight paths to each server. Using alternate paths reduces contention within the switch and can provide robustness against failures of switch hardware in internal columns. The SIGA contains a nine–bit route address mask register and a random route generator, a random number generator described in the note below. In each bit position where the mask contains a "1", that bit is taken from the random route generator. Where the mask contains a "0", the corresponding bit comes from the logical route address. The result is the physical route address used in the message header.

Figure 3–16 Example of a switch with alternate paths.



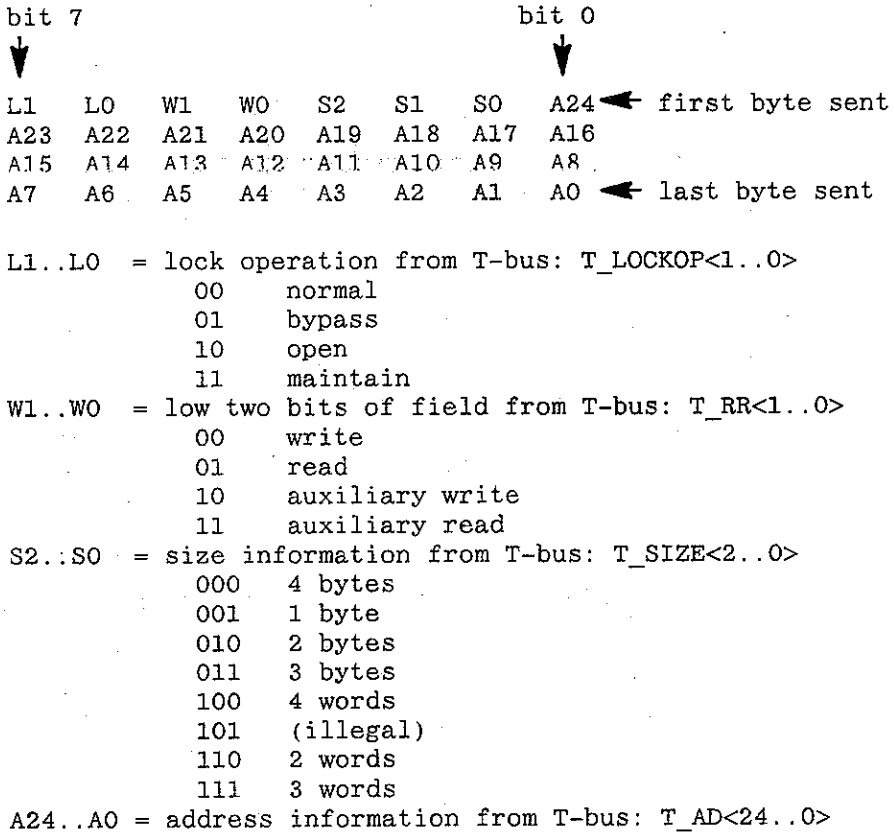
NOTE

TECHNICAL DETAIL
The random route generator is nine bits of an on-chip, 11-bit maximum sequence generator ($x^{11} + x^2 + 1$), updated at the T-bus clock rate, and reset via the T-bus. Resetting the generator ensures it is not in its dead state (all ones). Typically, the TCS or a bootstrap program will reset the generator during system startup. Each SIGA has two random number generators: the random route generator described here, updated at the T-bus frequency; and the generator used in the random strategy for message transmission, updated at the switch clock frequency.

3.11.6 Message Command

The command component, present only in every downstream message, is shown in Figure 3-17. Locking is described in a section above.

Figure 3-17 Message command format.

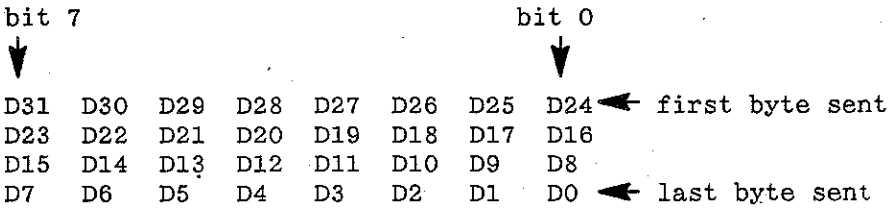


3.11.7 Message Data

The format of the data component of a message depends on several factors. The simple case is described below, followed by the changes brought about by special conditions.

In a write request or a response to a successful read operation, the message contains the data to be written or the data that was read, in "big-endian" format. That is, the most significant byte of each word is sent first. Figure 3–18 shows the format of data in a one–word write request message, or a one–word successful read response message.

Figure 3–18 Big–endian data format.



Two–, Three– and Four–word Transfers

If two, three or four words are written or read, the words at successively higher addresses follow the first word in the data of the message.

One–, Two– and Three–byte Transfers

If only one, two or three bytes are requested, a full word nevertheless appears in the message. The bytes to be written or read appear in the message data in the same position they would in memory. For instance, a two–byte transfer of the most significant half of a word occupies D31 through D16 of the message. In a write operation, the unused bytes of a one–, two– or three–byte transfer are ignored, and in a read operation those bytes are whatever data accompanied the requested byte(s) on the T–bus. Thus it is the responsibility of the T–bus device originating a request to place write data in the correct position within a word, and to extract read data from the appropriate place in the response word.

Write Response

The data component in the response message to a write is always one byte, regardless of the size of the request or the success of the operation. If the write is successful, the numerical value of the response data byte is unspecified. If

the write is unsuccessful, the byte contains an error code. The error codes are described in a section below. The error bit in the checksum byte indicates whether an error occurred, and thus whether the data byte contents are unspecified or are an error code. In a multi-word write, it is impossible to tell, from the message alone, which word(s) caused the error.

Read Response

The response to a read request always contains a data component of one, two, three or four words. If the read operation is successful, the response contains the requested data.

If the read operation encounters a word with the stolen bit asserted, special action is taken. The server SIGA proceeds to read all requested words from the T-bus, but any data past the first word with the stolen bit asserted are not included in the response message. The SIGA asserts the stolen bit in the checksum byte of the response. Therefore, if the checksum's stolen bit is a "one", the requesting device must be aware that the message may not contain all the words it requested. The last word returned is the first word whose stolen bit was asserted on the server's T-bus.

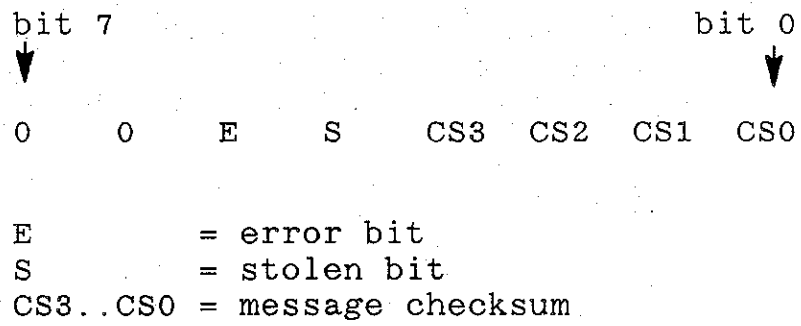
The response to an unsuccessful read request may be different in the switch than as delivered to the requester device. The server SIGA reads from its T-bus all the data that is requested, until the error occurs. Some of this data may be correct, depending on the server device; the T-bus specification requires only that an error be generated if the operation as a whole is flawed. If a stolen bit is encountered during this reading, any data after the word with the stolen bit is discarded. The least significant byte of what is now the last word is replaced by an error code. The server SIGA sends these word(s) as the message data, and asserts the error bit in the checksum byte. The requester SIGA, however, places on its T-bus only one word, with the error code in the low byte.

Thus, the response *seen by the requester device* to an unsuccessful read request is always one word. The numerical value of the high three data bytes is unspecified; the low byte (D7..D0) contains an error code described later. In a multi-word read, it is impossible to tell, from this response alone, which word(s) caused the error.

3.11.8 Message Checksum Byte

The last byte of every message is a checksum byte, shown in Figure 3-19.

Figure 3–19 **Checksum byte format.**



NOTE

HISTORICAL NOTE
Bit 7 of the checksum byte was once defined as "forward driver enable". Defined for downstream messages only, a "1" would enable the forward driver circuits on the next clock cycle. In the initial implementation of the SIGA, however, this bit is always a "0", and in the initial implementation of the SGA this bit is ignored.

Error Bit

The error bit is defined only for upstream messages, where a "1" indicates that the immediately preceding byte — that is, the last byte of the data component — contains an error code. A "0" indicates no error. This bit is always a "0" in downstream messages.

Stolen Bit

NOTE

STEAL MECHANISM NOT IMPLEMENTED
The switch hardware implements the stolen bit as described in this chapter, but the rest of the machine as produced does not use this bit. Therefore, the steal mechanism is not available to software.

The stolen bit is defined for both downstream and upstream messages. In a write request it asks to write the stolen bit of the given address. Normally, only a full word is stolen. Asserting the stolen bit on a one–, two– or three–byte write, however, is not prohibited by the switch and is presented as such to the server T–bus. Asserting the stolen bit on a multi–word write is defined as illegal by convention. This convention may be enforced by enabling "stolen verify" errors, but for this to be effective the message transmission anticipation

must be set for maximum delay, so the error is detected before the invalid request is transmitted. If the server SIGA receives a multi-word write request with the stolen bit on (in violation of the convention), the SIGA acts as if the bit were zero.

The stolen bit is always "0" in a write response (that is, upstream).

The stolen bit is always "0" in a read request (that is, downstream).

In a read response, the stolen bit indicates that the last word returned is stolen, and that subsequent words requested, if any, are omitted from the message.

NOTE

TECHNICAL DETAIL
In servicing a read request, the server SIGA reads all requested data from the T-bus, aborting the process only upon error. Encountering a stolen bit does not abort the reading; any further data requested is read from the T-bus, but is not sent over the switch in the read response.

Checksum Field

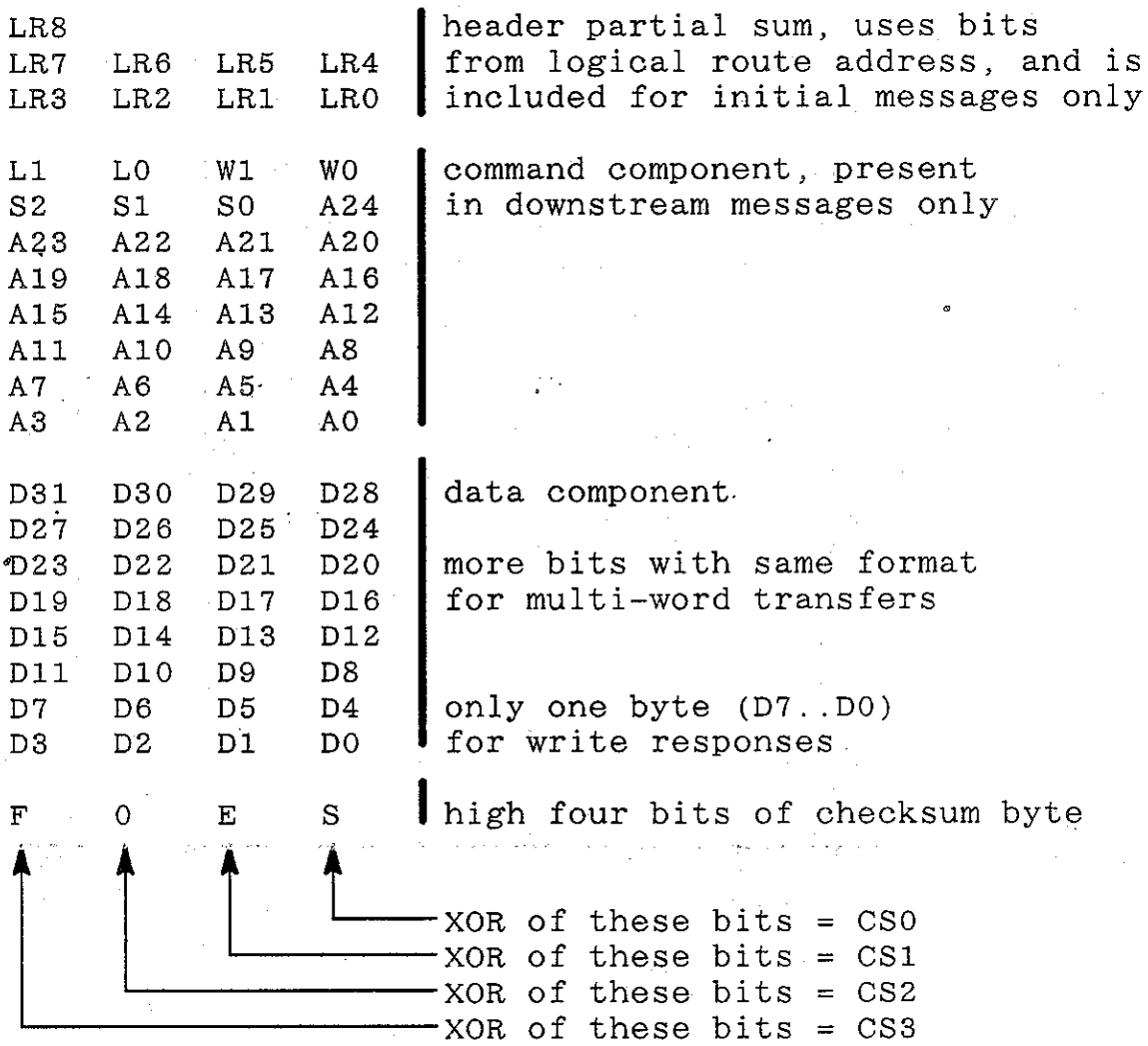
The checksum is generated and checked for every message, both downstream and upstream. For either an upstream or a downstream message, all bytes of the data component, and the high four bits of the checksum byte, contribute to a message's checksum. In downstream messages, all bytes of the command component (including the address) also contribute to the checksum.

If the message is an initial message, it contains a header, but the header is not included in the checksum. Instead, the requester SIGA uses the logical route address, an intermediate value obtained during computation of the physical route address as described above. The bits of the logical route address are combined to form a *header partial sum*, to which the checksum unit is initialized. For all other messages — both non-initial messages of a locked transaction, and all upstream messages — the checksum unit is initialized to zero.

To check the validity of initial messages, the server SIGA initializes its checksum unit to the header partial sum that would deliver a message to this SIGA if the switch is working correctly. The SIGA obtains this 4-bit value from its Server_ConfigA register, which in turn is initialized via the T-bus, typically by the Test and Control System during system startup.

Each bit of the checksum is the exclusive-OR of the contributing bits. Each contributing bit affects only one of the four checksum bits. This is shown in Figure 3-20.

Figure 3–20 Checksum calculation.



3.11.9 Message Sizes

The maximum message size is important because it places an upper bound on the amount of buffering required for a message. The largest message is 24 bytes, including all components. It is a four-word write request in a three-column switch. The header is three bytes, the command is four bytes, the data is sixteen bytes, and the checksum is one byte. The largest upstream message is a four-word read response, which has 17 bytes.

The minimum message size is important because it must not be mistaken for connection control functions. In particular, both a drop in frame and an assertion of reverse for one switch clock cycle have special meanings. The smallest message is two bytes, a write response, which always has one byte of data and one byte of checksum. The smallest downstream message is read request in

a two-column switch, which has two header bytes, four command bytes, no data component, and one checksum byte, for a total of seven bytes.

3.11.10 Switch Protocol Timers

The SIGA implements two timers to detect major problems in switch transactions. One timer is a reject timer, the other a connection timer. The reject timer limits waiting for a connection to be established, and the connection timer limits how long a connection, once established, may last.

Reject Timer

The reject timer limits how long a requester SIGA will repeatedly try to establish a connection in the face of repeated rejects. The reject timer is a four-bit counter that is initialized when the first transmission of an initial message begins, and is counted down. If it underflows, that is remembered. Each time the requester SIGA receives a reject, it checks whether the reject timer has underflowed. If not, the message is retried after a delay. If underflow has occurred, the requester SIGA makes no further retries and reports an error (code Rej_TO) to the T-bus device. A configuration register, available on the T-bus, contains two parameters of the reject timer. One parameter is the four-bit value loaded into the counter. The other selects the rate at which the counter is decremented. The decrement signal is taken from the low-to-high transition of one bit in the low half of the real time clock. The configuration parameter selects which bit is used. The clock counts at a one megahertz rate, and functions here as a variable prescaler. If the value of the four-bit prescale selection parameter is N, the output of clock bit N is used to decrement the reject timer. Thus the timeout value is

$$\text{initializer} \times 2^{(\text{prescale} + 1)} \text{ microseconds,}$$
$$\text{with an uncertainty of } 2^{(\text{prescale} + 1)} \text{ microseconds.}$$

Connection Timer

The connection timer limits how long a connection may be held. The connection timer is an eight-bit counter that is loaded each time an initial message transmission is begun, whether it is the first try or a retry. (Here, "initial message" means the first message on a connection, establishing the connection.) If the counter underflows, the requester SIGA tears down the connection, even if it is locked. The requester SIGA also reports an error to the T-bus device, but the code and timing depend on whether the requester SIGA is waiting for a message acknowledgment (assertion of reverse for at least two switch clock cycles). If awaiting an acknowledgment, the error code Wait_TO is reported immediately. If not awaiting an acknowledgment, the requester SIGA waits until the T-bus master makes another request of the SIGA, and then returns the error code Idle_TO. Meanwhile, the requester SIGA remains "locked". If some other T-bus device makes a request of the requester SIGA while it is

locked, the SIGA returns a REFUSED–LOCKED error to it, so the Idle–TO is not delivered to the incorrect device. If the correct device issues a FREE–LOCKS while the requester SIGA is locked, the Idle–TO error is forgotten.

The connection timer is initialized from eight bits of a configuration register available on the T–bus. It is decremented at one megahertz, so the timeout value is up to 255 microseconds, with a resolution and uncertainty of one microsecond.

3.11.11 The Switch as a T–bus Device

THIS SECTION SHOULD INCLUDE DISCUSSION OF ERROR DETECTION AND REPORTING IN GENERAL, AS SEEN BY THE REQUESTING T–BUS DEVICE. ALSO, SOME THINGS CAN GO IN SIGA CHIP DISCUSSION INSTEAD OF HERE.

SEE THE B2VME CHAPTER FOR LIST OF ERROR CODES, AND SIGA SPECIFICATION FOR DISCUSSION OF SIGA ERROR DETECTION LOGIC.

3.12 Switch Chip Set: SGA, LCON, SIGA

3.12.1 LCON

The primary purpose of the LCON (Level CONverter) is to translate signal levels between the switch and the function board. This conversion, and other aspects of the LCON, are discussed below. Last, there is a description of each signal pin.

Level Conversion and Signal Integrity

The SGA and discrete circuitry on the switch card are 100K ECL devices running between –4.5 volts and ground, with nominal signal levels of –0.9 and –1.7 volts. The SIGA and other circuits on the function board run at the conventional TTL voltage, +5 volts. The LCON converts between these levels. To accomplish this, the LCON is implemented using a gate array with "mixed mode I/O". That is, all internal gates are ECL, while macros (components of the chip) are available that accept TTL levels on input and drive TTL levels on output. To do this, the LCON requires power at both +5.0 and –4.5 volts.

NOTE

TECHNICAL DETAIL
The gate array device used for the LCON is intended to be run at –4.5 volts, like any other 100K ECL device. It runs satisfactorily at –5.0 volts, the only drawback being increased power dissipation. The Butterfly II runs it at –5.0 volts to simplify power supply requirements.

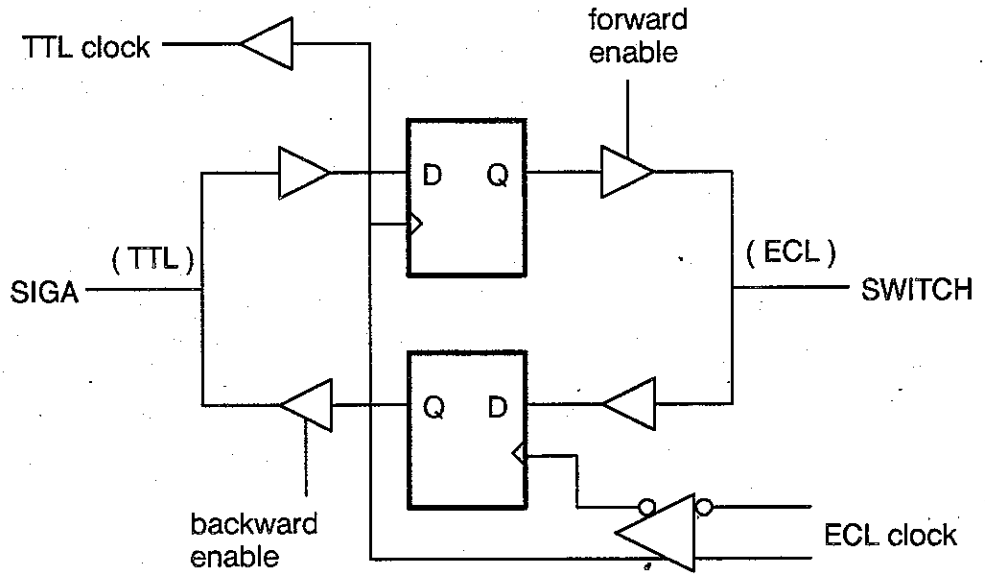
The Butterfly II signalling environment is 50 ohms. Bidirectional data lines are terminated in 50 ohms (to –2 volts) at both ends. Therefore, the LCON and SGA must drive a 25–ohm DC load. Further, the far end termination resistor should not be shunted with any low impedance from the inactive driver, so "cut–off" drivers are used. These drivers go to a low state that is lower than the standard 100K ECL level. This ensures that the base–emitter junction of the output emitter follower is reverse biased and therefore presents a high impedance.

Reclocking

The LCON reclocks the signals data, frame, reverse and the 65–millisecond pulse, to reduce timing uncertainty in these signals. As a signal passes through a series of gates and cables, the minimum and maximum propagation delays accumulate, resulting in a large period of uncertainty following a change in the input signal. By clocking the signal through a register (flip–flop) stage, however, the uncertainty is reduced to the clock–to–Q propagation delay of that register, at a penalty of one clock cycle delay.

Figure 3–21 shows the reclocking of a data signal in the requester section. Because data signals are bidirectional, reclocking is more complex for them than for other signals. A unidirectional signal requires only the upper or lower half of the circuitry shown. Signals from the switch to the SIGA are clocked on the falling edge, while signals from the SIGA to the switch are clocked on the rising edge.

Figure 3–21 LCON reclocking example.



Enabling Data and Control Signals

Because data pins are bidirectional, they must be "turned around" when the direction of data flow changes. The SIGA controls the turn around via the signals **nrqbck** and **nsvbck**. Frame and reverse pins are always enabled during normal operation, and are disabled only when it is necessary to isolate the LCON from the SIGA or from the switch for diagnostic purposes, or when the system is changing over from one switch to the other. When disabled, the frame and reverse outputs on the switch side are gated to produce a logical 0 level, and those on the SIGA side are tri–stated. While enabling or disabling these control signals, the switch (or both switches) should be quiescent to ensure no broken messages are produced.

The tri–statable clock output can be enabled and disabled independently of the LCON requester and server functions. In a function board that derives its processor clock from this pin, the separate enable allows testing and use of the switch path, and changing from one switch to the other, without disturbing the processor clock. The 65–millisecond pulse is always enabled.

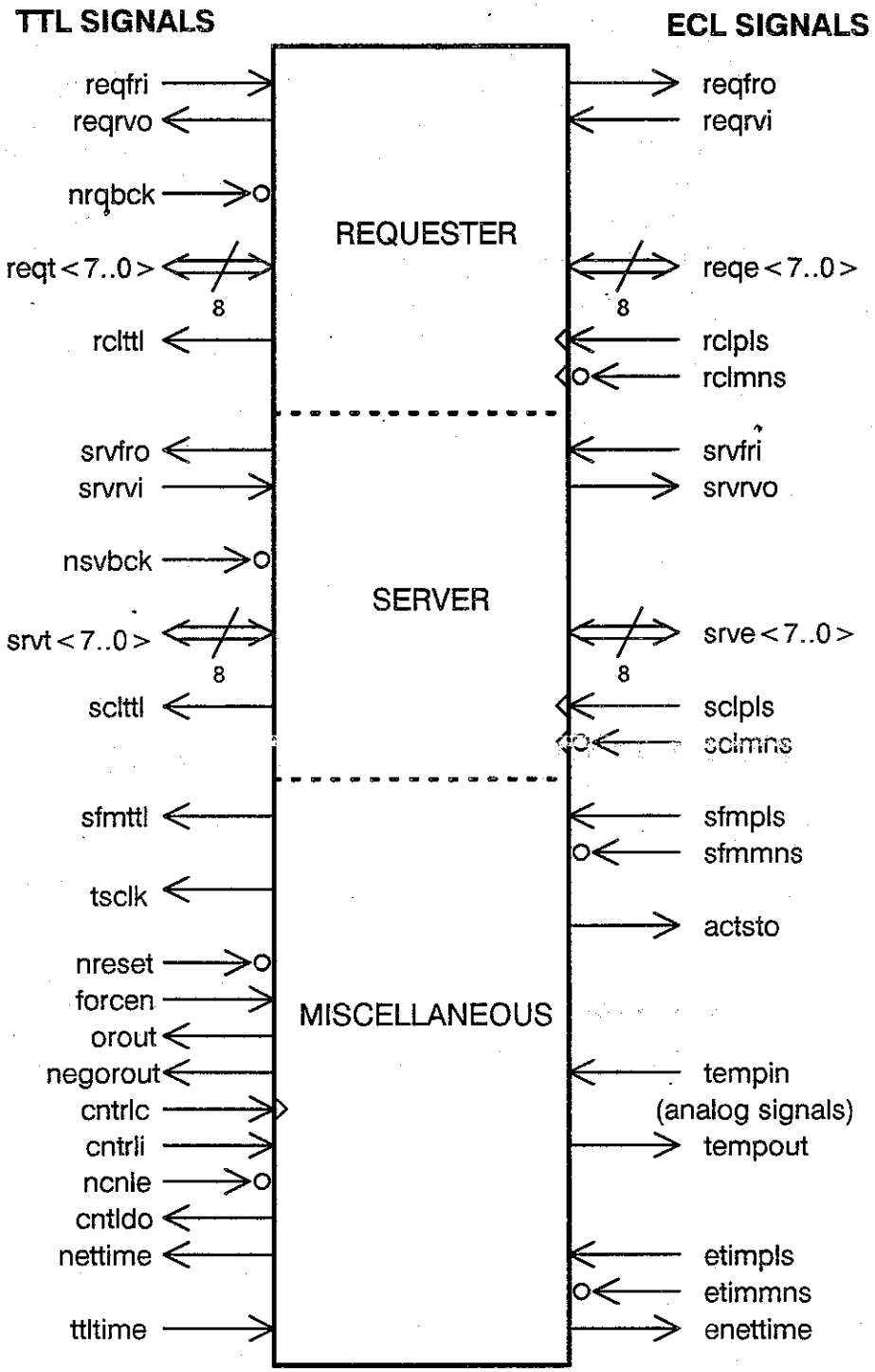
LCON Signal Pins

Figure 3–22 shows all electrical connections to the LCON, other than power and ground pins, and the three sections of the LCON. Signal names in the requester and server sections are identical except that a leading *r* or *s* specifies requester or server. The "data" pins are bidirectional; all other pins are unidi-

rectional, except for testability (described below). Signals shown on the left side of the LCON are TTL levels, and signals on the right are ECL levels. The left side connects to the SIGA, and the right side to the switch. The miscellaneous section is mainly the TCS interface.

Signals are asserted with a high level, except for the negative sides of the four differential pair clock inputs, and for the four signals *nrqbck, nsvbck, nreset* and *ncnle*. The nrqbck and nsvbck signals come from the SIGA and tell the LCON whether the direction of data flow is forward (high, de–asserted) or backward (low, asserted). These names are mnemonic for "not requester backward" and "not server backward". The nreset signal is discussed in the section on resetting the LCON, and the ncnle signal is part of the TCS interface.

Figure 3–22 LCON input and output signals.



The LCON signals are described in Figure 3–23. For signals that are passed through the LCON, the "direction" column uses the symbol ⇒ to indicate from SIGA (TTL) to switch (ECL), the symbol ⇐ to indicate the opposite, and ⇔ for a bidirectional signal. Note that ⇒ is the forward direction for the requester, *but is the reverse direction for the server section.*

Figure 3–23 LCON signals: requester, server and miscellaneous.

TTL signal	Direction	ECL signal	Type	Comments
reqt<7..0>	⇔	reqe<7..0>	reclocked bidirectional	data
reqfri	⇒	reqfro	reclocked unidirectional	frame
reqrvo	⇐	reqrvi	reclocked unidirectional	reverse
nrqbck	—	—	input	direction control
rclttl	⇐	rclpls rclmns	redriven unidirectional	clock
srvt<7..0>	⇔	srve<7..0>	reclocked bidirectional	data
srvfro	⇐	srvfri	reclocked unidirectional	frame
srvrvi	⇒	srvrvo	reclocked unidirectional	reverse
nsvbck	—	—	input	direction control
scllttl	⇐	sclpls sclmns	redriven unidirectional	clock
tsclk	—	—	output	function board clock
nreset	—	—	input	reset
cntrlc	—	—	input	TCS clock
cntrli	—	—	input	TCS data in
ncnle	—	—	input	TCS execute
cntldo	—	—	output	TCS data out
sfmttl	⇐	sfmpls sfmmns	redriven unidirectional	65 millisecond pulse
nettime	⇐	etimpls etimmns	redriven unidirectional	system net time to function board
ttltime	⇒	enettime	redriven unidirectional	function board net time to clock master card
—	—	actsto	output	AC parametric test
orout	—	—	output	OR tree
negorout	—	—	output	OR of negative inputs
forcen	—	—	input	force enable
—	—	tempin tempout	analog input analog output	temperature sensing diode

LCON TCS Interface

The LCON has a Test and Control System (TCS) interface that allows a simple microcomputer to enable, disable, monitor, test and reset the LCON. Other circuitry allows convenient testing of all LCON pins not testable by the TCS interface.

Some details of the LCON TCS interface are covered in the TCS chapter.

Resetting the LCON

There are two ways to reset the LCON: assert the "nreset" pin, and issue the reset command via the TCS interface. The latter clears only the latches containing the three enable bits (for requester section, server section and tri-state clock output) and the divider chains for monitoring clocks. It does not clear any of the flops used to pass data and control signals.

The nreset pin, however, drives the reset input of all internal flip-flops. Also, as an aid to in-circuit testing, as long as the nreset pin is held asserted all TTL outputs (except as noted here) are tri-stated and all ECL outputs are held in the low state. The TTL clocks (rclttl and sclttl) are not tri-stated because they are in the critical timing path and tri-state buffers are slow. Nor are "orout" and "negorout" tri-stated, as noted in the section on testability.

Because the requester and server clocks run continuously and the nreset signal is not synchronized inside the LCON, it is possible that the data, frame and reverse flops could go metastable if nreset were de-asserted just as a flop is clocked. The enable latches and the TCS command shift register are clocked only on command from the TCS, and so do not have this problem. Since the enable latches are guaranteed to be clear after reset there is no chance of an LCON sending garbage on system turn-on.

Dedicated Test Pins on the LCON

Besides the TCS interface, the LCON provides three facilities for testing before and during operation. These are the OR-tree, the AC test output, and the temperature sensing diode.

Asserting the nreset pin allows an external agent, such as an in-circuit tester, to drive the pins that normally are LCON outputs. These, and many input pins, are OR'ed together to drive the "orout" pin. To do this, the LCON pins that are functionally just outputs are implemented as bidirectional; the only connection made to the input side is to the OR-tree. The in-circuit tester uses orout by bringing all inputs low, then toggling them one at a time and watching to see that orout toggles also. This verifies a limited amount of device functionality and, more important, contact between LCON pins and the board. A requirement of the simulation and test tools is that differential pairs must always be driven with complementary values. Consequently, the negative sides of the

four differential pair clock inputs are OR'ed into a separate tree that drives the "negorout" pin.

NOTE

TECHNICAL DETAIL
The OR–tree has 57 inputs. The requester and server secctions contribute 22 inputs each. (Their TTL clocks are excluded as noted above, and their negative ECL clock is in the negout OR–tree.) The remaining 13 inputs are sfmttl, tsclk, forcen, cntrlc, cntrli, ncnle, cntldo, nettime, ttltime, sfmpls, aststo, etimpls and enettime.

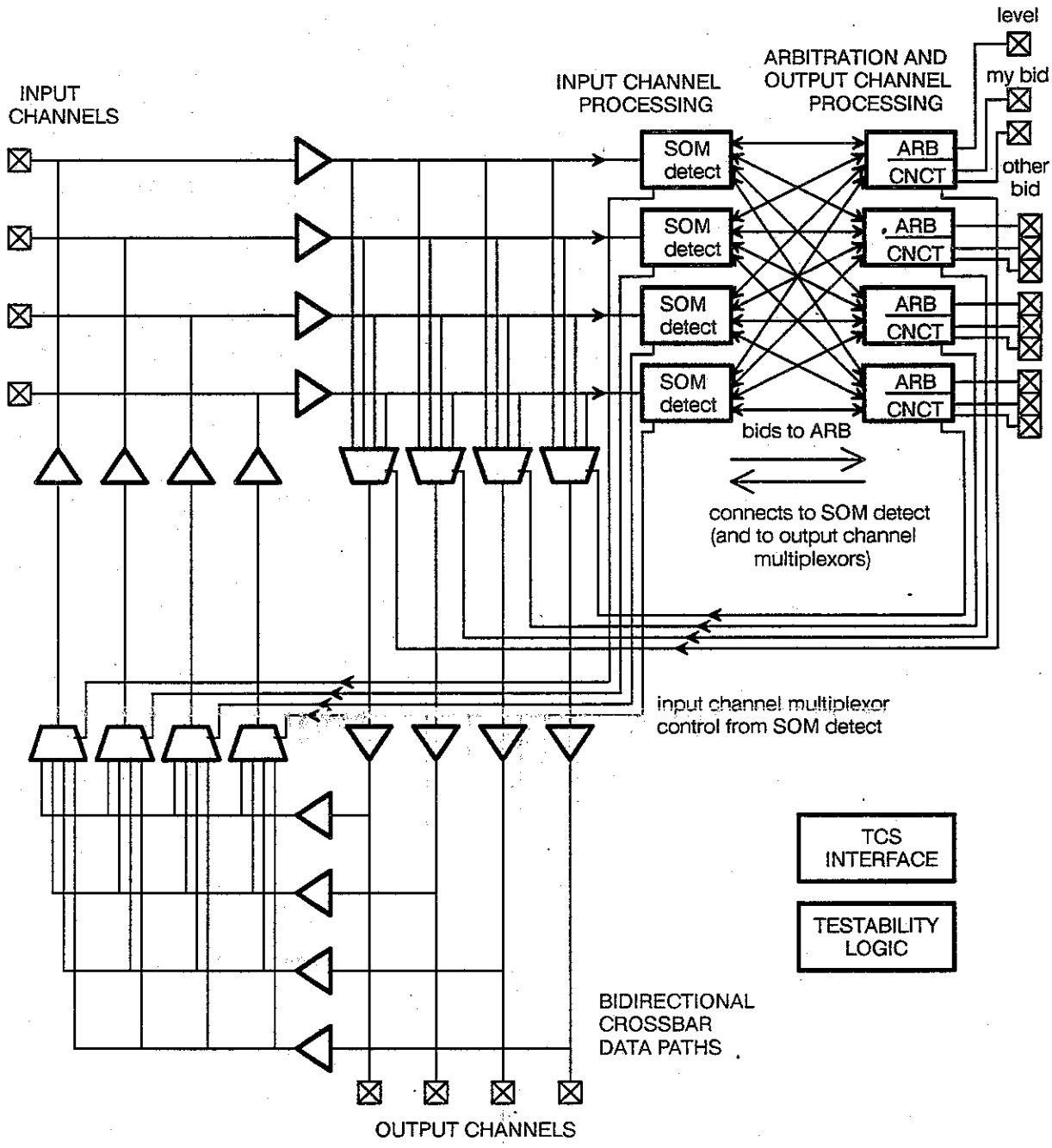
The "actsto" signal is the output of a 6–stage ripple counter driven from the requester clock. Its propagation delay is a rough indicator of device speed, and is large enough to be measurable on production test equipment.

The "tempin" and "tempout" pins connect to a diode on the chip. By injecting a known current (e.g., 100 microamperes) and measuring the voltage drop, the die temperature can be monitored during operation. The proportionality between temperature and voltage may differ from one fabrication lot to another.

3.12.2 SGA

Figure 3–24 shows the basic data paths and functional blocks of the SGA (Switch Gate Array).

Figure 3–24 SGA block diagram.



The SGA is essentially a 4 × 4, bidirectional crossbar switch. Each input channel and each output channel has eight bidirectional data lines, plus various control signals (frame, reverse, and signals for arbitration between SGAs). Clocked at 40 megahertz, the SGA provides a bidirectional throughput between input and output of 320 megabits per second per channel.

TCS Interface

The SGA has a Test and Control System (TCS) interface that allows a simple microcomputer to read and write bits within the SGA independently of the crossbar data paths. Using this interface, the TCS can monitor the status of message traffic traveling through the SGA, and turn input and output ports on and off in the event of a hardware failure. The TCS interface can also sense individual forward data path input signals and control individual forward path output pins to allow switch path connectivity testing in operational systems. Other circuitry allows convenient testing of all SGA pins not testable by the TCS interface.

Data Paths

The data path for all channels is similar. Downstream data arriving at an input channel is buffered immediately upon entering the SGA. The buffered data values are routed to the connection control logic and to the corresponding inputs of four multiplexors, one for each output channel. Each multiplexor is an 8-bit wide, 4-to-1 selector that determines which input channel, if any, is connected to a given output channel. The outputs of each multiplexor are connected to an 8-bit register (not shown), where the data is reclocked. The register outputs connect through high power bus drivers to the output channel data pins. Similar circuitry going from output channels to input channels makes each path bidirectional.

Control Logic

The functional blocks SOM detect (Start Of Message detector, also called SOMDET), ARB (arbiter) and CNCT (connection) process arriving control and routing signals and establish connections.

Each input channel has its own SOM detector block. This block monitors the channel's input control and data lines for the start of message condition (frame going high after being low for two switch clock cycles). When this is detected, the SOM detector requests a connection to the appropriate output channel, by asserting a *bid* signal for that output channel.

The on-chip bids for a given output channel are processed by the arbiter logic. It chooses which input, if any, will be connected to the associated output channel. The choice is made at random from the bids that have sufficient priority. The arbiter tells the connection block which input channel has been chosen.

The connection logic uses the input channel selection ("who") information from the arbiter and bids from another SGA to inform the SOM detector that a connection has been made, and maintains the connection until it is no longer used. The connection logic directly controls the select lines of the downstream multiplexors, while the upstream multiplexors' select lines are controlled by

the SOM detectors. The connection logic enables the output data drivers in time for transmission of the second byte of the incoming message; the first byte, carrying routing information, is consumed by the SGA.

For a connection to be made, the following conditions must be met:

- The output channel must be not busy (outgoing frame signal low for two consecutive switch clock cycles).
- Some input channel must be bidding for the output channel; and if the output port is at high priority, then the bid must be high priority also.
- The paired SGA, that can also drive this output channel, must not have a competing bid for the same output channel that takes precedence over this SGA's bid. Bids of equal priority are resolved by random choice in the CNCT logic.
- The output channel must be enabled.
- The SGA must not be in the process of being reset.

After making a bid, the SOM detector remembers that a connection attempt is in progress. If the connection block does not inform the SOM detector that a connection is made, the SOM detector generates a reject — it asserts its reverse upstream output for one clock cycle.

Correspondingly, a reverse coming into the SGA from downstream is detected by the connection logic. The connection logic then forces the outgoing frame low for two switch clock cycles and frees up the output channel for other work. The SOM detector notices the downstream reject and passes it along upstream.

Pin Signals

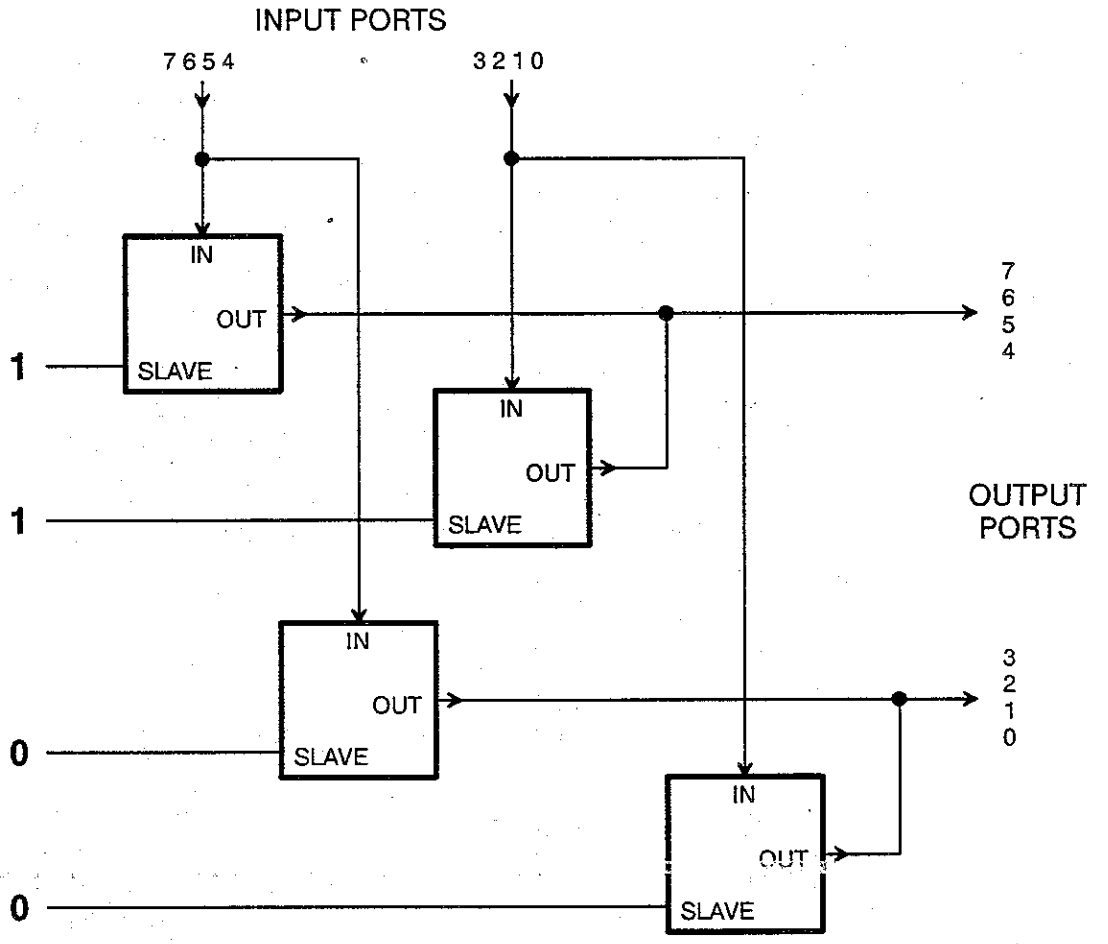
The SGA pins fall into six categories, listed below. When a signal is associated with a particular input or output channel, its name begins with "I" or "O" followed by the channel number, 0, 1, 2 or 3.

- Device control signals
- TCS interface signals
- Switch operation control signals
- Input channel data and control signals
- Output channel data and control signals
- SGA pair output channel arbitration signals

The signals in each category are described below. Except where noted, the high logic level is the asserted state and causes the action described.

Figure 3–25 SGA signals.

- Device control signals
  - ALLDIS — Input. Disables all but two output pins; enables those two (OROUT and DLYOUT), which are otherwise disabled.
  - MRESET — Input. Resets all SGA flops at the next clock high. Every flop is reset except the port disable latches of the TCS logic.
  - CLKMNS and CLKPLS — Inputs, differential pair, always the complement of each other. Synchronous clock used by all SGA flops.
  - TEMPIN and TEMPOUT — SGA junction temperature sensing diode. TEMPIN = diode anode, TEMPOUT = diode cathode.
  - OROUT — Output. The output of an OR gate whose inputs are all pins not otherwise testable for connectivity. Also see ALLDIS.
  - DLYOUT — Output. Delayed version of CLKMNS. Also see ALLDIS.
- TCS interface signals
  - CNTRLR — Input. Reset of port disable latches to the "all enabled" state.
  - CNTLDI — Input. Command data. Serial input to 13–bit command register.
  - CNTRLC — Input. Data input clock. CNTLDI is clocked into the command register on the rising edge of CNTRLC if CNTRLE is not asserted.
  - CNTRLE — Input. Command execute. A read is always performed, and a write is performed conditionally.
  - CNTRFRC — Input. Force the write command, even if disabled.
  - CNTRWD — Input. Disable the write command.
  - CNTLDO and NCNTLDO — Pseudo–differential output pair. Data from TCS interface. Use of the pair partially cancels SSO (simultaneously switching outputs) noise on Vcc within the SGA.
- Switch operation control signals
  - SLAVE — Input. Identifies to internal circuitry which set of output paths (0–3 or 4–7) can be driven by this SGA. This bit is compared to the most significant bit of the routing address of each message, to determine whether the message is requesting one of this SGA's output ports. See following illustration.



- RANDI0, RANDI1 and RANDIN — Inputs. Bits from a pseudo-random number generator, used by the output channel arbitration porcess to resolve simultaneous requests of the same priority level. RANDI0 and RANDI1 are used within an SGA. RANDIN arbitrates between the two SGAs that drive the same four output channels, so RANDIN must be opposite in value for the two SGAs of a pair. (Some swtch card documentation uses the name RANDOM for the SGA signal RANDIN.)
- HOLD — Causes each output channel to retain the high priority level message restriction after a high priority message arrives, until HOLD is de-asserted for two clock periods.

• Input channel data and control signals

- I0FRM, I1FRM, I2FRM and I3FRM — Inputs. *Frame* signal for input channels 0 through 3.

- I0RVRS, I1RVRS, I2RVRS and I3RVRS — Outputs. *Reverse* signal for input channels 0 through 3. Implemented as bidirectional, but input capability is used only for testing.
- I0D0, I0D1, I0D2, I0D3, I0D4, I0D5, I0D6, I0D7, I1D0, I1D1, I1D2, I1D3, I1D4, I1D5, I1D6, I1D7, I2D0, I2D1, I2D2, I2D3, I2D4, I2D5, I2D6, I2D7, I3D0, I3D1, I3D2, I3D3, I3D4, I3D5, I3D6, I3D7 — Bidirectional. Data lines for input channels 0 through 3.

- Output channel data and control signals
  - O0RVRS, O1RVRS, O2RVRS and O3RVRS — Inputs. *Reverse* signal for output channels 0 through 3.
  - O0FRM, O1FRM, O2FRM and O3FRM — Bidirectional. *Frame* signal for output channels 0 through 3. Wire OR'ed with corresponding pin of paired SGA. The input capability is used by internal logic to determine whether the channel is free.
  - O0D0, O0D1, O0D2, O0D3, O0D4, O0D5, O0D6, O0D7, O1D0, O1D1, O1D2, O1D3, O1D4, O1D5, O1D6, O1D7, O2D0, O2D1, O2D2, O2D3, O2D4, O2D5, O2D6, O2D7, O3D0, O3D1, O3D2, O3D3, O3D4, O3D5, O3D6, O3D7 — Bidirectional. Data lines for output channels 0 through 3.
- SGA pair output channel arbitration signals
  - O0LVL, O1LVL, O2LVL and O3LVL — Bidirectional. Priority level for output channels 0 through 3. Wire OR'ed with corresponding pin of paired SGA. See discussion below.
  - O0MBID, O1MBID, O2MBID and O3MBID — Implemented as bidirectional, but input capability is used only for testing. "My bid" outputs for output channels 0 through 3. Drives corresponding O0OBID, O1OBID, O2OBID and O3OBID inputs of paired SGA. The "my bid" pin for an output channel is asserted if this SGA has, for that channel, a bid whose priority is equal to the priority level of the channel.
  - O0OBID, O1OBID, O2OBID and O3OBID — Inputs. "Other bid" inputs for output channels 0 through 3.

SGA Pair Output Channel Arbitration

Each 8×8 crosspoint switch output channel is shared between two SGAs, either of which may connect one of its four input channels to the output channel. The SGAs must negotiate the connection setup. The signals "level", "my bid", "other bid" and RANDIN are used to arbitrate cases where both SGAs may have bids for the same output channel. We explain this arbitration from the

point of view of one SGA, "me"; the other SGA is "you". My problem is to decide whether I win the output channel.

If I presently have no bid, I of course do not win the channel. So assume I have a bid for it. I need to know how the priority of that bid compares to the priority of output channel.

The concept of output channel priority is presented earlier in this chapter, and the discussion there explains how an SGA determines channel priority. The SGAs of a pair must agree about the channel's priority. This is accomplished via the "level" signal. Normally, the channel is at low priority, indicated by a de-asserted level signal. When either of the SGAs receives a high priority bid for the channel, it asserts the level signal. Because the level pin is wire OR'ed between the two SGAs, both paired SGAs understand that the channel has gone to high priority. The level signal stays asserted until the first de-assertion of the *hold* signal. Both SGAs see the same hold signal, so they agree on when the level signal is de-asserted. The output channel's priority remains high until the *second* de-assertion of the hold signal. Both SGAs understand this, and know that the channel's priority remains high until that second de-assertion, even though the level signal is no longer asserted. Thus, the two SGAs are at all times in agreement on the priority of the output channel.

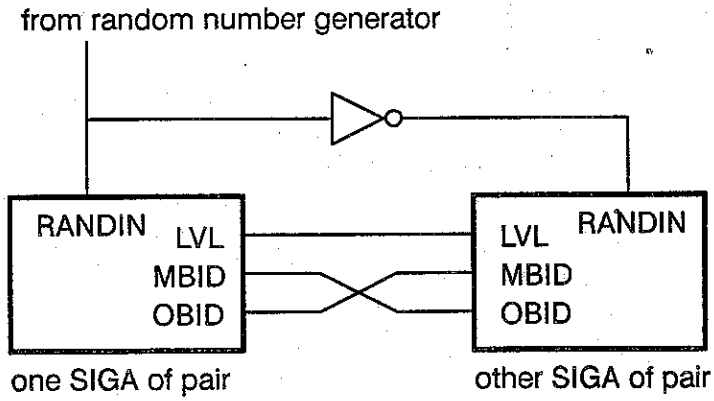
If the priority of the output channel is high, but the bid(s) I have for it are low, then I do not win the channel. So assume my best bid is of the same priority as that of the output channel. Either they are both low or they are both high. (Only in the special case noted below can the channel priority be low and my bid be high.)

The RANDIN pin tells me whether the random number generator chooses me in case of equal, conflicting bids between paired SGAs. If RANDIN says *me*, then I win the channel, regardless of whether the other SGA has a contending bid or not. So assume that RANDIN says *you*, the other SGA.

At this point, I will win the channel unless you also have a bid whose priority equals that of the channel. In that case, you will win. So I need to know whether you have such a bid. You tell me this by the signal that you call "my bid" and that comes into my input pin that I call "other bid". With this information, I now know who wins the channel, and therefore whether to accept or reject the bid I received. Because you need the analogous information about me, I supply it on my "my bid" pin.

Figure 3-26 shows the wiring.

Figure 3–26 Connection of arbitration signal pins.



NOTE

SIMULTANEOUS HIGH AND LOW BIDS
A high priority bid makes the channel priority high on the *next* bidding cycle, not immediately. The high priority bid does make the level output go high, but the signal used in arbitration is one clock period behind the level output and input. If high and low priority messages bid simultaneously for a channel that is initially at low priority, they compete on an equal footing and the random signals (RANDI0, etc.) determine which bid wins the channel.

Resetting the SGA

The MRESET pin clears all SGA edge-triggered flops. The majority of edge-triggered flops use a gate array macro (circuit component) that requires the clock to be asserted before the flop's reset will clear the flop. The suggested sequence is to assert MRESET while the clock is not asserted, assert the clock, release the clock, and release MRESET. Because of long delays associated with long reset nets in the SGA, MRESET should be de-asserted at least 20 nanoseconds before the clock pulse ends. MRESET clears all data flops, causing all data pins of all channels, both input and output, to be at the low voltage level. This insures there can be no bias driver overloading. (The way this works involves principles of ECL circuitry and gate array design, which is beyond the scope of this document.)

The MRESET pin does not affect latches used in the TCS interface to enable and disable individual channels. When the CNTRLR pin is brought high, these latches are reset to zero, enabling all channels. The action of CNTRLR is asynchronous; there is no need to cycle the clock with this operation.

Testability

[THIS SECTION MAY BE GENERIC TO ALL THREE CHIPS. IF SO, IT SHOULD BE MOVED TO A HIGHER LEVEL.]

The SGA, LCON and SIGA are designed for testing at the device, board and system level. At both the device level and the board level, three kinds of tests are made. *DC parametrics tests* check connectivity (shorted signals, open signals, and weak signals) and the ability to drive outputs both high and low. *Functional tests* check the pattern of output signals while input signals are stepped through an exhaustive sequence of test vectors. This set of vectors is designed to be a *toggle test*, which tests that each internal net (points wired together) can achieve both a high and a low state. Toggle test coverage is rarely 100%, but is made as high as practical. *AC parametric tests* check timing. At the board and system levels, *operational tests* check the device in actual operation.

During testing, the ambient temperature, the signal input levels and the power supply voltage levels may be varied. Burn–in is also a component of the test procedure.

Pins dedicated to testing, together with some of the testing capabilities of the TCS interface, support DC parametric testing. Those special pins are described here; the TCS chapter describes its use for this and other tests. The strategy for DC connectivity testing is to set all signal pin output drivers to a low logic level, then sense an externally applied stimulus on each input pin and force an externally observable high output level on each output pin. (Not all output drivers can be forced high; such pins are tested by sensing. The only output–only pins on the SGA are OROUT and DLYOUT.) Bidirectional pins are tested with either sensing, forcing or both.

Dedicated Test Pins on the SGA

When the ALLDIS pin is asserted, the connectivity of every pin not testable via the TCS can be observed via the OROUT and DLYOUT pins. On a bed–of–nails tester, this also tests the connectivity between each pin and the board etch. Further, all other output pins are disabled, allowing the board tester to back drive all the pins to detect shorts and opens in the board etch.

The approximate operating frequency can be determined by comparing the delay in the outputs OROUT, one of whose input gate inputs is the clock pin CLKPLS, and DLYOUT, a delayed version of the clock pin CLKMNS.

NOTE

OBSCURE BUG IN FIRST VERSION OF SGA
The first version of the SGA chip had an obscure bug. Various conditions triggered the bug, and the result was one port becoming unable to make connections until the chip was reset. One trigger is the simultaneous arrival at the SGA of the tail of a message going downstream, and a returning reject from upstream. It happens that none of the trigger conditions arise in a machine with four or fewer midplanes. In a machine with more than four midplanes, a work-around is to use the slotted strategy for all switch transmissions.

3.12.3 SIGA

Please refer to the SIGA specification, appendix A of this document, for details of the SIGA.

3.13 Analysis of Selected Cases

This section discusses specific situations that may arise in the switch. It includes both theoretical analysis and simulation results. Additional cases will probably be added in the future.

3.13.1 High Priority Bid and Hold De-asserted

There are subtleties associated with a message that arrives at an SGA during a clock cycle that hold is de-asserted. The output port of that SGA will not be set to high priority. This has two effects. First, suppose that the port is not in use. The high priority message will get the port, and proceed through the switch. Suppose it succeeds all the way through the switch. The path it takes will have high priority ports except at the one SGA in question. A subsequent high priority message on that path would bump the remaining low priority port up to high, but until it did, slightly different behavior could occur. In particular, low priority messages would be allowed on that port. It is unlikely that this would have any significant effect on switch performance in practice. This is especially true because the de-assertion of hold is intended to be fairly frequent, so the high priority track will be wiped out soon. The effect is innocuous because the high priority message did get through.

To consider the second effect, suppose the port is busy with a low priority message when the high priority message arrives at exactly the same clock cycle as hold is de-asserted. Then the high priority message will be rejected as usual, but the port will not be set to high priority. Therefore, further low priority traffic will be allowed on the port, so retransmissions of the high priority message may again have to contend with low priority traffic at this same SGA port. This undermines the intent of the priority mechanism, namely, that each at-

tempt into the switch will set at least one more switch port along the path to high priority. If the high priority message arrives one clock cycle earlier or later, this problem case does not occur. Although this case is statistically rare, it complicates the calculation of the bound on switch latency. A similar case arises when the high priority message gets through this SGA but is rejected further along.

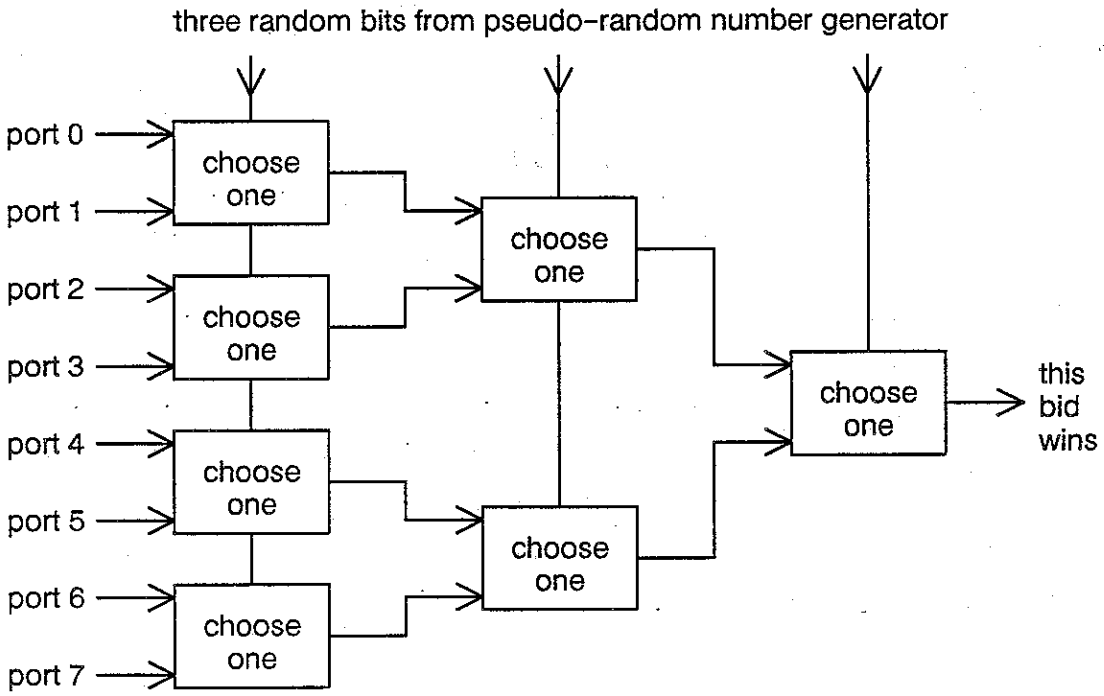
3.13.2 Bid Arbitration Unfairness

The original implementation of the arbitration among messages bidding simultaneously for one output port is not totally random, but is slightly biased. This unfairness arises in two ways, one due to the maximum sequence generator and one due to hierarchical selection of the winning bid.

The maximum sequence generator is seven bits and has one dead state, so in use it takes on 127 values. Three of its bits are used for bid selection in the arbiter. During a period of the generator, these three bits will take on seven of the possible eight values sixteen times, and will take on the eighth value only fifteen times. Thus, one of the eight input ports is slightly discriminated against, by a ratio of 16 to 15. That port is the one selected by the generator's dead state. The dead state is all ones, so the port selected by all ones is discriminated against.

The second unfairness is as follows. Selecting the winning bid with a one–out–of–eight selector circuit would be fair. The original SGA design, however, selects the winning bid in a hierarchy of successive one–out–of–two selector circuits. Within an SGA chip, input ports 0 and 1 contend with each other, and ports 2 and 3 separately contend with each other. The winner of each pair then contends with the winner of the other pair. Finally, the winner for this SGA chip contends with the winner from the paired SGA chip. Figure 3–27 shows this play–off hierarchy. Input ports 0–3 are served by one SGA of the pair, and 4–7 by the other SGA. The maximum unfairness arises when there is only one bid on one SGA's input ports and a bid on each of the four input ports of the other SGA. The lone bid on one SGA is guaranteed to get through that SGA, and ultimately succeeds with a probability of 50%. The other four bids split the remaining 50% equally, each succeeding 12.5% of the time on the average.

Figure 3-27 Unfair competition in hierarchical arbiter.



These unfairness effects arise very seldom. They occur only when message bids collide, contending for one output port at the same time. Further, if the output port is at high priority, any low priority bids are rejected and do not contend for the port. Experience with effects such as these in earlier models of the Butterfly family of computers strongly suggests that these effects will not hamper performance and will be detectable, if at all, only by software written specifically to stress the switch.

4

The Butterfly II Memory System

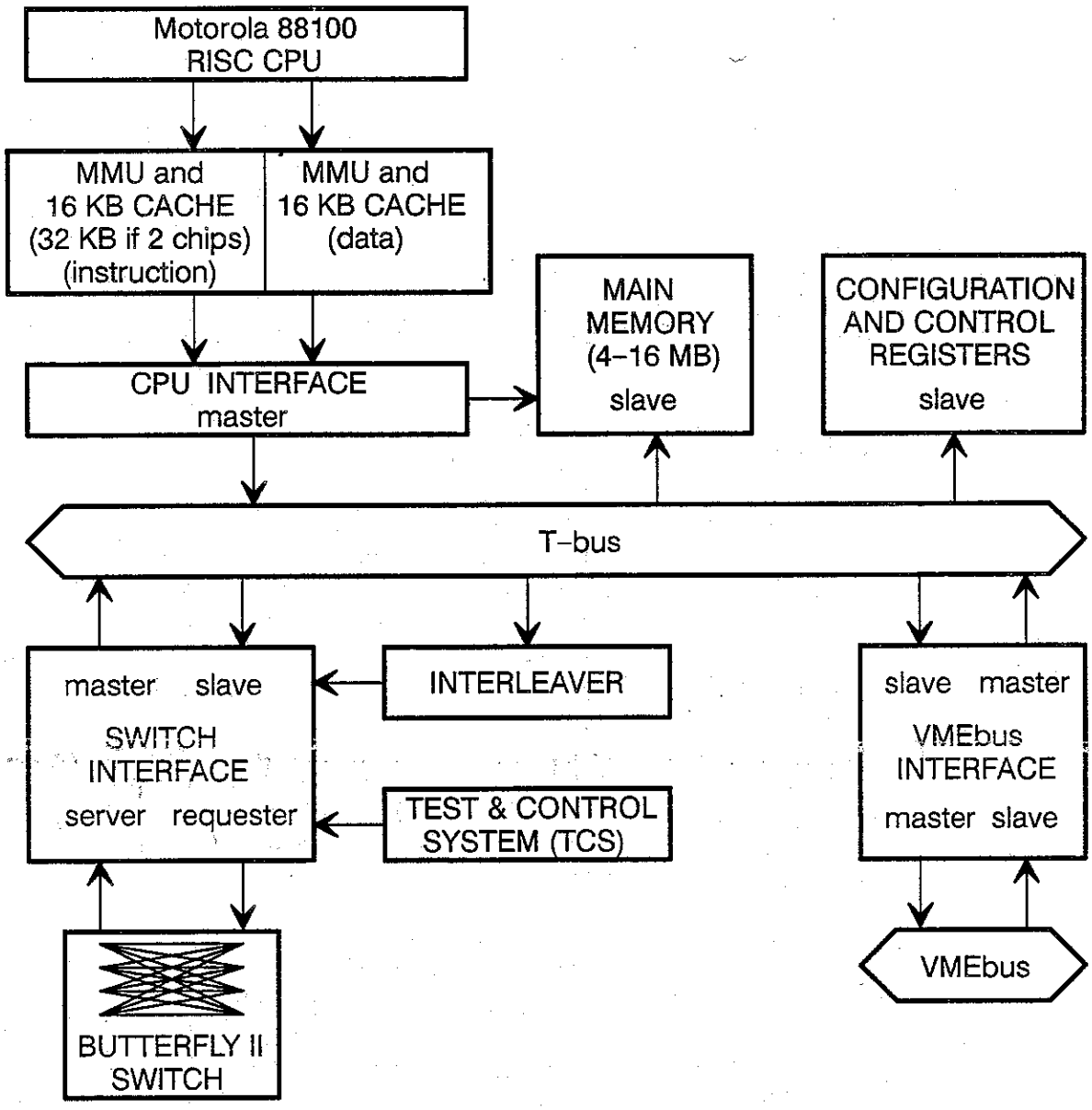
4.1 General Characteristics

The Butterfly II computer has a 34–bit global address space, providing addressing capability for up to 16 gigabytes. All addresses are byte addresses. Butterfly II main memory responds to byte, halfword and word reads and writes. Main memory resides on function boards. One such function board is the B2VME, which combines memory, a processor, and a VMEbus interface that provides windows between VMEbus memory and the Butterfly II global address space.

The address space seen by processes is mapped into the global address space of the machine, first by the MC88200 cache and memory management unit (CMMU) and then by specialized Butterfly II hardware. The Butterfly II supports demand paging via the CMMU.

Figure 4–1 illustrates components in the Butterfly II system that generate, transform or respond to addresses. The figure uses the B2VME as a basis. The principal originator of addresses is the CPU. VMEbus master devices may also generate addresses sent into the Butterfly II machine, and the Test and Control System generates addresses as part of its control and monitoring activity. The component that responds to the most addresses is memory. Registers are accessed by their address. VMEbus slave devices may also occupy addresses, when they respond to Butterfly II requests.

Figure 4–1 Address flow.



Addresses are transformed in five places. Addresses generated by a CPU are first mapped by either its instruction CMMU or its data CMMU, depending on the type of reference the CPU is making. Then the address is transformed by the CPU interface, and placed on the T–bus or sent via the fast path to local memory. An address to which the requester SIGA responds may be taken directly from the T–bus, or part of it may be transformed by the Interleaver. The other two address transformations occur in the VMEbus interface, where addresses on the T–bus are mapped to addresses on the VMEbus, and vice versa.

The remainder of this chapter discusses how the CPU accesses memory, and how the memory system responds to accesses from any source. The VMEbus interface and its mapping is described in the B2VME chapter and is not discussed further here. The TCS also is covered in a separate chapter. The address map — describing where memory, registers, interfaces, etc. appear in the address space of a function board — is covered in the chapter dealing with the specific function board (e.g., B2VME).

4.2 Local versus Remote Access

Butterfly II resources are called *local* or *remote* depending on whether they are on the same T-bus as the device that originated the access. Being on the same T-bus implies being on the same function board. A local access references a resource on the same function board, and therefore does not involve switch traffic. A remote access must be sent over the switch, and consequently experiences a delay while the request is sent and the response is received. Any access using the switch is called remote — even if it happens that the resource resides on the same function board as the originator, but is addressed via the switch rather than locally.

When a transaction is presented to the T-bus, the resource addressed is specified both by the global address (T_AD<33..0>) and by two path bits (T_PATH<1..0>). The T-bus path bits specify one of three paths the access will take: local, over one switch, or over the other switch. *Local* permits response only from devices on the local function board. Selection of one switch or the other indicates which SIGA should respond to the transaction. (A machine may be configured without a redundant switch.) The T-bus master that initiates the transaction drives both the T-bus address bits and the T-bus path bits.

The memory subsystem on a function board responds if the T-bus path bits specify a local access and the T-bus address is within the range it implements. Other local T-bus devices, such as control and configuration registers or the VMEbus interface on a B2VME, respond if the access is local and within the address range they implement.

If the T-bus path bits specify switch access, the T-bus address bits are sent into the switch. Nine of these bits are consumed in routing the transaction message to a remote switch port, where the remaining 25 address bits are delivered to the T-bus there (and zeros replace the nine bits). In the transaction at that remote function board, its T-bus path bits always specify a local access. Some device local to that remote T-bus will respond (or a timeout error will occur).

The SIGA server always sets T_AD<33..25> to zero and T_PATH to "local".

In the CPU interface, the T-bus path bits are derived from a computation based on three sources: the *path* bits of the Process Configuration register

(PCR), the *local* bit of the CPU Mapping RAM (CMR), and the *interleave enable* bit of the CMR. The T-bus address bits depend on the Physical Address produced by the CMMU, and the mapping performed by the CMR. How the CPU interface computes the T-bus address and path bits are described in the B2VME chapter.

Limit on Local Address Space Size

The Butterfly II address space of 16 gigabytes is spread out across the function boards connected via the switch. Each switch address is limited to 32 megabytes of the total address space. Sometimes the need for a large physical memory, but only a moderate number of processors, arises. A natural idea is to place a large amount of memory on one or a few function boards. The 32-megabyte limit prevents this. A brief explanation and justification of the limit is as follows.

The address produced by the CMMU is 32 bits, and is interpreted as two components, one specifying switch routing and one giving the address local to that switch port. In theory, the hardware could be configurable to set that boundary between the two components at any place in the address, maximizing the local address space size. In practice, this would lead to more complex hardware and to program optimizing more dependent on machine size.

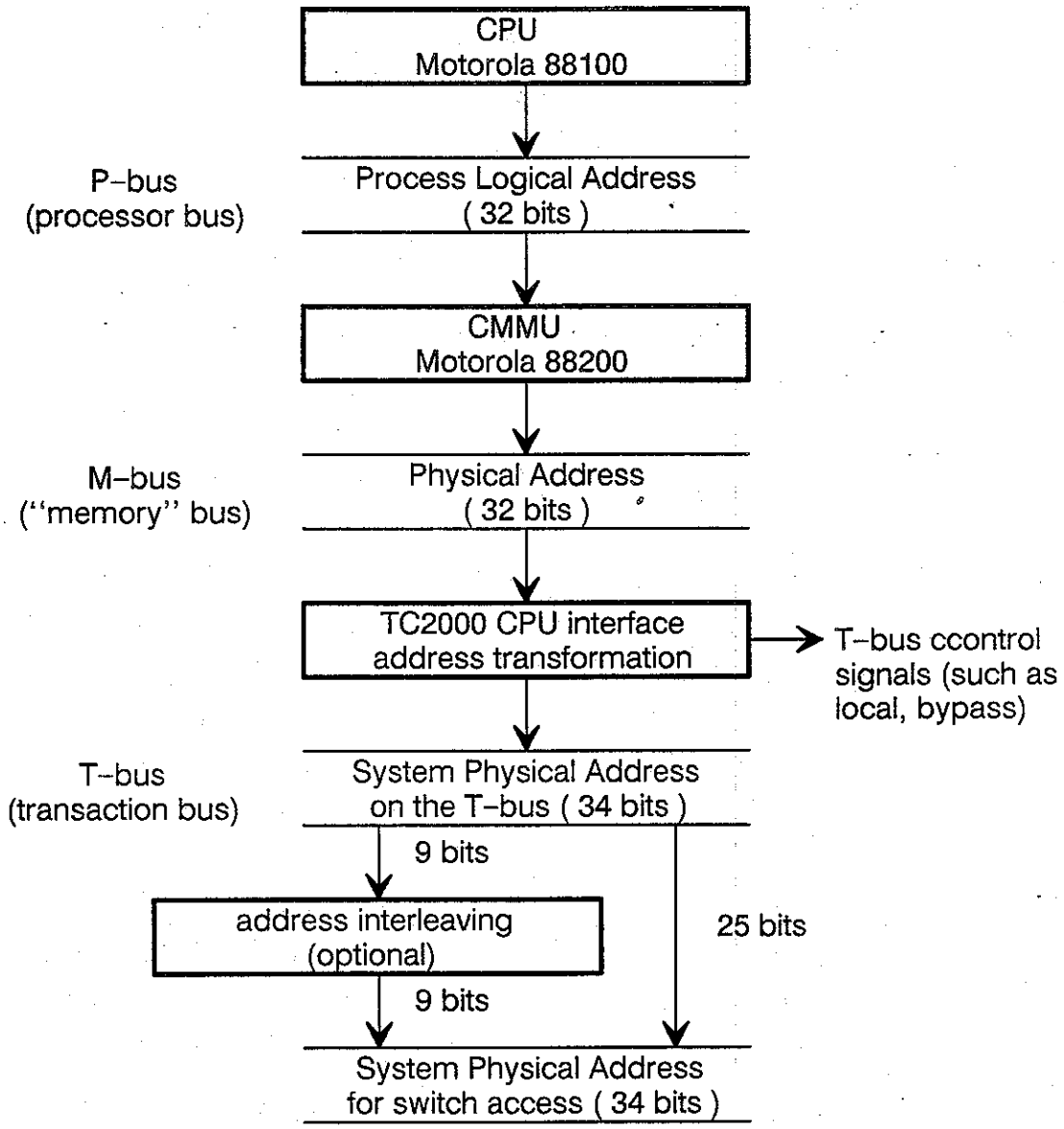
4.3 Private versus Globally Accessible

The operating system can impose access restrictions on a page-by-page basis through use of the CMMUs. The Butterfly II architecture does not impose any absolute constraints on accessibility; the entire global address space is accessible to any CPU if mapping and permissions are set up appropriately. Other T-bus masters have similar unrestricted access, although they may be implemented with constraints particular to the device, such as windowing in the VMEbus interface.

4.4 Address Formats

The path from CPU to memory uses three addresses formats. Figure 4-2 summarizes the path of address transformation for every address generated by the CPU. Each format is described below.

Figure 4–2 Addressing from CPU to T–bus and switch.



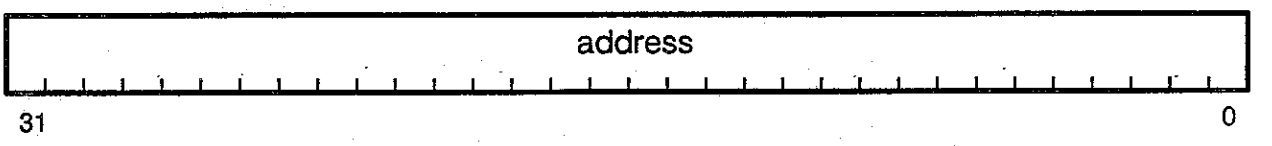
4.4.1 Process Logical Address

The Motorola MC88100 CPU operates in the virtual address space of 32–bit *Process Logical Addresses*. Examples of a Process Logical Address are an address stored in a CPU register, or a pointer in the C language. The user's program sees only Process Logical Addresses, and hence a 32–bit virtual address space. Therefore, many of the details in the following discussion are not important to the application–level programmer. Careful use of the page transla-

tion mechanisms in the CMMU and CPU Mapping RAM hides as much of these details as desired.

The Process Logical Address is a 32-bit field with no defined internal structure. See the Motorola *MC88100 User's Manual* for further discussion of the CPU operation and its use of addresses.

Figure 4–3 Process Logical Address format.



4.4.2 Physical Address

The 32-bit Process Logical Address generated by the CPU is transformed by a Motorola MC88200 CMMU into a 32-bit *Physical Address*. One CMMU transforms data addresses, and instruction addresses are transformed by a second (or an optional third) CMMU. The Physical Address is largely unstructured, but two points deserve mention.

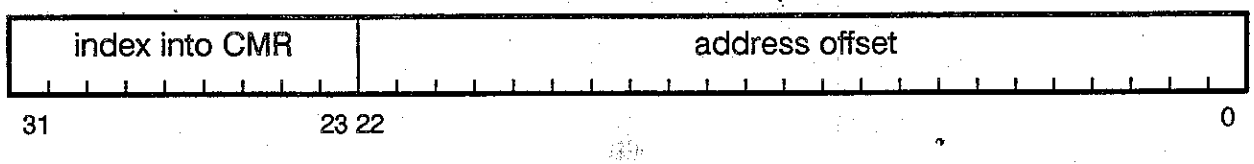
First, the CMMU implements a 4-kilobyte page size. The Butterfly II hardware is built around around an 8-kilobyte page size. The 8-kilobyte size is wired into the design of the VMEbus master and slave mapping hardware (described in the B2VME chapter) and into the Interleaver (described later in this chapter). The differing page sizes are handled in system software, by allocating two adjacent 4-kilobyte pages each time a new page is needed.

Second, the CMMU has a fixed, one-to-one mapping for the top one megabyte of supervisor address space, called the *control memory address space*. This area is intended for memory-mapped peripherals and I/O devices. Within this top one megabyte, four kilobytes are diverted to address the CMMU's internal registers. Accesses to the remaining 1020 kilobytes are passed through the CMMU as a Physical Address. Thus, the top one megabyte has a 4-kilobyte "hole" that cannot be used to access locations elsewhere in the Butterfly II machine. When the machine is powered up, a hardwired circuit determines where this hole lies by initializing a register within the CMMU. Because there are two (or three) CMMUs per CPU, there are actually two (or three) 4-kilobyte holes in the top megabyte of supervisor mode Process Logical Address space.

For more information on the Physical Address and operation of the CMMU, see the Motorola *MC88200 User's Manual.*

The CMMU places little constraint on how the Physical Address it produces is interpreted. The CMMU is designed so the Physical Address could be presented directly to a memory system. Only the two points mentioned above relate to the CMMU's concept of Physical Address format. In the Butterfly II, however, the Physical Address is understood to have fields with particular meaning. These fields are used by the CPU interface to transform the Physical Address into a System Physical Address. Figure 4–4 shows the format of the Physical Address as interpreted by the Butterfly II hardware.

Figure 4–4 Physical Address format.



As Figure 4–4 shows, the Physical Address high nine bits (31..23) are an index used to select an entry in the CPU Mapping RAM (CMR). The address offset field, composed of the remaining 23 bits (22..0), specifies a location in a space of eight megabytes. Thus, this format can *directly* address $512 \times 8 = 4096$ megabytes.

As described later, the transformation of Physical Address into System Physical Address permits *indirect* access, by changing the mapping, to additional address space. This results in a total address space of $512 \times 32 = 16{,}384$ megabytes.

4.4.3 System Physical Address

The *System Physical Address* is unique to the Butterfly II architecture. It is 34 bits. The CPU interface transforms the 32-bit Physical Address into the 34-bit System Physical Address, and in the process also produces other address-related signals. The System Physical Address from the CPU interface is placed on the T-bus. It is important not to confuse the Physical Address with the *System* Physical Address.

- The **Physical Address** exists only going from a CMMU to the attached CPU interface. Physical Address is a term employed by Motorola in their CMMU literature. It is 32 bits.
- The **System Physical Address** exists on the T-bus and in the switch. It is the common language by which all T-bus master devices address all T-bus slave devices. System Physical Address is a term defined by Butterfly II designers. It is 34 bits.

Figure 4–5 System Physical Address format.

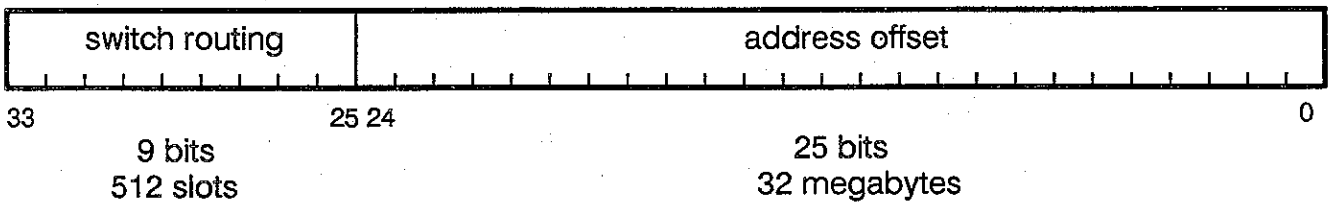


Figure 4–5 shows the System Physical Address format and its two fields. The switch routing field specifies the path through the switch, and therefore one of 512 slots. The address offset field of the System Physical Address addresses a total of 32 megabytes per switch port. The address space of the System Physical Address is 512 × 32 megabytes, or 16 gigabytes.

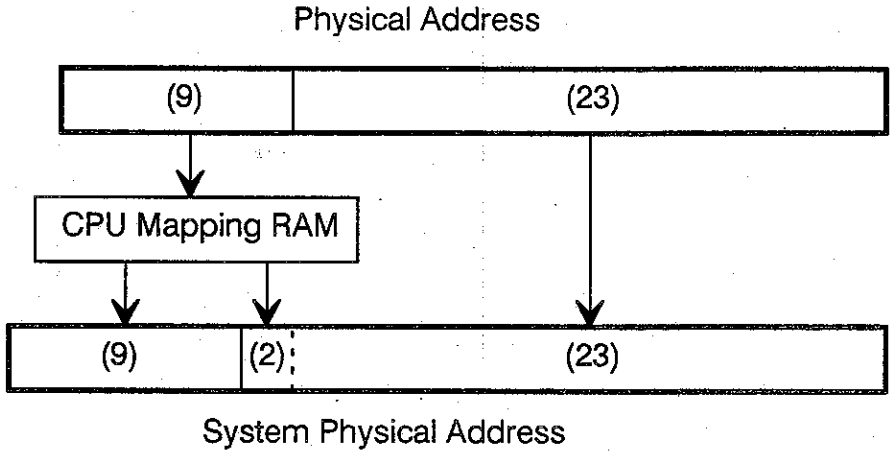
The System Physical Address does not contain any explicit indication of whether the address is on the local function board. This distinction is carried by two other T-bus bits, T_PATH < 1..0 > . These bits specify local access (that is, not over the switch), remote access via a SIGA (SIGA A), remote access via the other SIGA (SIGA B and the redundant switch), and one illegal value. The SIGA (A or B) determines whether to service a T-bus request by the T_PATH bits, not by any of the System Physical Address bits (T_AD < 33..0 >).

Most address transformations in the Butterfly II convert one format of address into another. There are two exceptions to this. In each case, the address before and after the transformation is a System Physical Address. One transformation is interleaving, which is described in a later section of this chapter. The other is the processing of an address as a message traverses the switch. When it enters the switch, part of the address specifies the routing through the switch, and thereby specifies the destination port that will receive the message. As the message passes through each column of the switch, parts of this routing information are stripped off. At the destination, the server SIGA replaces the discarded bits with zeros.

4.4.4 Address Transformation in the CPU Interface

The CPU interface receives a Physical Address from one of the two or three CMMUs, and produces a System Physical Address that is placed on the T-bus. Figure 4–6 shows this transformation,.

Figure 4–6 **Physical Address to System Physical Address.**



The high nine bits of the Physical Address select an entry in the CPU Mapping RAM. This entry supplies the high eleven bits of the System Physical Address. These eleven bits comprise nine bits that specify switch routing, and two bits that are the top bits of the address offset at the switch port addressed; these two bits are sometimes called the *bank* bits, since they select one of four 8–megabyte banks of address space. The low 23 bits of the Physical Address are copied unchanged into the low 23 bits of the System Physical Address, where they specify a byte offset within the addressed bank. The details of the CPU Mapping RAM are discussed in the B2VME chapter.

Access Control Bits from CPU Interface

Besides the transformation of Physical Address to System Physical Address, the CPU interface provides several signals that describe or control the access being made. These bits are:

- *local* — Used in determining whether the access is to local resources (on this function board) or remote resources (over the switch).
- *bypass* — Indicates whether the access should bypass the machine's memory module locking protocol.
- *interleave enable* — Used in determining whether the access is to interleaved memory; applies only to remote references.
- *intercept access** — Controls a mechanism to fake a response to the access, used in speeding up certain block transfer operations. The "*" is part of the bit name, indicating that "0" asserts the action.
- *fast path disable* — Prevents the accelerated response by local memory, normally available by a mechanism called the "fast path".

These access control bits are not strictly a part of the address, and are described in detail in the B2VME chapter.

4.5 Interleaving

The discussion of interleaving begins with introductory discussion that the advanced reader may wish to skip.

4.5.1 Overview of Interleaving

Interleaving is a form of address mapping. A range of addresses that the CPU, and therefore the program, sees as contiguous is mapped into several sub–ranges that lie in separate memory subsystems. The first several addresses go to one memory subsystem, the next several addresses to a different memory subsystem, the next several to yet another, and so on. Often there are more sub–ranges than there are interleaved memory subsystems, so the mapping eventually cycles back through the memory subsystems and maps many sub–ranges to each memory subsystem.

4.5.2 Motivation for Interleaving

Why Interleave At All?

The programmer does not see any direct effect of interleaving. The purpose of interleaving is to improve efficiency, thus speeding computation. This improvement depends on the pattern of memory references being somewhat sequential, a hypothesis which is very well met by the instruction fetches of almost all programs, and is usually moderately well met by their data fetches as well. This is called *locality of memory access*. Programs often refer to elements of a data structure that resides in contiguous locations of virtual address space. For instance, indexing through array elements is a common operation.

As a counterexample, a program segment that fits entirely in the cache, making memory references at random (such as to a hash table), derives no direct benefit from interleaving. In a multiprocessor environment such as the Butterfly II, however, even this program segment will probably execute faster due to the *indirect* benefit of reduced switch and memory contention and congestion.

Historically, an early use of interleaving was to improve the effective speed of memory. Large memories, such as banks of magnetic core memory, were relatively slow compared to the CPU speed achievable. Interleaving allowed the next memory reference to begin while the previous reference, to a different memory subsystem, was completing. This application used fine grain interleaving; typically, each successive word address was mapped to a different memory subsystem. This was often implemented by taking some of the low bits of the virtual address and moving them to a high position in the physical address.

Why Interleave This Way?

The Butterfly II interleaving is motivated by other concerns and is implemented in a different way. Several processes may be accessing data structures that are stored near each other in a shared, virtual address space. Without interleaving, these data structures would usually reside in nearby physical locations, usually in the same memory subsystem. When the memory subsystem is servicing an access request from one process, other access requests to it will be delayed. The memory subsystem becomes a bottleneck. Also, switch paths converging on the memory subsystem's server port will become heavily loaded as access requests are rejected and retried.

The crux of this bottleneck is that several popular data structures all lie in one memory subsystem. The solution is to spread out the data among several memory subsystems. The access pattern is then spread out, and the performance approximates average behavior instead of "hot spot" behavior.

The amount of data mapped to each memory subsystem in turn — the interleaving granularity — need bear no relation to the actual size of the data structures the processes are referencing. The Butterfly II interleaver maps each clump of 16 bytes to a different switch port, and therefore a different memory subsystem. This scatters the data across many memory subsystems.

The 16–byte clump size is well matched to the maximum switch message data size, also 16 bytes. If a single switch message could access more than the interleaving clump size, then such an access to interleaved memory would necessarily refer to more than one switch port.

In the historical interleaving, it was common for the entire virtual address space to be interleaved. On the Butterfly II, interleaving is performed on the basis of quad–pages. Each quad–page (4×8 kilobytes) may be entirely interleaved or entirely non–interleaved. The quad–page–based interleaving permits flexibility in the use of interleaving. Butterfly II interleaving is more like memory mapping than the historical interleaving, both in implementation and use.

In the Butterfly II, some memory subsystems may be busy with other tasks. Some may be physically not installed, or be configured out of service. The number of memory subsystems may be different than when the system was last brought up, and may often not be a power of two. All of these variations are accommodated well by the page–based, mapping style of interleaving, and would be very difficult to accommodate with interleaving that relied simply on a shuffling of bits within the address.

4.5.3 Uniform Use of Butterfly II Interleaving

The Butterfly II interleaving design permits considerable flexibility in use. Each switch interface has associated with it its own copy of the interleaving

hardware, and each T-bus master has its own capacity for deciding whether a given access is to an interleaved page.

The hardware was designed with machine-wide uniformity of interleaving in mind. The idea is that all interleavers will be set up identically. Also, that all hardware that decides whether a given page number is interleaved or non-interleaved will be set up identically. We suggest that the reader approach Butterfly II interleaving with this model in mind, and that the programmer consider using interleaving this way to avoid complexities in coding and debugging.

Various deviations from this uniform model are possible. Some would produce very peculiar effects and should rarely if ever be used, while others may be valuable in certain applications. An example of very strange use of interleaving is to access a given physical page sometimes as interleaved and sometimes as non-interleaved. This has the effect of scrambling and/or hiding portions of the data. While one might imagine a sorting or hash-coding algorithm based on this "use" of interleaving, it's more of a misuse.

A second example of peculiar use of the interleaver is to permit access to slot zero from the CPU. As described earlier, the CPU interface interprets a Physical Address that has bits 28..23 all zero to be a local memory reference. Ordinarily, these bits must be part of the switch routing, so slot zero is inaccessible from the CPU. For remote access to an interleaved page, the interleaver replaces these bits with a new value, so the interleaver could be programmed to supply a switch routing field of zero for certain "interleaved" pages. This would be weird and is not advised.

As a third example, in an application where the Butterfly II is serving several users, each user is probably allocated a number of processors for the user's exclusive use. The operating system might restrict the user's access to memory by preventing access to physical pages outside the user's cluster of processors. If the machine is thus partitioned, the operating system could set up a given page number as interleaved within that cluster, but non-interleaved in the rest of the machine. This example is more of theoretical interest than practical use, because of difficulties in implementation.

Again, we suggest that the standard use of Butterfly II interleaving is uniform. If a given page number is interleaved for any access, we suggest it be interleaved for all accesses in the machine.

4.5.4 Implementation of Butterfly II Interleaving

Overview as a "Black Box"

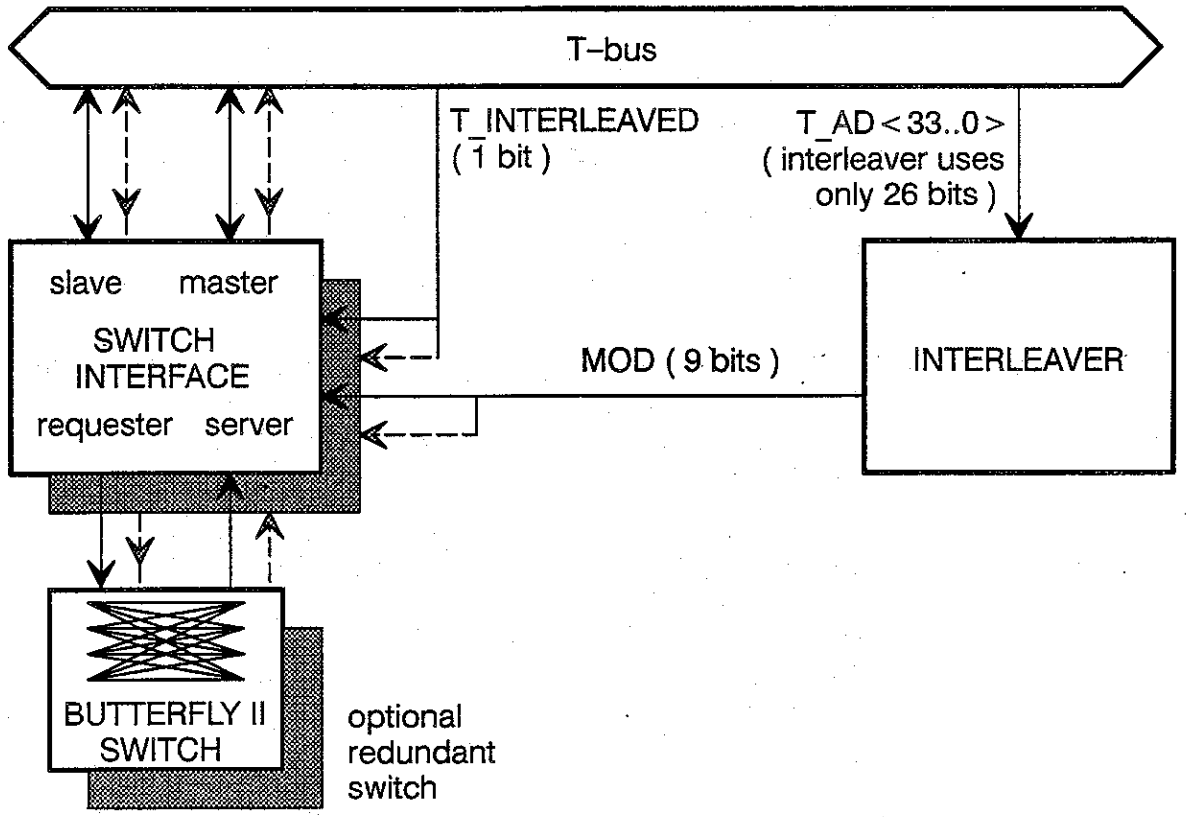
The interleaver input is most of the address bits on the T-bus, therefore a 34-bit System Physical Address. The interleaver output is nine bits, all of

which are inputs to the SIGA. For the switch route portion of the address, the SIGA uses either T-bus bits T_AD<33..25> or the nine bits from the interleaver. A 1-bit control signal associated with the T-bus, T_INTERLEAVED, tells the SIGA which source to use.

Figure 4-7 shows how the interleaver fits into the Butterfly II design. Its only action is to change the switch routing used by the SIGA, based only on the T-bus address. Therefore, the following points are true.

- Interleaving applies to remote (over the switch) references only. Local references are not interleaved. References to memory on the same function board may be interleaved only if they access that memory over the switch.
- The effect of interleaving is to change the switch port to which an access is directed. The 25 bits of local address are unchanged; the access will have the same local address at the new switch port as it would have had at the other switch port without interleaving.
- T-bus masters that originate accesses over the switch, such as the CPU interface and the VMEbus interface in the B2VME, should drive the T_INTERLEAVED signal consistently. Unusual — and probably undesired — results would ensue if one T-bus master accessed a page as interleaved, and another T-bus master accessed the same page as non-interleaved.

Figure 4–7 Overview of the interleaver.



On the B2VME, the interleaver drives its 9–bit output to both SIGAs in parallel, so the interleaving they see is the same. Even if both SIGAs are active, only one SIGA requester will use those signals because the T–bus T_PATH bits specify only one or the other SIGA.

Internal Workings of the Interleaver

The interleaver contains two RAMs and an adder. Its organization is shown in Figure 4–8. T_AD < 33..25 > and T_AD < 12..4 > provide the two 9–bit inputs to an adder. T_AD < 33..28 > supply the high six address bits, and T_AD < 22..15 > the low eight address bits, to the pool RAM. The address supplied to the pool RAM selects one of 16,384 3–bit locations. The three bits from the selected location drive the high three address bits of the modulus RAM, and the low ten bits of its address come from the adder. This selects one of 8,192 9–bit locations in the modulus RAM. The nine bits from that location are the output of the interleaver. They go to the SIGA.

Figure 4–8 Interleaver internal processing.

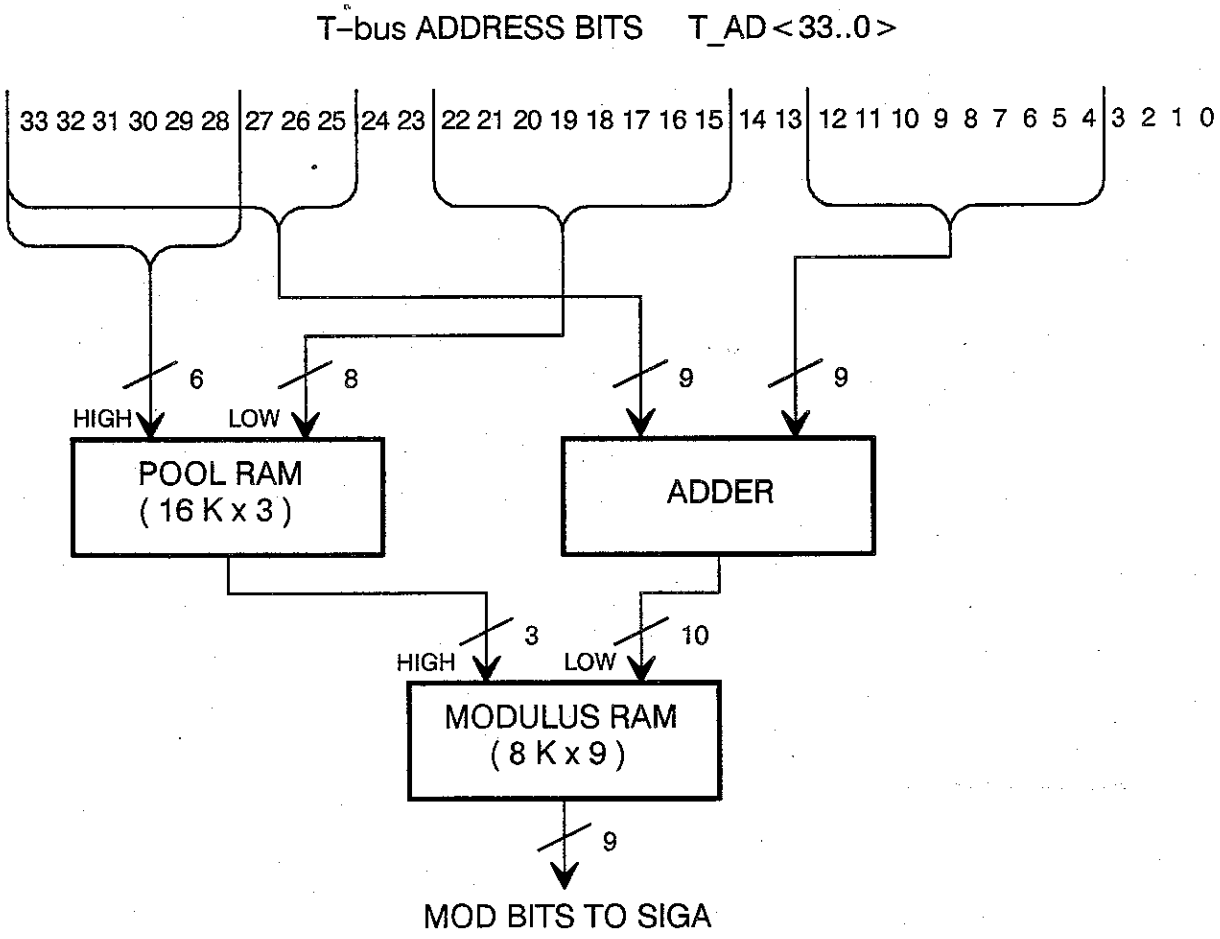
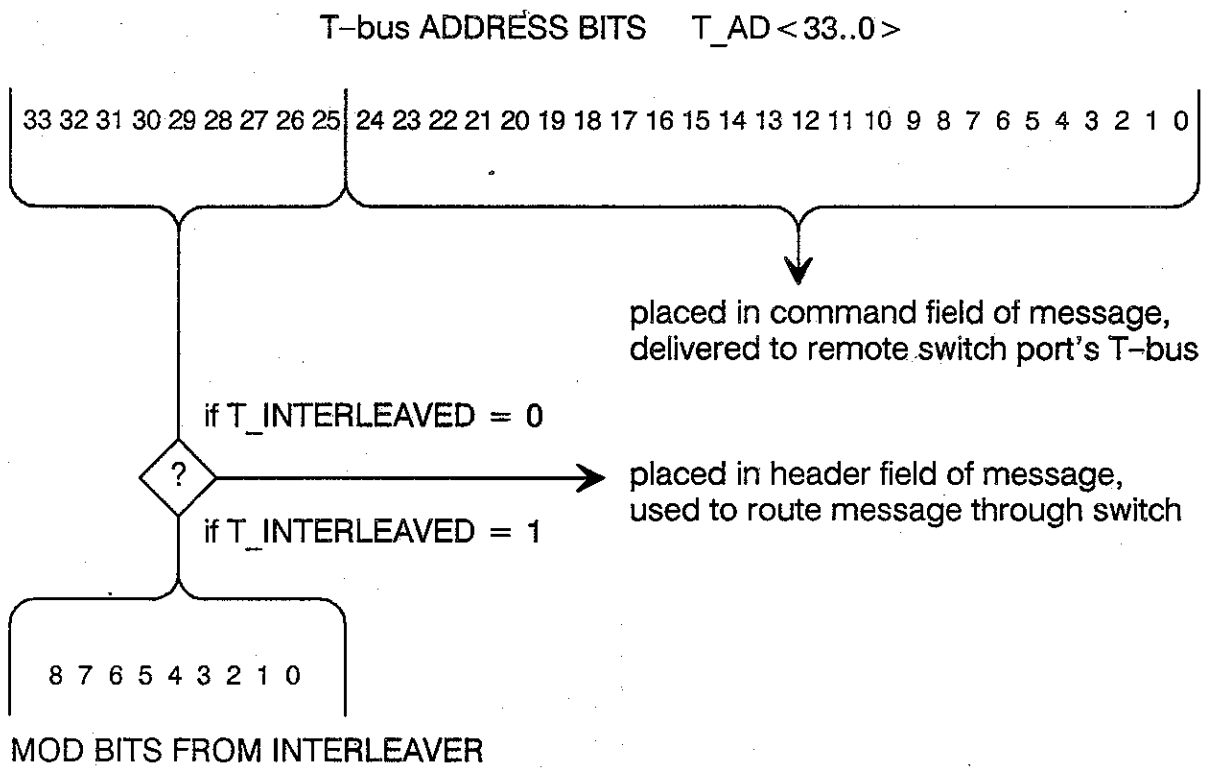


Figure 4–9 illustrates how the SIGA uses the T-bus address bits, the MOD bits from the interleaver, and the T_INTERLEAVED bit. However, for the SIGA to use interleaving at all, interleaving must be enabled by setting a bit in the SIGA's Requester Configuration B register. This bit is described in the *SIGA Specification*.

Figure 4–9 Interleaving processing in the SIGA.



The CPU interface derives T_INTERLEAVED from its Interleave Decision RAM, gated with the *interleave enable* bit in the CPU Mapping RAM entry used by the access.

The VMEbus interface derives the bank bits and also the T_INTERLEAVED bit from its selected VMEbus Slave Map RAM register, independently. The intended use is that software will set up this RAM such that T_INTERLEAVED is never asserted on banks other than "00", although VMEbus interface hardware does not restrict interleaving to only bank "00".

4.5.5 Conceptual Operation of the Interleaver

The interleaver hardware was designed with a particular model of interleaving in mind, although the design is relatively general and could be used in various ways. This section describes the intended use of the interleaver and motivates the hardware design. We examine each task the interleaver performs, identifying the hardware that implements that task.

Mapping Clumps to Switch Ports

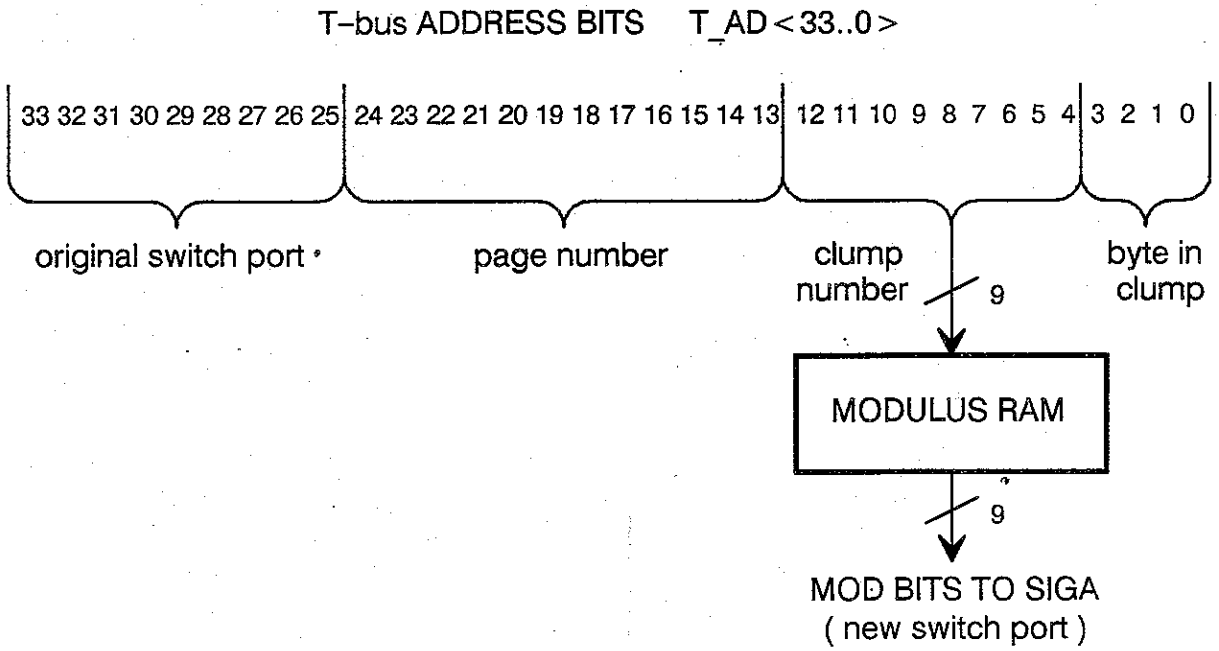
The primary task of the interleaver is to distribute different clumps of bytes within a page to different memory subsystems. The switch port identifies the memory subsystem, so the interleaver maps each successive clump to a different switch port.

The interleaver operates on the System Physical Address. For all addresses within a page (eight kilobytes), the high 21 bits of the System Physical Address are the same. Therefore, at this most basic level, the high 21 bits are not relevant to the interleaving operation. Only the bottom 13 bits change, so only they can contribute to the interleaver's mapping of address to new switch port. (In actual use, groups of four pages, "quad-pages", are the unit that software normally allocates for interleaving. This is because of how the pool RAM operates, which is discussed later.)

Further, all 16 bytes of each clump should be mapped to the same memory subsystem, so the low 4 bits of the System Physical Address do not contribute to the interleaving. *Only System Physical Address bits 12..4 drive the* ***basic*** *interleaver mapping*. These bits identify the clump within the given page, and therefore are called the *clump number*.

The basic task of mapping the clump number to a new switch port is performed by the modulus RAM. The clump number supplies the address to this RAM, and the contents in the selected location are the new switch port where that clump of addresses in the interleaved page resides. Figure 4-10 shows this basic task.

Figure 4–10 Mapping clumps to switch ports.



The modulus RAM could be programmed to map each of the 512 clumps in the page to any "random" switch port. Indeed, this flexibility is important, because only certain of the possible 512 switch ports may be available when interleaving is set up, typically during system initialization. The modulus RAM is intended for use in a more structured way, however. When interleaving is set up, the software should decide which switch ports are available to supply clumps of memory in the interleaved page. These switch ports constitute a *pool* of switch ports. The modulus RAM assigns clumps to each of the switch ports in turn, round robin fashion, recycling through the pool as many times as necessary.

Usually there will be fewer than 512 switch ports in the pool; suppose there are six. The first six clumps are mapped to different switch ports. The seventh clump is mapped to the same switch port as the first clump. The eighth clump is mapped to the same as the second clump. And so on. Figure 4–11 shows this use of the modulus RAM. In this example, the modulus RAM computes the function, port = clump modulo 6. Hence the name, *modulus* RAM.

Figure 4–11 Modulus RAM use — example 1.

```
clump 0 ⇒ port 0     clump 6 ⇒ port 0     clump 12 ⇒ port 0    ...
clump 1 ⇒ port 1     clump 7 ⇒ port 1     clump 13 ⇒ port 1    ...
clump 2 ⇒ port 2     clump 8 ⇒ port 2     clump 14 ⇒ port 2    ...
clump 3 ⇒ port 3     clump 9 ⇒ port 3     clump 15 ⇒ port 3    ...
```

```
clump 4 ⇒ port 4     clump 10 ⇒ port 4     clump 16 ⇒ port 4     ...
clump 5 ⇒ port 5     clump 11 ⇒ port 5     clump 17 ⇒ port 5     etc.
```

In fact, the switch port numbers produced by the modulus RAM need not be the numbers 0 through 24. Suppose that the switch ports with available memory were every third port starting with port 10: 10, 13, 16, 19 and so on. Figure 4–12 shows how the modulus RAM would be set up.

Figure 4–12 Modulus RAM use — example 2.

```
clump 0 ⇒ port 10    clump 6 ⇒ port 10    clump 12 ⇒ port 10    ...
clump 1 ⇒ port 13    clump 7 ⇒ port 13    clump 13 ⇒ port 13    ...
clump 2 ⇒ port 16    clump 8 ⇒ port 16    clump 14 ⇒ port 16    ...
clump 3 ⇒ port 19    clump 9 ⇒ port 19    clump 15 ⇒ port 19    ...
clump 4 ⇒ port 22    clump 10 ⇒ port 22   clump 16 ⇒ port 22    ...
clump 5 ⇒ port 25    clump 11 ⇒ port 25   clump 17 ⇒ port 25    etc.
```

The available switch ports need not be in any particular order. Figure 4–13 shows the same switch ports as in Figure 4–12, but in a scrambled order. This will work fine. In other words, any permutation of the switch ports in the pool may be used in the round-robin assignment of clumps to ports.

Figure 4–13 Modulus RAM use — example 2.

```
clump 0 ⇒ port 19    clump 6 ⇒ port 19    clump 12 ⇒ port 19    ...
clump 1 ⇒ port 25    clump 7 ⇒ port 25    clump 13 ⇒ port 25    ...
clump 2 ⇒ port 16    clump 8 ⇒ port 16    clump 14 ⇒ port 16    ...
clump 3 ⇒ port 13    clump 9 ⇒ port 13    clump 15 ⇒ port 13    ...
clump 4 ⇒ port 22    clump 10 ⇒ port 22   clump 16 ⇒ port 22    ...
clump 5 ⇒ port 10    clump 11 ⇒ port 10   clump 17 ⇒ port 10    etc.
```

In the above examples we have seen considerable flexibility in use of the modulus RAM. What is important is that *all interleavers that will access the interleaved page must use the same mapping*. The software may restrict this access to certain processors, such as one cluster of processors allocated to a particular user. This would mean the interleavers for other processors would not have to contain the same mapping for this page. On the other hand, the software implementation may take the far less complex route of mandating, by convention, that all interleavers in the entire machine hold the same mapping.

Efficient Use of Interleaving

Figure 4–14 shows the same mapping as Figure 4–12, but in graphic form. Each "A" indicates that the clump number shown to the left is in the memory subsystem at the switch port shown below. The bottom "A" says clump 0 is mapped to port 10, the next "A" maps clump 1 to port 13, etc.

This diagram pertains to one interleaved page at each of the six ports. Interleaving this page has no other effect on any other page in the machine. The software, however, may implement a convention that the same-numbered page at every port also be interleaved.

Figure 4–14 Modulus RAM use — one stripe.

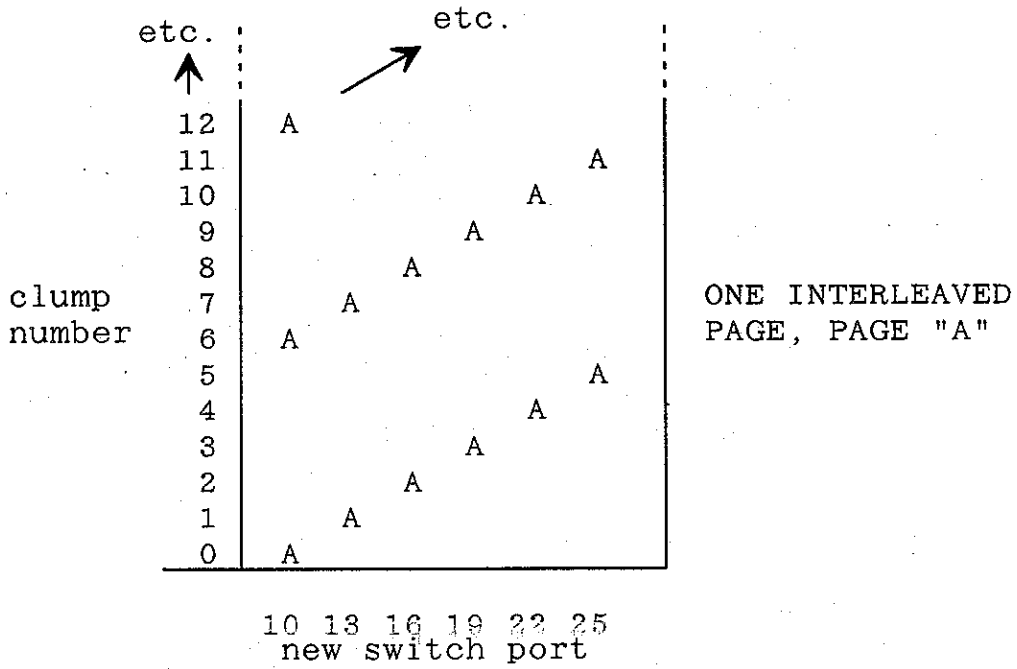
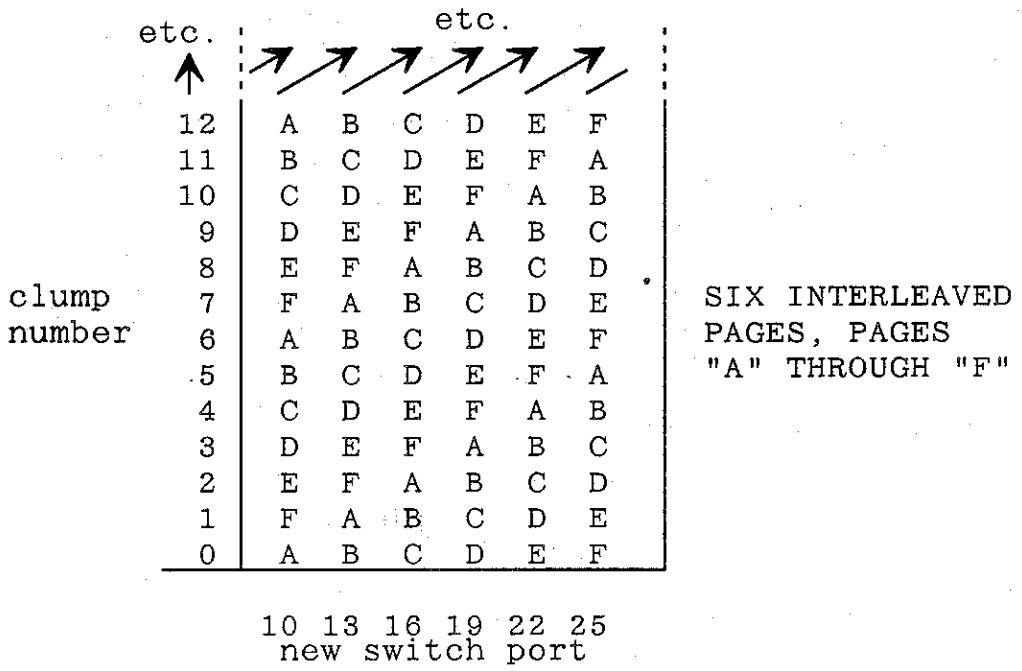


Figure 4–14 shows that successive clumps are mapped to higher and higher areas in the interleaved page on each node. We call this pattern of interleaved addresses a *stripe*, like stripes on a barber pole. This stripe is a result of cycling through the switch ports in a repetitive fashion. Although we allocated a page of physical memory at each of the six ports in the pool, we have achieved only one interleaved page of System Address Space. This would be a waste of system resources except for an additional feature of the interleaver.

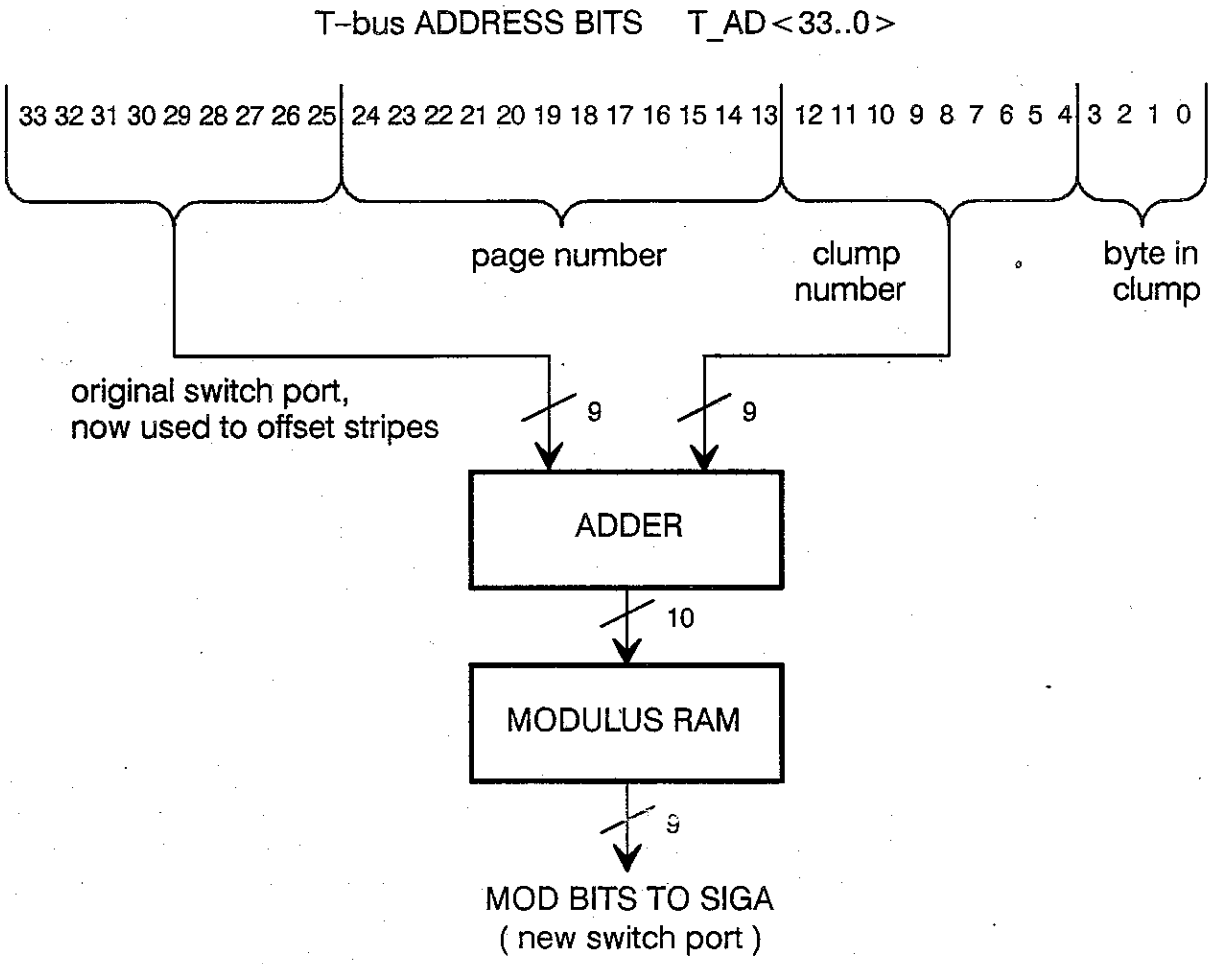
The unused clumps — blank in Figure 4–14 — can be used by other stripes, each offset from the others. Figure 4–15 shows how six interleaved pages "A" through "F" fill up the six physical pages of the pool. Page "B" starts at the next switch port in the pool than where page "A" starts, and thereafter has the same cyclic striping obtained by the modulo function as "A" has. Page "C" starts at the next available page in the pool, and so on. The number of stripes that this algorithm makes is the same as the number of switch ports in the pool, in this example six. This uses all the physical memory allocated to interleaved pages very efficiently.

Figure 4–15 Modulus RAM use — six stripes.



To map different interleaved pages to different stripes, the address supplied to the interleaver's modulus RAM is given an offset. In the example of Figure 4–15, page "A" would have no offset, page "B" an offset of one, "C" an offset of two, and so on. The interleaver provides this capability by adding T-bus address bits 33..25 to the clump number. In an address in a non-interleaved page, bits 33..25 specify the switch port; but in an interleaved page these bits are replaced by the MOD bits from the interleaver. Therefore, these bits are available and provide a convenient place to specify this offset. Figure 4–16 shows the interleaver hardware to implement this.

Figure 4–16 Using an offset to pack stripes.



NOTE

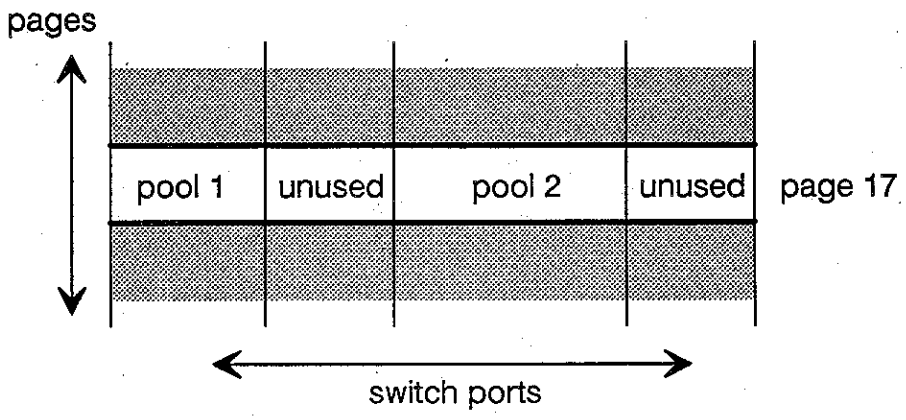
STRIPE OFFSET OF ZERO

To specify a stripe offset of zero, as with interleaved page "A" in Figure 4–14, the T-bus bits T_AD<33..25> must be all zero. That is, the switch routing field of the System Physical Address must be zero. This implies that the switch routing field of the Physical Address must be zero, and in particular bits 28..23 of the Physical Address must be zero. But this is precisely the condition that indicates a local reference, so the SIGA will not respond, the access will not go over the switch, and the interleaver will not be used. This is an accepted limitation of the design: *the stripe offset cannot be zero*. If the pool of switch ports is of good size, there will be many stripes available, so the sacrifice of one stripe is not serious.

Several Interleave Pools

The suggested uniform use of interleaving requires that a page that is interleaved on one node be interleaved on all nodes. Nevertheless, that page on various nodes could be in different interleave pools, as illustrated in Figure 4–17. There, page 17 on some nodes is in pool 1, on other nodes it is in pool 2, and on yet other nodes it is unused.

Figure 4–17 Multiple interleave pools.



There are several reasons why it might be convenient to have several interleave pools.

- Efficient use of resources.

  The suggested model for Butterfly II interleaving is uniformity of mapping. This implies that if a given page number (an 8-kilobyte range of System Physical Address space) is referenced as interleaved at one switch port, it will be so at all switch ports. Therefore it is necessary to allocate that physical page, at every port in the machine, to interleaving — even if only a few switch ports are actually used in the interleave pool. Having several interleave pools might allow partially relaxing this model, while still using uniform interleaving within clusters of switch ports. This could allow tailoring the allocation of interleaved pages to the application.

- Independent, redundant copies for reliability.

  One way to achieve robustness against errors or component failures is to store critical data structures in duplicate, in separate places. One way to do this is to use separate stripe offsets for the two copies. This has the disadvantage that failure of a switch port loses part of the data in each copy, so neither copy is usable; the data must be reconstructed by culling parts from each stripe. A simpler solution uses two interleave pools that

have no switch ports in common. One copy is stored in each pool. Now one or the other copy is still intact after an error or failure.

- Performance control by reducing interactions.

  The execution speed of any program is affected by contention for access to memory. Usually this interaction is negligible. Benchmark programs and some intensive applications, however, can be sensitive to the contention. For these, it may be useful to give the program its own interleaved pool. This may help to isolate its memory accesses from those of the other processors. The benchmark's results are then more repeatable and meaningful, and the intensive application performs reliably. It could be that a exerciser test or diagnostic program would make such heavy use of its memory that isolating it from the rest of the machine would avoid degrading the service other users receive. (How

- Matching pool size to contention reduction needs.

  With separate interleave pools, each pool can contain a different number of switch ports. Data structures that would suffer only a moderate amount of access contention at a single switch port can be placed in a small interleave pool. Data structures that would engender significant contention even in a small pool can be placed in a larger pool, which may help their performance.

(Note, however, that implementing multiple interleave pools in an operating system can be difficult. The above benefits are therefore of a theoretical nature, and may or may not be realizable in a practical operating system.)

The Butterfly II interleaver provides for eight interleave pools. (The hardware supports eight pools per interleaver, but the uniform use of interleaving, as section 4.5.3 recommends, limits this to eight pools per Butterfly II machine.) Each pool operates as described above, with clumps mapped to stripes distributed among the memory subsystems of the pool. The multiple pool capability is implemented by having eight times as much modulus RAM as would otherwise be needed. By selecting a given pool, one of the eighths of the modulus RAM is selected.

The three additional address bits driving the modulus RAM come from the interleaver's pool RAM. The selection of a pool, that is, the address driving the pool RAM, is based on two fields of T-bus address bits.

- The first component of pool selection is T_AD<22..15>. These bits specify a 32-kilobyte *quad-page* within the 8-megabyte bank zero.
  - Because T_AD<14..13> are not included in pool selection, individual pages cannot be independently assigned to interleave pools. Rather, all interleaved pages in a quad-page belong to the same pool. This is consistent with the CPU interface design, that decides whether a page is interleaved or non-interleaved based not on the entire page number, but on the quad-page in which the page lies.

- Interleaving is restricted to bank zero, that is, to T_AD<24..23> = 00. The CPU interface enforces this restriction for references from the CPU, and conventional use of the VMEbus interface by software implements this restriction for references from the VMEbus. If this restriction were violated, the given quad-page *in each of the four 8-megabyte banks* would belong to the same interleave pool, because T_AD<24..23> are not included in the pool selection.

- The other component of pool selection is T_AD<33..28>. In an non-interleaved remote access these bits are part of the switch routing; in an interleaved access they are part of the stripe offset.
  - Because T_AD<27..25> are not included in pool selection, individual stripes cannot be independently assigned to pools. Rather, groups of eight stripes all belong to the same pool.

This concludes the conceptual operation of the interleaver. Figure 4-8 (above) shows the complete hardware implementation.

[NOTE: We need a discussion of suggested approaches to programming the interleaver — at least one possible approach, especially showing the use of pools. And maybe ways of thinking about the double use of T_AD<33..28>.]

4.5.6 The Interleaver Loader

The modulus RAM and pool RAM are not read or written from the T-bus, but through the Switch Interface Gate Array (SIGA). The SIGA mechanism that performs this is called the *interleaver loader*, and is described in the B2VME chapter and in the SIGA Specification.

4.6 Error Detection and Error Rate

In the B2VME, error detection is performed by byte-wide parity generation and checking, making a total of four parity bits per word. Each memory subsystem reports parity errors on the T_PARITY signal of its local T-bus; for a remote switch access, the server SIGA propagates this error back to the requester device with a bus error code.

Only main memory has parity hardware. Registers, mapping RAMs, etc. do not have parity hardware. Every switch message is protected by a parity bit.

The error rate is estimated by using the data from the memory chip manufacturer, and scaling for the number of chips in the machine. We assume independent error statistics for the chips, after the initial burn-in during system integration and test.

The implementation of the Butterfly II machine has the following characteristics:

- All main memory is distributed on B2VME function boards. Each B2VME holds four megabytes of dynamic RAM arranged as 1M × 36–bit words.
- Each word has four 8–bit bytes with one parity bit per byte.
- The B2VME therefore has 1 megaword × 36 chips per megaword = 36 memory chips.
- The Butterfly II architecture permits up to 512 function boards.

4.6.1 Detected Soft Error Rate

The error rate for a memory chip is often specified as the number of errors per one billion hours of operation, also called *FIT* (failures in time).

The MTBF (mean time between failures), in hours, is given by:

$$\text{MTBF} = 10^9 \div (\text{FIT of chip}) \div (\text{number of chips})$$

In the following, a FIT of 1000 is used. This is appropriate for the high confidence level (90 percent) we use in design.

For the B2VME, MTBF = $10^9 \div 1000 \div 36$ = 27,778 hours = 3.17 years.

Figure 4–18 lists the soft error rate for several sizes of Butterfly II system. For a Butterfly II with 512 B2VMEs (2048 megabytes total main memory), MTBF = 54.3 hours. This suggests that for very large machines and machines with more memory per node, the MTBF with current hardware may not be within desired *design* limits. Therefore, error correction may be employed on later versions.

Figure 4–18 System memory soft error rates.

Number of B2VMEs	Total Memory (MB) (4 MB per board)	Mean Time Between Soft Errors (hours)	(days)	(years)
1	4	27778	1157	3.17
8	32	3472	145	0.396
16	64	1736	72	0.198
32	128	868	36.2	0.099
64	256	434	18.1	0.050
80	320	347	14.5	0.040
96	384	289	12.1	0.033
128	512	217.0	9.04	0.025
192	768	144.7	6.03	0.017
256	1024	108.5	4.52	0.012
384	1536	72.3	3.01	0.008
512	2048	54.3	2.26	0.006

4.6.2 Undetected Soft Error Rate

The undetected soft error rate is *extremely* small. Because an undetected error is so serious, however, it is important to examine the rate to ensure that it is negligible. The undetected soft error rate is slightly more complex to calculate because it depends on the length of time in a "window" between references.

To calculate the chance of two bits being in error simultaneously, we must define "simultaneously". Here, we take simultaneously to be a period between reads of the byte, because errors go unnoticed until the data is read. Rather than the average time between reads, we use an estimate of the longest time that a data byte remains unread in main memory and is then read. We call this window of time "w", in units of hours.

In estimating the undetected soft error rate we make the following assumptions.

- We assume the errors measured in the FIT number are single-bit errors. If the errors involve several bits — or worst, all bits in the chip — the undetected error rate would be higher. Because the B2VME design employs a separate chip for each bit in the word, a failure of several bits on

a chip is manifested as several words going bad. Consequently, this assumption is appropriate for our memory architecture.

- The error rate is dominated by errors in the memory chips. We disregard the chance of an error in the parity checking logic. This is reasonable because the Butterfly II design includes a feature that permits diagnostic software to exercise and check the parity hardware — a mode in which the wrong parity is intentionally written. Note also that the parity is generated and checked on every switch message by the SIGAs.
- Out of nine bits, the chance of more than two bits being in error is negligible compared to the chance of less than three bits being in error.
- We assume there are no idle, de–allocated areas of memory, in which an error would go undetected. This is a conservative assumption. In fact, picked bits — bits that should be "0" but erroneously become "1" — might be detected in free memory that is later allocated with the assumption that it is still zero.
- We assume all of main memory is equally at risk. That is, all bytes are used and referenced infrequently, at the window rate described below. This is a conservative assumption, for three reasons.
  - Not all memory is likely to be in use.
  - The error rate computation assumes that all allocated memory is susceptible to the two–error window. In practice, some memory will hold code and frequently–referenced data, so errors will be detected as single errors and cannot accumulate to an undetected double error.
  - Some memory is written. When a location is written, the possible existence of any errors in the previously stored data is lost.

By definition, the sum of all possibilities that may occur during the window is one. Considering nine bits, we have:

$$1 = p(\text{no bits fail}) + p(\text{one bit fails}) + p(\text{two bits fail}) + \ldots + p(\text{all nine fail})$$

Or, using our assumption that terms beyond the first three are negligible:

$$p(\text{two bits fail}) = 1 - p(\text{no bits fail}) - p(\text{one bit fails})$$

Using "f" to indicate the probability that any given bit fails during the time window, the last two terms can be expressed and approximated as:

$$p(\text{no bits fail}) = (1-f)^9 = 1 - 9f + 36f^2 - \ldots \approx 1 - 9f + 36f^2$$

$$p(\text{one bit fails}) = 9f(1-f)^8 = 9f(1 - 8f + \ldots) \approx 9f - 72f^2$$

Combining these approximations into the equation for two bits failing:

$$p(\text{two bits fail}) = 1 - (1 - 9f + 36f^2) - (9f - 72f^2) = 36f^2$$

Using the definitions of FIT (error rate per 10^9 hours per chip) and the window w, we see that f can be expressed as:

$$f = w \times FIT \div 10^9 \div (\text{number of independent bits per chip})$$

Here, the "number of independent bits per chip" depends on the way the FIT number is measured. We have assumed that single bits fail, one at a time, so the "number of independent bits per chip" is the same as the number of bits per chip. Further, this is reasonable because of the B2VME memory architecture of one chip per bit.

Using this, we can write the probability of an undetected error in nine bits during the window of w hours:

$$p(\text{undetected error in byte during window}) = 36\ FIT^2\ w^2 \div 10^{18} \div (\text{number of independent bits per chip})^2$$

The probability of an undetected error per byte *per hour* is the above divided by the window size, w. And the MTBF is the reciprocal of that per-hour probability:

$$MTBF(\text{undetected, per byte}) = 10^{18} \times (\text{number of independent bits per chip})^2 \div (36\ FIT^2\ w)\ \text{hours}$$

Now let's plug in some representative numbers. Take FIT = 10^3 errors per 10^9 hours per chip, 10^6 independent bits per chip, and w = 1 week (168 hours). And consider the error rate for a B2VME board with four megabytes of memory.

$$MTBF(\text{undetected, per B2VME, w} = 168) = 10^{18} \times (10^6)^2 \div (36 \times 1000^2 \times 168) \div (4 \times 10^6)\ \text{hours} \approx 4.7 \times 10^9\ \text{years}$$

Scaling this up to a Butterfly II with 512 B2VME's:

$$MTBF(\text{undetected, per 2048-megabyte machine, w} = 168) \approx 9.2 \times 10^6\ \text{years}$$

This is a long time. There is ample room for degradation if other chips have a worse error rate than that used here, and for larger memories. Using 4-megabit DRAMs, the B2VME design permits drop-in upgrade to 16 megabytes per B2VME. Expansion to the Butterfly II addressing design limit of 32 megabytes per switch port also leaves the undetected error rate negligible.

4.6.3 Hard Error Rate

After a soft error, the machine will reboot and all hardware can still be used. After a hard error, the memory subsystem with the error must be replaced or configured out of the system. The hard error rate is of importance in large

machines such as the Butterfly II computer, but of less importance here than in many other machines because of the ability to configure around failed function boards.

4.6.4 Write Wrong Parity

The B2VME contains a feature to test and exercise its memory parity hardware. Under software control, data can be written with the wrong parity. This is controlled by the *write wrong parity* bit of the Machine Configuration register, described in the B2VME chapter.

4.7 Historical Note: Steal Bit

The original Butterfly II design included a mechanism called "steal". **The steal mechanism is not available to software in the machine actually produced.** Although this mechanism has not been fully implemented, parts of the machine (namely the SIGA) do support it. The mechanism affects design of the CPU interface, switch and memory. It is described here because it is fundamentally an aspect of memory.

Associated with each 4-byte word of Butterfly II memory is an additional, 33rd bit called the *steal* bit. The intent of this tag bit is that when it is set to "1", the associated word is being used in a computation that must be allowed to write its result into the word before another process reads the word. The steal bit is a type of lock, locking one word.

Comparison of Steal and Lock

The steal bit is a different and separate mechanism from the machine's locking protocol.

- The steal bit applies to one word of memory, is not keyed to who stole the word (except if a software convention is employed), and persists for an indefinite time (until the word is un-stolen). The *sneak* mechanism gets around stolen words.
- The locking protocol applies to a T-bus resource (typically an entire memory on a remote function board), is explicitly tied to the CPU that initiated the locked sequence, and lasts for a short time (limited by the CPU Lock Timer). The *bypass locks* facility gets around locks for local memory references, and instruction fetches automatically bypass locks.

Summary of Steal Mechanism

The important points of the steal mechanism are summarized below.

- Each word of Butterfly II main memory (RAM) has a 33rd bit, the *steal* bit. The bit applies to the entire word; the word cannot be partly stolen and partly not stolen. There is no indication of who stole the word, unless established by software convention. Each steal bit was protected by its own parity bit.
- Parts of the Butterfly II address space other than main memory (RAM), such as control registers and VMEbus address space, do not implement the steal bit. Attempting to steal any locations in these devices results in a bus error.
- A word is stolen in either of two ways.
    - The word is written while a bit in a CPU interface control and configuration register is set to "1". (In an early function board, this bit was the *steal* bit in the Augmentation register.)
    - An XMEM instruction is executed while the Lightweight Steal register's lightweight steal bit is set to "1". This also clears the register bit to "0". (The Lightweight Steal register existed only on an early function board.)
- An access of any size (byte, halfword or word) steals the entire word. A switch message attempting to steal more than one word is illegal.
- Stealing a word that is already stolen is permitted; no error is generated.
- Reading (without the *sneak* mechanism) a stolen word results in a bus error, with a code indicating that the word is stolen.
- Reading any portion of a stolen word (byte, halfword or word) results in the same bus error. If a switch message requests reading more than one word, the response is truncated after the first word that is stolen. If any words read during a T-bus burst read is stolen, a bus error occurs. (In the initial design, only the CMMU generated burst reads, to fill a cache line. Cacheable data should never be stolen, as noted below.)
- If the Augmentation register's *sneak* bit is set to "1", reading is immune to whether the location accessed is stolen; the "stolen" bus error does not occur, the data is returned normally, and further, the Process Configuration register's *sneak data* bit is set to the value of the word's *steal* bit. (The AR *sneak* bit and PCR *sneak data* bit existed only on an early function board.)
- A word is un-stolen by writing to it while the Augmentation register's *steal* bit is cleared to "0". This clears the word's steal bit to "0".
- A write access of any size (byte, halfword or word) un-steals the entire word. A switch message writing more than one word un-steals any stolen words that it writes.
- The steal, "stolen" bus error on a read, and un-steal operations all depend on the memory holding the word actually being accessed. If some error, such as an access protection violation, occurs before the memory

is accessed, then the operation does not occur — the word is not stolen, the error code does not specify "location read is stolen", or the word is not un-stolen, respectively.

- Having the Augmentation register's *steal* bit set to "1" has no effect on a read operation, and having the register's *sneak* bit set to "1" during a write has no effect — with the execption of cacheable locations, noted below.
- Cacheable locations cannot be stolen or sneaked. Attempting to do so results in a bus error. Examples of cacheable locations are:
  - All instructions — "text" or "code" pages
  - Memory management page tables
  - Exception vectors
  - Supervisor stack

Problems with the Steal Mechanism

The steal mechanism was abandoned mainly for three reasons:

- The effect of normally reading a stolen location — namely, a bus error — was deemed fairly drastic. Servicing a bus error invokes significant software machinery and takes a significant amount of time. Also, a reference from a VMEbus that encounters a stolen location could be hard to handle productively.
- How to efficiently save and restore the 33rd (steal) bit on each word in a demand-paged, disk swapping operating system environment was not clear.
- The additional circuitry required to support the steal mechanism exacerbated the electrical demands on function board T-bus implementation — run length and loading, and therefore signal quality and propagation time.

6

The Butterfly II Test and Control System (TCS)

6.1 Introduction

The Test and Control System (TCS) is a network of microcomputers within the Butterfly II machine. The TCS controls, monitors, and tests the Butterfly II hardware components. These functions are implemented by the TCS rather than the main processing components of the machine to avoid the problems associated with a computer testing and controlling itself.

At the center of the TCS is the TCS master, a small computer with disk and console. A backup, or redundant, TCS is implemented in some configurations. For simplicity, this chapter omits discussion of redundant TCS components until section 6.6. The TCS master communicates with and controls a set of TCS slave processors. Each TCS slave processor is a single-chip microcomputer that monitors and controls an element of the Butterfly II machine. The TCS performs the following tasks:

- **Testing** — The TCS controls the loading and operation of machine diagnostics. After loading a diagnostic program into a function board, the TCS can start its execution and read the results of the test. This capability eliminates diagnostic PROMs from the system's circuit board designs.
- **Bootstrapping** — The TCS controls the initialization of system components. It also provides the system's cold start mechanism by loading bootstrap code into system memory and starting its execution. This capability eliminates bootstrap PROMs from the system's circuit board designs.
- **Monitoring** — The TCS ensures that the system's hardware is functioning by continually checking critical components and signals. The TCS also monitors temperature and power supply voltages.

- **Control** — The TCS controls circuit board power and switch access. It can set power on, off, or to margining values for board testing. The TCS can cut off switch access to individual function boards, and it can cut any individual data path out of the switch.
- **Configuration management** — The TCS can detect the number and type of circuit boards in the system. The TCS checks this against its system configuration file to detect component outages.
- **Operator interface** — The TCS console provides operator access to TCS functions and information. It also serves as the operating system console for the Butterfly II machine. A modem port allows access for remote field service.
- **System software interface** — Software running on Butterfly II function boards can transfer information to and from the TCS, and invoke TCS functions.

Following sections of this document provide an overview of how the TCS is used; describe the individual components of the TCS (master, bus, slaves, front and back panels, and power); discuss the system-level functions that the TCS can perform; present the B2TCS card functional specifications; discuss the support for redundancy; and describe the TCS bus protocol.

6.2 Overview of TCS Use

The Test and Control System is intended for use in manufacturing, field service, and operational environments. Because the TCS can control the major components of the machine at a very low level, initialization, testing and bootstrapping of the entire system or any of its modules is greatly simplified. Since the TCS monitors the machine at a level that is not usually accessible without specialized test equipment, we expect that it will be a powerful tool for performance measurement, fault detection, fault isolation, and fault prediction.

The TCS master processor is a small computer that is the heart of the Test and Control System. When a Butterfly II system is first powered up, only the TCS master gets power. After it passes its own self tests, the TCS master is responsible for turning on the bulk power supplies, turning on all of the circuit cards in the system, conducting power-on self tests, deconfiguring modules that fail, loading the first level system bootstrap software, and enabling the processors on the function boards.

While the machine is up and running, the TCS master continuously monitors the circuit cards in the system, watching for changes in the machine configuration. When a card is added or removed (by operator intervention or hardware failure), the TCS master detects the change and updates a machine configuration file.

A serial terminal connected to the TCS master serves as both the TCS console and the Mach system console. A command interpreter running on the TCS master responds to TCS commands. Mach console traffic is passed unmodified through the TCS master to the Mach master node. A modem port on the TCS master can be used to give off-site field service personnel access to the Mach and TCS consoles.

The TCS master communicates with the Mach operating system through a data structure on the Mach master node. In addition to passing console traffic back and forth, the Mach kernel may issue direct requests to the TCS master using a simple message protocol. The kernel may request measurement data, such as the ambient temperature at one or more function boards, or it may request a service such as rebooting the machine.

Data collected by the Test and Control System can be used in many different ways. Performance measurement software can use switch traffic measurements to detect memory hot spots. A demon process can periodically survey temperatures and power supply voltages throughout the machine and log the results in a history file that can be checked by field service personnel. The Test and Control System has been built out of simple computing elements in order to keep it as inexpensive and reliable as possible. As a result, the TCS master processor is not well equipped to perform extensive data analysis, nor is it capable of storing large amounts of data. The TCS therefore serves as a powerful data collection tool, but data logging and analysis are left to other, more capable, processors.

The TCS master has a library of diagnostic software on a hard (or floppy) disk. It can retrieve routines from this disk and load them into the system's function boards when appropriate. During power-up, these tests are systematically loaded and run by the TCS master. During field service, an operator can type commands at the TCS terminal to invoke these tests, plus more sophisticated tests as needed to isolate faults. Other commands turn circuit cards (or the whole system) on and off, read the serial numbers and revision levels of system components, or blink LEDs to help service personnel locate cards.

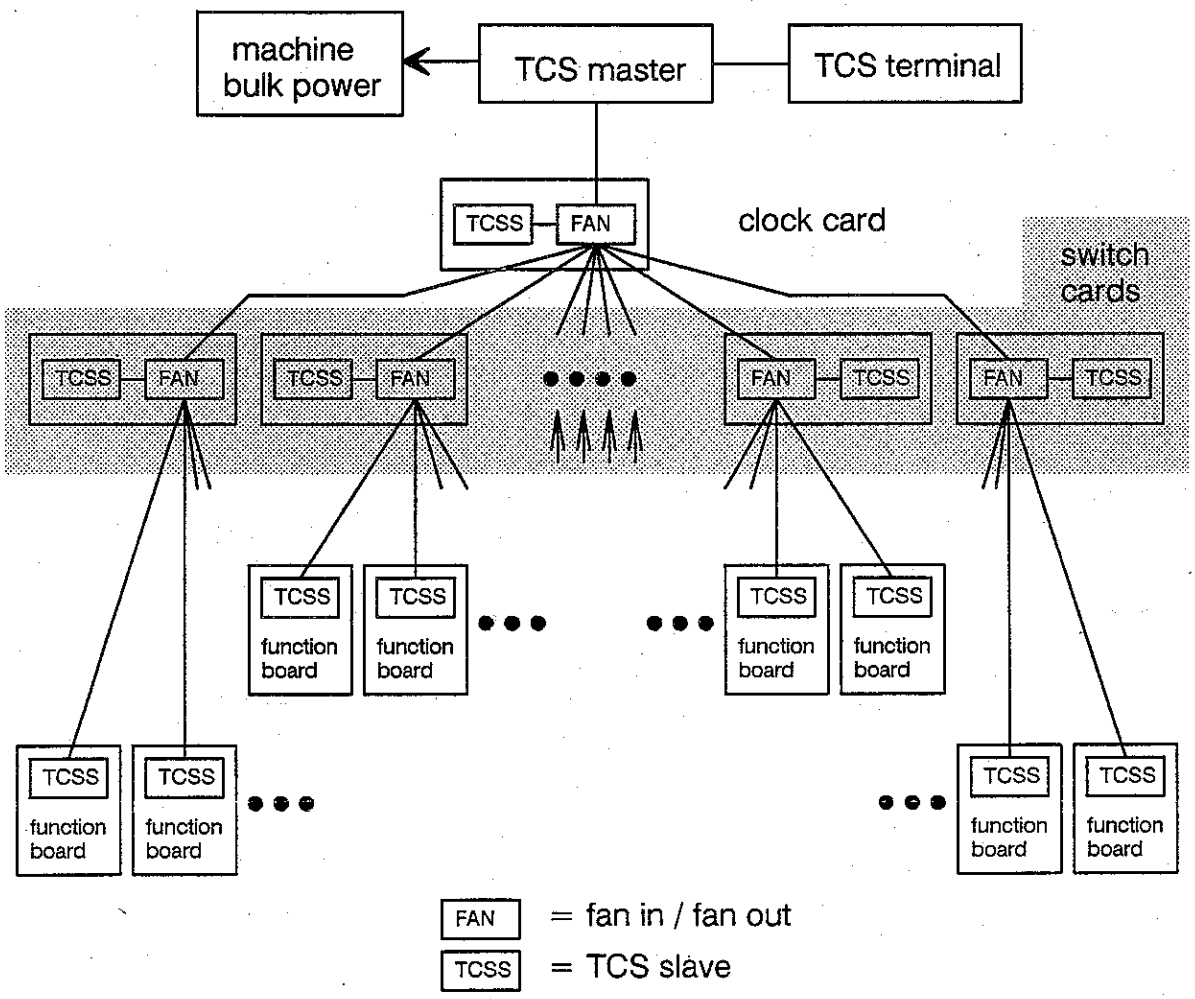
The TCS is designed to manipulate circuit cards via a simple interface. In the manufacturing environment this means that a stand-alone circuit card can be plugged into a TCS master with a cable, loaded with code from the master's disk, and exercised. Field service operators can manipulate and probe individual circuit cards without removing them from the system. Various diagnostic programs are available from within the Test and Control System master and can be invoked with TCS terminal commands. A goal of the TCS is to diagnose Butterfly II component failures as completely as possible using just the TCS.

6.3 Hardware Components

The basic structure of the TCS is shown in Figure 6–1. The TCS is made up of the following components:

- TCS master — The TCS master controls and coordinates all TCS activity, and thereby oversees the operation of the entire machine.
- TCS slaves — Each TCS slave controls and monitors the individual circuit card of which it is a part.
- TCS bus — The communication system ("bus") links the TCS master to all TCS slaves in the machine.
- TCS front panel — Top-level controls and indicators are mounted here.
- TCS back panel — The TCS terminal and phone line connect here.
- TCS power supplies — One supply powers the TCS master, and another powers all TCS slaves in a 64-slot machine. An additional TCS slave power supply is required for each additional 64 slots.
- Machine bulk power control — The TCS controls the power for the rest of the machine, with provision for independently powering on or off separate sections of it.

Figure 6–1 TCS block diagram — overview.



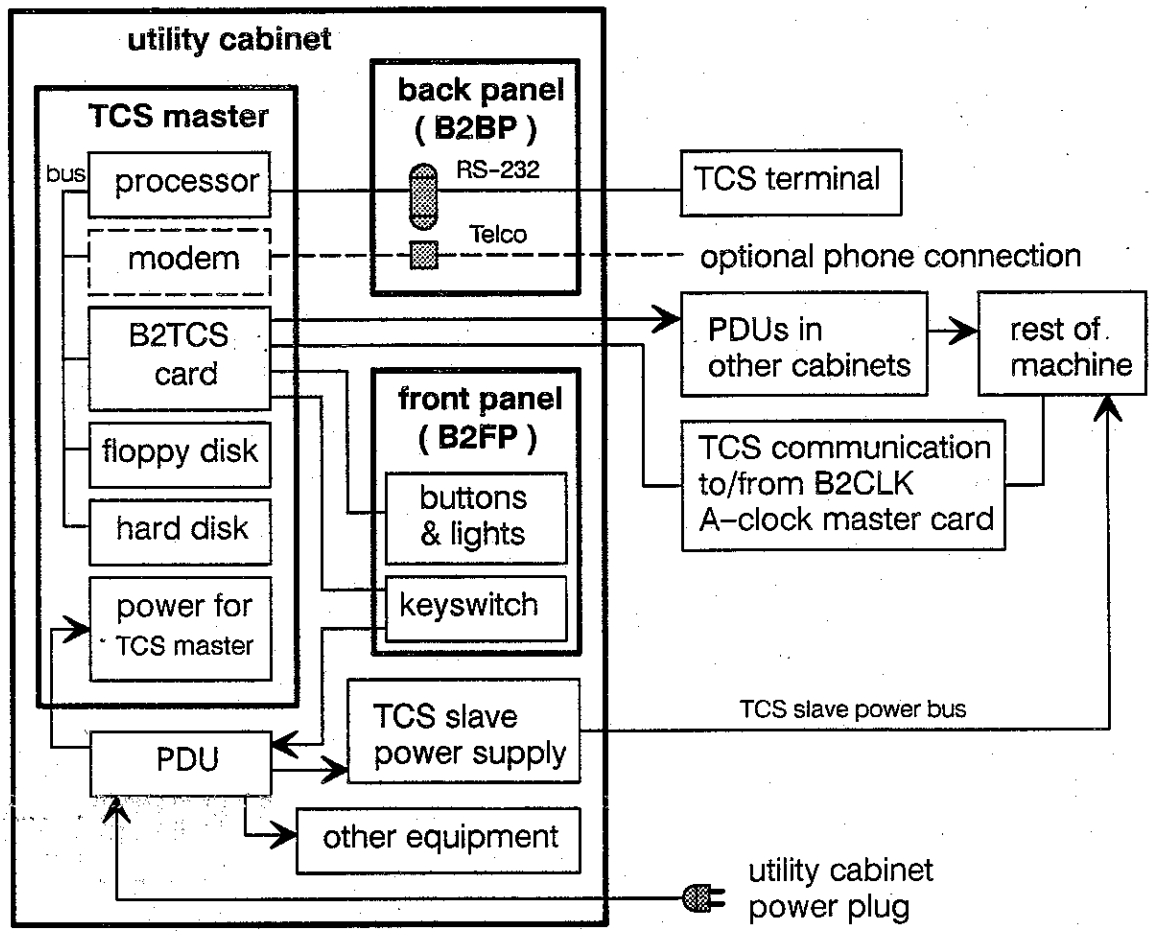
The logical operation of the TCS communication system resembles that of a bus. However, the hardware implementation is tree-structured, containing active elements that fan data out from the master to the slaves and fan data in from the slaves to the master.

The standard TCS terminal is a DEC VT320. If hard copy is needed, a VT320 with a printer connected can be used.

6.3.1 TCS Master

The TCS master, shown in Figure 6–2, is the brains of the Test and Control System. It is made up of an IBM PC compatible central processing unit and peripheral cards in a PC-bus card cage with a passive backplane connecting them together.

Figure 6–2 TCS master and associated equipment block diagram.



Master CPU Card

The TCS master CPU card is an IBM PC/AT compatible computer that is built on a single IBM PC form factor printed circuit card. The CPU card includes:

- SCSI port (1.5 megabytes per second)
- Floppy disk port
- Two RS–232 serial ports
- Real time clock / calendar with battery backup
- 512 kilobytes of memory

The CPU controls the following system support devices: a hard disk drive, a floppy disk drive, a serial port, the real time clock and calendar, the special

B2TCS interface card described below, and, optionally, a modem. The system boots DOS from either the hard disk or the floppy disk, both of which are connected to the CPU card via the SCSI bus.

Cabling between the CPU card and the back panel supports an RS–232 port for the TCS terminal.

TCS Interface Card — B2TCS

Except for the B2TCS card, the TCS master is an ordinary personal computer. The B2TCS is a PC/AT–bus format circuit card that implements all the circuitry particular to the TCS. The B2TCS card includes:

- TCS bus interface
- Front panel keyswitch and reset button interface
- Front panel LED drivers (8 provided, 3 currently used)
- Control for machine bulk power
- PC/AT bus interface
- TCS master watchdog timer
- PROM for Butterfly II–specific bootstrap code

The TCS bus interface is a 2681 DUART (Dual Universal Asynchronous Receiver/Transmitter). The TCS uses this chip set up for 9–bit data exchanges, as does the serial communication interface (SCI) of the Motoroia 68HC11 employed as the TCS slave processor. This DUART has address mark multidrop receiver addressing capability, as does the Motorola 68HC11 microcomputer's SCI. CMOS drivers and receivers connect this DUART with the TCS bus connectors.

The front panel interface on the B2TCS connects to three indicator LEDs and two switches — a reset button, and an off/on/secure keyswitch. The "off" position directly turns off the power distribution unit serving the TCS master, and thereby the power for the entire machine. When the keyswitch is in the "on" position, power is supplied to the TCS master, which in turn controls all of the other power supplies in the machine. When the keyswitch is in this position, pressing the reset button forces a reset of the TCS master CPU. When the keyswitch is in the "secure" position, power is supplied to the TCS master but the reset button is disabled. The CPU can sense whether the keyswitch is in the "on" or "secure" position by reading a register on the B2TCS.

The front panel LED drivers control three LEDs on the front panel. These indicate TCS power on, bulk power on, and TCS enabled. The "TCS enabled" LED indicates whether the comminication line from the master to the TCS slaves is ready for use.

Modem Card

A modem in the Butterfly II TCS master allows field service personnel to establish a remote TCS terminal connection.

The modem in the Butterfly II TCS is a modem card installed in the TCS master card cage on the PC bus. It is connected to an RJ-11 phone jack on the TCS the back panel with a cable. Using a changeable modem card allows us to supply equipment that is already qualified by the FCC and European local PTTs.

The modem card, and its connection to a phone line, are optional. While the design intention is that every Butterfly II machine have this capability, the phone line need not be connected, nor the modem card present, at sites where telephone access is undesirable.

Hard Disk

The TCS master contains a Winchester (hard) disk of at least 20-megabyte capacity that is controlled over a SCSI interface. The list of files below is representative of the files on the hard disk. For a detailed discussion of hard disk files, please refer to TCS software and operations documentation.

- TCS master operating system (DOS)
- TCS master application code
- TCS system diagnostic
- Machine configuration file
- Machine log
- Function board Power-On Self Test (POST) code
- Function board diagnostic code
- Switch card and clock card diagnostics (run in function boards and/or TCS master)
- Bootstrap loader code (runs in a function board with access to a disk)
- Bootstrap server code (runs in other function boards)

Floppy Disk

The TCS master contains an IBM PC/AT compatible, 1.2-megabyte, 5.25-inch floppy disk drive controlled over a SCSI interface. The intended principal uses of the floppy drive are to load new versions of TCS software, and to load special diagnostics during field service. However, the drive is available for any I/O needs of the TCS software.

6.3.2 TCS Bus

The TCS communication bus, or "TCS bus" for short, is a cable that originates at the TCS interface card (B2TCS) discussed above. The following four wires comprise the cable:

TCS transmit–direction data (master to slave)
TCS receive–direction data (slave to master)
TCS master identity (signal named A/B*)
ground

Use of the transmit and receive wires is described below. The identity wire is described in section 6.6 on redundancy support. The asterisk in A/B* means negation, not a footnote reference; the identity can be thought of either as "A" or as "not B".

Electrically, the TCS bus is a multi–drop (plus fan–in and fan–out), asynchronous, serial bus system that carries data between the TCS master and the TCS slave processors. The TCS bus carries TTL level (5 volt) signals that are distributed through the Butterfly II machine from the TCS master via CMOS fan–out and fan–in logic distributed among clock and switch cards.

At each slave, resistors to ground prevent noise on the receive data wire when the TCS master (or the fan–out circuit driving the slave) is removed or powered off. Without this, noise could be erroneously parsed by the slave as commands.

Transmission Speed and Timing

The master communicates with slave processors over the asynchronous serial TCS bus. The TCS bus runs at 125 kilobits per second. Using asynchronous communication with ninth–bit message framing, each byte takes eleven bit times — each byte has nine data bits, one start bit and one stop bit. This results in a peak byte transfer rate of 11,363 bytes per second. The same rate is used for both the master to slave direction and the slave to master direction.

Receive (Slave to Master) Direction

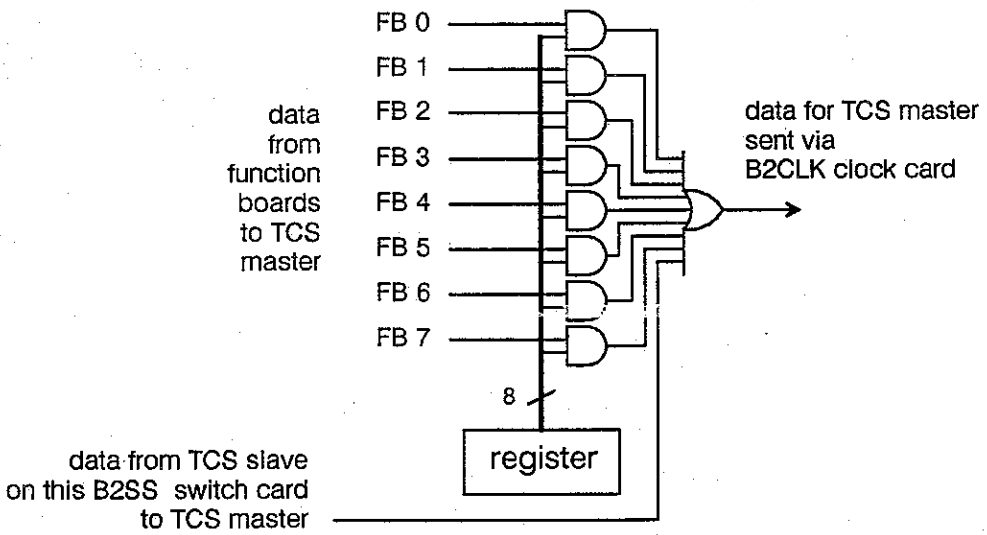
In the slave to master direction, the TCS bus connects all the slaves to the master. On each midplane, the slaves on the eight function boards are connected to the B2SS switch card. The wire from each function board is bused over the midplane to the switch card, where it connects to a schmitt trigger input AND gate. The other input to that AND gate comes from a register that is used to amputate function board slaves from the slave to master bus. This means that there are eight AND gates on each B2SS switch card that receive data from eight function boards. (A ninth gate is present to support redundancy, discussed in section 6.6.) These gates, and the B2SS card's own data to the TCS master, are OR'ed together and drive a serial line connected to the B2CLK clock card. The register that controls the enabling of the function board slaves

is controlled by the slave on the B2SS switch card. A B2SS slave can amputate any of the eight function boards from the slave-to-master bus.

When the data from a B2SS switch card arrives at a B2CLK clock card, it is AND'ed with a control bit and OR'ed with data from other B2SS cards before being sent on to the TCS master. The enable signals on these AND gates are controlled by the TCS slave on the B2CLK.

Figure 6-3 illustrates the AND and OR fan-in of the TCS bus in the receive (slave to master) direction, using a B2SS card as an example. At each stage of TCS bus fan-in, such a circuit is used. In Figure 6-1, discussed elsewhere, this fan-in goes from the function boards at the bottom to the TCS master at the top.

Figure 6-3 Slave to master TCS bus fan-in.



In normal operation, each TCS slave sends data to the master only in response to a command from the master, and the master is careful to let only one such command be outstanding at a time. Therefore, data collected by the fan-in circuitry is normally never garbled by other, interfering data. However, if a part of the hardware malfunctions, the TCS master amputates it by disabling data from the failed hardware. In this way the TCS amputates the failed component and continues to use the remaining, enabled portions of the fan-in circuitry.

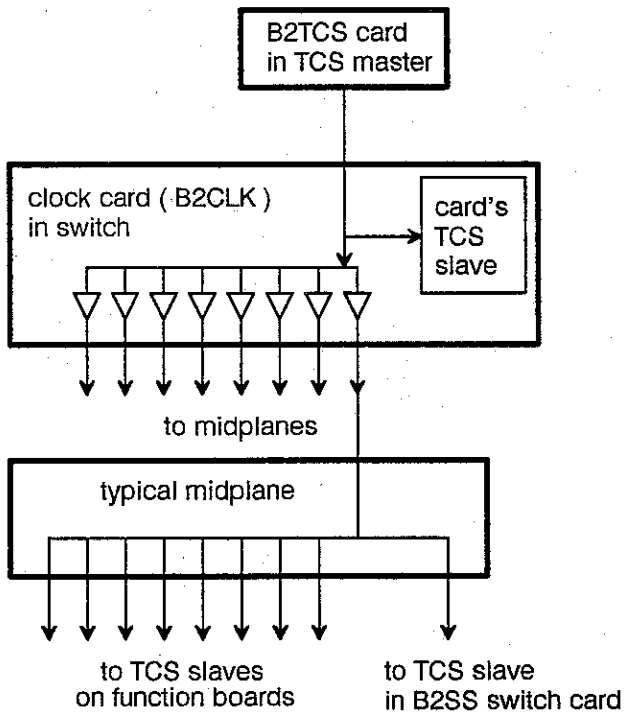
Transmit (Master to Slave) Direction

In the master to slave direction, the TCS bus is a buffered TTL level signal driven by a 74HC240 and received at the slaves with 74HC14 low input current schmitt trigger inverters.

Eight copies of the TCS bus are driven by each B2CLK clock card and sent out over the clock cables to the Butterfly II midplanes, where the signal is distributed to eight function boards and a switch card pair. In each B2SS/B2SR pair, the B2SS card holds the TCS slave that services the pair.

Figure 6–4 shows the fan–out of the TCS bus data from the master to slaves. In a machine with more than eight midplanes, an additional layer of clock cards is used to fan out TCS bus data.

Figure 6–4 Master to slave TCS bus fan–out.



TCS Bus Transactions and Protocol

TCS communications refers to how the master and slave talk to each other, and what commands are implemented. This protocol is described in section 6.7. The protocol supports monitoring and control functions of the general

nature described here, but whose details are presented in TCS software and operations documentation.

All transactions on the TCS are initiated by the TCS master. A TCS slave never sends an unsolicited message. Each TCS message has an address and a command in the message header. All the slaves receive all the messages that pass over the TCS bus. Each slave examines each message to extract the address information. If the address of a message matches the slave's address, then the slave carries out whatever command the message indicates. If the message's address does not match the slave's, the slave processor commands its TCS bus receiver chip to ignore further characters until a new start of message arrives. This way all slaves constantly resynchronize their reception on the start of each message.

There are two types of address, one that addresses a particular slave, and another that addresses multiple slaves. A message that addresses more than one slave is called a broadcast message. Broadcast messages can be addressed to all circuit cards, or to all of the cards of a given type (such as all B2VME boards). Broadcast messages are used during the power–up sequence for loading power–up and diagnostic code into cards of the same type simultaneously.

During normal operation, the TCS master periodically polls the slaves for status information. That status (one byte) indicates whether there are error conditions.

6.3.3 TCS Slave Processors

TCS slave processors perform the TCS commands issued by the TCS master processor. There is a TCS slave processor on every Butterfly II component to be controlled: function board, switch card pair, and clock card. (The TCS master also controls machine bulk power, but not via a slave.)

NOTE

A TCS slave never takes any action controlling its associated hardware unless explicitly told to do so by the TCS master — with one exception. That exception is that if the card temperature exceeds an "alarm" threshold, the slave turns off card power.

The TCS slave processor is a Motorola 68HC11 CMOS microcomputer. This single–chip computer is equipped with ROM, RAM, EEPROM, a counter/timer, a serial communications interface, a serial peripheral interface, and an 8–channel analog to digital converter. Temperature transducers and card power supply voltages are connected to the analog to digital converter inputs for sensing this information.

Slave processors use a power supply and clock that are independent of the cards they monitor and control. TCS power comes from the TCS power supply, and is brought to the slave via the midplane. The clock is supplied by an 8.0 MHz crystal attached to the 68HC11 microcomputer.

The 68HC11 is actually a family of similar microcomputers. The specific part used in the original Butterfly II design is Motorola XC68HC811E2FN. The "811E2" signifies two kilobytes of on-chip memory, and that this memory is EEPROM rather than ROM.

The discussion here covers aspects common to all TCS slaves, and presents some characteristics of slaves on particular card types. The slave on each card type is fully described elsewhere, in conjunction with the description of each type of card. Details of the slave processor are covered in two Motorola publications:

MC68HC11E9 HCMOS Single-Chip Microcontroller provides technical data on the processor as a hardware component.

M68HC11 HCMOS Single-Chip Microcontroller Programmer's Reference Manual describes the processor from a programming viewpoint.

The subsections below describe in detail the interactions TCS slaves have with their surrounding hardware. These fall into two broad categories:

- Aspects of the slave's environment that it monitors or controls
  - SIGA, LCON and SGA chip interfaces
  - Card reset
  - Card temperature monitoring
  - Card voltage monitoring
  - Card voltage control
  - Card LED control
- The nature of the slave itself and its use of the TCS bus
  - Addressing (identity)
  - Configuration parameters
  - Bus selection
  - Amputation from the bus

SIGA Interface

The slave processor on a function board communicates with the board's T-bus, and thus with every other section of the board — including its processor, memory, and I/O subsystem — via the Switch Interface Gate Array (SIGA).

The SIGA interface is based on fourteen 8-bit registers that are accessible to the slave. These registers are:

T-bus access data	(4 bytes/registers)
T-bus access address	(4 bytes/registers)
T-bus command	(1 byte/register)
T-bus command modifier	(2 bytes/registers)
T-bus response	(1 byte/register)
CSU page address	(2 bytes/registers)

These registers, except for the CSU page address register, permit the TCS slave to read and write the T-bus. The SIGA contains additional registers in its Control and Status Unit (CSU). The CSU page address register specifies the 8-kilobyte page at which the additional SIGA registers appear. The TCS slave can access the additional SIGA registers by first setting the CSU page address register appropriately, and then using the other registers listed above to make a T-bus access through the SIGA's TCS interface out onto the T-bus, with the address of the desired register in the CSU. Refer to the *SIGA Specification* (an Appendix) for further details of SIGA registers.

The SIGA to TCS interface is designed to write these registers using the four wires:

Clock
Data In
Data Out
Execute

The "Clock" signal shifts a command into a SIGA shift register via the "Data In" pin. The "Execute" pin signals that the complete command is in the shift register and the SIGA should write its TCS interface register.

Reading the T-bus Response register triggers the T-bus access that is set up in the registers named above.

The TCS is responsible for initializing the SIGAs on function boards at system power-up time.

LCON Interface

The slave processors use the Level CONverter gate array (LCON) interface for three principal functions.

1. The slave can enable or disable a function board's switch port and clock; disabling these amputates the board from the Butterfly II switch.
2. A slave can individually assert any of the LCON pins that connect a function board to the Butterfly II switch.

3. A slave can also individually read the state of any of these pins. This ability is used in conjunction with a similar function in the Switch Gate Array on the switch card to test the continuity of the connections between the function board and the switch card.

The LCON TCS slave interface is similar to the SIGA interface in that it is based on the same four wires connected to the slave processor's serial peripheral port, and is based on a shift register in the gate array. An interface command is shifted into the array and an "execute" signal causes a register access. The format of the actual command in the LCON, however, is different from that in the SIGA.

There are two registers within the LCON, one that enables and disables the switch ports and clock, and another that monitors the I/O pins of the array. At system power-up time the TCS is responsible for selecting which clock is used to run a particular function board. This is done in the LCON, and thus with the LCON/TCS interface.

The B2VME has two LCONs and two SIGAs. The slave on this board has logic that selects which of the four gate arrays the slave serial peripheral interface talks to.

On a B2VME, the TCS can calculate the approximate fraction of time that a particular switch port is busy by using the LCON interface to sample the "frame" and "reverse" control signals.

SGA Interface

The TCS interface in the Switch Gate Array (SGA) implements the following capabilities:

- Set any bit on any output port (other bits are cleared) — data or control
- Read the state of any bit on any input port — data or control
- Enable or disable any of the four input ports
- Enable or disable any of the four output ports
- Read whether an input port is busy
- Read whether an output port is busy
- Read the priority bit of any output port, i.e., whether a low priority message will go through
- Read the version number of the SGA

The low-level interface is based on the four wires: Clock, Data In, Data Out, and Execute. These operate in the same manner as their counterparts on the SIGA and LCON.

The slave in a switch card pair (on the B2SS card) also has two output bits that clear the SGA's random number generator, or force it into the dead state for testing.

Processor Reset Control

The TCS can reset a function board's processor by writing a register designated as the processor reset register. (For example, in the B2VME, the TCS slave can assert the 88100 CPU's reset line. It can also independently reset each SIGA and the board as a whole, though these latter three are usually asserted simultaneously.) In addition to the reset function, this capability is also used to stop the processor while code is loaded into its memory space.

Temperature Monitoring

The function board's temperature is monitored using a temperature transducer and support circuitry connected to one of the slave's analog to digital converters.

Voltage Monitoring

The slave monitors card voltages by connecting them, scaled and offset, to the slave's analog to digital converters. Each slave monitors +5, –4.5 and –2 VDC, and other voltages appropriate to the particular card type. Bulk power is not monitored directly, but is inferred from the other voltages.

Voltage Control

The slave controls the card power supply by asserting a 3-bit value interfaced with the voltage converter block that converts bulk power to card power. The slave can control the power block output voltage of the +5 VDC and the –4.5 VDC supplies. Each of these two supply voltages can be set to one of six levels:

Off
On (nominal voltage)
+5% nominal voltage
—5% nominal voltage
+10% nominal voltage
—10% nominal voltage

Circuit Card LED Control

An assortment of LEDs are visible on each circuit card while it is installed in the card cage, typically indicating:

- TCS VCC present (green)
- Bulk 48 VDC (±24 VDC) present (green)
- Card VCC present (green)
- Card VEE present (green)
- TCS flag (amber)
- Card-specific data (on a B2VME, four green LEDs indicate "frame" and "reverse" for each switch port)

Of these, two are directly controlled by the TCS. One indicates "card TCS power on" and is connected with a resistor to the card's TCS VCC power. The other is controlled by the TCS slave and can be set on, off, or blinking. Blinking the LED on and off is one of the TCS slave's tasks, and does not require further intervention from the TCS master. Two blink rates are defined in the original design, fast (about 3 Hz) and slow (about 1 Hz).

The TCS flag amber LED is intended to point out cards that fail diagnostics, and as an aid to a service person in locating a particular card in a large system. The suggested convention for use of this LED is as follows:

on	= dead card
slow blink	= diagnostic in progress, or failure
off	= passed diagnostic
fast blink	= card locater signal

A hardware reset causes the LED to turn on. The slave should be programmed to leave the LED on at startup, so that an uninitialized or totally broken card will have its light on continuously.

The TCS master should start the LED blinking at the slow rate when it begins testing or configuration discovery. This indicates that the TCS has discovered the card, but has not yet approved it for use.

Once the TCS has completed diagnostics, the LEDs on cards that pass should be turned off. Cards that fail or become non-communicative will continue to blink at the slow rate.

The TCS master should provide a command that causes the LEDs on one or more slaves to blink at the fast rate to help a service person locate a particular card. The command should be arranged so that leaving the LED blinking indefinitely is difficult, probably by starting it blinking and then waiting for a carriage return to stop it again.

If the TCS discovers a failed card in the course of normal operation, it should try to set the LED blinking at the slow rate.

Finally, note that the LEDs on each B2SS/B2SR pair of switch cards are both controlled by the same slave, on the B2SS. This hardware connection should be masked by the TCS master. The two switch cards should be treated as separate entities, and their LEDs controlled independently as described above.

Slave Address Sensing

When a Butterfly II card is installed, eleven wires in the TCS slave interface stay high or are pulled low by the wiring on the card slot connector. This encodes that slot's TCS slave address. The slave processor compares this address to the address in TCS messages, in determining whether the message addresses this slave. The slave can read its 11–bit TCS address via a decoder circuit that selects the address register.

The TCS slave address is similar to the 9–bit *processor node number* that each function board obtains from the midplane: three bits each of bay, midplane and slot identity. For function boards, the bay, midplane and slot fields of its *TCS slave address* have the same values as those fields of its processor node number. The 9–bit scheme permits up to 512 function boards, the Butterfly II design limit; but it leaves no addresses for switch or clock cards. Slot field values above the 0–7 range are used to address these cards in the TCS slave address format.

The TCS slave address format is described further in section 6.7 on TCS bus protocol.

Slave Configuration Information

Several slave configuration values are written into the slave's EEPROM at the factory during final assembly and test. These values are listed below, including the number of bytes (ASCII characters) allotted. The list below reflects the initial implementation. Because the definition of EEPROM contents is tied to TCS firmware and software, not to hardware, this list should be taken as highly suggestive but not definitive. Please refer to TCS software and operation documentation for precise details.

- **Card type** (1 byte) — Describes what kind of card this slave is on. This value is used to implement broadcast messages and to select card–specific slave routines.
  0 = B2SS switch server card
  1 = B2SR switch requester card (has no slave in current implementation)
  2 = B2CLK clock card
  3 = reserved for clock buffer card for large machines
  4 = B2VME function board with 4 megabytes of RAM
  5 = reserved for B2VME function board with 16 megabytes of RAM
  6 = B2NFC null function card used in development and testing
  9 = reserved for B2SM switch middle column card
  Other values are not currently assigned.
- **Circuit card serial number** (16 bytes)
- **Artwork revision level** (2 bytes)
- **Electrical revision level** (2 bytes)

- **TCS slave code (EEPROM) revision level** (2 bytes)
- **Analog to digital converter calibration** — There is A/D converter calibration information for the 68HC11's analog to digital converters. This data calibrates the transducer systems that read the card voltages and temperatures. Having a calibration value(s) eliminates the need for precision voltage references and components. The details of this data vary from one card type to another; see the individual software specifications for each slave.
- **Temperature alarm setpoint** — Temperatures above this value will signal an error condition in the slave status byte during polling. See the individual software specifications for each slave for details.
- **Voltage within specifications setpoint** (1 byte) — Voltages deviating from nominal by more than this value, either above or below, will signal an error condition in the slave status byte during polling.
- **Timers** — Timeout values used by the slave program, including a T-bus access timeout (2 bytes).

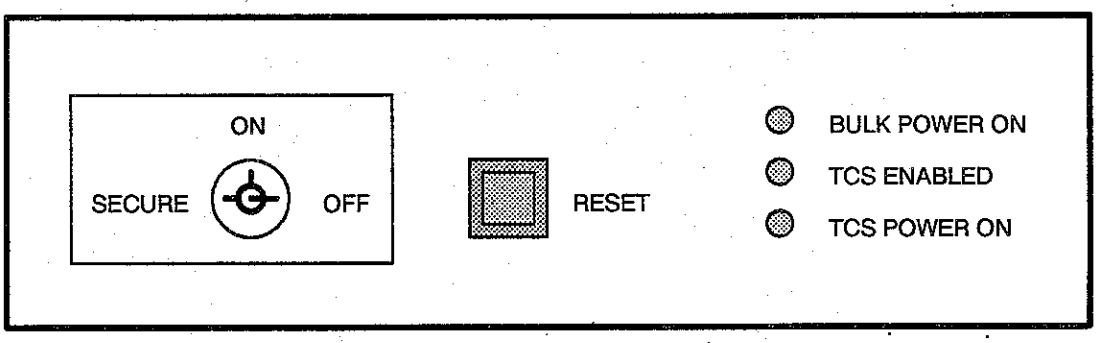
Amputating a Slave from the TCS Bus

The slave on each B2SS switch card can individually turn off the TCS slave-to-master serial signals from the eight function board slaves that are associated with that switch card. An 8-bit register accessible by the B2SS slave has a bit corresponding to each of the function board slaves. A bit in another register similarly controls slave-to-master data from the B2SS in the other switch.

6.3.4 TCS Front Panel

The TCS front panel contains three LEDs that indicate "TCS power on", "main system (bulk) power on", and "TCS enabled"; a "reset" button; and a 3-position keyswitch that selects "on", "off" and "secure".

Figure 6–5 TCS front panel (conceptual).



The **keyswitch** performs three functions.

1. Controlling power — Turning the keyswitch to the "on" position turns the TCS master power on; the "off" position turns it off. After the TCS master is on, the master is used to power on, start and reset the main system. The TCS master power is on in both the "on" and "secure" positions.
2. Disabling reset — The "secure" position disables the "reset" button, by electrically preventing the reset signal from reaching the TCS master CPU.
3. On/secure indication to TCS master — The TCS master can sense whether the keyswitch is in the "on" or the "secure" position, and take appropriate action. See TCS software and operations documentation for discussion of how TCS behavior is affected.

WARNING

Turning the keyswitch to "off" while the machine is powered up will immediately and completely power down the entire machine.

The **reset button** forces a reset of the TCS master CPU, if the keyswitch is in the "on" position. The TCS master CPU will then, upon successfully rebooting itself, examine the machine state and TCS configuration information to determine what action to take. For example, the master might run diagnostics, or wait for operator commands, or reset and reboot the Butterfly II machine, or continue the normal TCS monitoring and control of an already running Butterfly II machine.

The **LED indicators** provide the following information:

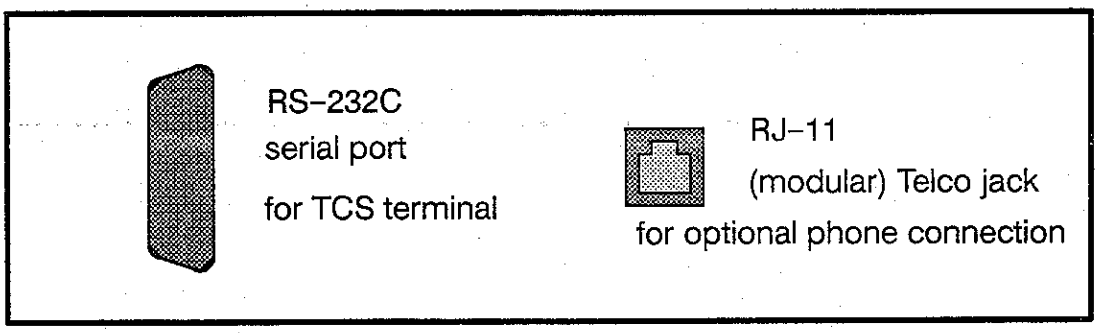
- **TCS power on** — This LED is illuminated by TCS power. It should be on when the keyswitch is in the "on" or "secure" positions, and dark when the keyswitch is "off".

- **Main system power (bulk power) on** — This LED is under control of the TCS master, and indicates that the master is commanding the machine's power distribution units to supply bulk power to the machine.
- **TCS enabled** — This LED is under TCS master control, and indicates that messages sent by the TCS master are driven onto the TCS bus to the TCS slaves in the rest of the machine.

6.3.5 TCS Back Panel

A panel at the back of the machine brings an RS–232 port and an RJ–11 phone line connector from the TCS master to the outside world. The RS–232 port is used to connect a machine control room terminal to the TCS master for use as a TCS console. The phone line connector is used for connecting a remote (over the phone line) terminal to the TCS for use by field service personnel for remote diagnostics.

Figure 6–6 TCS back panel (conceptual).



6.3.6 TCS Power Supplies

Two power supply systems are associated exclusively with the TCS, and the TCS obtains its power only from these supplies. One powers the TCS master, and is a simple, line–powered supply. The other powers the TCS slaves distributed throughout the machine, and is bused over the midplane system to the TCS slaves. One TCS slave power supply suffices for a machine of 64 function boards, and an additional supply is required for each additional 64 slots.

Powering the TCS slaves with a supply separate from the Butterfly II main power system is necessary for several of their monitoring and control features, such as turning individual cards on and off, and checking for proper card power voltages before allowing the card to run. A slave could not perform these operations if it drew power from the same source as the card it controlled.

The TCS master supply is a simple, line-powered supply that powers the TCS master.

6.3.7 Bulk Power Control

The Butterfly II bulk power supply system can be turned on and off by the TCS. There is direct control from the TCS interface card (B2TCS) in the form of multiple, independent, SPST relay contact closures. The 2-wire power control from each relay connects to each Power Distribution Unit (PDU) in the rest of the machine. Each PDU then applies the external AC power to the bulk DC power supplies within the PDU's cabinet. (The utility cabinet PDU, however, is controlled by the front panel keyswitch.)

The purpose of multiple power control relays is to allow staged power-on sequencing, to reduce power line transient surge. It also permits powering some sections of the machine while others remain off, such as during servicing. The original B2TCS has two relays; more are planned for later versions.

6.4 TCS Operational Capabilities

Sections above describe features of the TCS hardware. This section describes how those features can be used in operation of the Butterfly II machine. In keeping with the hardware context of this document, the capabilities described in this section reflect the intent of the design and are strongly suggestive of the operational capabilities implemented in TCS firmware and software. For a current, accurate and complete description of the capabilities, however, please refer to the TCS software and operations documentation.

The TCS plays a significant role in the Butterfly II machine in three principal areas: power-up; normal operational monitoring; and testing during manufacture, final assembly, and field service.

In this section, the acronym **POST** stands for Power-On Self Test, a test that a given hardware component performs upon itself, typically during power-up but optionally at other times as needed, to assure its proper functioning.

6.4.1 Power-On and Bootstrapping

The TCS manages the power-up sequence of the Butterfly II machine. This sequence includes the following steps:

- Bring up TCS — master runs power-on self test
- Examine, power up and initialize the Butterfly II machine
  - Read machine configuration and compare to the configuration file
  - Turn on circuit card power, card by card

  - Initialize SIGAs
  - Use LCON to select clock that runs function board processors (In the B2VME, the TCS slave additionally selects between an on-board clock and the switch clock.)
- Check operation of function boards
  - Hold function board processors reset
  - Disable switch accesses to and from function boards
  - Broadcast POST code into memory of function boards
  - Release reset on function board processors to let POST run
  - Repeat the broadcast and run steps for each function board type (if the machine contains others; the first version has only B2VMEs)
  - Poll for POST completion and status, from each function board type
- Check switch operation
  - Hold the processors reset
  - Broadcast a switch POST into memory of each function board
  - Release processors from reset to execute switch POST
  - Poll for POST completion and status
- Signal and log any errors discovered by POSTs
- Start machine
  - Hold function boards reset
  - Select a switch system
  - Select a value of hold time for the SGAs
  - Load a bootstrap program into a master function board (likely one with access to a disk)
  - Load a bootstrap server program into all other function boards
  - Load the "working" machine configuration table into low memory of function boards
  - Enable switch accesses to function boards
  - Release processors from reset to start the bootstrap code/sequence running
- Start TCS monitoring application running in the TCS master

The following sections discuss parts of the sequence described above.

Power-On Self Tests

There are several different types of power-on self tests in the Butterfly II machine. The TCS master has one in PROM (available only if a keyboard and display are attached to the TCS master), self tests for each type of function board are stored on the TCS's hard disk, and a Butterfly II switch POST is stored on the TCS hard disk. Each is written for the type of processor to execute it; for example, the function board and switch POSTs are 88000 code.

The TCS master POST has several stages. First, the TCS CPU card has a POST in on-board ROM. Besides testing the CPU board, the CPU ROM POST also checks for the existence of ROM in TCS master device controllers such as the hard disk. If this test fails, code in EPROM on the B2TCS card interprets the resulting error code and generates an appropriate message on the serial port.

Upon passing its ROM POST, the CPU boots DOS and examines the *autoexec.bat* file for commands. Normally, this file directs the CPU to boot the TCS application software. This boot sequence serves as an indirect self test of the TCS master. The TCS software then checks whether it can operate B2TCS functions correctly, another stage of POST. Finally, the software enables the TCS bus and checks whether it can communicate with any TCS slaves. This serves as the final stage of POST; if any slaves respond appropriately, the TCS master has passed its POST.

TCS slaves have no explicit POST. Rather, each slave attempts to respond to commands from the TCS master. If its response is incorrect, the master takes appropriate action which may include amputating the slave from the TCS bus.

Function board self tests are loaded into function board memory via the TCS/SIGA interface while the processor is held reset. After the POST is loaded, the processor is released into execution. While the self tests in one type of function board are executing, POST code for the next function board type can be loaded and started. The TCS then polls, waiting for the test to finish, and then reads the results of the test from the POST status register on each function board.

Any errors or outages are reflected in the configuration table that is broadcast into the low memory of the function boards, is signaled at the TCS terminal, and is recorded in the machine log on the hard disk.

The Butterfly II switch POST consists of function board code that is loaded into all of the function boards and started simultaneously. The program has each of the processors write test data into a block of memory in each of the other processors' memory. Switch performance is monitored during the test to verify that it is within acceptable levels. After the test has executed, the memory is checked to see that the transfer was successful, and test status is written into a register that is polled by the TCS master.

Another kind of switch test, used only as a diagnostic tool, consists of TCS slaves manipulating function board LCONs and switch card SGAs to test the connectivity between these cards. (LCONs and SGAs can individually assert and monitor their I/O pins.) This test runs on the TCS master.

Configuration

The TCS determines what number and type of circuit cards are in the system, and which of them is operating properly.

At system power-up time, the TCS determines what circuit cards are present and working in the system, and provides this information to the Butterfly II application. To accomplish this, the master polls every card slot for card identification information and builds a table in its memory of what it finds. Power-On Self Tests are run on the cards that are present, and the configuration table is updated to reflect which of them are operating properly. The resulting table is compared with the configuration file stored on the TCS hard disk and, if there are any discrepancies, an error message is displayed on the TCS terminal and the machine error log is updated. (The configuration file stored on the hard disk was created during initial system installation and it presumably corresponds with what should be installed in the system.)

The final table — that lists what cards are working in the system — is written into low memory of the master node Butterfly II function board for quick access by operating systems and application processes. This table is used to inform such software what hardware resources are available and how to configure memory interleaving.

6.4.2 Monitoring

The TCS spends most of its time polling the Butterfly II machine, gathering switch activity statistics, gathering a system temperature history, gathering a system voltage history, watching for error conditions, and watching for system console I/O activity.

Temperature and Voltage Monitoring

The TCS monitors the temperature and voltage on each of the Butterfly II function boards. Temperatures or voltages over a defined error threshold are reported to the TCS terminal and are logged in the machine error log. If a large number of over-temperature reports occur, the master shuts down the entire system.

Bulk power supply failures are reported in a similar manner. If the TCS master, through polling slaves for voltage measurements, detects that the power supply voltage is out of specified limits, it reports the error and shuts down

the system if necessary. If the slave detects an overtemperature condition, it turns off card power.

Temperature and voltage for each of the function boards is periodically read and recorded in the machine log in addition to watching for error conditions. This information can be presented on the TCS terminal to diagnose system problems and monitor degradation characteristics.

Switch Activity Monitoring

The TCS can monitor switch activity while the machine is in use. The TCS can also load and run switch POST, exerciser and diagnostic programs in function boards and in the TCS master itself; however, those programs are not run *while the machine is available to users.*

The switch activity monitoring facility uses the ability of the LCON in each function board to monitor any one of several switch signals. The TCS master can command a function board TCS slave to repeatedly sample a given signal, using the LCON. In the original implementation, 1024 samples are tallied during 11 milliseconds. This count is then scaled down to an 8–bit value, and can subsequently be retrieved from the slave by the TCS master. This count represents the fraction of the time that the selected signal was asserted.

This monitoring function is particularly powerful when invoked with the broadcast capability of the TCS bus protocol. All function boards can be told to begin their tally at once. When the sampling interval is over, the TCS master reads the count from each slave. Thus, a machine–wide measurement of switch activity during that interval is obtained.

System Console I/O

The TCS master periodically polls one of the function boards, looking for application I/O. When the master sees a service request, it reads a memory location that describes what kind of service is being requested. One type of service that can be requested is system console I/O. This path is used when the system console device driver writes to the console.

A particular function board is defined (by a configuration parameter) to be the system console I/O handler. When console information is typed on the TCS terminal, it is shipped out to a predefined buffer in that function board's memory, and the processor is interrupted by writing an appropriate T–bus register. An operating system demon handles the information in the buffer.

Mach Service Requests

Note: the capability described in this section is possible but is not currently implemented. It could prove useful in diagnostic situations, for system exercis-

er programs to communicate with the operator at the TCS terminal. During multi-user time sharing operation, however, its utility is questionable and could pose a threat to system stability.

The Mach (operating system) service request interface provides TCS control and information to application processes. Application processes can issue any TCS command much like the TCS terminal issues them. The I/O between the TCS and the application is handled like the system console I/O.

A particular function board is defined (by a configuration parameter) to be the application interface I/O handler. When the TCS needs to inform the application of some kind of system error, or of an impending processor shutdown, an appropriate code is written into a predefined buffer in that function board's memory, and the processor is interrupted by writing a T-bus register. Another operating system demon handles the information in this buffer.

6.4.3 Field Service and System Management Commands

The TCS controls power cycling, runs diagnostics, installs or removes circuit cards from software resources, resets the system, reconfigures the system, and so forth. The typical operator of the TCS is a system manager or a field service technician. Understanding the hardware design is easier with an understanding of its intended use. To provide that context, some likely commands are described below. For the commands currently implemented, please see TCS software and operations documentation.

[NOTE: FOLLOWING COMMANDS NEED A PASS TO BRING THEM UP TO CURRENT STATE OF /usr/bfly2/src/tcs-m/george/dev/george.doc]

Card Power On/Off

Turn a particular circuit card on or off. The operator gives the single card power on/off command and is prompted for slot address information.

Turning off the card power is the one action a TCS slave may take without an explicit command from the TCS master. Namely, if the card temperature exceeds an "alarm" threshold, the TCS slave can turn off card power.

Card Install or Remove

Informs the Butterfly II application and the configuration table that a circuit card should be either added or removed from the system's resources. This command would be issued when a card is installed or removed from a Butterfly II machine. The operating system(s) probably will not support addition or deletion of function boards to/from their set of boards in use. The card power on/off command may do all that the install/remove command needs.

System Power On/Off

Turn the entire machine on or off.

When the machine is powered up and down, the TCS power control is used to reduce strain on the system's power supply by gradually turning cards on or off. Cards will likely be turned on or off one at a time with a short pause after each power command.

Single Card Reset

Reset a particular card. The operator is prompted for slot information, and the card in that slot is reset.

System Reset

Reset the entire system. Causes the master to broadcast a reset command to all of the slaves.

System Restart

Initiates the complete power–up sequence and boots up the machine.

Run System POST

Runs the system's Power–On Self Test. This command initiates the same testing sequence that is used during system power–up. Each component is tested with the appropriate POST code, and the results of each are displayed on the TCS terminal. This command would likely be used to debug a system that is failing its power–on self test.

Run Card or Switch POST

Runs the Power–On Self Test appropriate to a particular card, or to the Butterfly II switch, and reports the results on the TCS terminal. The test is the same one that is used during system power–up.

Run System Diagnostic

Runs an extensive set of diagnostics on the entire machine. Diagnostic code differs from POST code in that it is meant to determine *what* is wrong with the card rather than simply *whether* the card is working or not working. Diagnostic code is not required to complete as quickly as POST code.

Each type of function board has diagnostic code stored on the TCS hard disk. There is an analogous piece of code for the Butterfly II switch that includes an extensive test of continuity between the function boards and the switch cards using LCON and SGA I/O pin testing commands. Test results are reported to the TCS terminal.

Run Card or Switch Diagnostic

Runs diagnostic code on a particular function board or the switch. The operator is prompted for information about what they would like to test. Results are reported to the TCS terminal.

Run TCS System Diagnostic

Runs a diagnostic program to diagnose problems with the TCS system. The goal is to identify the field replaceable unit that has failed.

Capture Switch Traffic Data

Records average switch traffic rate for a specified switch port over a specified time interval.

Show Card/System Temperature/Voltage

Displays the temperature or power supply voltages on the specified card or group of cards, listed for each slot in the system, or averaged over all cards in the system.

Show Card Identity

Displays the following information about the card found in a slot the operator specifies.

- Card type
- Card revision level
- Slave code revision level
- Card serial number
- Gate array revision levels

6.5 B2TCS Functional Specification

6.5.1 Introduction

The B2TCS is a PC/AT format circuit card that comprises the Butterfly II machine's Test and Control System (TCS) interface circuitry. It is meant for use in the TCS master processor, a PC/AT compatible central processing unit and system bus.

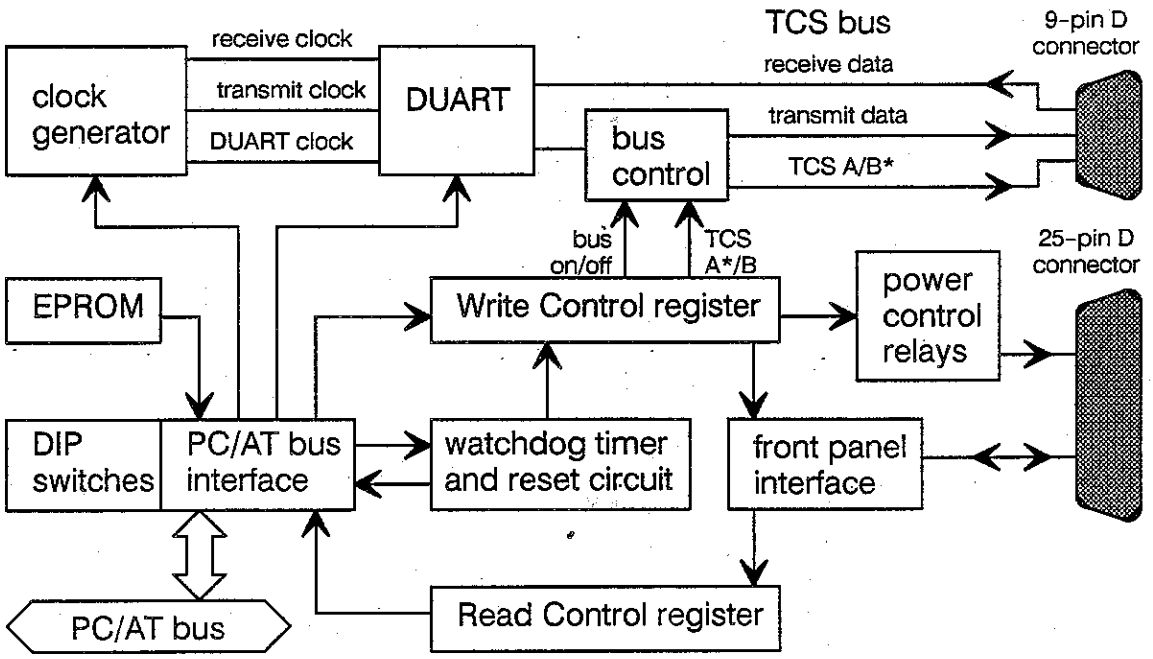
6.5.2 Major Card Functions

The B2TCS has the following major components:

- PC/AT bus interface
- TCS bus interface
- Front panel interface
- Power Distribution Unit (PDU) interface
- TCS master watchdog timer
- TCS master power-up EPROM

The functions performed by these components are described below. Figure 6–7 shows the functional organization of the B2TCS.

Figure 6–7 B2TCS functional block diagram.



PC/AT Bus Interface

The PC/AT bus interface decodes the B2TCS's devices and registers in the PC/AT's I/O space, and the B2TCS's EPROM in the memory space. The addressing parameters for the two spaces are independent.

TCS Bus Interface

The TCS bus is a TTL level asynchronous serial communication bus that connects a variable number of TCS slave processors to a TCS master processor. A TCS slave processor is a single-chip microcomputer equipped with a serial communication interface. There is one slave processor on each Butterfly II circuit card (except for the B2SR). The slave controls card functions such as power, memory access, and hardware test.

The TCS bus originates from a 2681 DUART on the B2TCS. The DUART receives all its clocks from an 82C54 programmable timer. Both the transmit and receive sides of the TCS bus operate at communication speeds of up to 125 kilobits per second. The TCS bus is brought to a DE-9 connector at the edge of the B2TCS, where a cable connects it to the B2CLK card for further buffering and distribution to the rest of the Butterfly II machine.

The transmit side of the TCS bus is driven by a CMOS buffer enabled under software control. After a TCS master reset, the TCS bus is disabled and must be re-enabled by the master processor before TCS messages can pass to the rest of the system.

Front Panel Interface

The TCS controls the Butterfly II front panel (B2FP). The front panel comprises the following:

- TCS master reset button
- Keyswitch with positions "power off", "power on", and "power on, secure"
- Main power on indicator LED
- TCS enabled indicator LED
- TCS power on indicator LED
- TCS general purpose indicator LED

These functions are accessible with the B2TCS's Control Read register and Control Write register. The B2TCS and the front panel are connected with a cable from the B2TCS's DB-25 connector.

The front panel's reset button is OR'ed with the reset signal generated by the B2TCS's watchdog timer. Depressing this momentary-contact reset button asserts reset on both the B2TCS and the master's CPU card. The TCS master processor re-boots in response to such a reset.

At boot time, the master may read the B2TCS Control Read register to determine whether it is powering up from a reset, or from a power cycle. The "main power LED on" and "relay X on" bits are cleared only by a power cycle, not by a TCS master reset.

The front panel keyswitch controls line power to the TCS by providing a contact closure to a Power Distribution Unit in the Butterfly II utility cabinet. Setting the keyswitch to "power on" turns on the TCS master processor. The TCS can distinguish between the "power on" and the "power on, secure" positions by reading the Control Read register on the B2TCS.

The front panel has three LEDs. One indicates that the TCS thinks the main system power is on; another indicates that the B2TCS's TCS bus drivers are enabled; and the last is a general purpose indicator LED. These are all controlled with the Control Write register.

PDU Interface

The TCS controls the Butterfly II Power Distribution Units (PDUs) with two relays on the B2TCS. The relays' contacts are brought out to a DB–25 connector on the card edge. The relays are controlled via the B2TCS's Control Write register described below. Their function at the connector can be modified with the jumpers JMP1, JMP2 and JMP3. Their state is read back in the Control Read register.

Watchdog Timer

The watchdog timer is a counter that is started and stopped with the "watchdog enable bit" in the Watchdog Control register. A write to the Watchdog Poke register resets the timer's count and lets it continue counting. If the counter reaches its end value, a reset signal connected to the TCS master's CPU card is pulled low, resetting the CPU. When the CPU is reset, the TCS master re–boots. The watchdog timer will fire if it is not poked at least once every 20 seconds.

B2TCS Reset

A B2TCS reset occurs when the watchdog timer expires, or when the front panel "reset" switch is pressed while the keyswitch is in the "on" position. It causes a standard PC/AT reset of the TCS master processor, and resets logic on the B2TCS card, with the exception that certain bits in B2TCS registers are unaffected. In particular, bits controlling main power to the rest of the machine are unaffected, so main power is not removed by resetting the TCS master. The effect on each bit of B2TCS control registers is described in the register summary.

B2TCS EPROM

The B2TCS includes socket(s) for EPROM memory. DIP switches described in section 6.5.3 configure the EPROM base address and size. The EPROM is the only addressable component of the B2TCS that resides in the PC/AT memory space; all others reside in I/O space.

Code in the B2TCS EPROM sends power–up error messages to the TCS terminal.

The present version of the B2TCS contains 64 kilobytes of EPROM. Future versions of the card might contain additional memory, holding an image of the DOS operating system and a subset of the TCS application program. This would improve the mean time between failure by allowing the TCS master to operate without its hard disk.

6.5.3 B2TCS Address Space

The PC/AT bus is addressed with a 24-bit address bus. (The four topmost bits are used for addressing extended memory; these must be latched at the beginning of the bus cycle. The 20 principal address bits are valid throughout the cycle.) There are two address spaces, a 24-bit memory space and a 10-bit I/O space. The two spaces share the same address lines, but are distinguished by different read and write strobes.

The B2TCS TCS bus, the watchdog timer, and the front panel control registers are decoded in the PC/AT's I/O space. The EPROM is decoded in the memory space.

B2TCS Memory Space

The B2TCS includes a 64-kilobyte EPROM. Its base address, and the size of the address range in which it responds, are configured by DIP switches. It may be configured as a block of 2, 4, 8, 16 or 32 kilobytes.

B2TCS EPROM Configuration Switches

There are two sets of related switches, one that sets the memory block size and another that sets the block's base address. The switches are divided up between the card's two DIP switch banks SW1 and SW2. Figure 6-8 defines the meaning of each switch.

There is a code version switch that forces the EPROM to respond with either the upper half or the lower half of its 64-kilobyte space. Thus the EPROM can hold two different code versions, and this switch selects between them. The setup of the base address and size remains unchanged by this switch.

NOTE

The EPROM base address must be set to a value between 0x0C8000 and 0x0E0000 on a 2-kilobyte boundary, because these are the locations which the PC's power-on self test (POST) looks for additional EPROM. See the *IBM PC/AT Technical Reference* for more information about EPROM code requirements.

Figure 6-8 EPROM configuration switch definitions.

EPROM size selection

DIP switch SW2 s8	s7	s6	EPROM memory block size
off	off	off	2 kilobytes
off	off	on	4 kilobytes
off	on	off	8 kilobytes
off	on	on	16 kilobytes
on	off	off	32 kilobytes

switch position key

off = open = ONE
on = closed = ZERO

EPROM base address selection

DIP switch SW2 s5 addr <19>	s4 addr <18>	s3 addr <17>	s2 addr <16>	s1 addr <15>	DIP switch SW1 s3 addr <14>	s2 addr <13>	EPROM base address
on	on	on	on	on	on	on	0x000000
off	off	off	off	off	off	off	0x0FE000
off	off	on	on	off	on	on	0x0C8000 (recommended)

EPROM code version selection

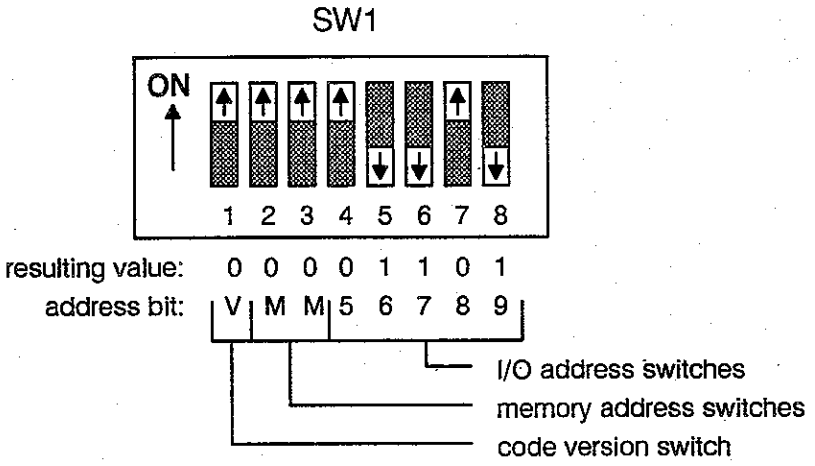
DIP switch SW1 s1	EPROM code version
on	code version zero (bottom portion of EPROM)
off	code version one (top portion of EPROM)

B2TCS I/O Base Address

The I/O base address of the B2TCS is selectable to avoid conflicts with future cards installed in the TCS master processor. The card's I/O base address is set with DIP switch SW1. SW1 sets the 5 most significant bits of the I/O base address (ADD<9..5>). Setting a switch to the "ON" position makes the corresponding address bit a zero. The recommended I/O base address is one set aside by IBM for prototype cards. All TCS software expects to find the B2TCS at I/O base address:

0x02C0

Figure 6–9 **Recommended I/O base address switch setting.**



Switch 1 on the I/O base address DIP switch bank selects between two versions of EPROM code. Switches 2 and 3 set EPROM base address bits 13 and 14, respectively. See Figure 6–8.

B2TCS Device Offsets

The offsets to devices and registers on the B2TCS card are listed below. The DUART and the Counter/Timer entries in the list are the offsets to the base address of the component; the address of each internal register of these components is the sum of the register number, the component's device offset, and the B2TCS's base address.

0x000	Control Read register and Control Write register
0x001	Watchdog Poke register (write only)
0x002	Watchdog Control register (write only)
0x004	Programmable Counter/Timer (four read/write registers)
0x010	DUART (16 read/write registers)

6.5.4 B2TCS Register Summary

Each of the B2TCS registers and addressable components listed in section 6.5.3 is described below, in alphabetical order. Access is by byte only.

Control Read Register

The B2TCS's Control Read and Control Write registers manage the Butterfly II front panel and PDUs. Note that the main power control bits and main pow-

er indicator bit are reset only at power-up, not by a reset of the TCS master. The Control Read register has the following bit definitions.

Figure 6–10 Control Read register.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	secure*	B2TCS revision level	watchdog timer on	on UPS power	main power LED on	TCS bus enabled*	relay 2 on	relay 1 on
power-on state:	EXT	REV	0	EXT	0	1	0	0
reset state:	EXT	REV	0	EXT	SAME	1	SAME	SAME

KEY: EXT = bit is driven by off-card source
REV = static card revision level indicator
SAME = bit retains value it had before TCS master reset

secure* This bit is driven by a keyswitch on the front panel. If no front panel is connected, the bit reads high (1).
0 — Indicates that the keyswitch is in the "power on, secure" position.
1 — Indicates that the keyswitch is in the "power on" position.

B2TCS revision level This bit differentiates between a B2TCS with EPROM and a B2TCS without EPROM. (Early versions had no EPROM.)
0 — Indicates that this is a B2TCS without EPROM.
1 — Indicates that this is a B2TCS with EPROM.

watchdog timer on This bit is controlled by the Control Write register described later. When this bit is asserted, the B2TCS watchdog timer must be poked at least every 20 seconds to avoid a reset to both the B2TCS and the CPU card. This bit is cleared at power-up, and at TCS master reset.
0 — Indicates that the watchdog timer is off.
1 — Indicates that the watchdog timer is enabled.

on UPS power This bit is driven by the DB-25 connector at the B2TCS's card edge. The connector's pin is connected to a signal on an Uninterruptible Power Supply (UPS) that indicates it is running on battery power rather than line power.
0 — Indicates that the UPS is running on line power.
1 — Indicates that the UPS is running on battery power.
The present Butterfly II implementation does not support support UPS power. This bit anticipates possible future implementations that do.

main power LED on This indicator bit is controlled by the Control Write register described later. When this bit is asserted, an LED illuminates on the front panel. This bit is cleared at TCS power-up, but not by TCS master reset.

0 — The TCS master processor believes that main power is off.
1 — The TCS master processor believes that main power is on.

TCS bus enabled* This bit indicates the state of the *TCS bus enable* bit in the Control Write register. This bit is an inverted version of that bit. This bit is set at power-up and at TCS master reset.

0 — Indicates that the TCS is able to send messages on the TCS bus; the front panel "TCS enabled" LED is on.
1 — Indicates that the TCS bus is disabled; the front panel LED is off.

relay 2 on
relay 1 on These bits indicate the state of the corresponding bits in the Control Write register. These bits are cleared at TCS power-up, but NOT by a TCS master reset.

0 — Indicates that the relay and the associated PDU are off.
1 — Indicates that the relay and the associated PDU are on.

Control Write Register

The Control Write register has the following bit definitions.

Figure 6–11 Control Write register.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	spare	spare	indicator LED on	TCS bus A*/B	main power on LED	TCS bus enable	relay 2 control	relay 1 control
power-on state:	–	–	0	0	0	0	0	0
reset state:	–	–	0	0	SAME	0	SAME	SAME

KEY: SAME = bit retains value it had before TCS master reset

indicator LED on This bit controls a general purpose indicator LED on the Butterfly II front panel. This bit is cleared ar power-up and by a TCS master reset.

0 — General purpose indicator LED is not illuminated.
1 — Illuminates the LED.

TCS bus A*/B This bit controls the TCS bus selector signal on pin 6 of the TCS bus. This bit is inverted before driving the bus wire, as implied by the names: the bit is A*/B, the signal is A/B*. This signal tells the B2CLK card which bus, A or B, it is on. The B2CLK sets its TCS address with respect to this signal. This bit is cleared at power-up and by a TCS master reset.

0 — Indicates that this master is using TCS bus A.
1 — Indicates that this master is using TCS bus B.

main power on LED This bit controls the "main power on" LED on the Butterfly II front panel. This bit is cleared at power-up, but not by TCS master reset.

0 — "Main power on" LED is not illuminated.
1 — Illuminates the LED.

TCS bus enable This bit controls whether TCS bus messages sent to the B2TCS DUART are driven out onto the TCS bus or not. It also controls the "TCS enabled" LED on the front panel. This bit is cleared at power-up and by TCS master reset.

0 — TCS messages are not driven onto the TCS bus; LED is off.
1 — TCS messages are driven onto the TCS bus; LED is on.

relay 2 control
relay 1 control These bits open and close two relays on the B2TCS. These relays are intended to control Power Dustribution Units, and thence main power for the rest of the machine, as described in section 6.5.2. At power-up, these bits are cleared, opening the relays, but a TCS master reset does not affect them.

0 — The relay is open.
1 — The relay is closed.

Dual Asynchronous Receiver/Transmitter (DUART)

The TCS bus is driven by a 2681 DUART. One channel is used for the TCS bus. The other channel is unused in the original version of the B2TCS, but reserved for use as a second TCS bus in a later version of the B2TCS. Sixteen internal registers control the DUART. See chip manufacturer documentation for details on addressing and programming the device. The 2681 is manufactured by Signetics, Motorola, and United Microelectronics.

Programmable Counter/Timer

The clocks that drive the 2681 DUART are generated by an 82C54 programmable counter/timer chip used as an interval timer. Four internal registers control the device. See chip manufacturer documentation for details on addressing and programming the device. The 82C54 is manufactured by Oki, Intel, Harris, Toshiba, and others.

Watchdog Control Register

Bit 0 is the only defined bit in this register; no other bits are implemented. This is a write–only register.

Figure 6–12 Watchdog Control register.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	spare	spare	spare	spare	spare	spare	spare	enable watchdog timer
power–on state:	–	–	–	–	–	–	–	0
reset state:	–	–	–	–	–	–	–	0

enable watchdog timer

This bit controls whether the watchdog timer is counting or not. Setting this bit does **not** clear the watchdog timer. Therefore, the software should poke the watchdog before setting this bit, or ensure by earlier use that the timer is not about to expire.

0 — Disables the watchdog timer.

1 — Enables the watchdog timer.

Watchdog Poke Register

There are no data associated with this register. A write of any data to the address of this register resets the count in the watchdog timer, thus poking the watchdog. This is a write–only register.

6.5.5 B2TCS Controls and Indicators

The only manually alterable controls on the B2TCS are two DIP switches for address control, and jumpers that select the type of PDU control used. The the function of the switches and jumpers is described below. The B2TCS physical layout is shown in Figure 6–13.

Connectors

J1 — reset signal to CPU card, B2TCS drives J1 pin 1 with TTL logic signal RESET*

J2 — two jumper pins, clock oscillator enable (jumper normally installed).

J3 — DE-9, 9-pin male, TCS bus
pin 2 = TCS_IN* (slave to master)
pin 3 = TCS_OUT* (master to slave)
pin 6 = TCS_A/B*
pins 7, 8 = ground
pins 1, 4, 5, 9 = no connection

J4 — DB-25, 25-pin male, front panel interface and power control

J5 — 12 jumper pins, DUART interrupt level to PC/AT bus (jumpers normally absent) — Jumper pins J7 allow the B2TCS DUART to assert an interrupt on the PC/AT bus. Normally this capability is not used, so no jumpers are installed on J7. Figure 6-14 shows how jumpers in J7 specify the interrupt level asserted by the DUART.

Jumpers

JMP1 — three jumper pins, power control relay number 2 —
jumper on pins 1 & 2 = relay provides contact closure between J4 pins 10 and 23
jumper on pins 2 & 3 = relay applies 24 VAC* to J4 pin 10, no connection to J4 pin 23

JMP2 — three jumper pins, power control relay number 1 —
jumper on pins 1 & 2 = relay provides contact closure between J4 pins 9 and 22
jumper on pins 2 & 3 = relay applies 24 VAC* to J4 pin 9, no connection to J4 pin 22

JMP3 — three jumper pins, front panel keyswitch contacts "number 2" —
jumper on pins 1 & 2 = switch closure connects J4 pin 12 to pin 25
jumper on pins 2 & 3 = switch closure applies 24 VAC* to J4 pin 12, no connection to J4 pin 25

* The application of "24 VAC" on the specified pins of J4 assumes that 24 VAC comes into J4 pin 13. Alternatively, J4 pin 13 could be connected to ground, in which case the specified pins would be grounded by the action of the keyswitch or relay.

DIP Switches

SW1 — DIP switch bank, B2TCS I/O base address, see section 6.5.3

SW2 — DIP switch bank, B2TCS memory base address, see section 6.5.3

Figure 6–13 B2TCS physical layout.

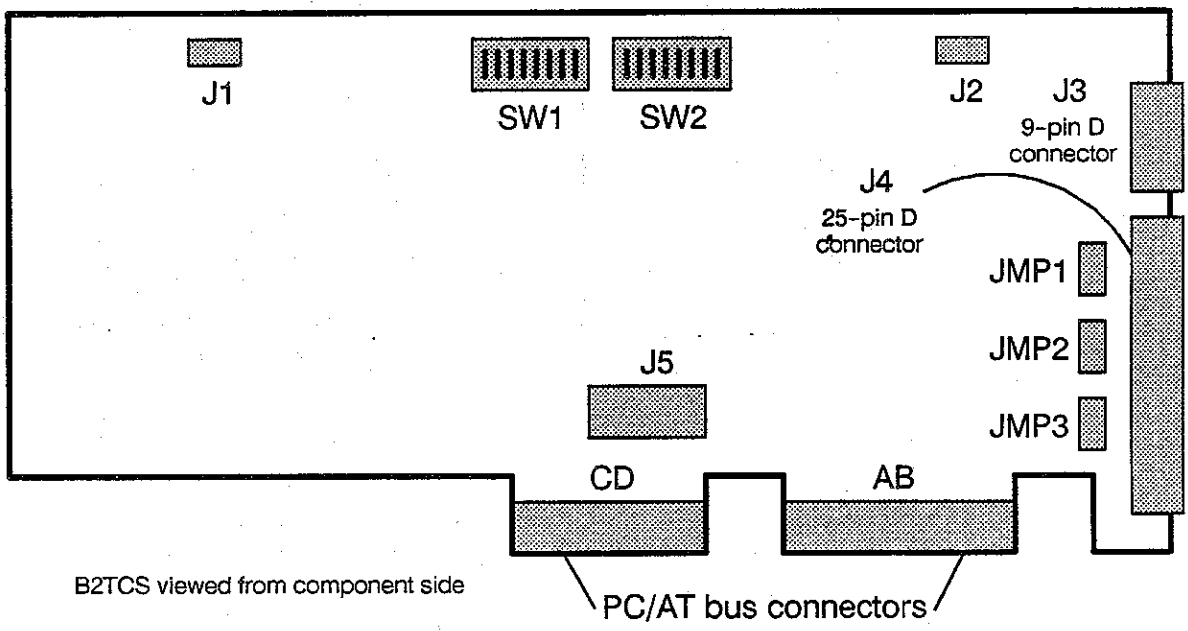
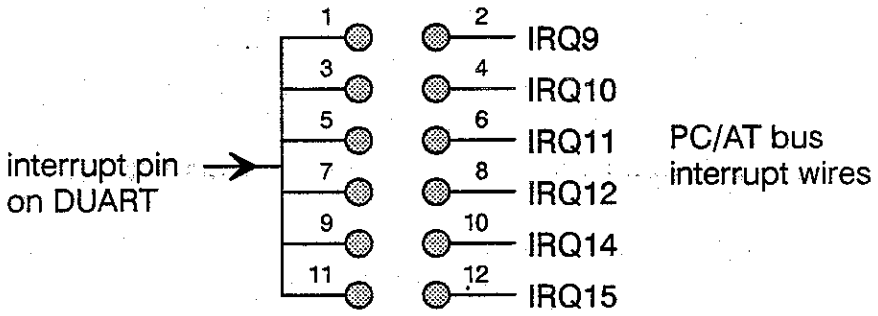


Figure 6–14 B2TCS DUART interrupt level select jumpers.



6.6 Redundancy Support

6.6.1 Redundant Modules

At the beginning of the design cycle for the Butterfly II hardware, support for non–stop operation at unattended sites was a requirement. To meet this requirement, the Butterfly II architecture supports hot sparing of hardware modules, amputation of failed modules, power–on servicing of failed modules, and other related features. The requirement for non–stop operation was dropped, but many of the hardware features were kept. In the first release of the machine, there is no software support for these features. In later releases, we expect to use these features to improve system availability.

The Test and Control System plays a central role in the management of backup modules, performing operations such as amputation and power–down of failed modules, switchover to backup modules, off–line testing of backup modules, and other functions. In addition, the design of the TCS itself includes a backup TCS bus, and has taken into consideration provisions such as backup TCS power and a backup TCS master. This section describes TCS hardware and software features designed to support backup modules. None of these features are supported in the first release of the machine, and only some of them are implemented. We also describe some problems that have not been addressed. This section should be viewed as a work in progress, to be completed as we develop marketing and service strategies to complement it.

The terms "redundant" and "backup" are used here interchangeably.

6.6.2 List of Backup Features

The following features are included in the Butterfly II design, to varying degrees:

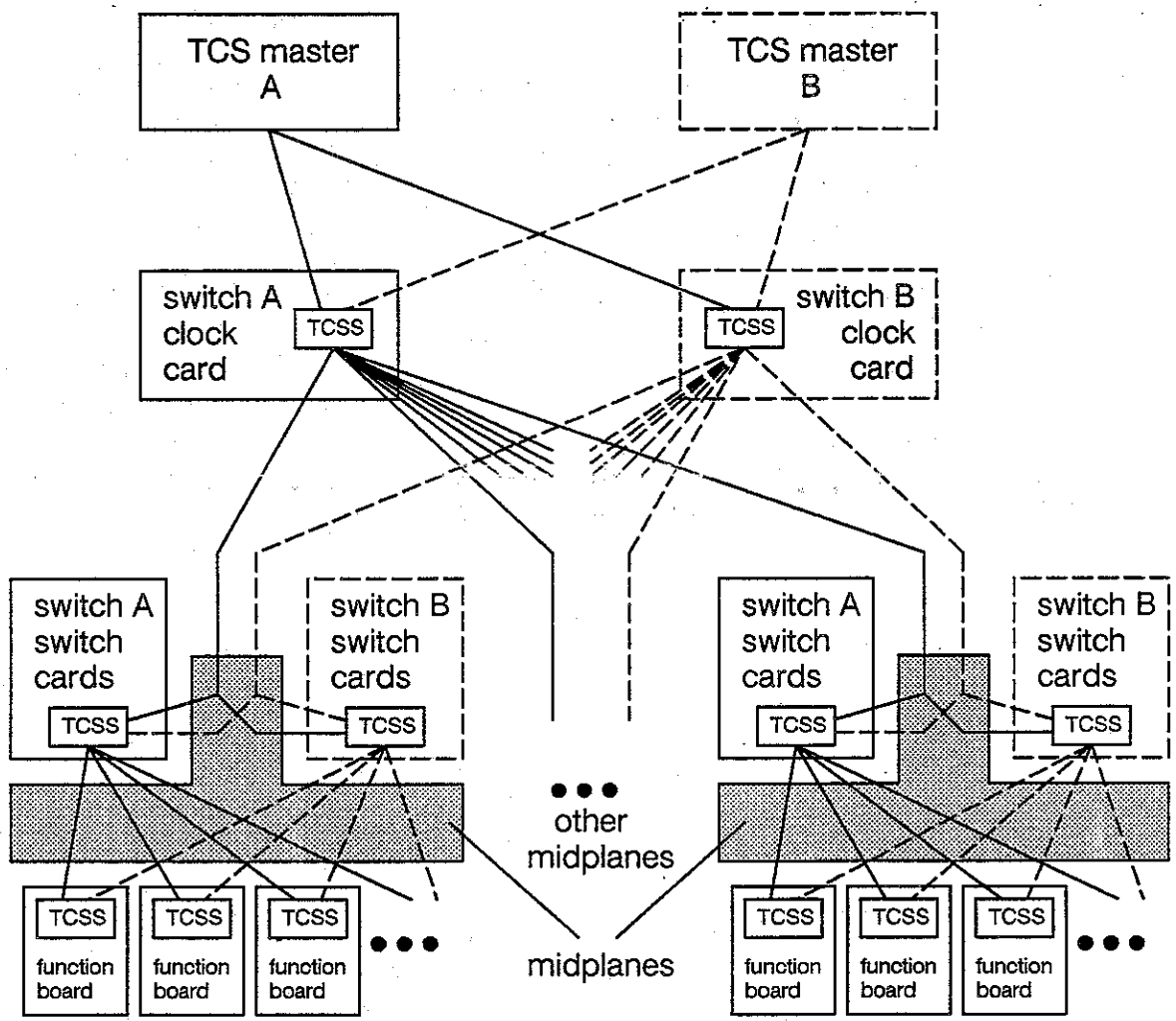
- Backup switch system — includes backup switch cards, a backup clock card, a backup switch interface (LCON and SIGA) on each function board, and means to select between the two switches
- Backup TCS master — includes a backup TCS master itself, a backup TCS bus distributed throughout the machine, and means to select between the two TCS buses at each TCS slave
- Backup TCS power — the TCS master could be powered by a battery backup system

These backup features are independent; any one or two, or all three, could be implemented. The support for redundancy is therefore somewhat complex, especially in the backup switch and/or backup TCS master area, so keep in mind their independence while reading the discussion below.

6.6.3 Redundant Switch and/or TCS Master

In a Butterfly II system configured for redundancy, there may be two independent TCS masters and/or two independent Butterfly switches. That is, system availability and reliability can be enhanced by duplicating the TCS master, or the switch, or both. Figure 6–15 shows these optional, redundant components with dashed lines.

Figure 6–15 TCS block diagram — communication and redundancy.



In a Butterfly II machine with a redundant TCS system, there are two TCS back panel connectors for the TCS terminal — one for each TCS master. If the primary master dies and the system operator needs to issue a TCS com-

mand, the terminal is disconnected from the primary terminal connector and connected to the backup one.

6.6.4 Redundant TCS Bus

A Butterfly II machine may be configured with a redundant TCS master, a redundant switch (and clock card), both, or neither. The arrangement of TCS bus(es) in these four configurations is as follows.

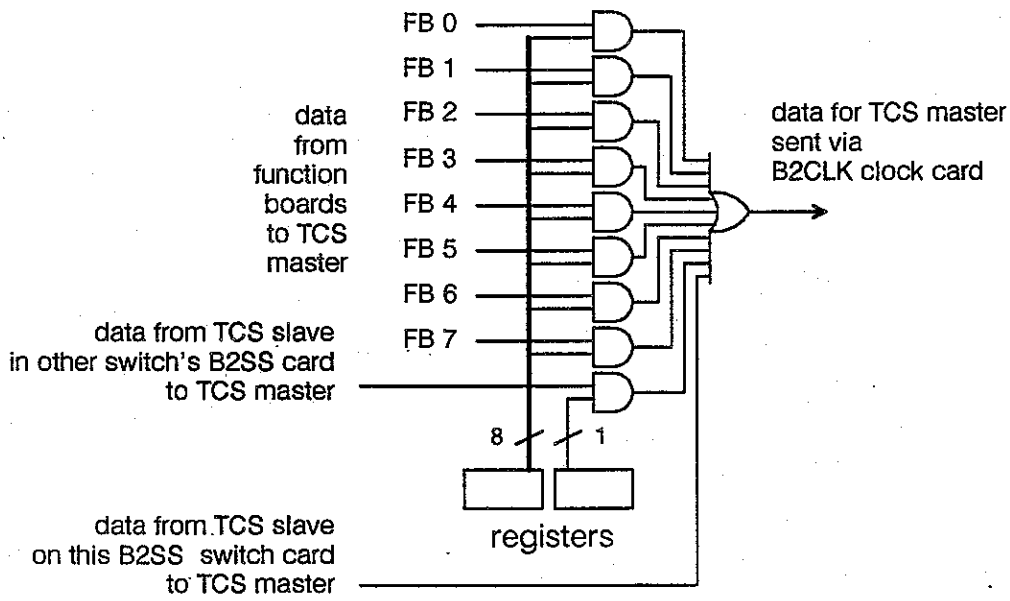
- In a simple machine without redundancy, one TCS bus cable is driven by the TCS master, to serve one B2CLK clock card. From there, the "bus" is fanned out for data from the master, and fanned in for data to the master.
- In a Butterfly II machine configured with a redundant TCS master but only one switch and clock, both TCS masters supply their own TCS bus to the clock card.
- In a machine configured with a redundant switch but only one TCS master, each switch has its own clock card, and the TCS master connects a separate TCS bus to each clock card. (In an early version, the TCS master connects to only one clock card, and that card in turn connects to the other clock card.)
- In a machine with not only a redundant TCS master but also a redundant switch, each TCS master supplies a TCS bus to each clock card, as shown in Figure 6–15. (Again, an early version has each clock card passing the TCS bus from its master on to the other clock card.)

In a machine configured with two TCS masters, there are two TCS buses, bus A and bus B. TCS bus A comes from the primary TCS master, and TCS bus B comes from the backup TCS master. Each bus consists of a transmit data wire (master to slave direction), a receive data wire (slave to master direction), a ground wire and a TCS identity (A/B*) wire. The A/B* wire tells the B2CLK clock card which TCS master is on the other end of the bus. The TCS bus A/B* wire goes only as far as the B2CLK card; it is not continued on to switch cards and function boards. (Switch cards have a *separate* A/B* wire that tells them whether they are a part of the A or the B *switch system*.)

6.6.5 TCS Bus Fan-in Details

On each midplane, the TCS slaves on the eight function boards are connected to *two* B2SS switch card slots: the primary switch card and an optionally installed, backup switch card. On each B2SS card, there are nine AND gates controlling slave-to-master data — the eight from function boards, and a ninth from the TCS slave on the corresponding B2SS card in a machine configured with a redundant switch. Thus, a B2SS slave can amputate the other B2SS from the slave-to-master bus, just is it can any of the eight function boards. Figure 6–16 shows this circuit.

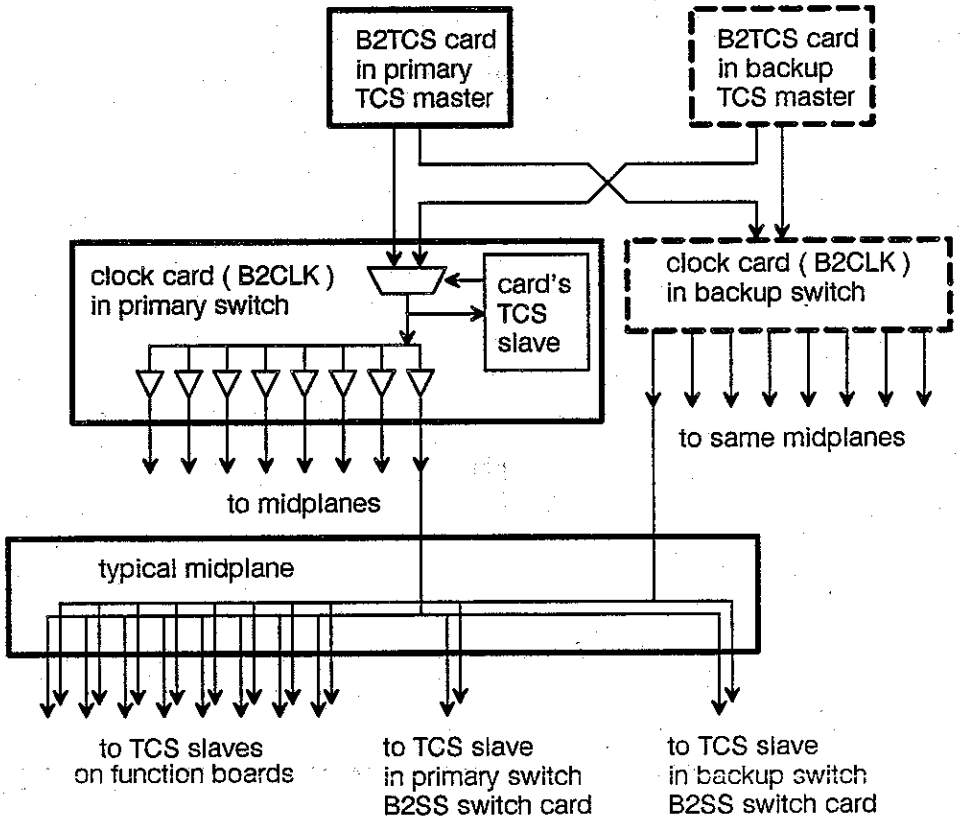
Figure 6–16 TCS bus fan–in with redundancy.



6.6.6 TCS Bus Fan–out Details

In the original B2TCS design, only one TCS bus is provided. The plan for future versions of the B2TCS is to include a second TCS bus, electrically independent of the first. The TCS master CPU controls which bus is used. The dual–bus design, shown in Figure 6–17, is more robust in systems with redundant switches. In the original design, the single bus from a TCS master is cabled, in a redundant–switch machine, to both B2CLK cards. This is less robust, since a failure in either B2CLK could corrupt the single TCS bus.

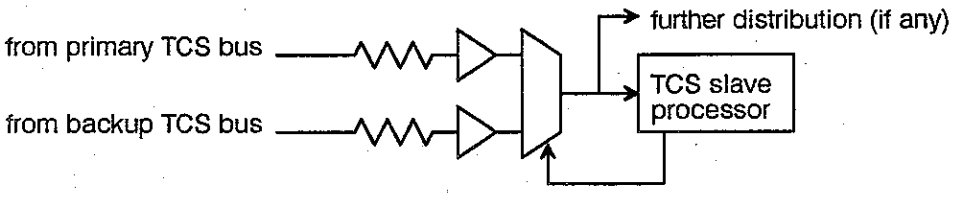
Figure 6–17 TCS bus fan–out with redundancy.



On each midplane, the TCS bus from each clock card is distributed not only to the eight function boards, but to *two* switch card pairs, primary and backup. In each pair, the B2SS card of the pair holds the TCS slave that services that pair.

Figure 6–18 shows the receiver circuitry at each TCS slave. Each resistor provides isolation from the bus, so that a failure in this slave does not corrupt the bus (such as by grounding it). (While the goal is immunity to single–point failures, calculations indicate the bus will withstand up to eight shorts to ground on the down–stream side of the protection resistors.) Alternatively or additionally, separate chips are used to receive the primary and backup buses to prevent a single point failure from locking up both transmit buses. The TCS slave processor controls a selector that chooses which TCS bus to receive. The chosen bus provides input to the slave processor, and also (in the case of the B2CLK) is redistributed further out the fan–out tree.

Figure 6–18 Master–to–slave circuit at each slave.



Among the TCS slave configuration information stored in EEPROM is a TCS bus monitor timeout value. The slave uses this value in determining whether to keep listening to the current bus, or try listening to the other bus.

6.6.7 Redundant Power for the TCS Master

The TCS master supply powers the TCS master(s). In the first implementation, this is a simple, line–powered supply. Two possibilities have been considered for increasing the machine's robustness: a UPS, and a redundant supply. The TCS master(s) would use an Uninterruptible Power Supply (UPS) only if the possibility of the TCS hard disk getting corrupted by a power failure is deemed too great. The TCS design includes the ability to read the "now on battery power" bit from a UPS. On the other hand, the Butterfly II TCS power supply may be backed up by a duplicate, redundant supply. The initial version has the capability for redundant TCS power, but the machine is not normally shipped with one.

6.6.8 Redundant Control of Bulk Power

The exact circuit for bulk power control with two TCS masters has not been decided; it is desirable that a failed TCS master neither prevent turning on bulk power, nor prevent turning it off.

6.6.9 Backup TCS Testing

The backup TCS (master and bus) is periodically tested to assure that it is available if the primary has a problem. This testing is similar to the mastership negotiation process described in section 6.6.10. The slaves are periodically told to listen to the backup master, and it polls the bus to verify that it can communicate with all of the TCS slaves.

6.6.10 Mastership Negotiation

This section applies only to a Butterfly II machine configured with redundant TCS masters.

At TCS power–up, the two TCS master processors negotiate with each other to determine which has control of the TCS system. The master processor that can communicate with the largest number of slaves wins mastership. In the case of a tie, one of the master processors is configured as the "A" or primary master and wins in a tie, while the other is the "B" or backup master. The primary and backup designations are also used to determine the order in which masters attempt to talk to slaves during negotiation; the primary master goes first.

After the master processors execute power–on self tests, they go on to poll the TCS bus for card ID responses from the slave processors. The two master processors take turns polling their respective buses, counting the number of slaves they find.

The configuration of data paths is as follows. Each master transmits on one of two TCS transmit buses (the transmit direction is defined as master to slave), and can select to listen to either of two receive buses for input. The slave processors control which bus they communicate with, by setting a control signal output. Each slave talks to one bus at a time, and both transmits and receives on that bus.

When the system first comes up, neither of the TCS masters have taken mastership, and the slaves have no idea which bus or master is working. The slaves alternate between the two buses, listening to one for a while, then listening to the other. When valid TCS messages are detected, the slave continues using that bus. A valid message is detected if the message parity checks with the rest of the message. This indicates that messages are being parsed correctly and that data is not corrupted. A message is detected by a slave even if it is not addressed to that particular slave. Thus, any activity on a bus that checksums properly will keep slaves listening to it.

The slave continues to listen to a bus until messages stop passing over that bus for some timeout period, or until that bus issues a "listen to the other bus" command to the slave.

When the system first powers up, the two master processors negotiate for mastership over the TCS system. After power on, the slaves are assumed to be in an unknown state, so we do not know which of the two buses the slaves are listening to. To synchronize the slaves into all listening to one bus, the primary master processor issues card ID requests while the backup master broadcasts several "listen to the other bus" commands. At this point the system expects all the slaves to be listening to the primary master, and both masters begin polling their respective buses for card IDs. The primary should get responses, and the backup should get none. If the backup master gets re-

sponses, there is a problem with the slave's connection to the primary or with the slave bus selection circuitry.

After the primary master polls all the card slots on its bus, it writes the number of slaves it could talk to into a register in each of the slaves. (Slave firmware should clear this register on power-up.) It then broadcasts a "listen to the other bus" command that switches the slaves over to the secondary bus. When the backup master processor sees that many slaves are responding to polling, it reads the primary's "number register". If there is a non-zero value in that register, the backup believes that the primary is alive and possibly well.

The backup master processor then polls its TCS bus for card ID information, and the number of slaves that respond overwrites the same register in the slaves. The backup master compares the number of slaves it can talk to with the number it finds in the slave for the primary master. If the primary's number is greater than or equal to the backup's, then the primary master wins mastership. Otherwise, the backup master wins.

6.6.11 Power-On with a Redundant Switch

In a machine configured with a backup switch, the TCS master has the additional power-on duty of checking the operation of the backup switch.

- Check switch operation
  - Hold the processors reset, then select switch "A"
  - Broadcast a switch POST into memory of each function board
  - Release processors from reset to execute switch POST
  - Poll for POST completion and status
  - Repeat above steps for backup switch "B"

6.6.12 Configuration with a Redundant Switch

The TCS handles the *switch configuration* of the Butterfly II machine; that is, which of the two switch systems the machine uses.

A Butterfly II machine configured for *switch redundancy* contains two complete switch systems. Each has its own circuit cards and clock generator. A configuration parameter designates one of the switches as the primary switch, and the other is a hot standby. The TCS selects which switch system is used via the SIGA and LCON TCS interfaces. Switch selection occurs at system startup or restart, not while the machine is running the operating system(s) or executing user programs.

6.6.13 Backup Switch Testing

In systems configured with redundant switch networks, the backup switch is repeatedly tested to assure that it will be fully available if required.

At this writing, it has not been determined how this testing is performed. It may be done simply by manipulating the backup SIGAs, LCONs, and SGAs. That may be able to test the continuity of the electrical connections, but it does not say a lot about how the switch will perform under stress. Refer to TCS software and operational documentation for details on this issue.

6.7 B2TCS Bus Protocol

THIS SECTION IS STILL IN PROGRESS.

Discuss TCS bus protocol here, at bit and message format level. Note this is message format, not electrical levels. Refer to "TCS software and operational document". Include, or otherwise refer to, the *TCS Bus Protocol Specification*.

This section must also cover the following topics:
— broadcast mode
— TCS slave address format
switch cards = "function boards" 8, 9 of the midplane?
clock cards = "function boards" A, B of ANY midplane?
clock buffer cards = C, D of a bay?
bay #31 = broadcast?
there are subtypes of broadcast?

Justin has TCS protocol document. Guy has additional text on TCS operation. Section 6.4 (TCS Operational Capabilities) "should reflect the real operational capabilities, bundled with the TCS slave protocol and the TCS master software document".

9

Butterfly II Switch Components

9.1 The Core of the Butterfly II Computer

This chapter discusses three components that together implement the Butterfly II switch. These components are:

- The Butterfly II Midplane (B2MP)
- The Butterfly II switch cards:
  - The Butterfly II Switch Requester card (B2SR)
  - The Butterfly II Switch Server card (B2SS)
- The Butterfly II clock card (B2CLK)

In addition to these four core components, the Butterfly II machine contains function boards, power supplies, cables, cabinetry, and peripheral devices such as VMEbus subsystems. These other components are discussed in other chapters.

9.2 The Butterfly II Midplane (B2MP)

9.2.1 Function and Motivation

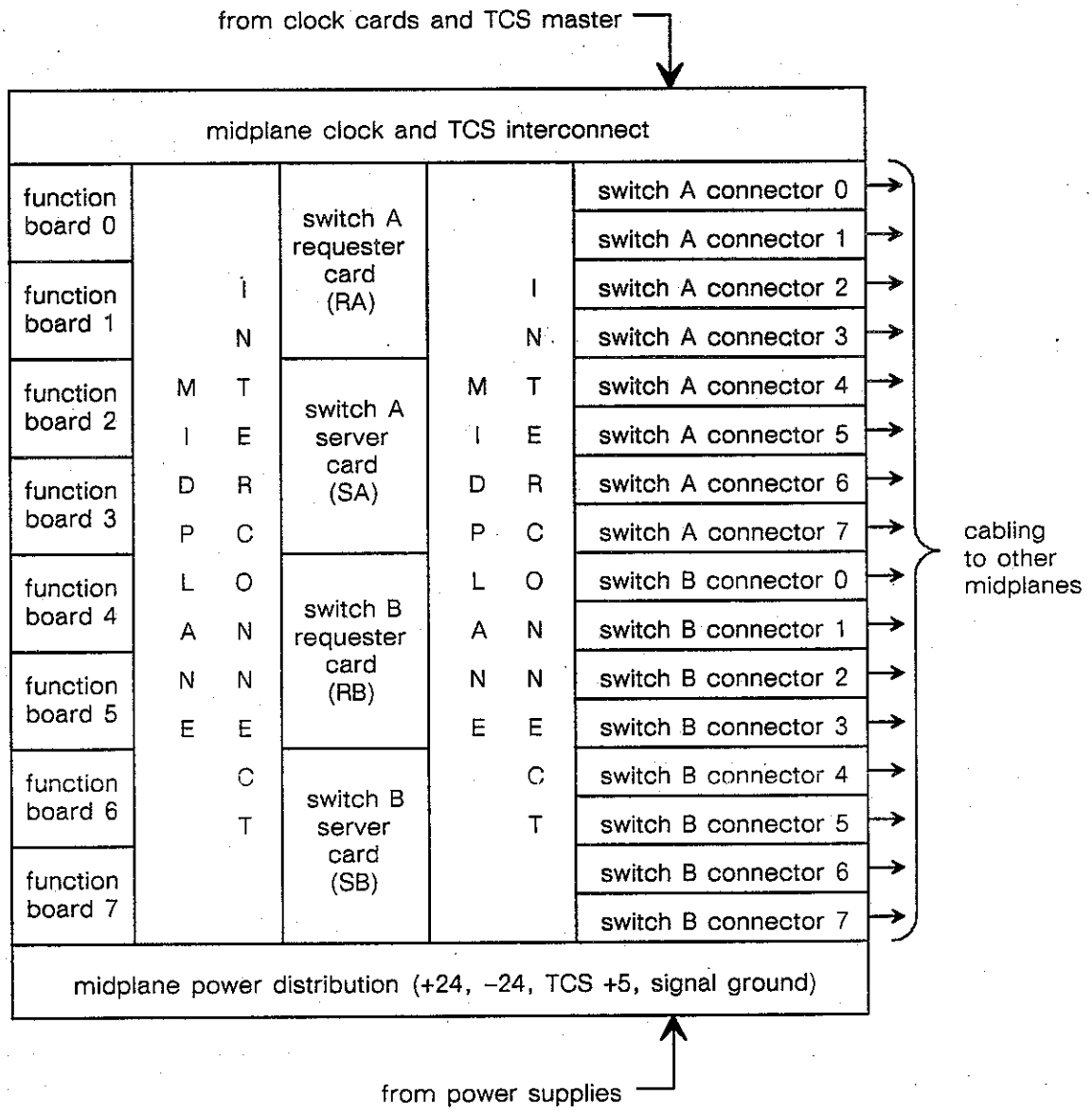
The Butterfly II Midplane provides the signal connections among the switch cards, function boards, and clock cables. It also supplies power to the switch cards and function boards. The number of interconnections required among these components is massive, and their short length with a controlled impedance is important in achieving the high performance of the Butterfly II design. The midplane corresponds to the backplane of more conventional machines. However, because function boards connect on one side and switch cards on the other, it is termed a *midplane*.

The midplane provides electrical connections and mechanical attachments for eight function boards plus enough switch hardware to support these function boards in configurations up to 63 processors and two switches. Thus, the midplane lies at the heart of an eight-slot module. The midplane interconnects the following components:

- Eight function boards, such as the TC/FPV function board
- Four switch cards: one requester—server pair, plus a second pair for the redundant switch
- Two switch clock cables: the first cable for the primary switch clock, and a second cable for a redundant switch
- Sixteen switch-to-switch data cables for connection to the rest of the machine: eight cables to switch cards throughout the machine, and a second set of eight for a redundant switch. In specific instances, some cables might be absent or replaced with TC/LOOP loop-back connectors, as described below.
- Power: +24 and -24 volts main power, +5 volts TCS power, and ground
- Three ground straps to ensure a low-impedance, system-wide ground: one to the midplane to the left, another to the midplane to the right, and a third to an I/O midplane above the midplane (if any, such as the TC/VMP). All three ground straps are on the function board side of the midplane.
- Midplane ID DIP switch, uniquely identifying a particular midplane in a machine.

Figure 9-1 shows the interconnections among these components. There are no active components on the midplane. Electrically, the midplane is an assembly of connectors and conductors plus a DIP switch.

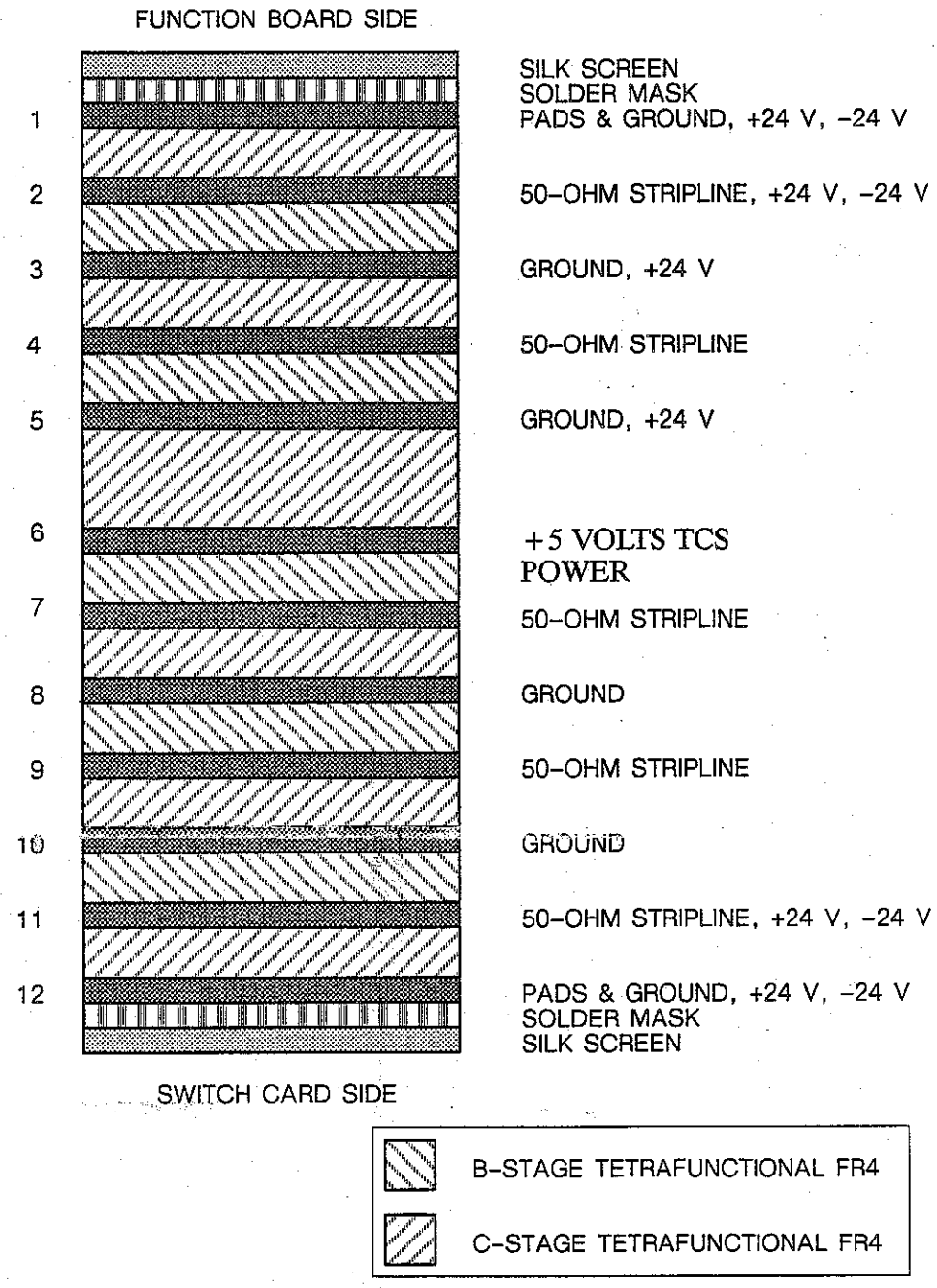
Figure 9–1 Midplane interconnections.



9.2.2 Physical Characteristics

The TC2000 midplane is a 9-inch by 19¾-inch printed circuit board with twelve conductive layers. Signals are carried principally on five layers with 50-ohm controlled impedance stripline. One layer carries the +5 TCS power. Four layers are mostly ground, and the outermost layer on each side is ground and connection pads. Many reference layers are split to distribute the +24 and -24 volt power. Figure 9-2 shows these layers.

Figure 9–2 Midplane layers.



NOTE

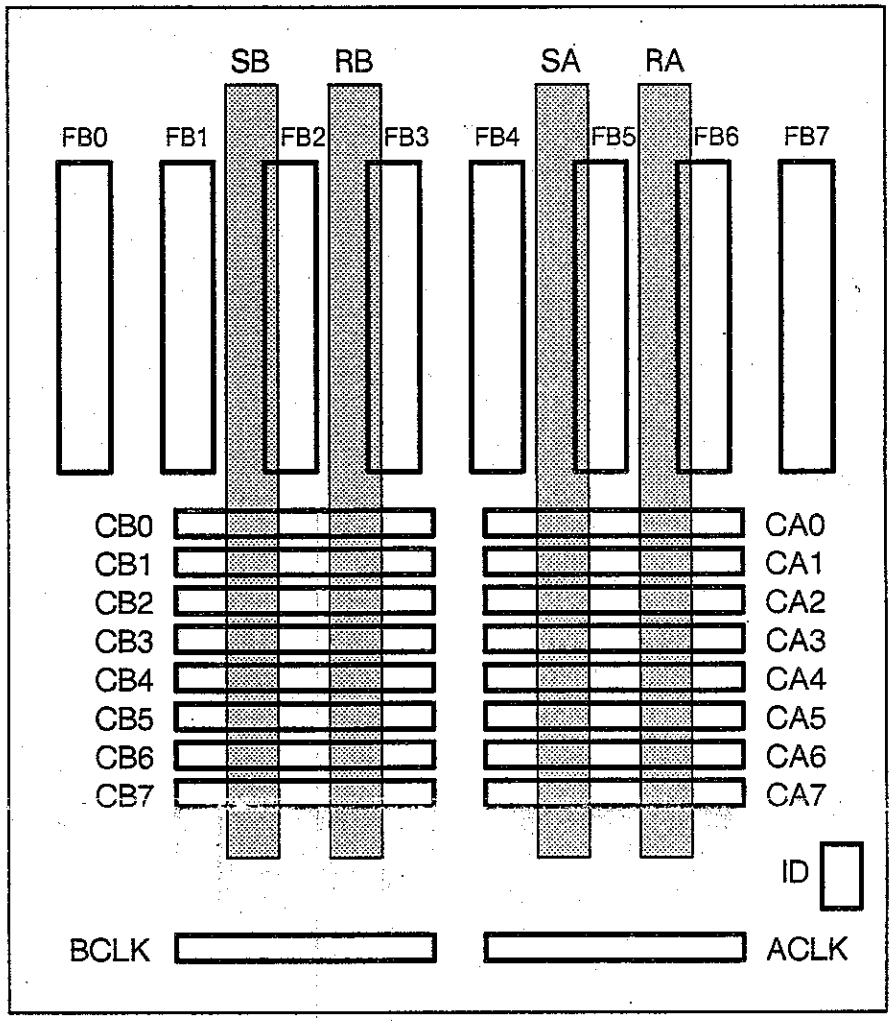
COPPER THICKNESS
Conductive layers (copper) are measured in the weight of copper per square foot of the layer. The midplane copper layers are 1–ounce copper (about 0.0014 inch).

NOTE

FR4
The FR4 dielectric material is made with a tetrafunctional modified epoxy. This decreases Z–axis thermal expansion, an important manufacturability feature for so thick a board. With tetrafunctional epoxy, there is a reduced risk of plated–through hole barrel cracking during wave soldering and repair.

Figure 9–3 shows the midplane connector layout, viewed from the function board side. The switch card connectors, shown shaded, are on the opposite side of the board, and are shown here to indicate their relative position through the midplane. The eight function board connectors are F0 through F7. The data cable connectors are CA0–CA7 for switch A, and CB0–CB7 for switch B. ACLK and BCLK are the clock cable connectors. SB, RB, SA and RA are the server and requester switch cards for switch B, and for switch A, respectively. "ID" is the midplane identity DIP switch. Boards attached to the midplane use the identity DIP switch setting to determine their location within the machine. The figure is not to scale.

Figure 9–3 Midplane connector layout.



view from FUNCTION BOARD side

The midplane layout is remarkable in meeting additional design goals. For example, there are no vias (layer changes on signal runs) on the board. This enhances manufacturability and reliability, and maintains the controlled impedance needed for clean signal propagation. Also, the total length of data paths is kept short to reduce propagation delay and skew, thus permitting faster operation.

There are no active components on the midplane. Electrically, the midplane is an assembly of conductors and a DIP switch. This switch sets the bay and midplane fields of the identification numbers supplied to the function boards and the switch server cards, as described later.

NOTE

In early domestic (United States) systems, the main power is 48 volts in magnitude, but signal ground may lie anywhere between the +48 and its return. Later machines will use +24 and -24 respectively.

9.2.3 Block Diagram

Figure 9–4 B2MP block diagram.

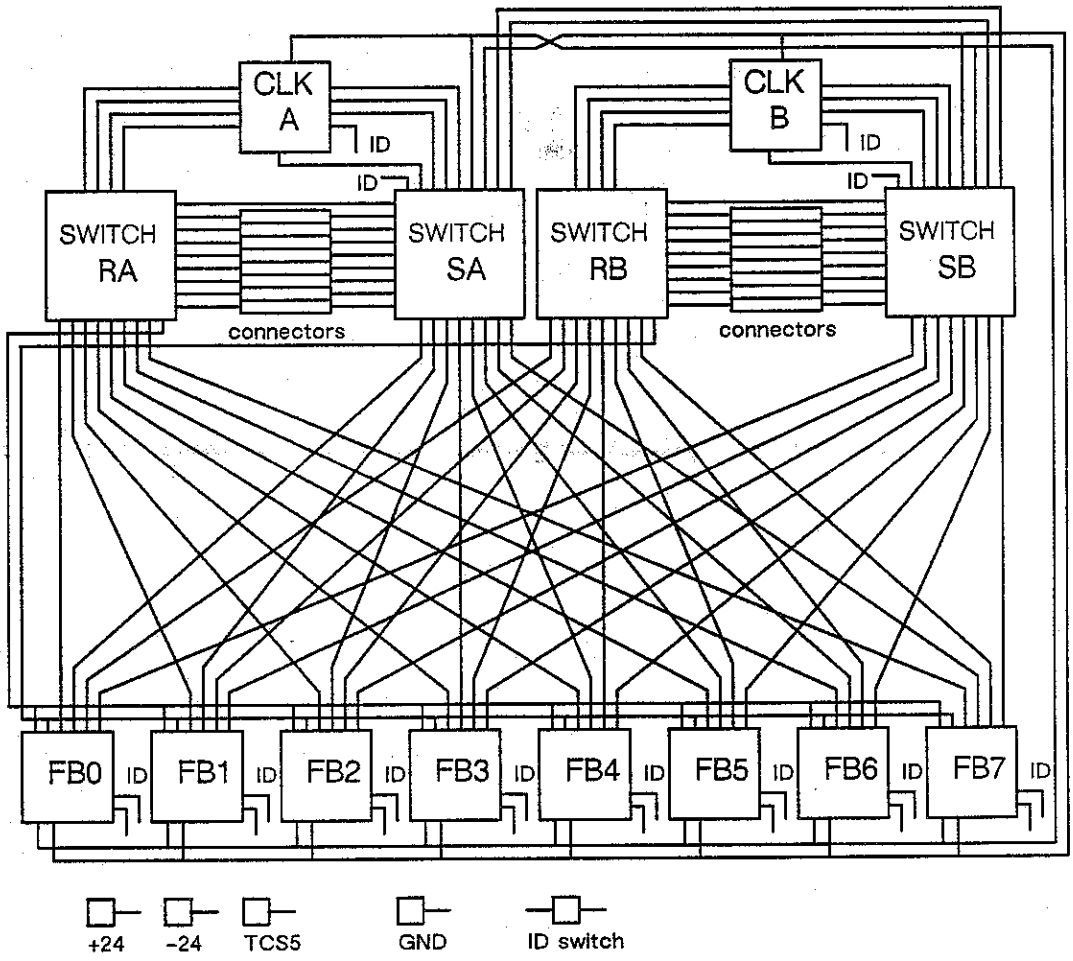


Figure 9–4 is a logical block diagram of the midplane. Each midplane contains full wiring for eight function boards and both an "A" and a "B" switch. Functionally similar parts of the wiring are discussed in the following sections.

All signals on the midplane use 100K ECL levels, with two exceptions:

- All communication between Test and Control System (TCS) masters and slaves uses CMOS levels

- Of the TCS-related signals between requester and server switch cards, some use ECL levels, some use CMOS levels, and some are analog signals, as noted for each such signal in section 9.2.8.

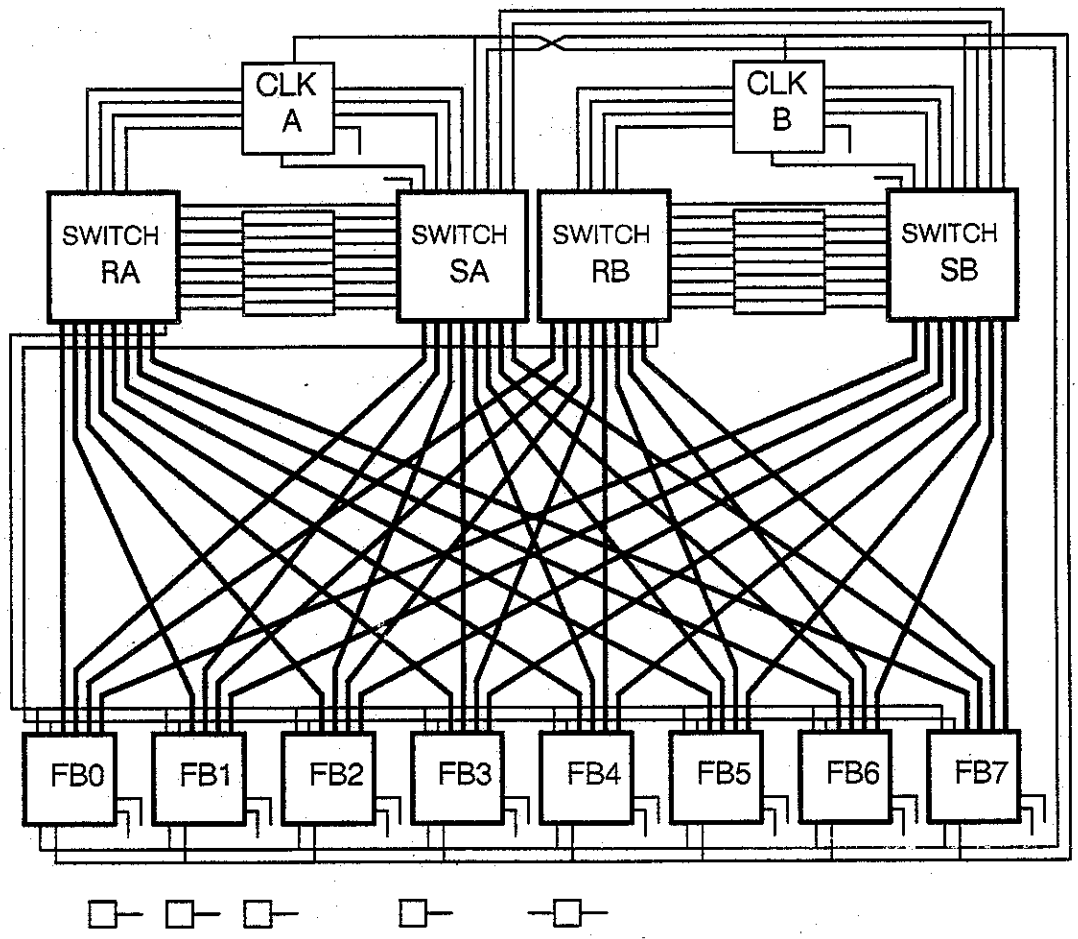
NOTE

DIFFERENTIAL PAIR SIGNAL NAMES
The companion signal name in a differential pair is generally formed by placing an asterisk after the signal name: CLK and CLK*. Some design software will not accept this, however, so the reader may occasionally see an L (for "low") before the signal name: CLK and LCLK. The two naming conventions are equivalent and interchangeable.

9.2.4 Wiring between switch cards and function boards

Each of the eight function board slots on a midplane is wired to each of the four switch card slots on the midplane. This wiring, except for certain TCS-related signals discussed later, is shown with heavy lines in Figure 9-5.

Figure 9–5 Function board — switch card wiring.



Each of these 32 paths, shown as a single line in Figure 9–5, is composed of sixteen signals, listed below. Except where noted, these signals are discussed in the chapter that describes the Butterfly II switch.

- D<7..0>, data, bidirectional
- FRM, frame, downstream (from requester to server)
- RVRS, reverse, upstream (from server to requester)
- CLK and CLK*, clock, a differential pair, from switch card to function board
- Two wires carry signals dependent on whether the connection involves a switch *requester* card or a switch *server* card:

  If the path is between a function board and a switch *requester* card:

  - MSEC65 and MSEC65*, 65-millisecond pulse, a differential pair, from switch *requester* card to function board

If the path is between a function board and a switch *server* card:

- TCSS_TO_MASTER, function board TCS slave data destined ultimately for the system TCS master via the switch *server* card, from function board to switch *server* card — discussed in the Test and Control System chapter
- NET_TIME, an external time pulse sensed by some function boards (on wires not on the midplane and not described here) and driven by those function boards for possible selection as the machine-wide net time signal, function board to switch *server* card

- SPARE1 and SPARE0, two spare wires

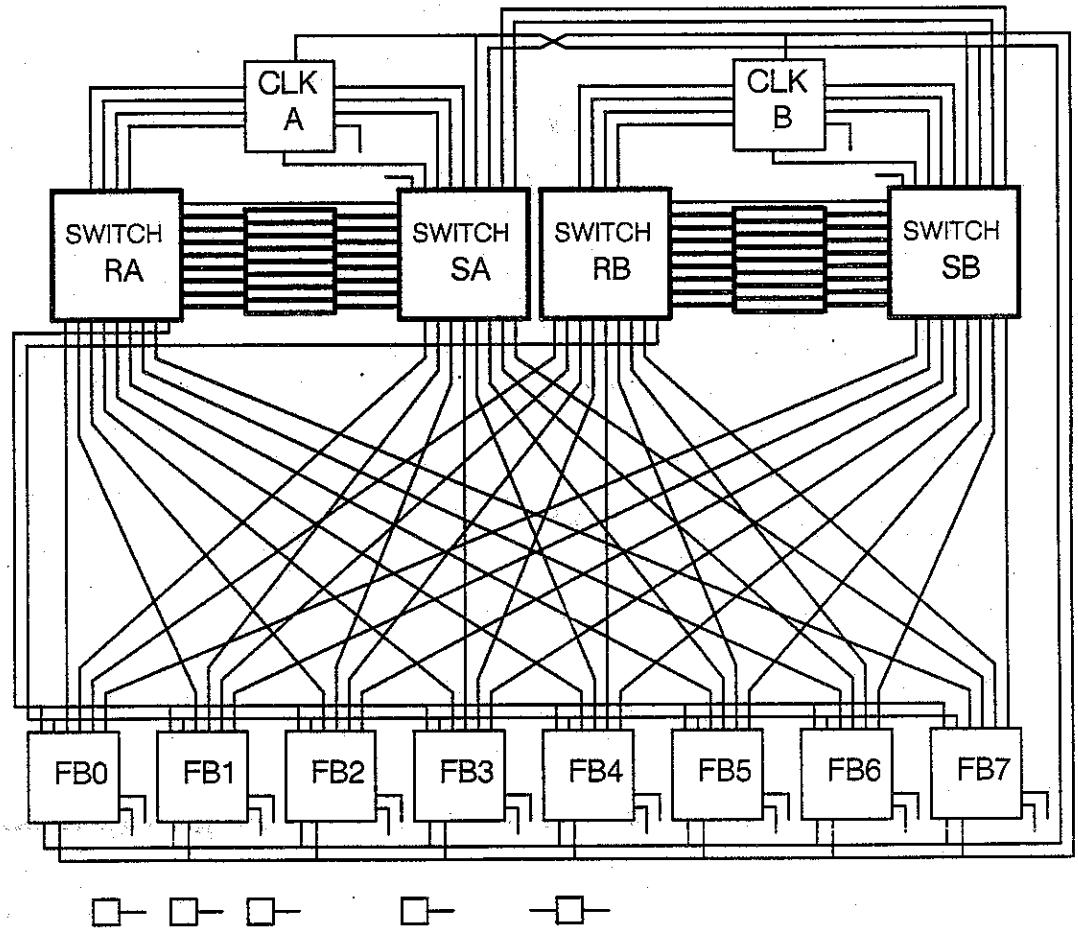
9.2.5 Wiring between switch cards and data cables

For communication with the rest of the machine, each switch card is wired through the midplane to connectors that accept cables. These cables connect to similar connectors on various midplanes. (For example, in a machine with a 2-column switch and exactly eight midplanes, one cable connects back to the same midplane, and the other seven cables connect to other midplanes.) Figure 9-6 shows this midplane wiring to these connectors with heavy lines.

NOTE

LOOP-BACK CONNECTOR
In any machine, at least one data cable from each midplane would loop back to that same midplane. A small printed circuit board, the TC/LOOP, implements the loop-back connection and is used in place of the bulkier cable. For example, an 8-midplane (64-slot) machine has eight TC/LOOPs, one on each midplane. For simplicity, every machine is shipped with eight TC/LOOPs, distributed appropriately on its midplanes. A 1-midplane machine has all eight TC/LOOPs on its midplane, and no data cables. Additional TC/LOOPs, beyond the first one installed on any given midplane, create alternate paths between the function boards on that midplane. If the machine is configured with a redundant switch, the TC/LOOP population in the primary switch is repeated in the redundant switch. The position of TC/LOOPs depends on the number of midplanes in the machine, and is described in the document *TC2000 Configuration Cabling*.

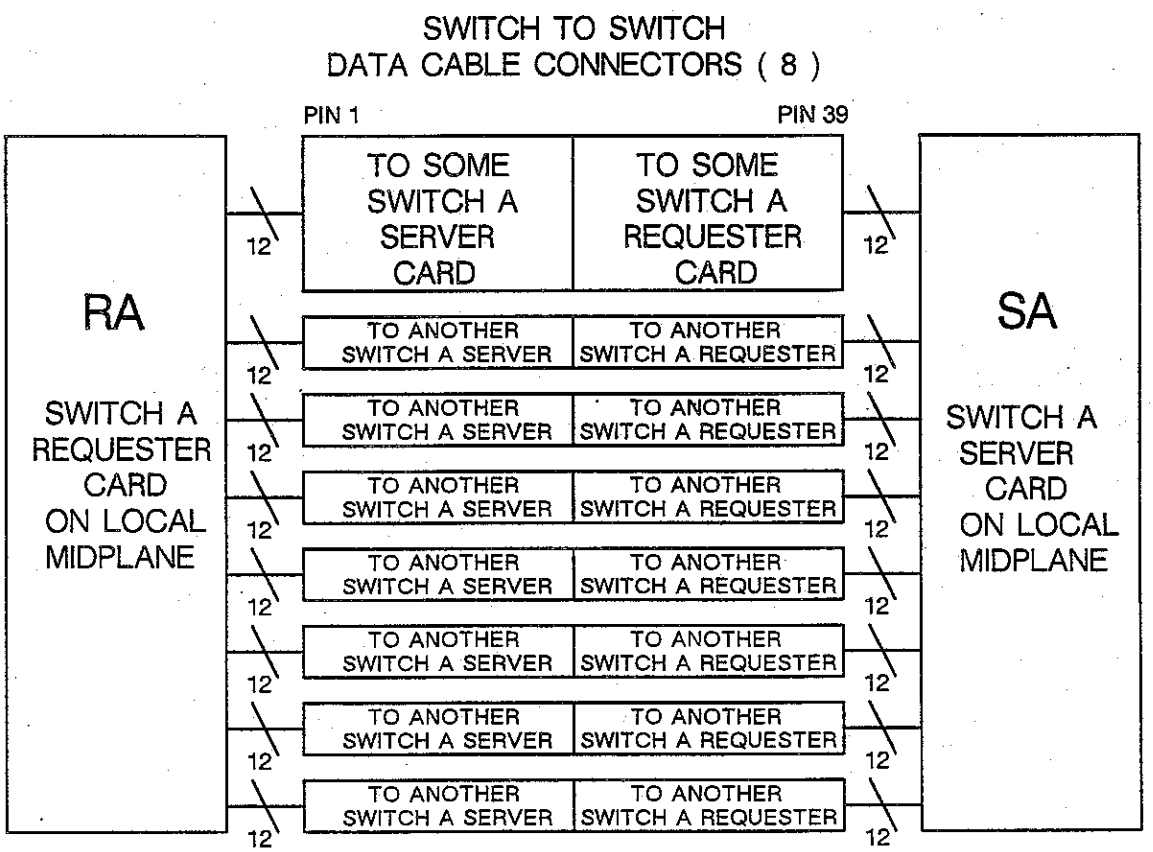
Figure 9–6 Switch card — data cable wiring.



A closer look at the data cable connectors for switch A is shown in Figure 9–7. No wires in this category connect a switch requester card and a switch server card just through the midplane; rather, each wire runs between one switch card and the connector. Logically, the connector can be thought of as two connectors, one for the local requester and one for the local server. Physically, it is implemented as a single connector. All of its wires go to the same place on another midplane, and are carried in one physical cable assembly.

The entire circuitry shown in Figure 9–7 is duplicated for switch B.

Figure 9–7 Data cable connector.



The twelve signals in each group from a switch card to a data cable connector are listed below.

- D<7..0>, data, bidirectional
- FRM, frame, from requester to server
- RVRS, reverse, from server to requester
- SPARE1 and SPARE0, two spare wires

Figure 9–8 summarizes the number of data cables in a Butterfly II system with one to eight midplanes. Each such machine also contains a set of eight B2LOOP loop-back cards. If the machine is configured with a second switch for redundancy, the number of data cables is twice that of a one-switch machine, and a second set of B2LOOPs is supplied. The data cables in a given machine are of various lengths, chosen to meet propagation time and packaging requirements.

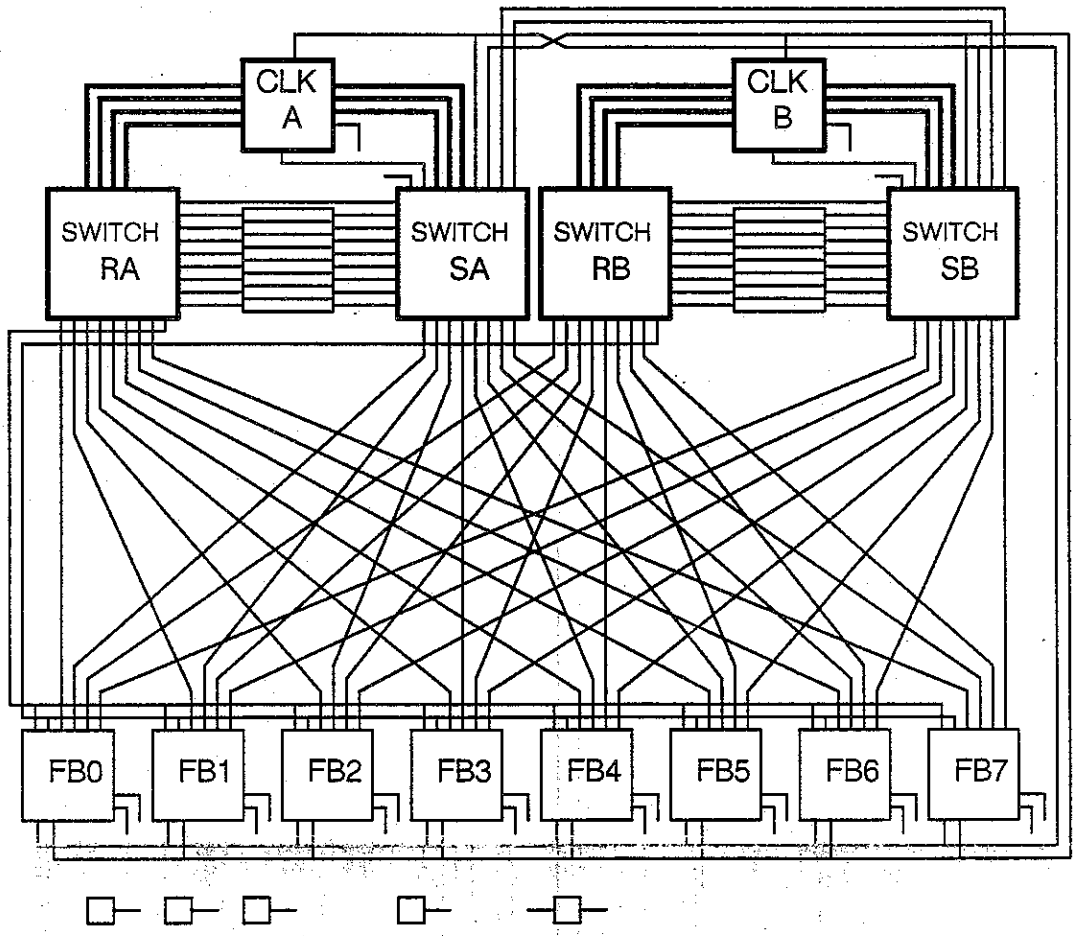
Figure 9–8 **Data cable count.**

number of midplanes	cables (of various lengths) one–switch system	two–switch system
1	NONE	NONE
2	4	8
3	6	12
4	12	24
5	10	20
6	15	30
7	21	42
8	28	56

9.2.6 Wiring between switch cards and clock cables

The primary purpose of midplane wires (actually circuit board traces) between clock cables and switch cards is to distribute the machine–wide clock signals. Several other signals are also carried in this group of wires, shown in heavy lines in Figure 9–9. Signals on these wires shown heavy are discussed immediately below. Other wires between clock cables and switch cards, not highlighted in Figure 9–9, relate to the Test and Control System and are discussed in a later section on TCS wiring.

Figure 9–9 Clock cable — switch card wiring.



As Figure 9–9 indicates, the midplane provides four signals to each requester card, and two signals to — and one from — each server card. Each of these signals is carried on a differential pair of wires. These signals are listed below.

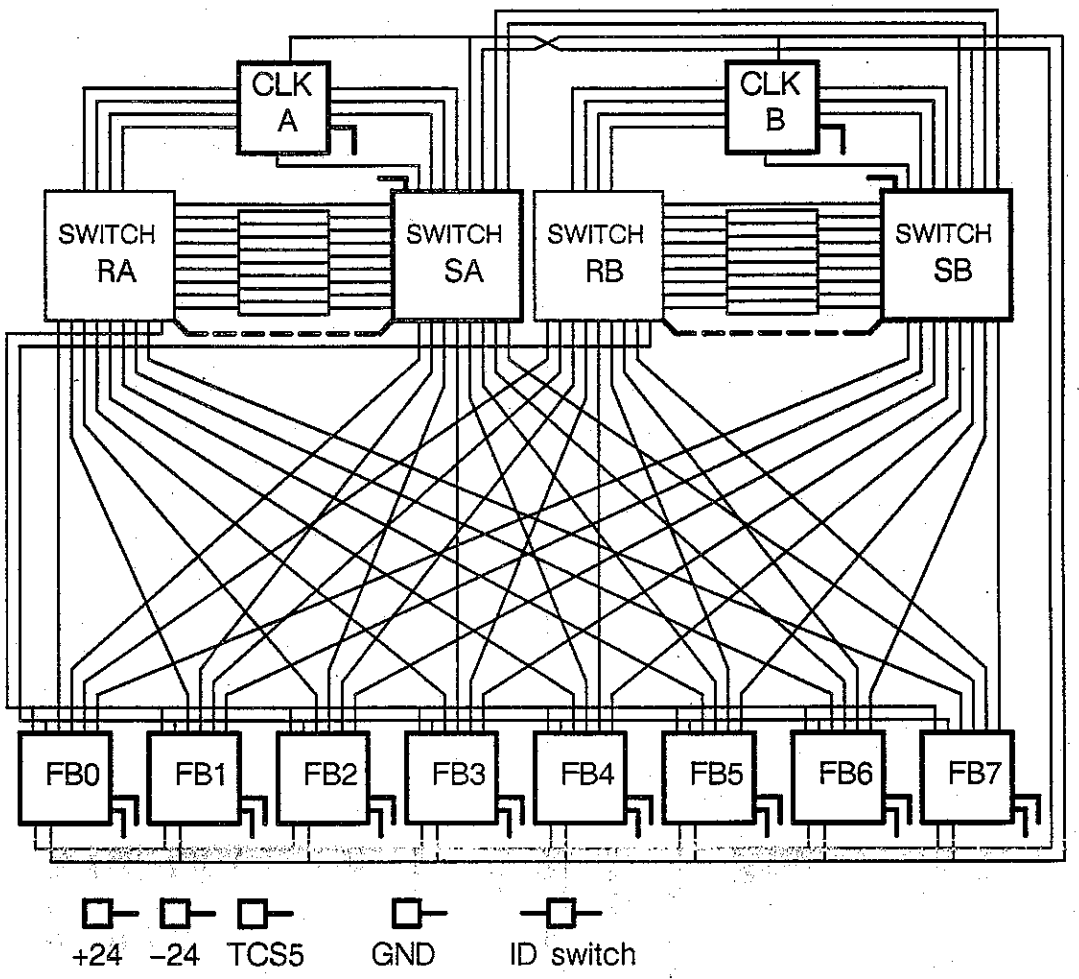
- Between a clock cable and a *switch requester* card:
  - R_CLK and R_CLK*, requester clock, a differential pair, from clock cable to requester card
  - R_HOLD and R_HOLD*, requester hold, a differential pair, from clock cable to requester card
  - MSEC65 and MSEC65*, 65–millisecond pulse, a differential pair, from clock cable to requester card
  - NET_TIME_SYS and NET_TIME_SYS*, the system-wide time pulse selected from those sensed by some function boards, a differential pair, from clock cable to requester card
- Between a clock cable and a *switch server* card:

- S_CLK and S_CLK*, server clock, a differential pair, from clock cable to server card
- S_HOLD and S_HOLD*, server hold, a differential pair, from clock cable to server card
- NET_TIME_SW and NET_TIME_SW*, a time pulse selected by the switch server card from among the eight external time pulse signals sensed by the function boards served by the server card, a differential pair, from server card to clock cable

9.2.7 Power and ID wiring

Figure 9–10 shows the midplane wiring for power and identity (ID) signals with heavy lines. The three boxes in the lower left symbolize main power supply, main power return and TCS power. These three are distributed to the eight function board connectors, the four switch card connectors, the two clock card power connectors, and nowhere else. The box to the right of these symbolizes ground. Ground is distributed to all connectors, including cable connectors and the ID circuitry. The switch requester card carries an on-board VEE supply, but no VTT supply. Requester card VTT is obtained from the paired server card's VTT, via a small bus bar on the midplane, shown in heavy dashed lines in Figure 9–10.

Figure 9–10 Midplane power and ID wiring.



The ID circuitry informs each function board, switch server card and clock card of its logical position in the Butterfly II machine, by grounding appropriate wires on each of these components. The ID is used by the TCS slave in each function board and in switch cards, to determine whether a message from the TCS master are addressing that slave. The ID is also used on B2VME function boards as a processor node number, which the software can read and use to uniquely identify the function board within the machine.

NOTE

ID AND SWITCH ADDRESS

The ID of a particular function board is determined partly by the DIP switch setting on the midplane holding the board, and partly by which slot of that midplane holds the board. The *switch address* of a function board, however, is the address that must appear in a switch message in order for the switch system to route the message to that function board. The switch address is determined partly by the data cable installation, and partly by which slot of its midplane holds the function board. Consequently, *the ID of a function board may be different from its switch address*. The low three bits of the two will always be the same, but only a consistent scheme of switch cabling and ID DIP switch setting will produce an ID and switch address identical in all nine bits.

The ID is a 9-bit number composed of three fields, shown in Figure 9-11. The bay field indicates a particular bay (cabinet), the midplane field indicates a midplane within that bay, and the slot field indicates one of the eight function board slots on that midplane. Each function board receives the full 9-bit ID, but the switch server cards receive only the bay and midplane fields (and the A/B* bit described below). The bay and midplane fields are set by a DIP switch on the midplane, shown in Figure 9-12.

Figure 9-11 Fields of the ID.

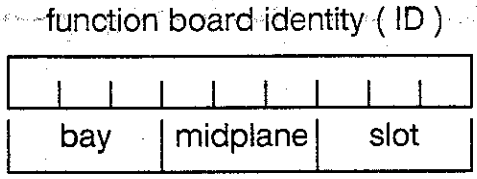
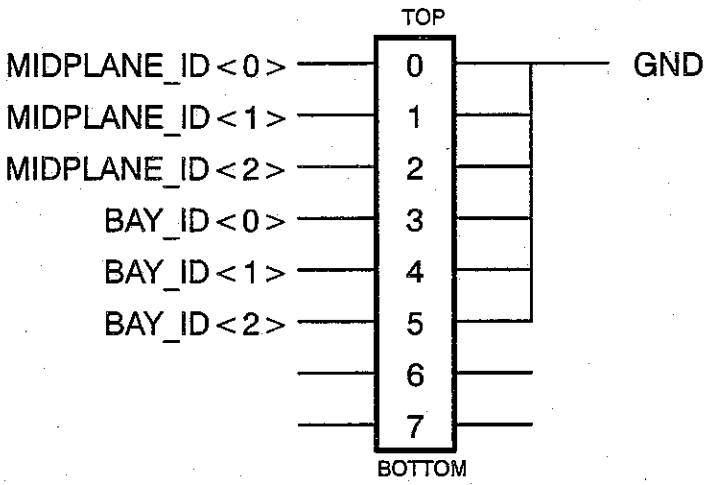


Figure 9–12 ID DIP switch.



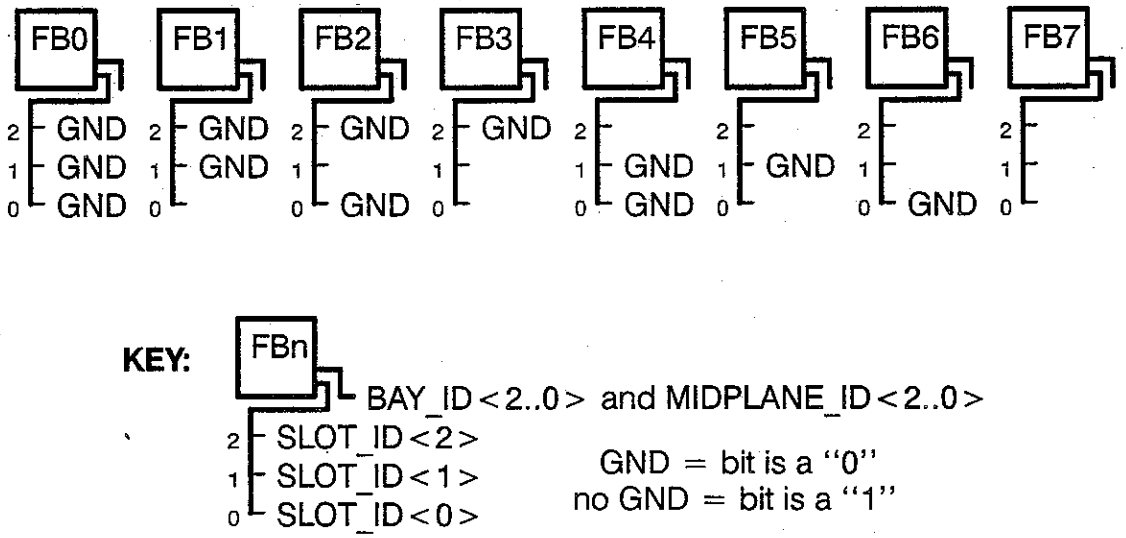
switch open = off = ID bit is "1"

switch closed = on = ID bit is "0"

See switch housing for open/closed (off/on) labeling.

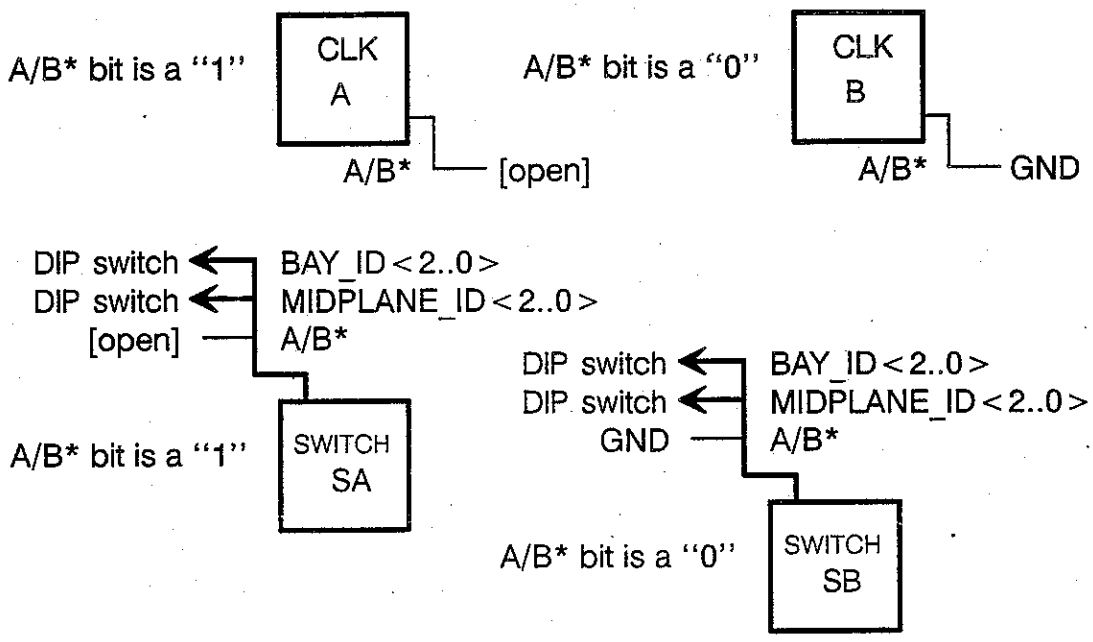
For all ID bits, the function boards and switch cards using the ID bit have pullups on these signals to create a "1" bit if the ID switch is open. If the midplane grounds the corresponding wire, the bit is a "0". The hard wired slot ID is shown in Figure 9–13.

Figure 9–13 Midplane wiring for function board slot ID.



Each switch server card and each clock card receives a signal that indicates whether the card is part of switch A or switch B. This signal is called A/B* and is shown in Figure 9–14. The switch server card TCS uses this bit, together with the bay and midplane fields of the ID, to determine whether data from a TCS master is addressing this slave. The clock card TCS slave processor, in contrast, is told by the TCS master which messages to accept. The clock card's A/B* bit is used merely as a consistency check, performed by the TCS master.

Figure 9–14 Midplane wiring for switch server and clock card identity.



Two- or Three-Column Switch

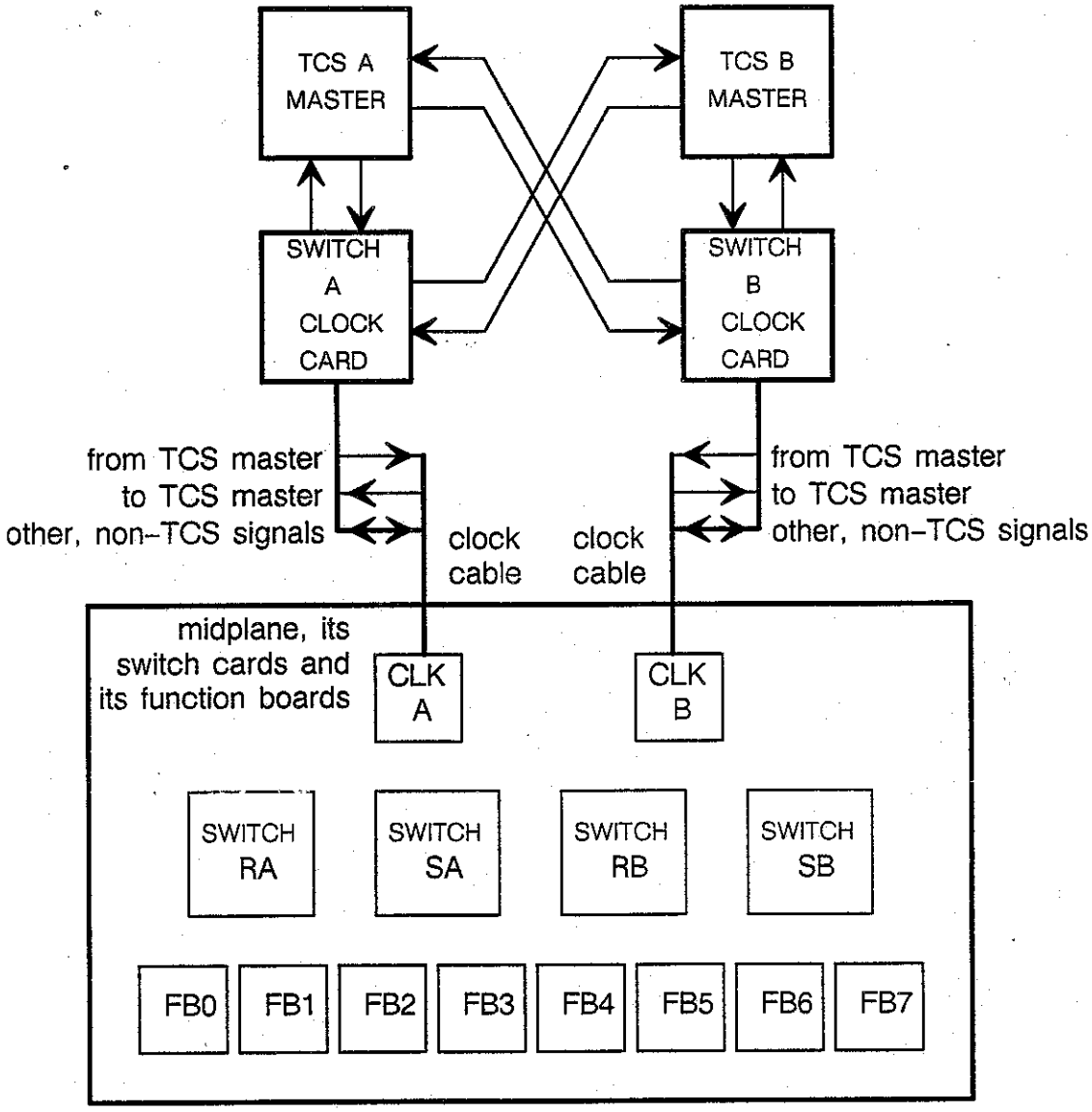
In the original midplane, a signal connecting to each switch server card tells these cards whether the system is configured for a two-column or a three-column switch. This signal, COLUMN_2_OR_3, is grounded to indicate a 2-column switch, and open for a 3-column switch. In the second version of the midplane (in which the clock card power connectors and more ground strapping points were also added), a similar signal, TYPE<0>, to both the switch requester cards and the switch server cards, is grounded for a two-column configuration and open for a three-column configuration.

9.2.8 TCS wiring

The midplane provides access to ten TCS slaves — one in each of the eight function boards, and one in each of the two switch server cards.

The midplane, its switch cards, and its function boards are connected to the rest of the Test and Control System (TCS) via the clock cables, as shown in Figure 9–15. Each clock cable contains two TCS–related wires. One distributes data from the TCS master, and the other sends collected data to the TCS master. Each of these travels through the clock card, where it is buffered, gated and/or processed. If the machine is configured with a second switch, the second clock cable provides two TCS–related wires, independent of those in the primary clock cable. The switch A clock cable nominally serves the TCS A master, and the switch B clock cable the TCS B master. However, as Figure 9–15 shows, the TCS systems are cross–coupled so either TCS master can use either clock cable. The intent is that only one master is active at any given time.

Figure 9–15 Midplane as a TCS subsystem.



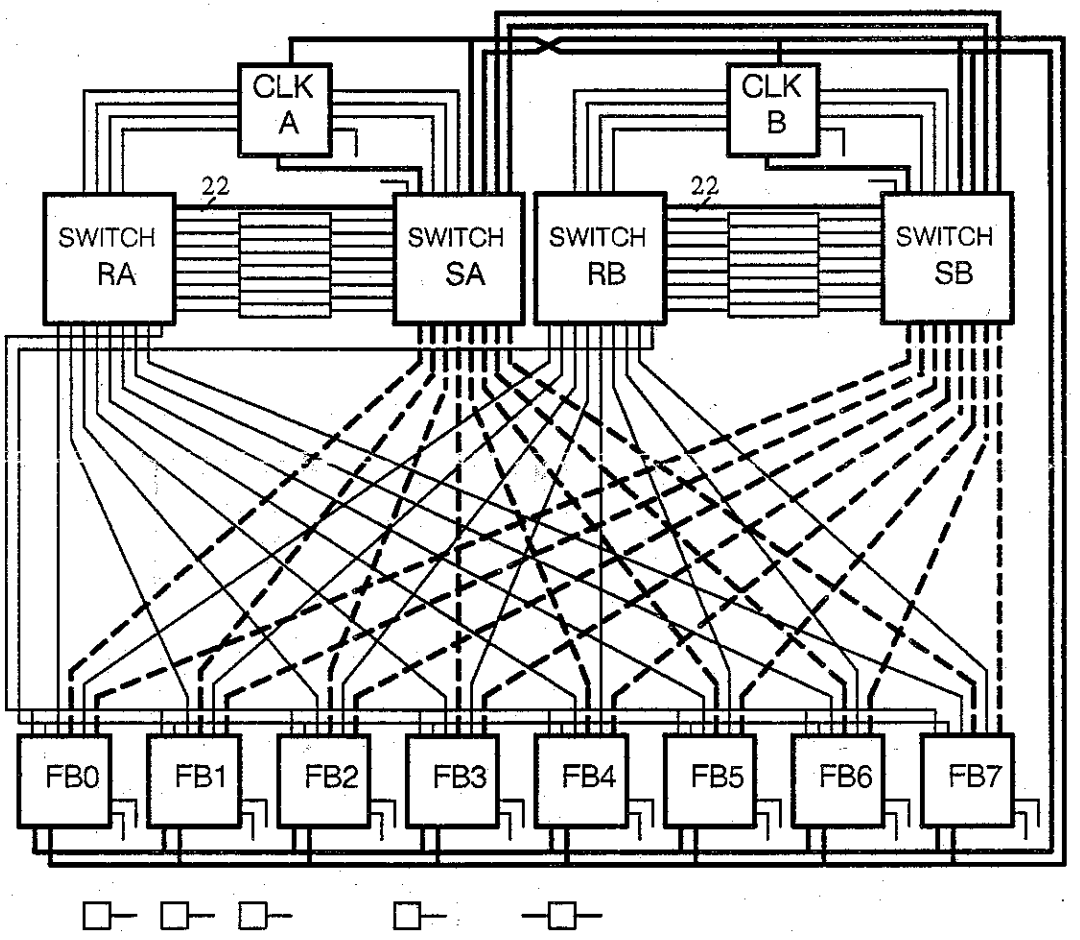
The TCS slave on a clock card is connected directly to the TCS master by a cable, involving no midplanes. The fan–in of TCS communications from the clock card's TCS slave and up to eight midplanes, and the fan–out of communications to the clock card's TCS slave and the midplanes, are performed on the clock card and thus do not involve any midplane wiring.

The midplane wiring supports connection of each TCS master (in a system with two masters) to each clock card (in a system with two switches). The midplane wiring also provides connection between the TCS slaves in the TC/SS switch cards (in a system with two switches). At each switch card TCS slave, data destined for the TCS master from the *other* switch card is combined with

data from the card's *own* TCS slave and data from the TCS slaves on the eight function boards. This is done to enhance reliability in systems with two switches.

Figure 9–16 shows in heavy lines the midplane wiring supporting the Test and Control System (TCS). Dashed lines indicate that only one of the 16 wires in each dashed line is TCS–related. The heavy lines between each requester and server card pair of a switch actually contain 22 wires. All other heavy lines in Figure 9–16 indicate only a single wire.

Figure 9–16 Midplane TCS wiring.



TCS–related signals on the midplane fall into the three categories listed below and described in the following paragraphs.

- Distribution of test and control information from TCS masters to TCS slaves
- Collection, for the TCS master, of the TCS slaves' responses to the master's commands

- TCS–related information exchanged between a switch requester card and its paired (same switch and same midplane) switch server card

Distribution from TCS master to TCS slave (TCSS)

Data from a TCS master is distributed by a wire that connects a clock cable connector to each function board and each switch server card. The signal name is as follows:

- On the clock connector, MASTER_TO_TCSS
- On the midplane, AMASTER_TO_TCSS (from CLKA clock cable), or BMASTER_TO_TCSS (from CLKB clock cable)
- On a function board connector, AMASTER_TO_TCSS or BMASTER_TO_TCSS
- On a switch server card connector, MASTER_TO_TCSS or OMASTER_TO_TCSS ("O" stands for "other", data from the other TCS master — TCS A is "other" to the SB card, and TCS B is "other" to the SA card)

NOTE

SIGNAL LEVEL
The MASTER_TO_TCSS signal idles low on the midplane. On each B2SS clock card and each function board, the signal is inverted to idle high just before going into the TCSS slave processor chip (68HC11).

Collection from TCS slaves to TCS master

Data to a TCS master is collected by fan–in on switch server cards and associated wiring on the midplane. Of the 16 wires from each function board to each switch server card, one is a signal from the function board toward a TCS master. The signal name is as follows:

- On the function board connector, the midplane, and the switch server card connector (eight signals, one from each function board slot), TCSS_TO_MASTER

Each switch server card OR's together the TCSS_TO_MASTER data from the eight function boards along with its own data to its TCS master *and data from the other switch server card's TCS slave*. The data from each function board and from the other switch server card can be individually disabled. This provides for operation despite cross–mode failure. The midplane supports this with wiring between the two switch server cards, as follows:

- On the switch server card connector, the signal TCSS_TO_OMASTER sends data from this card's TCS slave to the *other* TCS master

- On the midplane, the signals ATCSS_TO_BMASTER and BTCSS_TO_AMASTER carry data as indicated by their name
- On the switch server card connector, the signal OTCSS_TO_MASTER receives data from the other card's TCS slave

After OR'ing together data from the various TCS slaves, each switch server card sends the resulting data to its TCS master on the following wire:

- On the switch server card connector, the midplane, and the clock cable connector, TCSS_TO_MASTER

TCS-related signals between requester and server cards

The switch requester card has no TCS slave. The test and control functions required are performed by the TCS slave in the paired server card — that is, by the server card in the same switch and on the same midplane. The midplane supports this with 22 wires connecting the requester and server card connectors, as listed below. The levels used on each signal are also noted: CMOS, ECL or analog.

- Serial Peripheral Interface (SPI) signals, used by the TCS slave on the server card to communicate with SGA chips on the requester card
  - SPI_CLK, clocks data, CMOS from server card to requester card
  - SPI_MOSI, Master Out Slave In, SPI data, CMOS from server card to requester card ("Master" and "Slave" of SPI, not TCS)
  - R_SPI_MISO, Master In Slave Out, SPI data, ECL from requester card to server card ("Master" and "Slave" of SPI, not TCS)
- Random number generator signals, used to control the pseudo-random number generator used to resolve head-on collision of messages contending for a given switch output port
  - RAND_RESET, shift zeros into the maximum sequence generator register at the switch clock rate while this signal is asserted, ECL from server card to requester card
  - RAND_PRESET, shift ones (creating the dead state, for testing) into the register while this signal is asserted, ECL from server card to requester card
- SGA_RESET, asserts both *mreset* and *cntlr* pins on SGA chips to completely reset the SGAs and enable their output ports, ECL from server card to requester card
- EXEC_R00, EXEC_R01, EXEC_R10 and EXEC_R11, execute signals wired to the *cntrle* pin of SGA chips and used by the TCS slave to make each requester card SGA chip individually perform a previously-loaded command, ECL from server card to requester card

- PWR_ENABLE, turns on requester card ECL power, TTL from server card to requester card
- MARGIN_DISABLE, inhibits voltage margin testing of requester card ECL supplies, TTL from server card to requester card
- Requester card voltage and temperature monitoring (corresponding server card voltage and temperature monitoring occurs on that card)
  - R_VEE, requester card VEE voltage monitor, analog from requester card to server card
  - R_VTT, requester card VTT voltage monitor, analog from requester card to server card
  - R_SGA_VTT, monitors VTT voltage on each of the requester card's four SGA chips as selected by the command register, analog from requester card to server card
  - R_TEMP, monitors requester card air temperature near the SGAs or overall board air temperature near the board exit airflow as selected by the command register, analog from requester card to server card
- ID_SELECT, chip select for requester card EEPROM to allow TCS slave to read and write it, CMOS from server card to requester card
- Command shift register control signals, used by TCS slave in conjunction with the SPI signals, for functions such as voltage margining the requester card, selecting what requester card voltage and temperature to monitor, and turning the requester card amber LED "flag" on and off
  - NTTL_RESET, while asserted (low) clears the command shift register and the command latch register, CMOS from server card to requester card
  - R_CMD_EN, gates the contents of the command shift register into the command latch register, CMOS from server card to requester card
- SPARE0, SPARE1, SPARE2 — spare wires routed on the midplane

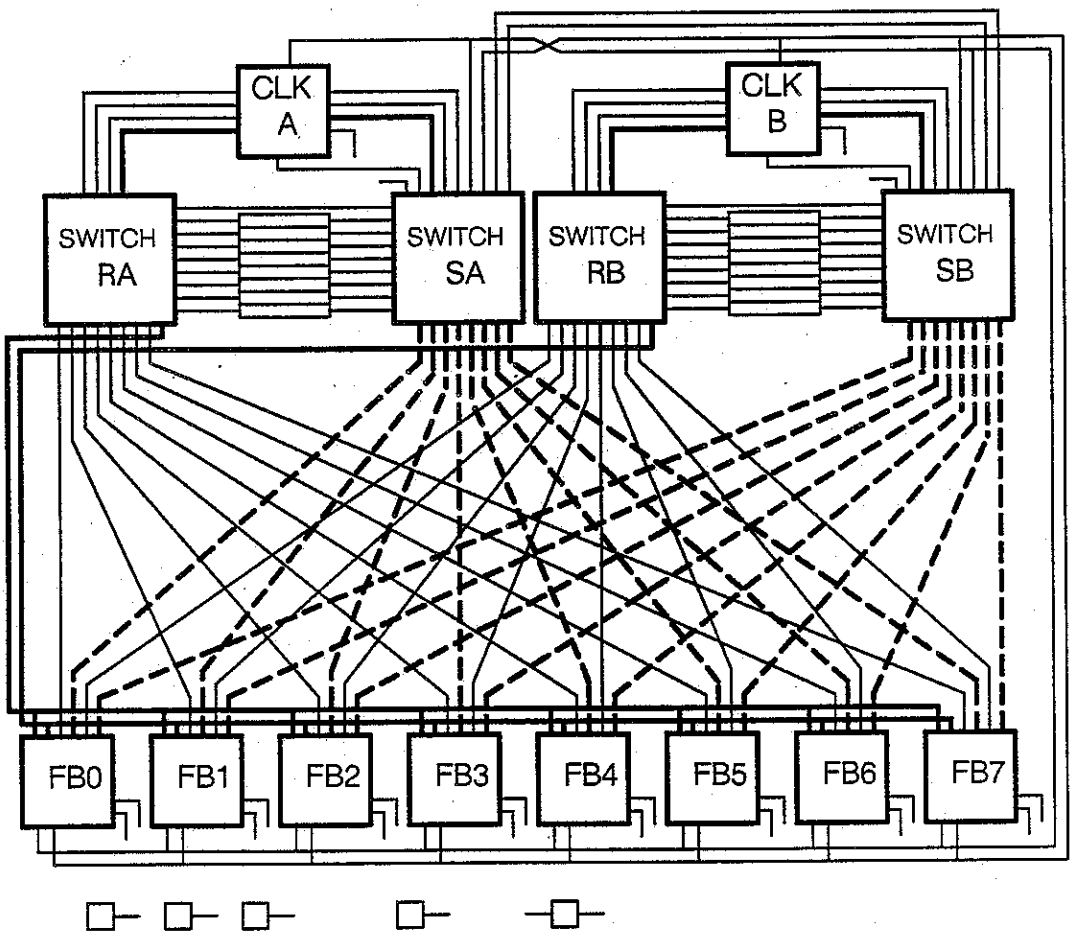
NOTE

HISTORICAL NOTE
In an early Butterfly II design, the switch requester card and switch server card were on a single board. The resulting large board and connector size led to implementing the two on separate boards. In the process of separating the curcuitry, it was decided that it was unnecessary to duplicate the TCS slave on both of the boards. Therefore, several wires are supplied to permit the one TCS slave (on the server board) to properly monitor and control the other board. These are the only midplane signal wires between the two cards. The only other midplane wire between them is a VTT power bus.

9.2.9 Net time wiring

Heavy lines in Figure 9–17 show the midplane wiring supporting the net time facility. A solid line indicates a path totally devoted to net time, and a dashed line indicates a group of wires of which only one is for net time.

Figure 9–17 Net time wiring.



Each function board may sense an external (to the Butterfly II computer) clock called *net time*, and report it to the switch server cards over midplane wiring with the signal name:

- On the function board connector, the midplane, and the switch server card connector, NET_TIME

Each switch server card selects one of the net time signals from its function boards and forwards that signal into the clock subsystem over the midplane wiring with the signal names:

- On the switch server card connector, the midplane, and the clock cable connector, NET_TIME_SW and NET_TIME_SW*, a differential pair

The clock subsystem selects one of the net time signals from its midplanes and redistributes it, over the clock cables, to switch requester cards with the signal names:

- On the clock cable connector, the midplane, and the requester card connector, NET_TIME_SYS and NET_TIME_SYS*, a differential pair

Each requester card distributes net time to its eight function boards over a differential pair bus with the signal names:

- On the requester card connector and the midplane, NET_TIME_SYS_FB and NET_TIME_SYS_FB*, a differential pair
- On the function board connector, both of the following:
  - A_NET_TIME_SYS_FB and A_NET_TIME_SYS_FB*, a differential pair from switch A
  - B_NET_TIME_SYS_FB and B_NET_TIME_SYS_FB*, a differential pair from switch B

The collection and distribution of net time is also discussed in the switch card section below, and illustrated there in Figure 9–30.

9.2.10 Midplane connector pinouts

This section illustrates the pin assignments in each type of connector on the midplane. (See Figure 9–12 for the ID DIP switch socket.) Figure 9–18 summarizes the connector types. Notes explaining and extending the information in the diagrams appear in a list after the diagrams.

Figure 9-18 Midplane connector summary.

connector type	modules per connector	symbolic name*	quantity on midplane
clock cable	1	CLKag	2
switch A data cable	1	CAgn	8
switch B data cable	1	CBgn	8
function board	4	Fng1 — Fng4	8
switch requester card (R-card)	10	Rag1 — Rag10	2
switch server card (S-card)	10	Sag1 — Sag10	2
clock card power cable	1?	(to be specified)	2

* KEY: (see notes)
a = A or B switch
g = J or P connector gender
n = 0 – 7 switch port or function board number

Clock cable connector

CLKag:

```
     |<-----to S------>|    |<-----to R------>|
pin 39                                      pin 1
     x x x C D x x G x o x I J x x M N a x x
     A B x x x E F x H x o x x K L x x x P Q
pin 40                                      pin 2
```

A S_CLK*
B S_CLK

C S_HOLD*
D S_HOLD

E NET_TIME_SW*
F NET_TIME_SW

G TCSS_TO_MASTER
H MASTER_TO_TCSS

o CLK_SPARE1
o CLK_SPARE0

I NET_TIME_SYS*
J NET_TIME_SYS

K MSEC65*
L MSEC65

M R_HOLD*
N R_HOLD

a A/B*

P R_CLK*
Q R_CLK

Switch A data cable connector

```
CAgn:
        |<---to local S--->|<---to local R--->|
   pin 39                                      pin 1
        o x x x x x x x x o o x x x x x x x x o
        A I G E D F C H J B b j h c f d e g i a
   pin 40                                      pin 2
```

S_TO_R<11..0>	R_TO_S<11..0>
A D<7>	a D<7>
B D<6>	b D<6>
C D<5>	c D<5>
D D<4>	d D<4>
E D<3>	e D<3>
F D<2>	f D<2>
G D<1>	g D<1>
H D<0>	h D<0>
I FRM	i FRM
J RVRS	j RVRS
o SPARE1	o SPARE1
o SPARE0	o SPARE0

Switch B data cable connector

```
CBgn:
        |<---to local S--->|<---to local R--->|
   pin 39                                      pin 1
        o x x x x x x x x o o x x x x x x x x o
        A I G C F E D H J B b j h d e f c g i a
   pin 40                                      pin 2
```

S_TO_R<11..0>	R_TO_S<11..0>
A D<7>	a D<7>
B D<6>	b D<6>
C D<5>	c D<5>
D D<4>	d D<4>
E D<3>	e D<3>
F D<2>	f D<2>
G D<1>	g D<1>
H D<0>	h D<0>
I FRM	i FRM
J RVRS	j RVRS
o SPARE1	o SPARE1
o SPARE0	o SPARE0

Function board connector

```
Fng1:    1    -----     main power (+24 volts)
         2    -----     main power return (-24 volts)

Fng2:    1  # a x c x   a = A_NET_TIME_SYS_FB, b = A_NET_TIME_SYS_FB*
         2  x b x d x   c = B_NET_TIME_SYS_FB, d = B_NET_TIME_SYS_FB*

         3  x x o x o   SB_TO_FB<15..0>
         4  x A x B x                        R-card              S-card
         5  # x C x D
         6  # E x F x                        o SPARE1            o SPARE1
         7  x x x x G                        o SPARE0            o SPARE0
         8  x H x J x                        A D<7>              A D<7>
         9  x I x x K                        B D<6>              B D<6>
        10  # x x L x                        C D<5>              C D<5>
        11  # x M x N                        D D<4>              D D<4>
                                             E D<3>              E D<3>
        12  x o x o x   RB_TO_FB<15..0>      F MSEC65            F TCSS_TO_MASTER
        13  x x A x B                        G LMSEC65           G NET_TIME
        14  x C x D x                        H CLK               H CLK
        15  # x E x F                        I LCLK              I LCLK
Fng3:    1  # x x G x                        J D<2>              J D<2>
         2  x H x x J                        K D<1>              K D<1>
         3  x I x K x                        L D<0>              L D<0>
         4  x x x x L                        M FRM               M FRM
         5  # M x N x                        N RVRS              N RVRS

         6  # x o x o   SA_TO_FB<15..0>
         7  x A x B x
         8  x x C x D
         9  x E x F x
        10  # x x x G
        11  # H x J x
        12  x I x x K
        13  x x x L x
        14  x x M x N

        15  # o x o x   RA_TO_FB<15..0>
Fng4:    1  # x A x B
         2  x C x D x                                              a = RACK_ID<3>
         3  x x E x F                                              b = RACK_ID<2>
         4  x x x G x                                              c = RACK_ID<1>
         5  # H x x J                                              d = RACK_ID<0>
         6  # I x K x                                              e = PANEL_ID<1>
         7  x x x j L                                              f = PANEL_ID<0>
         8  x M g N h                                              g = SLOT_ID<2>
         9  x e f c d   A = AMASTER_TO_TCSS                        h = SLOT_ID<1>
        10  # a A b B   B = BMASTER_TO_TCSS                        j = SLOT_ID<0>
```

```
11  # o x o x
12  x x o x o   o = unrouted spares (10)
13  x o t o t
14  x t o t o   t = TCS5 power (+5 volts) (6)
15  # o t o t

======================= END OF FUNCTION BOARD CONNECTOR ==================
```

Switch requester card (R-card) connector

```
Rag1:     1    -----     main power (+24 volts)
          2    -----     main power return (-24 volts)

Rag2:     1  # o x a b   a = NET_TIME_SYS_FB, b = NET_TIME_SYS_FB*
          2  x x o x x   o = unrouted spares (2)

          3  x o x o x   R_TO_FB4<15..0>    o SPARE1
          4  x x A x B                      o SPARE0
          5  # C x D x                      A D<7>
          6  # x E x F                      B D<6>
          7  x x x G x                      C D<5>
          8  x H x x J                      D D<4>
          9  x I x K x                      E D<3>
         10  # x x x L                      F MSEC65
         11  # M x N x                      G LMSEC65
                                            H CLK
         12  x x o x o   R_TO_FB2<15..0>    I LCLK
         13  x A x B x                      J D<2>
         14  x x C x D                      K D<1>
         15  # E x F x                      L D<0>
         16  # x x x G                      M FRM
         17  x H x J x                      N RVRS
         18  x I x x K
         19  x x x L x
         20  # x M x N

Rag3:     1  # o x o x   R_TO_FB3<15..0>
          2  x x A x B
          3  x C x D x
          4  x x E x F
          5  # x x G x
          6  # H x x J
          7  x I x K x
          8  x x x x L
          9  x M x N x

         10  # x o x o   R_TO_FB5<15..0>
         11  # A x B x
         12  x x C x D
         13  x E x F x
         14  x x x x G
         15  # H x J x
         16  # I x x K
         17  x x x L x
         18  x x M x N
```

```
          19  x o x o x   R_TO_FB6<15..0>
          20  # x A x B
Rag4:      1  # C x D x
           2  x x E x F
           3  x x x G x
           4  x H x x J
           5  # I x K x
           6  # x x x L
           7  x M x N x

           8  x x o x o   R_TO_FB0<15..0>
           9  x A x B x
          10  # x C x D
          11  # E x F x
          12  x x x x G
          13  x H x J x
          14  x I x x K
          15  # x x L x
Rag5:      1  # x M x N

           2  x o x o x   R_TO_FB7<15..0>
           3  x x A x B
           4  x C x D x
           5  # x E x F
           6  # x x G x
           7  x H x x J
           8  x I x K x
           9  x x x x L
          10  # M x N x

          11  # x o x o   R_TO_FB1<15..0>
          12  x A x B x
          13  x x C x D
          14  x E x F x
          15  # x x x G
Rag6:      1  # H x J x
           2  x I x x K
           3  x x x L x
           4  x x M x N

           5  # A x B x   S_TO_R_TCS<21..0>  A RAND_PRESET      L EXEC_R01
           6  # x C x D                      B RAND_RESET       M EXEC_R10
           7  x E x F x                      C SPI_CLK          N EXEC_R11
           8  x x G x H                      D SPI_MOSI         O R_TEMP
           9  x I x J x                      E R_SPI_MISO       P R_SGA_VTT
          10  # x K x L                      F SGA_RESET        Q R_VTT
          11  # M x N x                      G NTTL_RESET       R R_VEE
          12  x x O x P                      H R_CMD_EN         S ID_SELECT
          13  x Q t R x                      I MARGIN_DISABLE   T SPARE2
          14  x t S t T   t = TCS5 (5)       J POWER_ENABLE     U SPARE1
          15  # U t V t                      K EXEC_R00         V SPARE0

          [ blank space here on R-card, 5 more rows of pins on S-card ]

Rag7:      1    -----     VEE
           2    -----     VTT
```

```
Rag8:     1  # A x B x   R_TO_S0<11..0>     A D<7>
          2  x x C x D                      B D<6>
          3  x E x F x                      C D<5>
          4  x x G x H                      D D<4>
          5  # I x J x                      E D<3>
          6  # x o x o                      F D<2>
                                            G D<1>
          7  x A x B x   R_TO_S1<11..0>     H D<0>
          8  x x C x D                      I FRM
          9  x E x F x                      J RVRS
         10  # x G x H                      o SPARE1
         11  # I x J x                      o SPARE0
         12  x x o x o

         13  x A x B x   R_TO_S2<11..0>
         14  x x C x D
         15  # E x F x
Rag9:     1  # x G x H
          2  x I x J x
          3  x x o x o

          4  x A x B x   R_TO_S3<11..0>
          5  # x C x D
          6  # E x F x
          7  x x G x H
          8  x I x J x
          9  x x o x o

         10  # A x B x   R_TO_S4<11..0>
         11  # x C x D
         12  x E x F x
         13  x x G x H
         14  x I x J x
         15  # x o x o

         16  # A x B x   R_TO_S5<11..0>
         17  x x C x D
         18  x E x F x
         19  x x G x H
         20  # I x J x
Rag10:    1  # x o x o

          2  x A x B x   R_TO_S6<11..0>
          3  x x C x D
          4  x E x F x
          5  # x G x H
          6  # I x J x
          7  x x o x o

          8  x A x B x   R_TO_S7<11..0>
          9  x x C x D
         10  # E x F x
         11  # x G x H
         12  x I x J x
         13  x x o x o
```

```
          14  x o x o x   o = unrouted spares (5)
          15  # x o x o
          16  # G x o x
          17  x H x x x   A = R_CLK, B = R_CLK*
          18  x x x F E   C = R_HOLD, D = R_HOLD*
          19  x A x x x   E = MSEC65, F = MSEC65*
          20  # B x D C   G = NET_TIME_SYS, H = NET_TIME_SYS*

============  END OF SWITCH REQUESTER CARD (R-card) CONNECTOR  ============
```

Switch server card (S-card) connector

```
Sag1:     1    -----     main power (+24 volts)
          2    -----     main power return (-24 volts)

Sag2:     1  # o x o x   o = unrouted spares (4)
          2  x x o x o

          3  x o x o x   S_TO_FB1<15..0>    o SPARE1
          4  x x A x B                      o SPARE0
          5  # C x D x                      A D<7>
          6  # x E x F                      B D<6>
          7  x x x G x                      C D<5>
          8  x H x x J                      D D<4>
          9  x I x K x                      E D<3>
         10  # x x x L                      F TCSS_TO_MASTER
         11  # M x N x                      G NET_TIME
                                            H CLK
         12  x x o x o    S_TO_FB2<15..0>    I LCLK
         13  x A x B x                      J D<2>
         14  x x C x D                      K D<1>
         15  # E x F x                      L D<0>
         16  # x x x G                      M FRM
         17  x H x J x                      N RVRS
         18  x I x x K
         19  x x x L x
         20  # x M x N

Sag3:     1  # o x o x   S_TO_FB4<15..0>
          2  x x A x B
          3  x C x D x
          4  x x E x F
          5  # x x G x
          6  # H x x J
          7  x I x K x
          8  x x x x L
          9  x M x N x

         10  # x o x o   S_TO_FB5<15..0>
         11  # A x B x
         12  x x C x D
         13  x E x F x
         14  x x x x G
         15  # H x J x
         16  # I x x K
         17  x x x L x
         18  x x M x N
```

```
          19  x o x o x   S_TO_FB7<15..0>
          20  # x A x B
Sag4:      1  # C x D x
           2  x x E x F
           3  x x x G x
           4  x H x x J
           5  # I x K x
           6  # x x x L
           7  x M x N x

           8  x x o x o   S_TO_FB0<15..0>
           9  x A x B x
          10  # x C x D
          11  # E x F x
          12  x x x x G
          13  x H x J x
          14  x I x x K
          15  # x x L x
Sag5:      1  # x M x N

           2  x o x o x   S_TO_FB6<15..0>
           3  x x A x B
           4  x C x D x
           5  # x E x F
           6  # x x G x
           7  x H x x J
           8  x I x K x
           9  x x x x L
          10  # M x N x

          11  # x o x o   S_TO_FB3<15..0>
          12  x A x B x
          13  x x C x D
          14  x E x F x
          15  # x x x G
Sag6:      1  # H x J x
           2  x I x x K
           3  x x x L x
           4  x x M x N

           5  # A x B x   S_TO_R_TCS<21..0>  A RAND_PRESET      L EXEC_R01
           6  # x C x D                      B RAND_RESET       M EXEC_R10
           7  x E x F x                      C SPI_CLK          N EXEC_R11
           8  x x G x H                      D SPI_MOSI         O R_TEMP
           9  x I x J x                      E R_SPI_MISO       P R_SGA_VTT
          10  # x K x L                      F SGA_RESET        Q R_VTT
          11  # M x N x                      G NTTL_RESET       R R_VEE
          12  x x O x P                      H R_CMD_EN         S ID_SELECT
          13  x Q x R x                      I MARGIN_DISABLE   T SPARE2
          14  x x S x T                      J POWER_ENABLE     U SPARE1
          15  # U x V x                      K EXEC_R00         V SPARE0
```

```
          16  # x o t o   a = BAY_ID<2>       o = unrouted spares (3)
          17  x W t @ x   b = BAY_ID<1>       t = TCS5 (5)
          18  x t Y t Z   c = BAY_ID<0>          @ = COLUMN_2_OR_3
          19  x e g d t   d = MIDPLANE_ID<2>     W = TCSS_TO_OMASTER
          20  # f a b c   e = MIDPLANE_ID<1>     Y = OTCSS_TO_MASTER
                          f = MIDPLANE_ID<0>     Z = OMASTER_TO_TCSS
                          g = A/B* (a-f from DIP switch, g from connector)

Sag7:      1    -----     VEE
           2    -----     VTT

Sag8:      1  # B x A x   S_TO_R0<11..0>      A D<7>
           2  x x D x C                       B D<6>
           3  x F x E x                       C D<5>
           4  x x H x G                       D D<4>
           5  # J x I x                       E D<3>
           6  # x o x o                       F D<2>
                                              G D<1>
           7  x B x A x   S_TO_R1<11..0>      H D<0>
           8  x x D x C                       I FRM
           9  x F x E x                       J RVRS
          10  # x H x G                       o SPARE1
          11  # J x I x                       o SPARE0
          12  x x o x o

          13  x B x A x   S_TO_R2<11..0>
          14  x x D x C
          15  # F x E x
Sag9:      1  # x H x G
           2  x J x I x
           3  x x o x o

           4  x B x A x   S_TO_R3<11..0>
           5  # x D x C
           6  # F x E x
           7  x x H x G
           8  x J x I x
           9  x x o x o

          10  # B x A x   S_TO_R4<11..0>
          11  # x D x C
          12  x F x E x
          13  x x H x G
          14  x J x I x
          15  # x o x o

          16  # B x A x   S_TO_R5<11..0>
          17  x x D x C
          18  x F x E x
          19  x x H x G
          20  # J x I x
Sag10:     1  # x o x o
```

```
 2  x B x A x   S_TO_R6<11..0>
 3  x x D x C
 4  x F x E x
 5  # x H x G
 6  # J x I x
 7  x x o x o

 8  x B x A x   S_TO_R7<11..0>
 9  x x D x C
10  # F x E x
11  # x H x G
12  x J x I x
13  x x o x o

14  x o x o x   o = unrouted spares (5)
15  # x o x o
16  # H x o x   A = S_CLK, B = S_CLK*
17  x x G x x   C = S_HOLD, D = S_HOLD*
18  x x x F E   E = NET_TIME_SW, F = NET_TIME_SW*
19  x A x x x   G = MASTER_TO_TCSS
20  # B x D C   H = TCSS_TO_MASTER

==============  END OF SWITCH SERVER CARD (S-card) CONNECTOR  ==============
```

Midplane connector notes

1. All connectors in this section are shown looking toward the midplane, viewed from whichever side the connector being described is mounted on.

2. The following letters are used as variables in connector names:

   a A or B — specifies which switch connects here, switch A or switch B

   g J or P — specifies connector gender, jack (J) or plug (P)
   All connectors on the midplane are J. All connectors that mate with the midplane are P (this includes daughter boards (function boards and switch cards), clock cables, and data cables). See also male/female in note 5.

   n 0 through 7 — specifies which function board or which switch port connects here

   FOR EXAMPLE:
   CLKag is the generic name for a clock cable connector.
   CLKAJ is the midplane connector for the switch A clock cable.
   CLKAP is the connector on the cable that plugs into CLKAJ.

3. The following letters are used for pin names:

   A to **Z** the signal name is given in a table nearby, or if the pin is in a repeated group the names are given where that kind of group first appears in this connector

o a spare pin
If midplane wiring connects the pin to another spare pin, the pin is given a name such as SPARE1.
If no midplane wiring connects to the pin, it is noted as an *unrouted spare* in the connector diagram.

t TCS power, +5 volts from the TCS5 plane of the midplane

x ground — For switch cards and function boards, see also the ground pin note below.

ground on daughter boards (switch cards and function boards), but pin position is vacant on midplane connectors (see note below)

—— blade contact used for main power, VEE and VTT

4. KEYING

DAUGHTER BOARD KEYING — The pin format and keying of connectors for function boards, switch R-cards and switch S-cards is designed so that none of these components can be plugged in upside down or into a connector not intended for that type of component. However, components *of a given type* are interchangeable. For example, a switch requester card (R-card) can plug into either R-card connector, but not into an S-card connector or a function board connector.

DATA CABLE and CLOCK CABLE KEYING — Data cables use the same kind of connector as clock cables. It is possible to plug a data cable into a clock connector, or a clock cable into a data cable connector. Further, one side of the midplane connector is removed, permitting use of the B2LOOP loop-back card in data cable connectors. However, this permits plugging a data cable or a clock cable in upside down. To avoid the miscabling of data or clock cables, the following procedures should be followed.

- CLOCK CABLES carry a RED stripe or marking.
  DATA CABLES carry a BLUE stripe or marking.
  Observe this color coding when cabling the machine. (Note: cables in some early, in-house machines do not follow this standard.)
- Data cables and clock cables have a key on the plug at each end of the cable. When the cable is properly installed, this key fits into a slot in the jack on the midplane. Be sure this key is on the slotted side of the jack when cabling the machine.
- The B2LOOP loop-back card cannot be plugged in upside down, because its printed circuit board acts as a key.

5. The projecting metal signal pins are positioned on the midplane connectors, not on the daughter boards or cables. This reduces the risk of bent pins, because the boards and cables are moved and handled, but the midplane is relatively protected by the surrounding cabinetry. Also, if the flexible contact springs in receptacles wear out, this placement makes repair easier because the receptacles are on the easily removable boards and cables rather than on the midplane.

6. DAUGHTER BOARD CONNECTORS

   The daughter boards (switch R-cards, switch S-cards and function boards) have a more complex connector design than the cable connectors. They are made of several adjacent connector modules. Each module is given a name. Both signal and blade connector module types are used. Blade contact modules are used for power. For example, F0J3 is function board (F) number zero (0) midplane connector (J) module three (3). F0J2 is next to it on one side, and F0J4 on the other side.

   Each daughter board signal connector module has either 15 or 20 rows of pin positions, with six pin positions per row. On the midplane connectors, only positions 1 through 5 have pins installed; on the daughter boards, all six positions have contacts installed. The pin *positions* are numbered consecutively, left to right, not the actual pins. The midplane pin and daughter board pin that connect have the same number. However, every sixth pin number is absent from these midplane connectors. This is shown in Figure 9-19. On daughter boards, all pins in that sixth column are connected to ground. (Ground is also carried on all column one pins in both the midplane and the daughter boards, and on many pins scattered throughout the connectors.)

Figure 9-19 Midplane and daughter board pin numbering.

```
 1  2  3  4  5
 7  8  9 10 11
13 14 15 16 17
19 20 21 22 23
25 26 27 28 29
31 32 33 34 35
37 38 39 40 41
43 44 45 46 47
49 50 51 52 53
55 56 57 58 59
61 62 63...
...
```

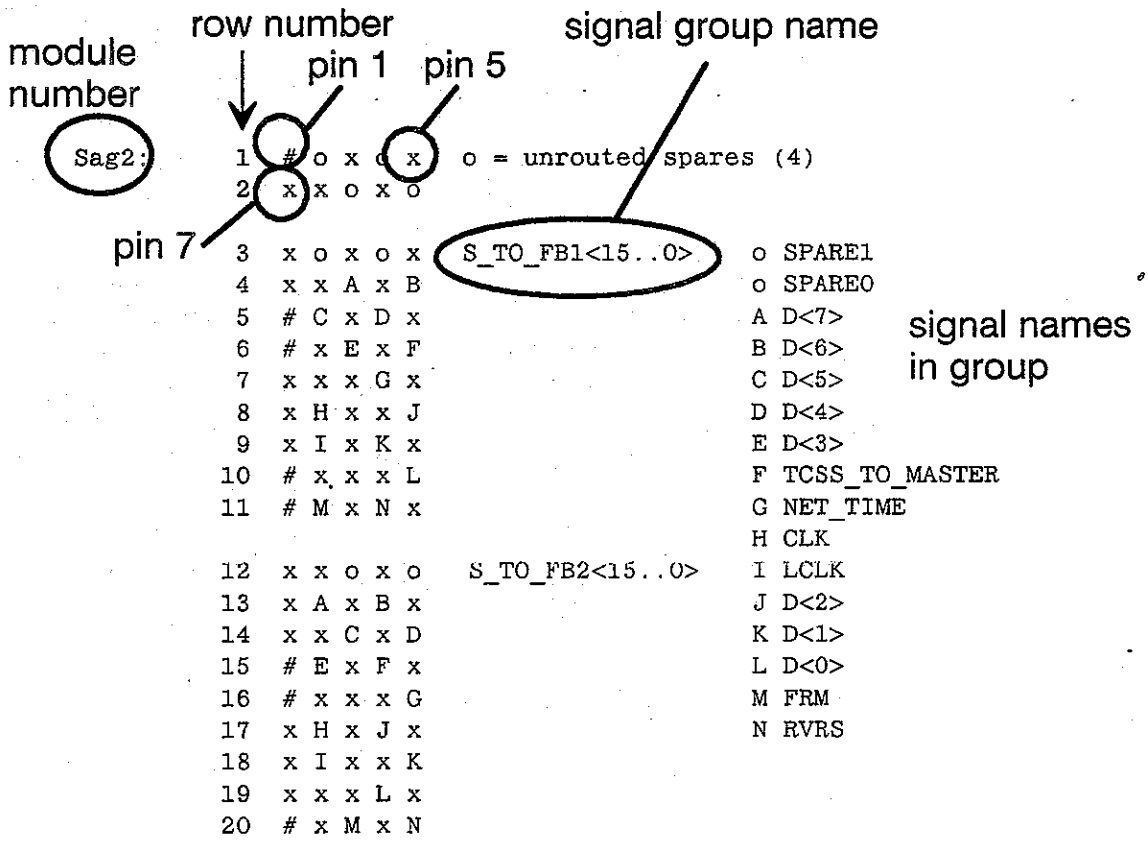
midplane (J)

```
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36
37 38 39 40 41 42
43 44 45 46 47 48
49 50 51 52 53 54
55 56 57 58 59 60
61 62 63...
...
```

daughter board (P)

Figure 9–20 shows how the midplane's daughter board connectors are diagrammed. At the left is the name of each module within the connector, here Sag2. The second column is the row number within the module. For example, module Sag2 has 20 rows of pins. The next five colums are pins. Here, pin 1 is ground (#), pin 2 is an unrouted spare, an so on. The sixth pin in each row is not shown because it is absent on the midplane connectors.

Figure 9–20 Daughter board connector legend.



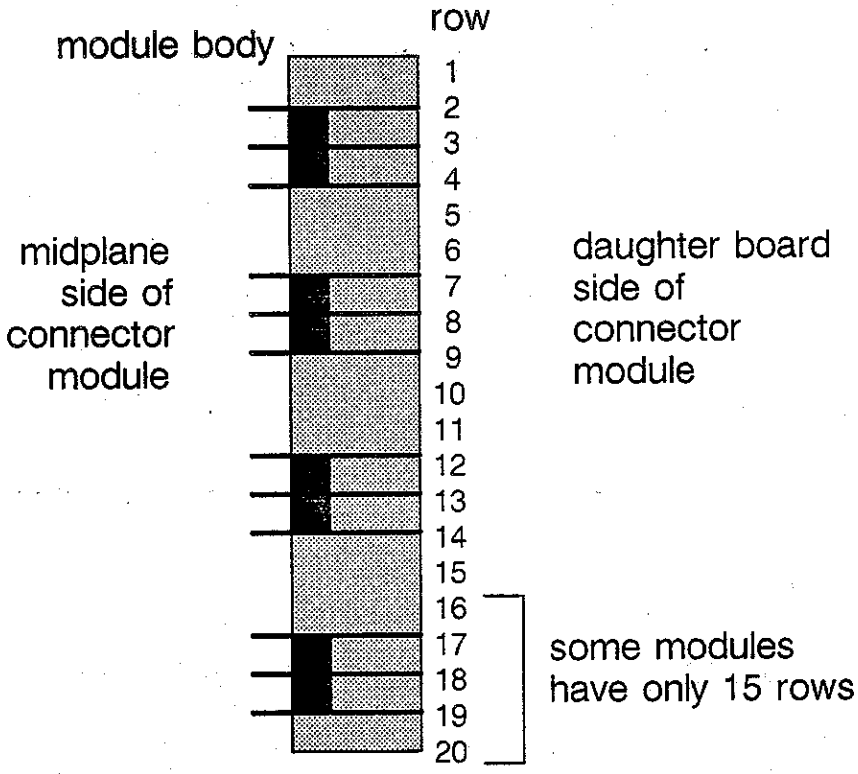
Signals in rows 3 through 11 here are a group named S_TO_FB1<15..0>. The 16 signals in each such S_TO_FB (switch S-card to function board) group are lettered A through N, plus two spares (o, o). The signal name for each of these letters is given to the right. Thus, pin "A" (in row 4) carries signal D<7> to function board 1. Rows 12 through 20 are another group of the same kind, so pin "A" in row 13 carries another signal, also named D<7> but to function board 2. A blank line separates rows in different signal groups, such as between rows 11 and 12. This is only to aid the reader; there is no extra space on the real connector.

7. DAUGHTER BOARD COLUMN 1 GROUND PINS

Any pin in column 1 is always ground. All daughter boards have pins present in column 1 of every row. The situation on the midplane is more complicated, however, and is discussed here.

The midplane modules have column 1 pins in the middle three of every five pin positions. For example, the midplane (J) connector shown in Figure 9–20 has no column 1 pins in rows 1, 5, 6, 10, 11, 15, 16 and 20. The three pins in the middle of each five positions are bussed together within the connector. This structure is shown in Figure 9–21.

Figure 9–21 Column 1 midplane pins in daughter board connectors.



A further complication arises from the back-to-back placement of switch cards and function boards on opposite sides of the midplane. In a few places, column 1 pin positions of a switch card fall in exactly the same spot as column 1 pin positions of a function board. Where this happens, the pin on the switch card is trimmed off so that it does not protrude beyond the module body into the midplane. This occurs in 9 rows on each of the four switch card connectors, and is shown in Figure 9–22 and Figure 9–23. When one of the projections is cut off, the group of three pins bussed within the module are still well grounded by the remaining two projections.

Figure 9–22 Column 1 pin trimming — detail.

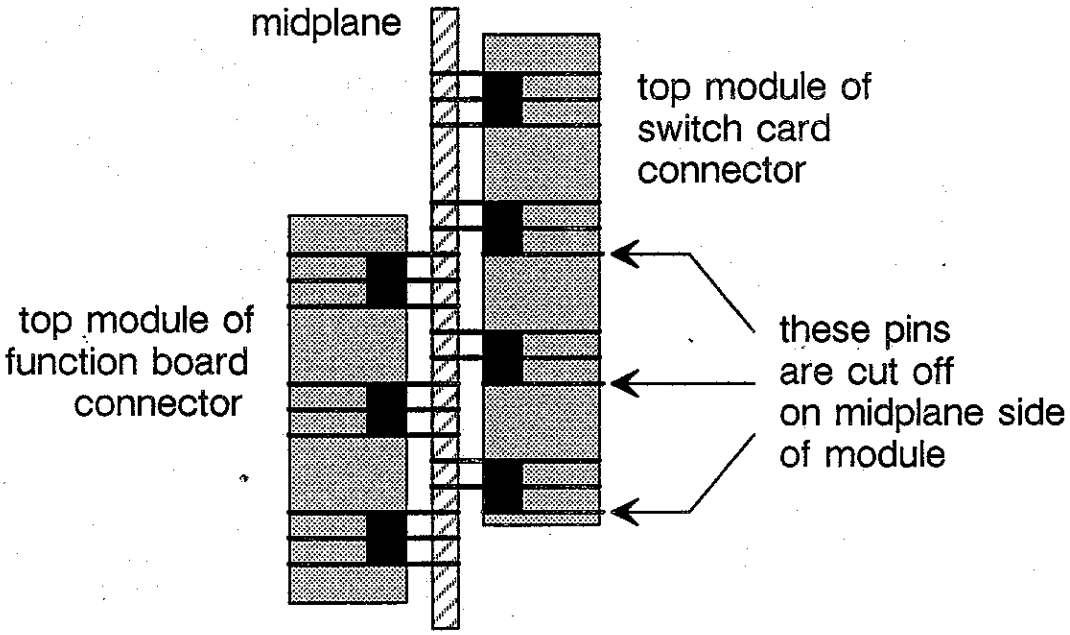
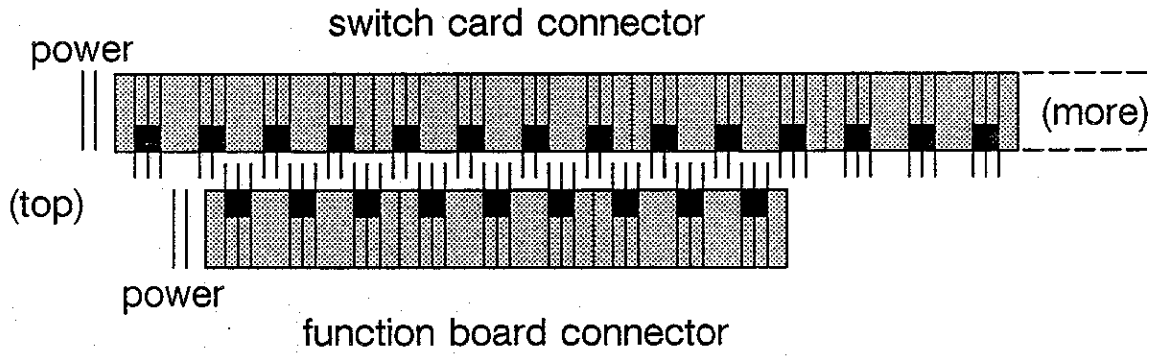


Figure 9–23 Column 1 pin trimming — nine pins.



END OF MIDPLANE CONNECTOR NOTES

9.3 Butterfly II Switch Cards (B2SR, B2SS)

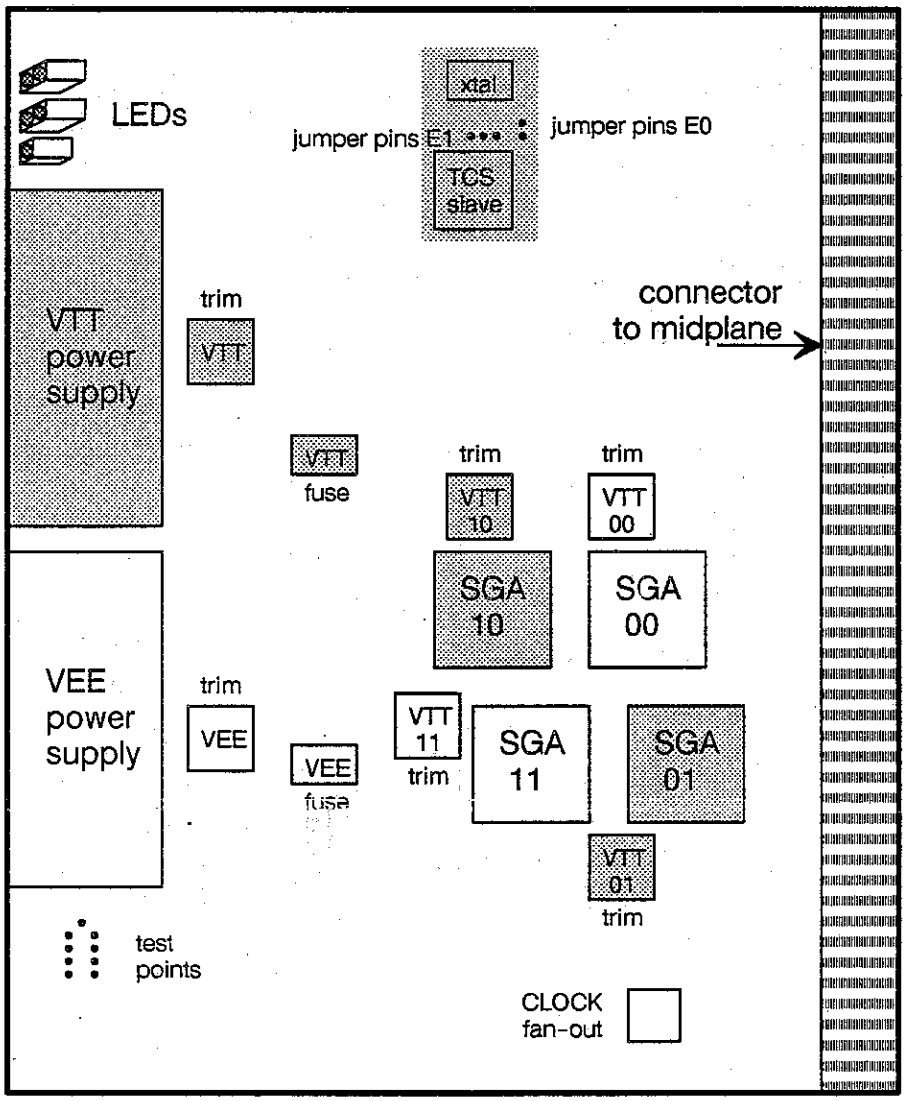
9.3.1 Introduction

The Butterfly II switch provides the means of communication for up to 512 function boards. This is done via a 512 by 512 virtual crossbar which is implemented using a modulo eight, 2- or 3-column Butterfly switch. "Modulo eight" refers to the size of the basic switching element: an 8 by 8 crossbar. A 2-column switch connects a system with up to 64 (8×8) slots; a 3-column switch connects a 512 ($8 \times 8 \times 8$) slot system. The first round of Butterfly II switch development implements a 2-column switch with a maximum size of 64 slots. Future development will expand the maximum size to three columns (512 slots).

The 2-column switch is implemented with an equal number of two printed circuit cards, the Butterfly II Switch Requester (B2SR) card and the Butterfly II Switch Server (B2SS) card.

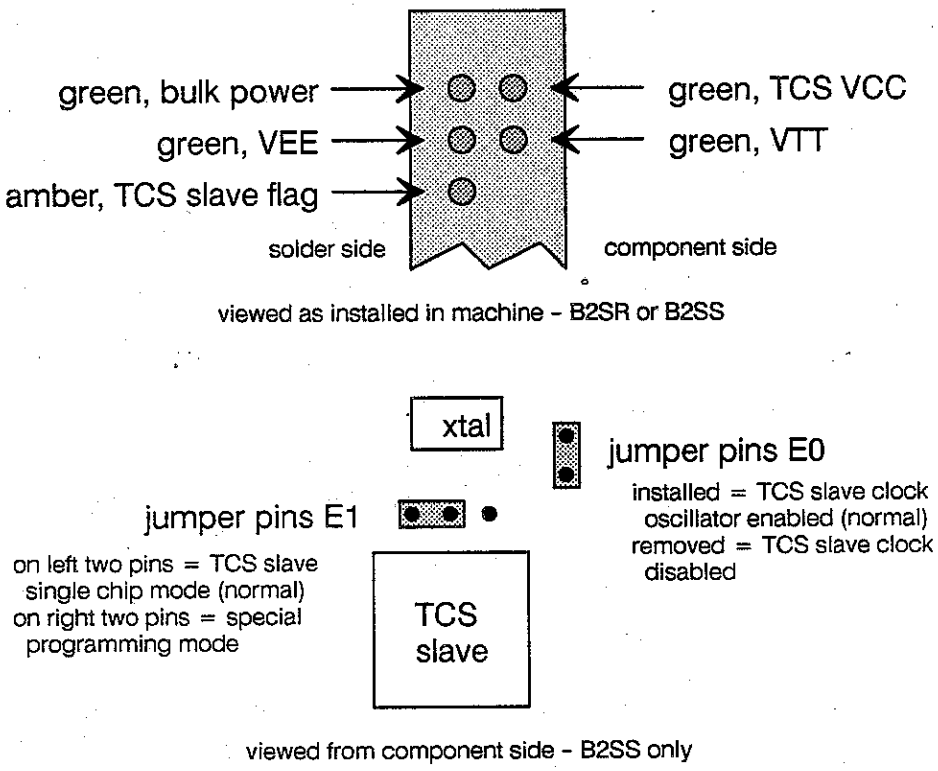
The B2SR and B2SS cards are similar in function and in physical layout. Figure 9-24 shows the layout of the B2SS card. Both cards have separate VTT supplies for the logic circuits internal to each Switch Gate Array (SGA) chip, and associated trim pots. The clock fan-out is a 100E111 chip, as discussed later. Figure 9-25 shows what each LED indicates, and the correct B2SS jumper positions for normal operation. The parts shown shaded in Figure 9-24 are different on the B2SR. The B2SR derives its VTT power from the B2SS, so it has no VTT supply. Nor does it have a TCS slave of its own, because it is controlled by the TCS slave on the B2SS2 card. The SGAs are arranged differently on the B2SR — the positions of SGA 10 and SGA 01 are swapped.

Figure 9–24 B2SS and B2SR physical layout.



viewed from component side

Figure 9-25 Switch card indicators and jumpers.



9.3.2 Glossary

The terms below are used in discussing switch cards.

FUNCTION BOARD: Any board that plugs into a Butterfly II switch for communication with any other board attached to that switch (e.g., B2VME).

MIDPLANE: The interconnect printed circuit board for eight function cards and one (two if redundancy is configured) pair of switch cards. This board serves the function of a backplane but has cards on both sides, thus the name "midplane".

REPLY: When a request has been issued, a server may be required to respond (as in the case of a remote "read"). This response message is a "reply".

REQUEST: When a function board needs to communicate over the switch, the requester's SIGA attempts to initiate a message. This is a "request".

REQUESTER: This refers to the part of the SIGA that initiates switch communications.

SERVER: This refers to the part of the SIGA that responds to requests. It "services" the request.

SIGA: Switch Interface Gate Array.

TCS: Test and Control System.

9.3.3 Function

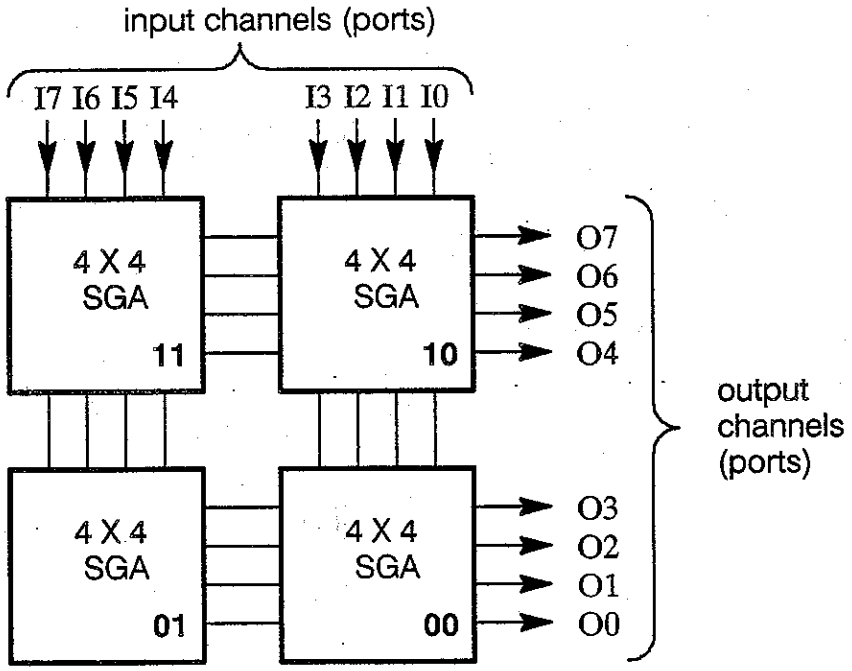
The switch cards perform three basic functions:

- They provide either an input or an output crossbar for eight function boards.
- They provide fan-in and fan-out for several signals that must be distributed in parallel to/from all function boards.
- The switch card TCS slave can control and monitor the individual switch connections. They can enable and disable any crossbar port and can measure certain performance parameters of crossbar ports.

9.3.4 Crossbar

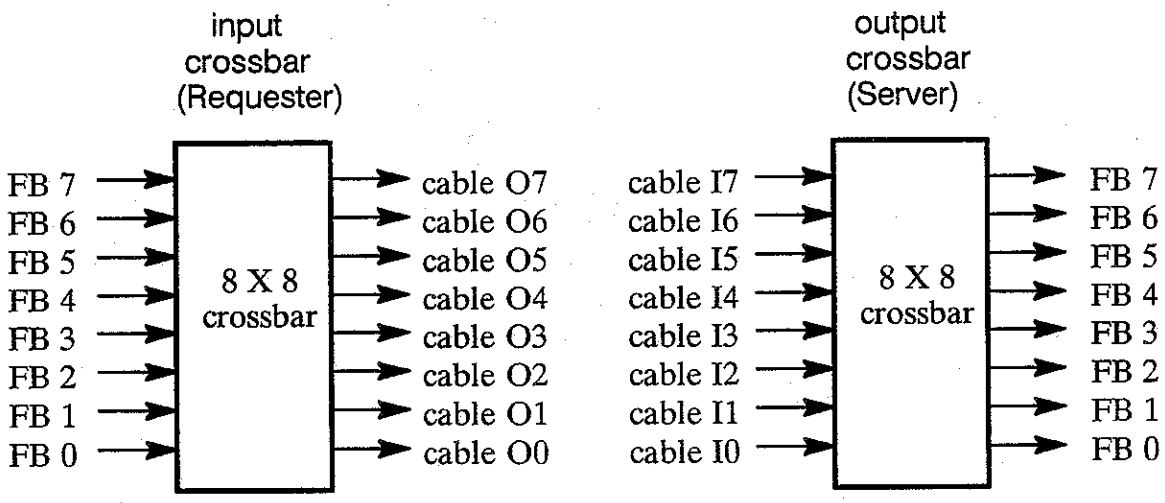
Each Butterfly II switch crossbar is implemented using four bipolar gate arrays (SGAs). Each SGA implements a 4 by 4 slice of the 8 by 8 crossbar. Each of the eight ports contains eight bidirectional data wires, a frame signal, and a reverse signal. Figure 9-26 shows how the SGAs are wired together. Each input wire is an input to two SGAs; each output wire is connected (wire-OR'ed) to two SGA outputs.

Figure 9–26 Butterfly II eight–by–eight switch unit.



The 8–bit data paths through the Butterfly II switch are bidirectional; that is, a request goes out and the reply comes back over the same wires. While the paths are bidirectional, switch messages are always initiated from function board to switch requester column to switch server column to function board, as indicated by the arrows in Figure 9–26 and Figure 9–27.

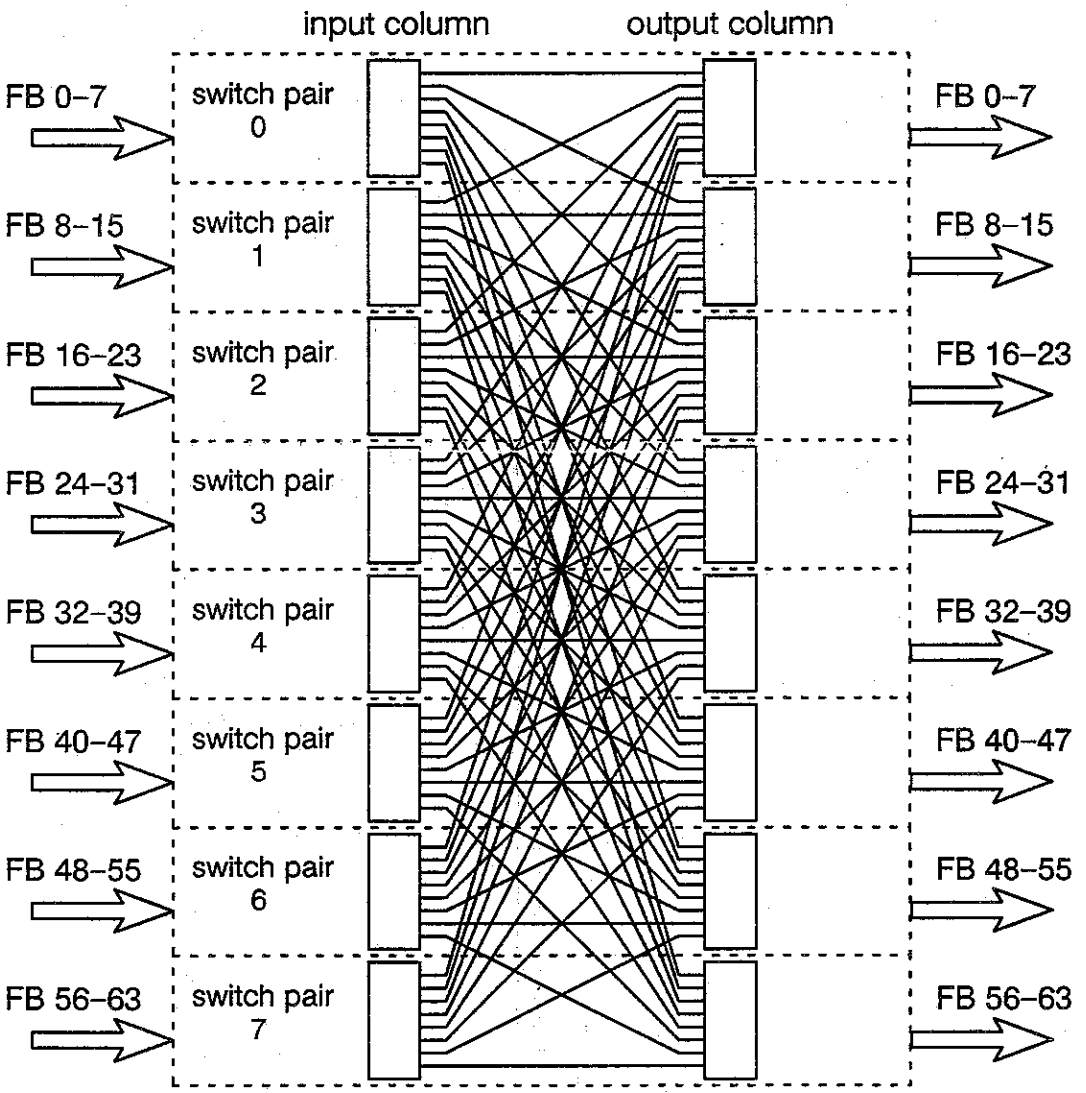
Figure 9–27 Switch connection initiation.



9.3.5 Card Pairs

A 64–slot Butterfly II switch is implemented with two types of switch card — the B2SR and the B2SS. The B2SR contains a requester (input) crossbar, switch support circuitry, and several fan–out signals; the B2SS contains a server (output) crossbar, switch support circuitry, several fan–out signals, and a TCS slave. A switch pair (or R–S pair) refers to the B2SR and B2SS that are connected to the same eight function boards. The TCS slave on the B2SS controls both cards of the R–S pair. There are eight R–S pairs in a 64–slot machine, arranged as shown in Figure 9–28.

Figure 9–28 64–slot Butterfly II switch.



9.3.6 Switch Support Circuits

The crossbars require a small amount of support logic. This logic includes a random number generator, a "hold priority" resynchronizer, and a "reset" synchronizer. A 3-bit random number (RAND0, RAND1, RANDOM) is used for arbitration in the SGAs. RAND0 and RAND1 are for internal arbitration; RANDOM is for arbitration between the two crossbars that share output wires. Thus, it is mandatory that RANDOM for SGAs 11 and 01 (see Figure 9-26) be opposite in value to RANDOM for SGAs 10 and 00, so that they don't both drive the same output port at the same time. Because a pseudo-random number is sufficient, a 7-bit maximum sequence generator ($x^7 + x^3 + 1$) is used to implement the random number. The generator can be set to all 1's or all 0's for testing purposes.

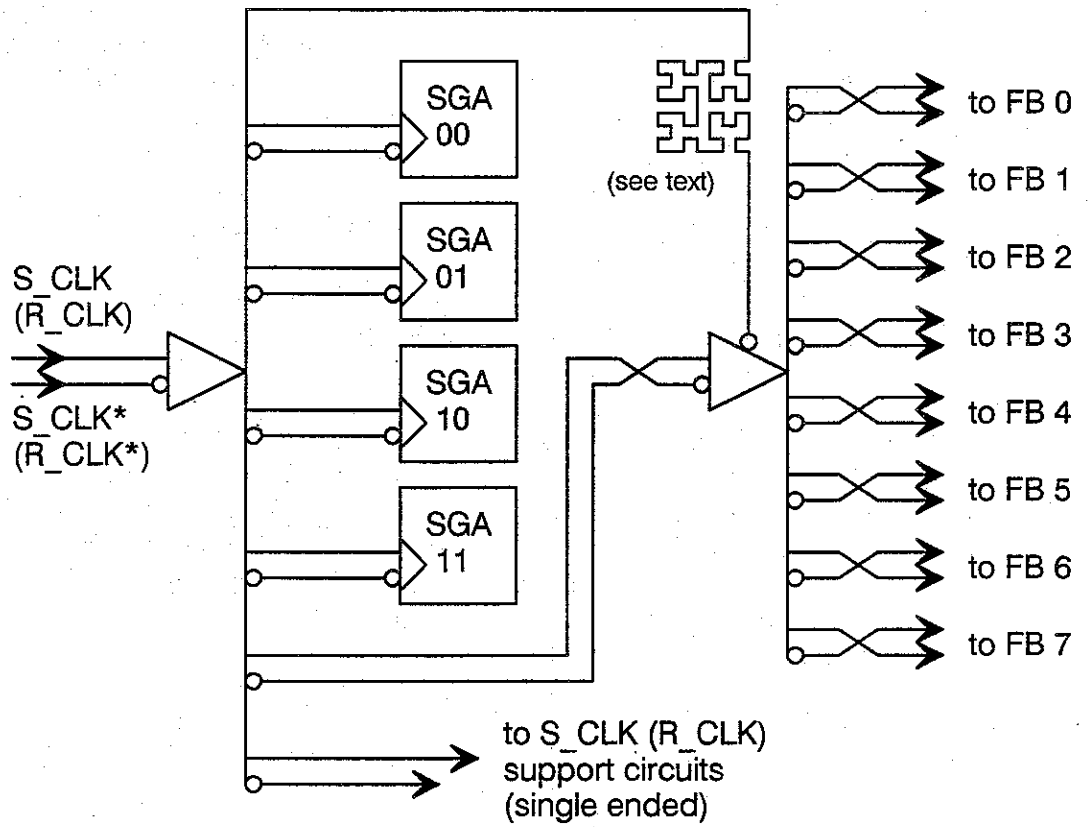
The dead state for the random number generator is all 1's. Once in the dead state, the random number generator remains there until the TCS slave injects a zero. For every S(R)_CLK tick that S_TO_R_TCS<RAND_RESET> is held high, a zero is shifted into the low bit of the random number generator, and is then shifted up on the subsequent clock tick. The random number changes every S_CLK (or R_CLK on the B2SR) tick. Only the three top bits are sent to the SGAs.

The SGA **hold** signal indicates to the SGAs that the current priority level of output ports should be held. The hold signal going false instructs the SGAs to forget that they have seen a high priority message. This signal is generated on the clock master card (B2CLK), and is distributed differentially to each of the switch cards. The period of the hold signal is in the range of 1 to 32 microseconds, and is TCS programmable. There are actually two hold signals distributed throughout the system: S_HOLD clocked by S_CLK (for the B2SS cards), and R_HOLD clocked by R_CLK (for the B2SR cards). The SGAs clock the hold signal on the rising edge of S(R)_CLK. The hold signal is clocked onto the switch cards on the falling edge of S(R)_CLK, and stays low for one period.

9.3.7 System Clock Distribution

The 40 MHz system clock is fanned-out via the switch cards. There are actually two clock signals, one for the requester switch column and one for the server switch column. R_CLK and S_CLK are guaranteed to run at exactly the same frequency but may be 180° out of phase. This phase difference can be selected to optimize the switch cable length. If the two clocks are 180° out of phase, the cable delay must be large enough to have at least one "½ bit" in the cable. In a system running at 40 MHz, ½ bit in the cable corresponds to an extra delay of half the clock period (½ of 25 = 12.5 nanoseconds), or about 12 feet of cable. The clock phase (0° or 180° out of phase) is selected by a jumper on the clock master card. Each of the two clocks has its own fanout tree. Figure 9-29 shows the circuit, which employs an ECLiPS family chip that has low output-to-output skew and is expressly intended for fan-out buffering.

Figure 9-29 Switch card clock fanout.



Note that each function board is isolated from the switch clock by a buffer. This guarantees that a failure on a function board will not bring down the switch clock on other function boards. The same holds true upstream for the switch cards at the clock master.

The function board clock buffer is driven by two clock signals in a scheme to adjust the duty cycle of the function board clock signal for highest speed operation. The circuit does this by stretching the duration of time that the clock is high. This is accomplished by using the De Morgan equivalent of the normal buffer application. Normally, the low-true "enable" input to the buffer creates a logical AND function at the outputs. By inverting the input and outputs, the "enable" input now functions as a high-true logical OR input. Because the input and outputs are differential, they can be inverted simply by exchanging the positive and negative polarity signals of each pair.

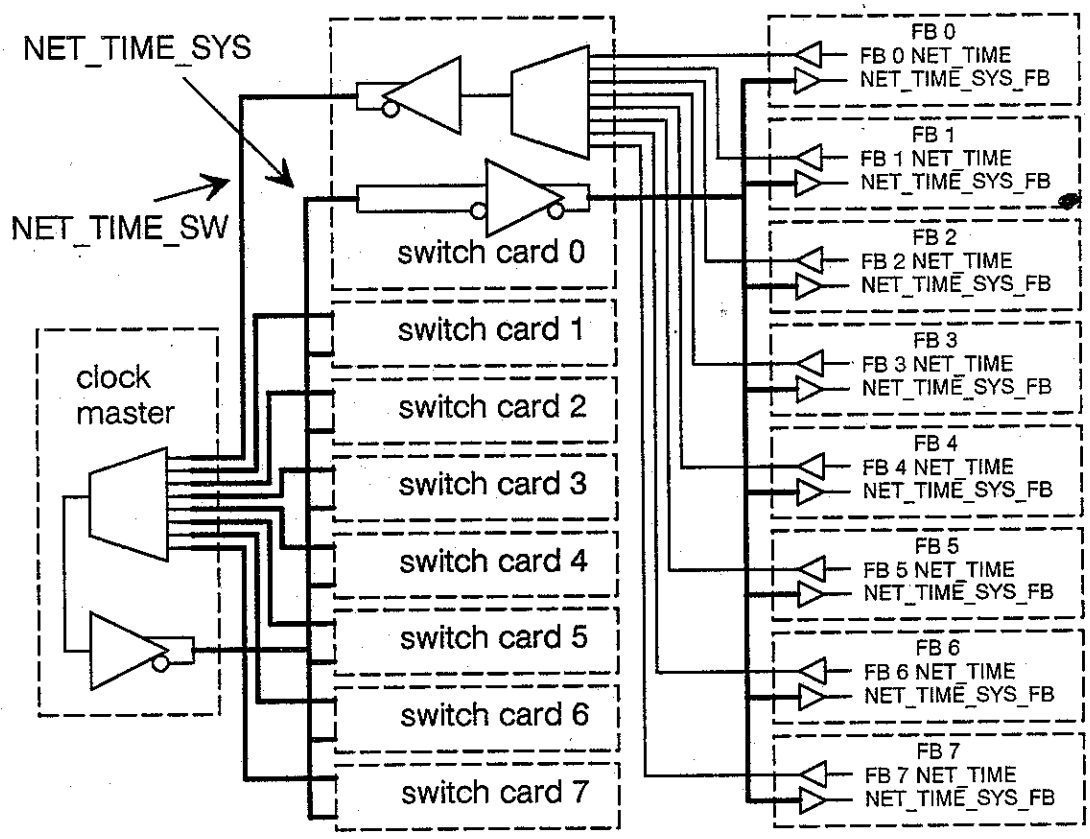
The differential inputs of the function board clock buffer are driven over the shortest practical circuit board trace, while the "enable" input is driven through a long trace. Thus, the differential input signal controls the timing of the rising edges at the buffer outputs, and the "enable" input controls the falling edges. The length of the long trace is selected to generate the proper duty cycle of the clock signals sent to the function boards.

9.3.8 NET_TIME

In some applications, such as certain kinds of circuit switching, there is a requirement that all the I/O boards know the network time. The source of the system **NET_TIME** signal may be any I/O board in the Butterfly II machine. The B2SS selects one of the function cards' NET_TIME to supply this signal. This signal is sent back to the central clock board over the clock cable as NET_TIME_SW. The clock master card selects the signal from one of the B2SS cards and distributes it (as NET_TIME_SYS) back out to all of the B2SR cards. Each B2SR card then busses the NET_TIME_SYS_FB signal to the eight function boards attached to it.

The individual NET_TIME signals are transmitted single-ended with ECL levels from the function cards to the B2SS card; NET_TIME_SW is transmitted differentially with ECL levels from the switch cards to the clock master card. NET_TIME_SYS is transmitted differentially with ECL levels from the clock master card to the B2SR cards; NET_TIME_SYS_FB is transmitted differentially from the B2SR cards to the function cards. In Figure 9–30, heavy lines indicate differential signals.

Figure 9–30 Collection and distribution of NET_TIME.



9.3.9 TCS Slave – Communication

The TCS slaves communicate with the TCS master via a 2–wire serial bus. One wire carries data from the TCS master to TCS slaves; the other carries data from TCS slaves to the TCS master. The format is similar to normal asynchronous transmissions.

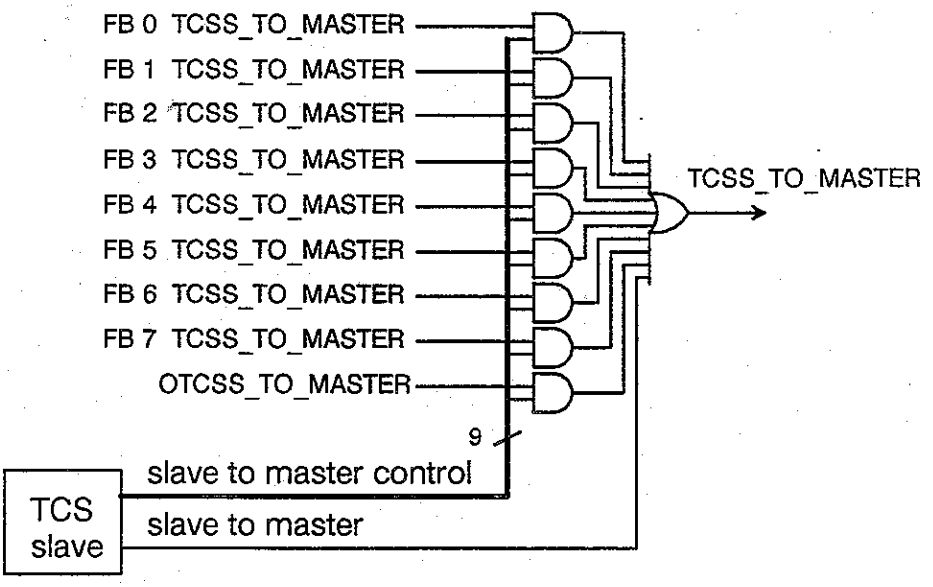
Master to Slave

The TCS master to slave communication signals are bussed on the midplane to the switch cards and the function boards. Resistors in series with the master–to–slave receivers on both the function cards and the switch cards guarantee that a failure in an individual function board or switch card slave will not disable the TCS bus. Both the A and the B TCS buses are received by all function boards and B2SS cards.

Slave to Master

The TCS slave on the B2SS card controls access of the slaves to the master. This is to prevent a failure on one slave from jamming the slave–to–master bus. Figure 9–31 shows how this is implemented. If a slave erroneously drives the bus out of turn, the TCS master can use this circuit to determine which slave is causing the problem, and then disable that slave's slave–to–master communication.

Figure 9–31 TCS slave to master circuitry.



Redundant Switch and TCS Support

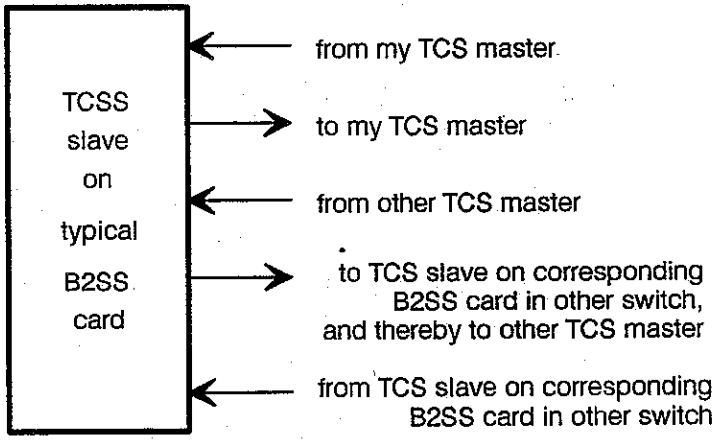
A Butterfly II machine may be configured with a redundant switch and/or a redundant TCS master. In a machine configured with two switches, each B2SS card in one switch corresponds to a particular B2SS card in the other switch — they serve the same eight function boards. The slave–to–master data signal from the TCS slave in each of these two corresponding B2SS cards can be sent (via its respective clock card) not only to the TCS master, but also can travel across to the corresponding B2SS TCS slave in the other switch. This allows a Butterfly II system to have a redundant switch without requiring a redundant TCS master. In Figure 9–31, the signal OTCSS_TO_MASTER is the slave–to–master data from that corresponding B2SS TCS slave. As the figure shows, it is individually controlled by the local B2SS TCS slave.

In the master–to–slave direction, no special hardware is needed on the B2SS to support redundant switches.

A machine configured with both a redundant switch and a redundant TCS master contains two complete TCS buses, one from each master. Each B2SS receives data from each TCS master, and chooses which one it will listen to. In the slave–to–master direction, each B2SS sends data to one master directly through its clock card, and to the other master indirectly through the TCS slave on the corresponding B2SS in the other switch. See Figure 9–32.

A machine with one switch but two TCS masters can also be supported. Each B2SS TCS slave receives both masters' data, as above. In the slave–to–master direction, each B2SS TCS slave transmits normally to its own master. A jumper, installed where the redundant corresponding B2SS would be, takes OTCSS_TO_MASTER to the other master.

Figure 9–32 **Support for redundancy.**



9.3.10 TCS Slave – Control

Power Supply Control

The TCS slave on the B2SS can turn on and off the power supplies as well as margin them ±5% and ±10%. This is possible because the TCS is powered separately from the rest of the system.

Board Flag LED

There is one amber LED on each B2SR card and each B2SS card. This LED is used to flag the board for field service. The TCS slave can turn either (or both) LEDs on and off when instructed to do so by the TCS master.

SGA Control

Each of the eight SGAs in an R–S pair has a 6–signal interface to the TCS slave. On the SGAs the signals are:

mreset This is the master reset for the SGA. It resets everything on the array. (asserted high)

cntrlr When this pin is pulled high, all of the input and output ports are (asynchronously) enabled. (asserted high)

cntldi This is the (serial) input to the (13–bit) SGA control word register. The SGA documentation describes the command words. (asserted high)

cntldo This is the (serial) output of the SGA control word register. A negative polarity version of this signal, cntldo*, is also generated by the SGA, but it is simply terminated on the switch card to minimize simultaneous switching noise within the SGA. (asserted high)

cntrlc The SGA control word register is clocked by the rising edge of ctrlc. (asserted high)

cntrle This is the execute pin, that signals to the SGA to execute the current command in its control word register. (asserted high)

In a switch card pair, all eight SGA *mreset* pins are connected to the signal called S_to_R_TCS<RESET>, and all eight SGA *cntrlr* pins are also connected to S_TO_R_TCS<RESET>.

The TCS slave processor (Motorola 68HC11) uses its serial peripheral port (SPI) to communicate with the eight SGAs in the R–S pair. The SPI_MOSI (Master Out Slave In) signal is connected to the *cntldi* pins of each of the eight SGAs. The SPI_CLK signal clocks (on the rising edge) the control word regis-

ter of all eight SGAs. The logical OR of the four B2SR card SGAs' *cntldo* is R_SGA_DATA_OUT, and the logical OR of the four B2SS card SGAs' *cntldo* is S_SGA_DATA_OUT. Both of these signals are among those that can be selected by the TCS slave to be read as SPI_MISO (Master In Slave Out). Each of the eight SGAs' *cntrle* pins has a dedicated control wire from the TCS slave (SGA_EXEC_S11, SGA_EXEC_S10, etc.).

SGA Crossbar Port Control

Each input port or output port of the crossbars may be independently enabled and disabled by the switch card TCS slave. Enabling or disabling a port interrupts any activity on that port; the switch is not "well behaved" during an enable or disable. The TCS arranges the cooperation of the function boards in quieting switch activity before disabling any ports.

SGA Individual Data Line Control

The TCS slave can also individually read input port signals or assert output port signals — the data, frame, or reverse signal in any one of the eight SGAs on the switch card pair. This feature can be used in diagnostics to isolate interconnect or gate array failures.

Switch NET_TIME Control

The TCS slave on the B2SS controls which of the eight NET_TIME signals from function boards gets sent to the clock master card by setting the NET_TIME_SELECT<2..0> bits to the appropriate value.

9.3.11 TCS Slave – Monitor Functions

Environment Monitoring

The TCS slave on the B2SS monitors the voltage levels of VEE and VTT on both cards and the board temperatures of both cards in the R–S pair via analog sensors read by the analog input port of the TCS slave processor (68HCll).

Cable Signal Monitoring

All signals that cross the midplane connector from the clock cable, and those to or from the function boards, can be selected to be monitored by the TCS slave, via the 68HC11's SPI_MISO signal. (Some signals are monitored indirectly, and spare wires are not monitored.) As mentioned above, each of the SGA port wires can be individually controlled (and monitored) via the SGA/TCS interface. Additionally, the following signals can be monitored:

- On the B2SR card: MSEC65, NET_TIME_SYS_FB, and R_HOLD.
- On the B2SS card: NET_TIME_SW and S_HOLD.

Note that monitoring S(R)_HOLD also indirectly monitors S(R)_CLK, because S(R)_HOLD won't change if S(R)_CLK is not changing. S(R)_CLK is monitored only indirectly to avoid adding the circuitry that would be required to divide S(R)_CLK down to a TCS-readable frequency.

9.3.12 TCS Slave — Card Identity

The TCS slave on the B2SS card reads the card identity (ID) from the midplane connector. The fields of the switch card ID are BAY_ID<2..0>, MIDPLANE_ID<2..0>, and A_OR_B_SWITCH. A_OR_B_SWITCH is read in the SLOT<0> bit of the TCS location word. SLOT<2..1> are hardwired to "0" on the B2SS card. In addition, bits TYPE<3..1> of the TCS location word are hardwired to binary "000" to indicate a B2SS-type switch card. The TYPE<0> bit is wired to the midplane connector pin COLUMN_2_OR_3, for use in 3-column machines. On the midplane, TYPE<0> is wired to "0" (ground) for a 2-column machine, and to "1" (open) for a 3-column machine. The TCS slave processor on the B2SS card stores the card's serial number and revision number in (on-chip) EEPROM. The B2SR card's serial number and revision number are stored in a TCS-programmable, 256-bit serial EEPROM that is read by the TCS slave.

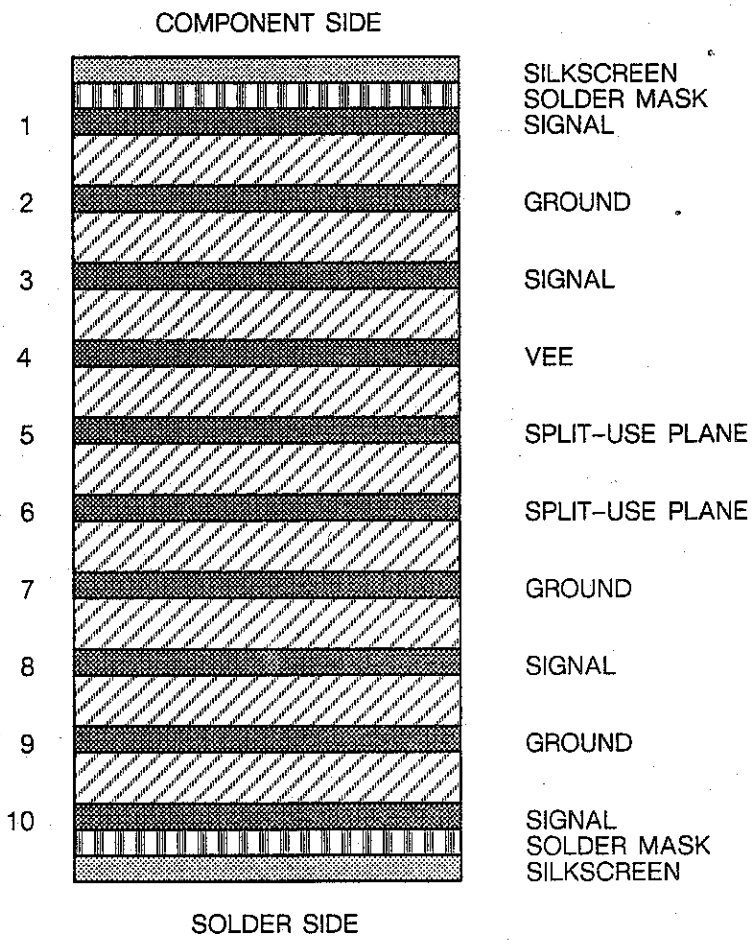
9.3.13 65-millisecond Pulse

A B2SR card contains fan-out circuitry for the 65-millisecond pulse. This signal, MSEC65, is used by the SIGA to maintain the synchronization of the real-time clock. The ECL signal MSEC65 is received by the B2SR differentially from the clock master card. It is clocked on the B2SR by the rising edge of R_CLK and is fanned-out differentially to each of the eight function boards. The individual fan-out on the B2SR card insures that a failure on any one function board will not interfere with the 65-millisecond signal on any of the other function boards.

9.3.14 Printed Circuit Board

The B2SS and B2SR boards are each 11-inch by 16.9-inch, 10-layer — 4 signal, 6 power/ground — controlled impedance (50Ω stripline and microstrip) PC boards. Figure 9-33 shows the use of the ten conductive layers.

Figure 9-33 B2SS and B2SR board layers.



Characteristic Impedance

The high speed ECL circuitry on the board requires a controlled impedance environment to operate properly. Simulations show that 50Ω ±10% is an acceptable range. All switch port signals (FRAME, REVERSE, D<7..0>) are parallel terminated in 50Ω to VTT at both ends. This is required for each of the bidirectional lines, so that no matter in which direction the line is being driven, it will be terminated at the receive end. Note that this makes the load on the SGA drivers 25Ω (50Ω in parallel with 50Ω = 25Ω). Thus, the SGA's drivers are all 25Ω drivers. Drivers that are "off" present a high impedance to the wire. 10-pin, 6-resistor SIPs that include bypass capacitors and extra VTT leads were chosen for termination to reduce switching noise at the resistors.

Routing

The pin assignments of the 301-pin SGAs were made with the routing of the switch cards in mind. The most dense routing on the switch card is in the neighborhood of the four SGAs and their terminators. Given the regularity of the routing, and the pinout of the SGA, the B2SR and B2SS were routed with four signal layers.

9.3.15 Displays and Switches

There are no mechanical switches on the B2SS or the B2SR. There are five LEDs on each card.

green — VEE power-on indicator
green — VTT power-on indicator
green — TCS VCC power-present indicator
green — main power (±24 volts) power-present indicator
amber — TCS flag, to aid field service in locating the board

9.3.16 Signal Lists

These signals go to the midplane connector, from where they are routed as described below. See the midplane description for pinouts.

Switch Card to Switch Card

On the midplane, these signals are routed to a switch cable. The R_TO_Sn and S_TO_Rn signals for a given pair are routed to the same switch cable. There is one set of these signals per port, n = 0 to 7.

Signal Function	B2SS Name	B2SR Name
frame	S_TO_Rn<FRM>	R_TO_Sn<FRM>
reverse	S_TO_Rn<RVRS>	R_TO_Sn<RVRS>
data7..data0	S_TO_Rn<D7..D0>	R_TO_Sn<D7..D0>
spares	S_TO_Rn<SPARE1..0>	R_TO_Sn<SPARE1..0>

summary: 12 signals times 8 switch ports = 96

Switch Card to Function Board

There is one set of these signals for every function board, n = 0 to 7.

Signal Function	B2SS Name	B2SR Name
frame	S_TO_FBn<FRM>	R_TO_FBn<FRM>

reverse	S_TO_FBn<RVRS>	R_TO_FBn<RVRS>
data7..data0	S_TO_FBn<D7..D0>	R_TO_FBn<D7..D0>
65 msec.		R_TO_FBn_MSEC65
		R_TO_FBn_MSEC65*
requester clock		R_TO_FBn_CLK
		R_TO_FBn_CLK*
server clock	S_TO_FBn_CLK	
	S_TO_FBn_CLK*	
net time	S_TO_FBn<NET_TIME>	
TCSS to master	S_TO_FBn<TCSS_TO_MASTER>	
spares	S_TO_FBn<SPARE1..0>	R_TO_FBn<SPARE1..0>

summary: 16 signals times 8 function boards = 128

The signal below is bussed from each B2SR card to the function boards.

NET_TIME_SYS_FB
NET_TIME_SYS_FB*

Clock Cable to Switch Card

Signal Function	B2SS Name	B2SR Name
R clock		R_CLK
		R_CLK*
R hold		R_HOLD
		R_HOLD*
65 msec.		MSEC65
		MSEC65*
system net time		NET_TIME_SYS
		NET_TIME_SYS*
S clock	S_CLK	
	S_CLK*	
S hold	S_HOLD	
	S_HOLD*	
switch net time	NET_TIME_SW	
	NET_TIME_SW*	
TCS master to slave	MASTER_TO_TCSS	
TCS slave to master	TCSS_TO_MASTER	

B2SS to B2SR TCS

Signal Function	Name
ID select	S_TO_R_TCS<ID_SELECT>
random control	S_TO_R_TCS<RAND_PRESET>
	S_TO_R_TCS<RAND_RESET>
Serial Peripheral Interface	S_TO_R_TCS<SPI_MOSI>
	S_TO_R_TCS<R_SPI_MISO>
	S_TO_R_TCS<SPI_CLK>
SGA TCS command execute	S_TO_R_TCS<EXEC_R00>

	S_TO_R_TCS<EXEC_R01>
	S_TO_R_TCS<EXEC_R10>
	S_TO_R_TCS<EXEC_R11>
SGA reset	S_TO_R_TCS<SGA_RESET>
command register clear	S_TO_R_TCS<NTTL_RESET>
command register latch	S_TO_R_TCS<R_CMD_EN>
ECL power control	S_TO_R_TCS<POWER_ENABLE>
voltage margining	S_TO_R_TCS<MARGIN_DISABLE>
VEE voltage monitor	S_TO_R_TCS<R_VEE>
VTT voltage monitor	S_TO_R_TCS<R_VTT>
SGA VTT voltage monitor	S_TO_R_TCS<R_SGA_VTT>
temperature sense	S_TO_R_TCS<R_TEMP>
spares	S_TO_R_TCS<SPARE2..SPARE0>

9.3.17 Power

There are two power levels required for the ECL circuitry on the B2SR and B2SS: VEE ≈ -4.5 volts and VTT ≈ -2 volts. (The precise voltage range to which each of these is set during manufacture and repair is given in the test specification document.) Both the B2SR card and the B2SS card have connections to the midplane main power and main power return (+24 volt and -24 volt) planes. The B2SS card generates VEE and VTT from ±24 volts, and busses VTT to the B2SR card via a bus bar on the midplane. The B2SR card generates its own VEE from main power.

The TCS slave circuits are powered by the centrally regulated TCS +5 volt supply. The B2SS card and the B2SR card midplane connectors have five TCS +5 pins each. The TCS slave can turn on/off VEE and VTT, and can margin them ±10% and ±5% while monitoring the voltage levels through its ADC.

There is one green indicator LED per power supply (main power, VEE, VTT, TCS +5) to indicate that power is present.

Additional topics — Future expansion of this section (B2SR and B2SS) should cover the following topics.

- functional block diagram – adapt from page 2 of schematics
- further details of TCS operation
  - description of TCS slave from programmer's viewpoint
  - accuracy of analog margining and sensing – from Cinsy
- glossary part should be merged into an overall glossary

9.4 The Butterfly II Clock Card (B2CLK)

9.4.1 Introduction

The B2CLK provides the switch clocks and other timing signals for a Butterfly II system of up to, and including, sixty-four function board slots.

All of the parts of a Butterfly II switch network are synchronous between the function board Switch Interface Gate Arrays (SIGAs). High-speed data transfer across a physically large synchronous machine demands great precision in the clock distribution system, so that the clocks received by each switching element are closely matched under all circumstances. Signal quality must also be held as high as possible, all without making the machine infeasible to build or market. The circuitry of the B2CLK is optimized to provide the tightest tolerances and best signal quality possible to the clock cables which distribute these clocks and other signals throughout the machine.

The clocks may be generated by either a fixed-frequency crystal oscillator, in the fashion of more traditional computers, or from a programmable frequency synthesizer. The synthesizer will allow the system to be operated through a range of frequencies.

A Butterfly II with more than sixty-four slots will require a two-level hierarchical clocking system with a master clock card and up to eight slave clock cards. Each slave clock card will take the place of the B2CLK for one sixty-four slot subsystem.

For a low-level description of the board-level implementation, see the B2CLK Logic Description and Logic Diagram.

9.4.2 Overview

The B2CLK generates the Requester and Server clocks (Rclk and Sclk) for a Butterfly II machine. The requester clock is distributed through the B2SR requester switch card to the function boards and the requester portion of the SIGA. It is used to synchronize the transmission of downstream or "request" messages and the reception of upstream replies or "responses". The server clock is likewise distributed through the B2SS server switch card and the server portion of the SIGA. Sclk synchronizes the reception of requests and the transmission of responses. Both signals are carried as differential ECL pairs with a 50% duty cycle. Rclk and Sclk are treated independently, though they share common ancestry and pains are taken to minimize their divergence. Although their frequencies will be matched, there is a configurable phase difference between the two. In addition to Rclk and Sclk, several other, non-clock signals are conducted between the B2CLK and the switch cards. The major signals in this category are listed below.

- **Sixty-five milliseconds** is a differential ECL periodic timing signal used by the SIGAs. This signal serves to maintain global synchronization of Real-Time Clocks (RTCs) across all the SIGAs in the machine. Sixty-five milliseconds is synchronized with Rclk. While the period of this signal is fixed at 65 ms, its relationship to Rclk is programmable to compensate for different Rclk frequencies.
- **Rhold** and **Shold** are also differential ECL periodic timing signals, and are used by the SGAs. The hold signal is required for the priority control scheme of the SGA internal output-port arbitration circuitry. These signals are identical except that Rhold is synchronized with Rclk, and Shold is likewise associated with Sclk. A programmable divider converts the switch clock into the hold signal which is the precursor to both Rhold and Shold.
- **Net time** is optionally generated by one or many function boards, and a selected version of it is then fanned out to all the function boards. The architecture of the Butterfly II performs both fan-in and fan-out in layers. Each switch card pair selects one signal from the eight function boards to which it connects, and passes that one on to the clock card. The B2CLK selects one output from the switch cards and fans it back to all switch card pairs. Each pair then fans the signal to its eight function boards. All of the selection is performed by the TCS, which can sample the selection to decide whether it seems reasonable (active). The upstream signal is called **net-time-switch**, while the signal sent back out is called **net-time-system**. Net time is not synchronized at any level in this selection/distribution subsystem. It is carried by differential ECL throughout.
- Test and Control System serial interface signals from the TCS master processor are fanned out to each midplane through the B2CLK. The serial interface responses from the TCS slave processors (also referred to as TCSSP or MC68HC11) are likewise fanned in and returned to the master processor. The signals are CMOS in both directions.

The B2CLK also communicates with the TCS master processor(s) through bidirectional TCS serial ports. One of the ports may be used to interconnect two B2CLKs so a TCS master may communicate with two B2CLKs through one cable. The interconnection schemes are discussed more fully in section 9.4.10.

9.4.3 Fallback Techniques

Butterfly II switch timing is quite aggressive. Two interdependent techniques have been devised to cope with timing problems in a reasonable manner, should they arise. Refer to section 9.4.7 for a discussion of different switch data (configuration) cable lengths.

The first technique, modifying clock phases and switch data cable lengths, will compensate for fairly minor performance problems such as large clock skew or poor cable quality. In this case the maximum length of a one–bit cable has been reduced by an unforeseen source of timing variability or delay, so that it cannot interconnect a thirty–two slot machine. The clock phases are changed to a one–and–a–half–bit scheme that requires long cables. A machine of this type can run at full speed with almost no degradation in the communication rate or latency. The main disadvantage for such a machine is that the excessively long cables boost system cost and make packaging difficult.

Another possibility is that a component such as a gate array fails to operate at a switch speed of 40 MHz. In this case the frequency synthesizer is set for a frequency lower than 40 MHz, to a point where all components will function reliably. For a machine connected with one–bit cables, the high–speed lengths are fully compatible with the lower speed so no hardware modification is necessary. Any one–and–a–half–bit cables in the machine will need to be lengthened for the lower speed, or they may be replaced with one–bit cables because the maximum allowed length of the one–bit cables is extended when the machine speed is reduced. The penalty for this repair is clear: switch bandwidth is diminished by an amount proportional to the reduction in switch speed.

9.4.4 Fixed Crystal Clock Generation

A commercial, moderate–precision TTL 80 MHz hybrid crystal oscillator will supply the fixed–frequency clock. Since it will be divided down to a 40 MHz signal by a flip–flop, its output duty cycle is unimportant. The precision of this oscillator will probably prove inadequate for long–term time–of–day timekeeping. It is assumed that a more precise time base will be available from a third–party peripheral (VME) board in machines which require good long–term timekeeping.

9.4.5 Frequency–synthesized Clock Generation

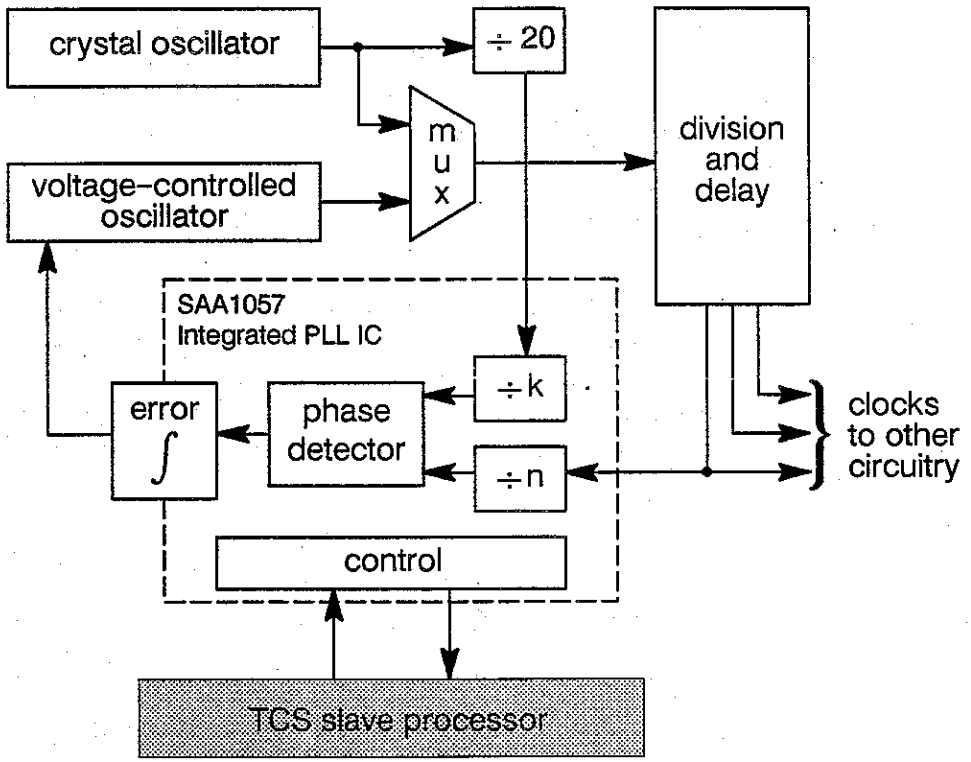
A programmable frequency synthesizer is included in the B2CLK to facilitate the operation of the machine at frequencies other than 40 MHz. This feature will enable the manufacturing group to margin system timing as a part of quality assurance inspection. Control and monitor functions related to the synthesizer are fully under the control of the TCS, in keeping with the philosophy of that system. The TCS interface will also enable the TCS to test the synthesizer — this is particularly valuable for the analog circuitry incorporated in the phase–locked loop. The synthesizer is locked to the 80 MHz crystal oscillator, so its long–term precision and stability will be equal to that of the crystal.

The Real Time Clock (RTC) circuitry in the SIGA imposes a restriction on the frequencies at which a Butterfly II may operate and maintain correct real time. The RTC is run by the output of a programmable one–microsecond prescaler. This prescaler divides the switch frequency by a programmed value to produce

a 1 MHz signal. The prescaler will only operate correctly if the switch frequency is an integer multiple of 2 MHz. In addition, the design of the frequency synthesizer allows an output range of only about 2:1, which has been placed between 30 and 44 MHz (switch clock frequencies).

A frequency-synthesizing phase-locked loop (PLL) is comprised of many unusual components. Also, the B2CLK integrates several features into the traditional PLL design, further obfuscating the PLL function. A simplified block diagram of the B2CLK synthesizer, presented in Figure 9-34, is described here to introduce the components and issues related to frequency synthesizer design.

Figure 9-34 **Block diagram of the B2CLK frequency synthesizer and clock selection circuitry.**



The heart of any phase-locked loop is a *phase detector*, which appears near the middle of Figure 9-34. This detector compares two signals and produces an (analog) signal describing the sign and magnitude of the difference between their phases. The phase difference or error signal is presented to an *integrator*, which changes its output level according to the "correction" specified by the phase difference signal. If the phase detector produces a positive error pulse, for example, the integrator will increase the value of its output. The output level of the integrator is called the PLL *control voltage*, because it specifies the output of the *voltage-controlled oscillator* or VCO. This oscillator produces

a frequency proportional to the control voltage. In the B2CLK, a multiplexer (mux) determines whether the master clock source is derived from the synthesizer or from the crystal oscillator. In order for the PLL to function, this multiplexer must select the VCO signal. This VCO output is further processed by B2CLK-specific division and delay circuitry, but a version of the VCO signal is finally presented back to one of the phase detector inputs, through a programmable divider, labelled "÷n" here. The other phase detector input is supplied by a form of the board's crystal oscillator, after division by two stages of fixed prescalers, labelled "÷20" and "÷k". The ÷n counter is programmed by the TCS slave processor to divide the VCO-derived signal such that its frequency will match the output of the ÷k section if the VCO produces exactly the desired frequency. The phase detector impels the VCO toward the desired frequency through a form of negative feedback.

Imagine, for example, that the B2CLK has just been turned on. It is reasonable to suppose that the control voltage is zero at this point, since there has been no signal to integrate. Some value is loaded into the ÷n counter, and immediately the phase detector becomes active. It observes that the VCO-derived, or *variable frequency* is lagging the crystal or *reference frequency*, so it sends a positive correction signal with large magnitude to the integrator. This signal results in an increase in the control voltage level, which boosts the VCO frequency (closer to the desired frequency), and the process continues until the desired frequency is reached. When the frequency (and phase) of the variable and reference signals become matched, the error signal from the phase detector becomes very quiet, reporting only very small errors resulting from capacitor leakage, temperature changes and the like. In this quiet state, the PLL is said to be *locked*, because the variable frequency and phase is actively matched (locked) to that of the reference frequency. The B2CLK synthesizer reports this locked state to the TCS slave processor.

Note that the discussion above treats the terms "PLL" and "frequency synthesizer" as roughly equivalent. The frequency synthesizer is properly a subset of PLL, because the synthesizer uses a programmable divider in the path of the variable frequency, while the PLL may use a fixed divider instead. Since the divider value in the Butterfly II synthesizer is programmable, it can generate a variety of frequencies, while a fixed-divider PLL can produce only one frequency.

SAA1057 Frequency Synthesizer

The Signetics SAA1057 is a highly integrated phase-locked loop radio tuning IC that supplies most of the circuitry required for the frequency synthesizer. It includes a prescaler, fifteen-bit programmable frequency-division counter, two phase detectors, a reference oscillator, programmable current amplifier and op-amp, plus logic to control the phase detectors and a serial command interface, all integrated into a single eighteen-pin DIP package.

The serial interface is synthesized from three direct (non–multiplexed) I/O lines from the TCS slave processor.

The programmable counter that determines the set frequency has tremendous resolution. It yields adjustability in *2 KHz* steps. While this is probably more resolution than is required, no premium is paid for it.

One of the phase detectors uses an analog sample–and–hold technique to provide clean and precise feedback while in the lock state. The analog phase detector does not have sufficient corrective power to acquire a lock to a new frequency, as is required when the machine is turned on. To acquire lock, a digital phase detector is also included which is activated in addition to the analog detector when a new frequency is set. The digital unit 32 times faster than its analog counterpart, and generates a correction signal 100 times more powerful, so initial lock is quickly achieved. Once lock is acquired, the digital detector is disabled, and the TCSSP is notified. The detection of lock is often difficult with less–integrated PLLs.

The reference oscillator is designed for use with a 4 MHz crystal, but may be supplied by a 5V p–p square wave with the addition of two passive components. A CMOS buffer will produce this square wave from a 4 MHz TTL signal derived from the 80 MHz crystal oscillator.

The comparatively narrow range of frequencies (30–44 MHz) demanded of the synthesizer enables the design to incorporate wide frequency margins in the typical case. These margins, along with the negative feedback PLL architecture, will compensate for variations in component values and other factors, eliminating the need for hand–tuning of the loop filter and VCO components.

The optimization of the SAA1057 for consumer radio circuits presents surprisingly few "quirk" side effects. The main oddity is that the feedback frequency from the VCO must be divided by four to produce a nominal 20 MHz signal. This is because the FM input, which could take the 80 MHz directly, is not guaranteed to work below 70 MHz, preventing the computer from being run below 36 MHz. The AM input is rated for 512 KHz to 32 MHz, so the VCO, divided by four, will not be constrained by the input range of the circuit. Compensation for this division, along with other details, will be part of the TCS master routine for frequency selection and control.

The other small disadvantage of the chip is that the actual phase of the reference frequency is inaccessible from the pins. This effectively prevents the alignment of phase between the 80 MHz crystal oscillator and the VCO operating at 80 MHz. Such an alignment would be difficult to achieve even if the phase detectors were built from scratch, but it might have made it possible to switch between the fixed and variable clocks without interrupting a working computer.

Unfortunately, however, the IC is single–sourced from Signetics, Inc. If the part becomes unavailable, however, the B2CLK may be stuffed without it (and

possibly without the other synthesizer components), and used in the fixed 40 MHz mode.

Loop Filter

The loop filter components of a PLL are charged with converting the AC error-correction signal into a noise-free DC control level for the VCO. This is implemented with an analog integrator followed by a passive low-pass filter. The SAA1057 incorporates an op-amp so that the integrator is implemented with a total of two external components. The simplicity of this loop filter is feasible because the timing requirements for the output frequency in the B2CLK are quite forgiving compared with the analog environment for which the IC was designed. For example, high-order loop filters are common in high-performance communications gear, but there they act to minimize FM sidebands in equipment which must achieve lock rapidly. In contrast, the B2CLK frequency synthesizer, by virtue of its digital output, is highly tolerant of sidebands, as it is of static phase shift, and it is unusually stable because its lock time is on the order of 100 milliseconds.

MC1648 Voltage-Controlled Oscillator

The MC1648 is a 10KECL VCO suitable for generation of frequencies between 8 MHz and 220 MHz. Internal buffering and an automatic gain compensation stage provide a clean square wave at ECL levels across a wide frequency range. The resonant tank circuit is external to the VCO, so properties such as frequency range and control voltage range may be matched to its application.

Unfortunately, these external components include a varactor diode and a toroidal inductor, both of which are somewhat troublesome for manufacturing. The varactor has highly variable parameters (typically $\pm 10\%$), and is difficult to analyze. This variability seems to be the factor responsible for the inclusion of a tunable capacitor in many tank circuits. The B2CLK tank circuit has a control voltage range from 2V to 10V. The toroid is physically cumbersome, though small, and probably must be custom-wound. Also, the 10KECL MC1648 will force the addition of a small 10KECL power supply (a more negative VEE supply than that for 100KECL) to the B2CLK.

TCS Monitor and Control Functions

The MC68HC11 TCSSP includes powerful circuitry which can be exploited for observation and manipulation of the PLL. The built-in analog-to-digital converters (ADCs) and event-triggered timers are particularly useful for measuring the unusual mix of analog circuitry in the PLL.

TCS — Initialization and Debug Functions

A measure of the VCO control voltage is useful for several different tasks. Note, however, that the range of the control voltage will exceed the native range of the MC68HC11 ADCs, so an attenuator is placed between the loop filter and the TCSSP.

During the initial test of a stuffed B2CLK, a TCS master routine may run a test to measure the actual VCO control voltage versus frequency curve of the PLL as a whole. This would take into account the real values of each component, and yield the critical parameters for each B2CLK individually. In addition to providing information for the pass/fail evaluation on the manufacturing floor, this information would prove useful to anyone charged with reworking a failed board. A brief summary of the critical parameters may be stored in the EEPROM of the TCSSP itself, so that the information is sure to "stick" to the corresponding circuitry. This test may be run as many times as desired, and would take approximately one minute to complete.

A second parameter of interest is the time required to lock the set frequency. The MC68HC11 features a built-in, event-triggered timer that is well matched to this measurement, requiring no additional components. The TCSSP code starts the counter immediately after triggering the SAA1057 to seek a new frequency. A TCS master program could run a small routine to produce a table of time-to-lock versus set frequency. This table may be quickly digested to find minimum and maximum set frequencies which will lock within a specified time. These two numbers constitute a very high-level statement about the function of the entire PLL, with all of its analog circuitry and variable parameters. By further comparing this minimum-maximum range to a specification for the board, the TCS master would have enough information to issue a pass/fail declaration on the entire PLL without external circuitry or testing. These limits will be stored in MC68HC11 EEPROM as capacity allows.

TCS — Routine Functions

The second category of TCSSP-PLL interaction is functions that will commonly be encountered by users of an established, production Butterfly II machine. Here, the TCSSP simply sets the desired frequency and checks for proper operation.

Early in the boot sequence of the machine, the TCS master will assemble a command string to set the frequency and other parameters of the SAA1057. Presumably, the frequency information will come from a file of default system settings or from a manual entry. The command string will be passed to the B2CLK TCSSP and inserted into the SAA1057 serial port, and the frequency-seek procedure will begin.

After a short time, the TCSSP will observe the transition to locked state. It will report this information to the TCS master, which may then proceed with

the boot sequence. The TCSSP may also return time-to-lock and the magnitude of the locked control voltage which the TCS master could compare to a historical table of PLL performance.

9.4.6 Clock Selection and Buffering

The master timing source for both Rclk and Sclk is derived either from a fixed crystal oscillator that supplies 80 MHz, or from a programmable frequency synthesizer that can produce a range of frequencies including 80 MHz. The selection is controlled by the TCS slave processor. The selected source is then divided by two to produce a nominal 40 MHz signal (with 50% duty cycle). Although the B2CLK hardware supports changing the clock source while a machine is powered up, the TCS master processor should warn the user that the resulting effective phase shift will probably violate timing assumptions made by the hardware.

Active components in the clock lines are notorious for degrading the delicate timing of a machine such as the Butterfly II computer, but the clock selection and buffering section is an exception. The TTL crystal oscillator is actively translated to 100KECL, and then a multiplexer selects the crystal or phase-locked oscillator to produce the master 80 MHz source. All of this occurs before the divide-by-two stage, however, so degradation of the duty cycle is unimportant (as long as a minimum pulse width to the divide-by-two is ensured). Further, all of the circuitry in this section is an example of common ambiguity, so *it does not contribute to the degradation of the clock signal in any way*. This "free" logic allows the TCS to control the selection of clock, as the philosophy of the TCS dictates it should. (Here, the TCS philosophy refers to the desire to control virtually all of the machine's configuration with the TCS.) Note that the other side of the clock configuration coin, the selection of clock phase, cannot reasonably be made into common ambiguity and therefore must violate the TCS philosophy.

9.4.7 Timing Interface Circuitry

The timing of the B2CLK is designed in two parts. The timing interface circuitry buffers or latches each signal directly into the clock cable connectors. The routing for the timing interface circuitry has been balanced by hand to minimize skew between output channels. The rest of the circuitry is considered to have more "local" timing requirements: it need only satisfy the input requirements of the timing interface section.

Termination Jumper Blocks

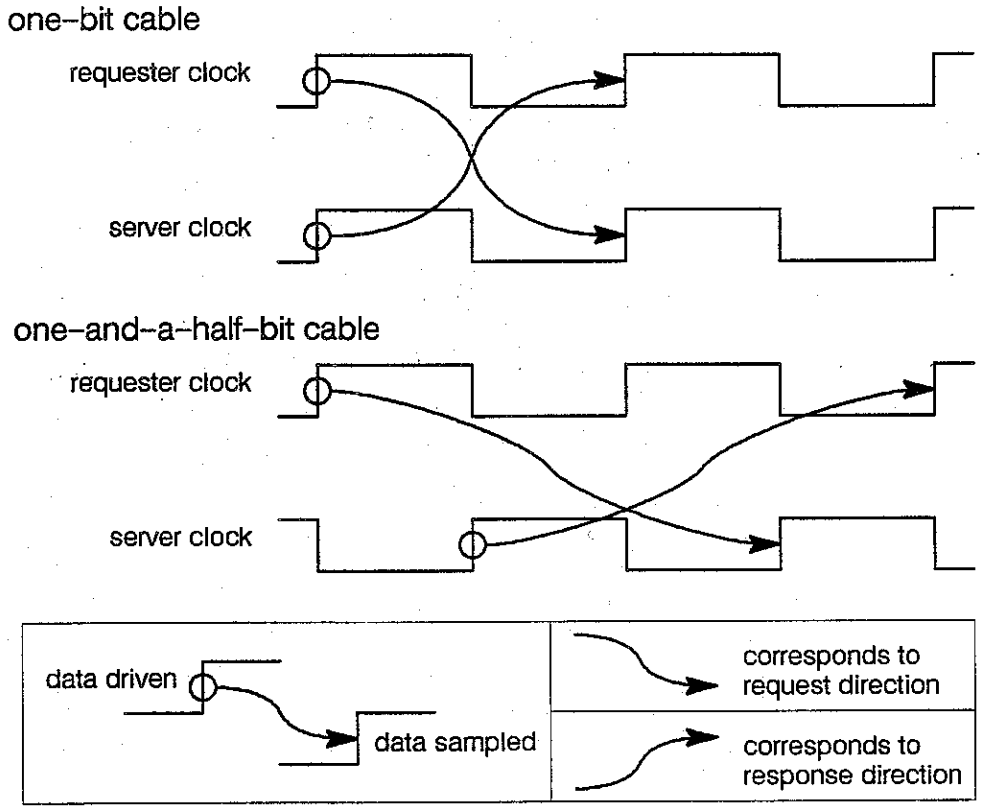
If an ECLiPS part is used for the clock buffer, each of its outputs must be terminated to achieve low skew between output channels. Between one and eight clock cables will be connected to the B2CLK, each with its own output driver

from the part. To ensure that all outputs are terminated, then, each of the output channels must be "artificially" terminated when no clock cable is connected. Eight jumper blocks on the B2CLK optionally terminate the clocks on the channel to which each is attached. In order to keep the transmission line stubs very short when the termination is disconnected, an inexpensive relay must not be used, and a simple electronic switch would add considerable noise or timing variability. It is infeasible, then, for the TCS to control this local termination. The position of each jumper block may be read by the TCS slave processor, so the TCS master can check the settings against a file of standard configurations.

Phase Jumper Blocks

Two kinds of switch data cables may be used in two-column Butterfly II machines: one-bit and one-and-a-half bit electrical lengths. When two SGAs are run with their clocks in phase (matched), then data is sent on a rising edge and received on the next, so the path between them is said to have "*one bit in the cable*". If the SGAs are too far apart for that to be feasible, they are run with their clocks 180° out of phase, and the sampling edge is placed an extra half-cycle away from the originating edge, placing a total of "*one-and-a-half-bits*" in the cable. This timing is shown in Figure 9-35.

Figure 9-35 Definition of number of bits in the cable.



The range of physical cable lengths for both types of cable is highly dependent on system timing variables, including the AC parameters of the SGAs and other logic, board wire routing, and switch frequency. At 40 MHz and using the first version of SGA with ECLiPS buffers, the maximum length for a one–bit cable is currently about 2½ feet, and the *minimum* for a one–and–a–half bit cable is over 9 feet. In practice, the cables will probably be as short as possible, within the range indicated by the timing model and still compatible with packaging. Cable length is also limited by attenuation, but an experiment performed on ten–foot samples of both brands of cable currently being explored indicates that attenuation will not be an obstacle for cable lengths likely to be used. No phase shift resolution smaller than 180° is useful because it becomes asymmetric with respect to data direction. The 180° shift is also easy to implement—the differential pair carrying the clock signal is simply crossed.

The phase of R and S clocks to each clock cable will depend on the configuration (size) of the machine and on the timing performance of its components. Four–channel wide shunt blocks on the B2CLK card control the phase of the clocks for each channel, with the exception of phases which are invariant across all configurations.

The functional layout of the jumper blocks is shown in Figure 9–36, and example phase jumper positions appear in Figure 9–37. Note that a total of four clock wires arrive at each clock cable connector: two wires comprise a differential pair for Rclk; another pair handles Sclk. The discussion is geared for sixty–four slot machines, but is applicable to smaller machines as well.

Figure 9–36 **Phase and termination jumper blocks.**

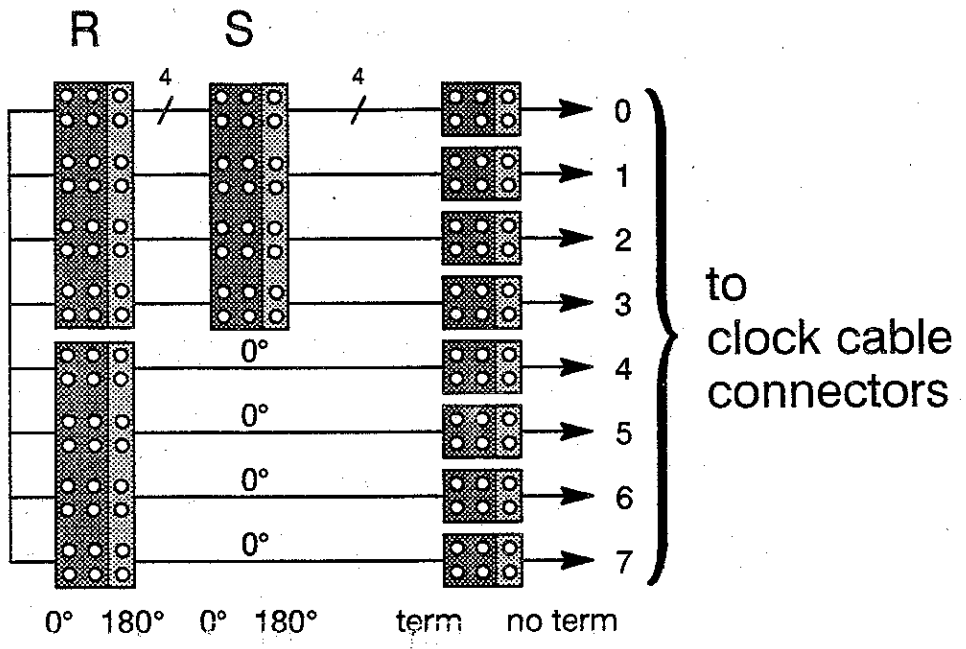


Figure 9–37 **Phase jumper blocks — examples.**

Example A R°/S°	Example B R°/S°	Example C R°/S°
0/0	0/180	0/0
0/0	0/180	0/0
0/0	0/180	0/0
0/0	0/180	0/0
180/180	0/180	0/0
180/180	0/180	0/0
180/180	0/180	0/0
180/180	0/180	0/0

Example "A" above is the one most likely to appear in a production sixty–four slot machine. The "top" half of the machine is interconnected with one–bit

cables, as is the "bottom" half. Cables between the halves are too long to be one-bit, so they are run as one-and-a-half bits.

Example "B" is a contingency plan which may be used in early machines. Here, the timing of one-bit cables is such that they must be shorter than the longest length required to build a thirty-two slot machine. The scheme used in Example A becomes impossible in that case, so instead the longer one-and-a-half-bit cables are used throughout the machine.

Example "C" is a more radical contingency plan. If some part of the system fails to function near 40 MHz, and thereby forces the switch frequency down, it may become possible to cable an entire sixty-four slot machine with one-bit cables. While it may sound odd to optimize cable length for such an unlikely case, it proves sensible given that one-and-a-half-bit cables designed for high speed will not work at much slower speeds. Also, slow-speed one-and-a-half-bit cables would be extremely long and quite difficult to package.

Note that three large phase jumper blocks are required to support all three of these scenarios. As hardware becomes proven, some of the contingency plans may be discarded, allowing more of the phase options to be hard-wired. The selected jumper block features very short internal electrical lengths which is important for preserving noise immunity.

The TCS is capable of reading the position of every jumper block on the B2CLK. The TCS Master will therefore be able to read the whole configuration of the B2CLK and verify the settings compared to a "standard configurations" file.

NOTE

PHASING AND FAIRNESS

If part of the machine is operated on a different clock phase than the rest of the machine, the two parts will have different access times to reach a given resource (such as a memory module). If function boards in both parts simultaneously begin accesses to one resource, a function board in the part of the machine with faster access will arrive first, and establish a switch connection if the resource is free. The access attempt from function board(s) in the part with slower access will then lose. If the random strategy is used for switch transmissions, this unfairness effect will be slight, perhaps unnoticeable. If the slotted strategy is used, however, the unfairness can be pronounced. Therefore, if the slotted strategy is to be used at all, it may be desirable to cable the machine entirely with one size of switch data cables and run it entirely on one clock phase.

9.4.8 Generation of 65 Milliseconds and Hold

Sixty–five milliseconds is a periodic signal, synchronized to Rclk, that has a period of exactly 2^{16} (65,536) microseconds, and a pulse width of one switch clock period. The production of this signal is complicated somewhat by the frequency synthesizer; a programmable divider is required to compensate for the programmability of the synthesizer, since its output is divided to produce the sixty–five millisecond period. This programmable divider is functionally identical to the prescaler for the same purpose in the SIGA—in both cases it is loaded with a value related to the actual switch frequency. The five–bit setting is controlled by the TCSSP, and will accept the same value as the SIGA prescaler.

The **hold** signal is also periodic and has a pulse width of one switch tick, but it is synchronized to both the R and S clocks, producing Rhold and Shold. Its period is roughly equal to the maximum delay between the request of the SIGA requester and its retry. The SIGA retry delay is the sum of request–to–reject time plus the maximum timeout period of the SIGA retry timer. The SIGA retry timing parameters are programmable over a wide range, so the hold generator also must be configurable. The maximum request–to–reject time is on the order of twenty switch ticks. The SIGA random–retry timer is a twelve–bit programmable counter that decrements at the full switch frequency. In light of this, the hold timer is implemented in two stages: the switch clock signal is prescaled by a fixed $\div 16$ counter that in turn feeds an eight–bit programmable $\div n$ counter. This provides a linear range equal to that of the SIGA retry counter, with the resolution of a fast retry. The $\div n$ counter, like the 1–microsecond prescaler, is controlled by externally latched MC68HC11 parallel I/O port bits.

Note that the hold period does not use the programmable prescaler, so its period will vary with switch frequency. In the random retry mode, the SIGA retry period is related only to the number of switch ticks, so the B2CLK counter is correctly matched. If the SIGA uses slotted retry, however, its period *is* compensated for frequency by the SIGA programmable prescaler, so the hold period setting must take the switch frequency into account.

Both sixty–five milliseconds and hold have extremely long periods in terms of the capabilities of 100KECL technology, though they only remain asserted for one switch clock period each cycle. The circuitry that produces the periods for these signals takes advantage of the slow signals to substitute FAST logic for expensive 100KECL parts. Area, power, cost and parts count is reduced, particularly for the programmable prescaler and programmable hold period generator, which must be controlled by the CMOS/TTL levels of the MC68HC11.

9.4.9 Net Time Selection and Buffering

One of the eight incoming net time signals from the clock cables is selected, buffered and fanned back to all of the cables. The selection is performed by the TCSSP on the B2CLK, which is also able to monitor the selected output. The monitor function enables the master to determine, through trial and error, which inputs are active without relying on *a priori* information. No synchronization is performed on any part of the net time distribution system.

The function and circuitry on the B2CLK which executes selection and buffering of net time is identical to the corresponding section of the switch server card (B2SS). All net time circuitry is implemented in 100KECL. The period of the selected net-time-switch may be monitored to high resolution by the TCS slave processor, to verify a net time source. None of the timing parameters are critical.

9.4.10 TCS Serial Lines

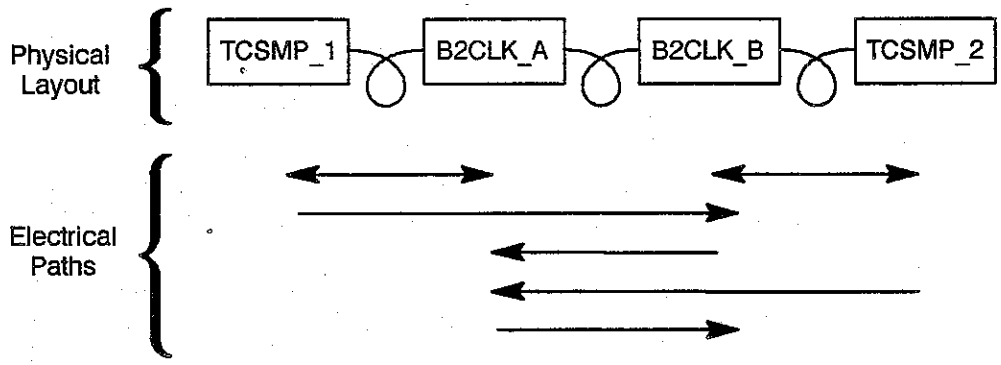
The input and output multidrop serial lines that implement the TCS master-slave interface are buffered by the B2CLK, and are carried by the clock cables. In addition to the buffering function, the on-board MC68HC11 TCSSP can disable any combination of the serial inputs, so that a defective switch card or cable may be disabled and thereby prevented from interfering with reception from functional switch cards. No synchronization of the TCS serial lines is performed on the B2CLK card.

Note that CMOS signals are sent through long lines without proper termination, but this situation has already been deemed acceptable by the TCS interface designers. Termination is not required because the TCS serial interface has too large a period (10 microseconds) and too slow an edge (200 nanoseconds) for transmission line effects to significantly affect signal quality.

The B2CLK is the point of connection between redundancy in the switch/clock distribution system and redundancy in the Test and Control System (TCS) master processors. There are four possible combinations of redundancy: single or redundant switch/clock distribution, times single or redundant TCS masters. The B2CLK will accommodate all four configurations in a straightforward manner that is also simple for the B2CLK local TCS slave processor (TCSSP) to discover. The circuitry for this function is given in abstract form below, in Figure 9–38 and Figure 9–39.

One further complication is the possibility of two different cabling schemes to cope with redundancy. The first scheme is already incorporated in the TCS master interface board (B2TCS), while the second promises improved behavior in the event of a failure in a partially-redundant Butterfly II machine. The first scheme places a cable between the clock cards to transfer data between the B2CLK and the "more distant" TCS master, as indicated in Figure 9–38.

Figure 9–38 First B2CLK TCS cabling scheme.



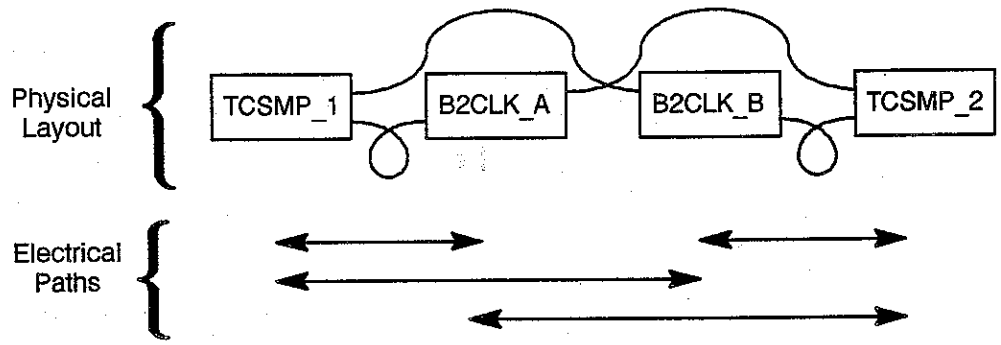
Each B2CLK has a clear, bidirectional pathway to the "nearest" TCS master, and can receive data directly from the "more distant" one. The transmission path to the more distant TCS master is indirect, however — it is gated by logic on the intermediate clock card. With the above cable system, this gating is necessary to allow the one return path to the master to be driven by the "A" TCS serial interface or the "B" interface, but not both. The TCS serial interface must be driven by only one slave at a time in order to function — there is no way to ensure that one interface is silent while the other is in use, so the master must lock out one of the two interfaces.

The shortcomings of this scheme become apparent when one considers a machine with one TCS master and two switch/clock distribution systems. (Omit TCS master 2 in Figure 9–38.)

- If the "central" B2CLK (B2CLK_A) is to be replaced while the machine is running, the TCS serial interface must be compromised for a short time until the cable from the master can be connected to the functioning clock card (B2CLK_B).
- If the "central" clock card (B2CLK_A) should fail, its gating of the signals from the "more distant" clock card (B2CLK_B) may also fail, preventing the master from communicating with the machine at all. In this way, the usefulness of a redundant switch/clock system without a redundant master is greatly diminished.

The second cabling scheme, illustrated in Figure 9–39, eliminates both of the limitations of its predecessor while adding very little complexity. Here, both TCS master processors have a cable to each clock card, eliminating intermediate logic along any path. Some complexity is added to the TCS master circuitry in this case, in the form of an extra connector and gating logic to select which cable is attended, but the clock card is free to omit the circuitry that performs the same function under the first scheme. The B2CLK design has been made compatible with both cabling schemes, so the complexity is retained, but it is very small.

Figure 9–39 Improved B2CLK TCS cabling scheme.



An abstract schematic of the clock card is given in Figure 9–40, highlighting the logic for TCS serial interface support. Note that all logic is asserted–true in the diagram. Real assertion levels, along with other details, are left to the schematics. All AND gates are controlled by the local TCSS to select which lines are enabled.

Figure 9–40 Abstract of B2CLK TCS serial line circuitry.

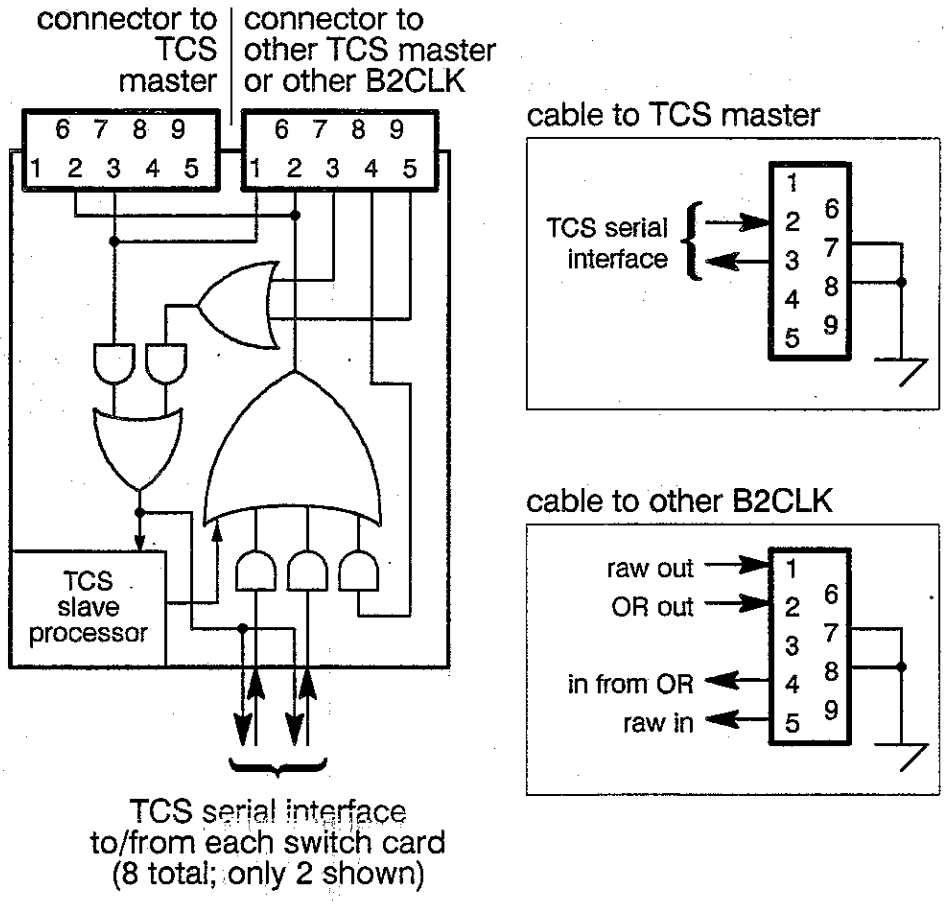
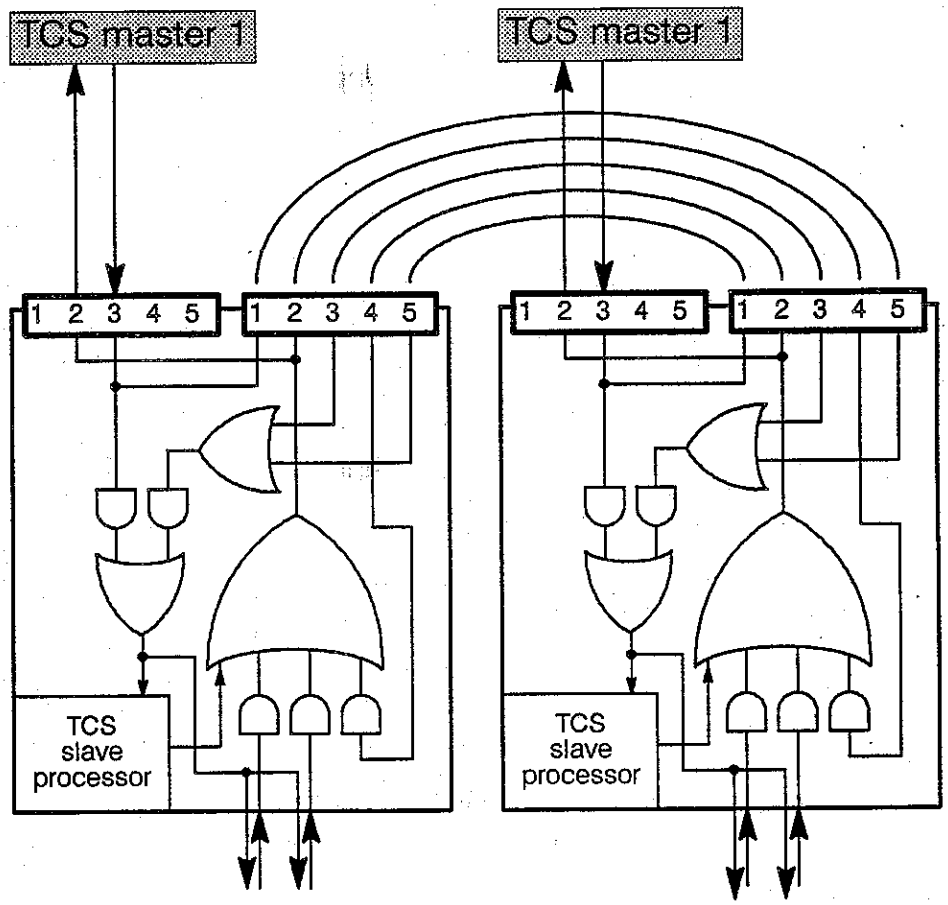


Figure 9–41 describes the interconnection of a fully redundant machine using early–revision TCS master interface cards (B2TCSs). Note that the cable between clock cards is a simple one–to–five cable that may be implemented as a mass–terminated ribbon cable assembly.

Figure 9–41 **Interconnection of redundant B2CLKs with early revision B2TCS cards.**



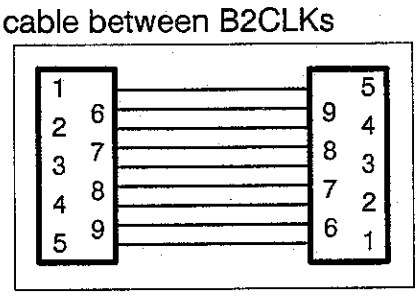
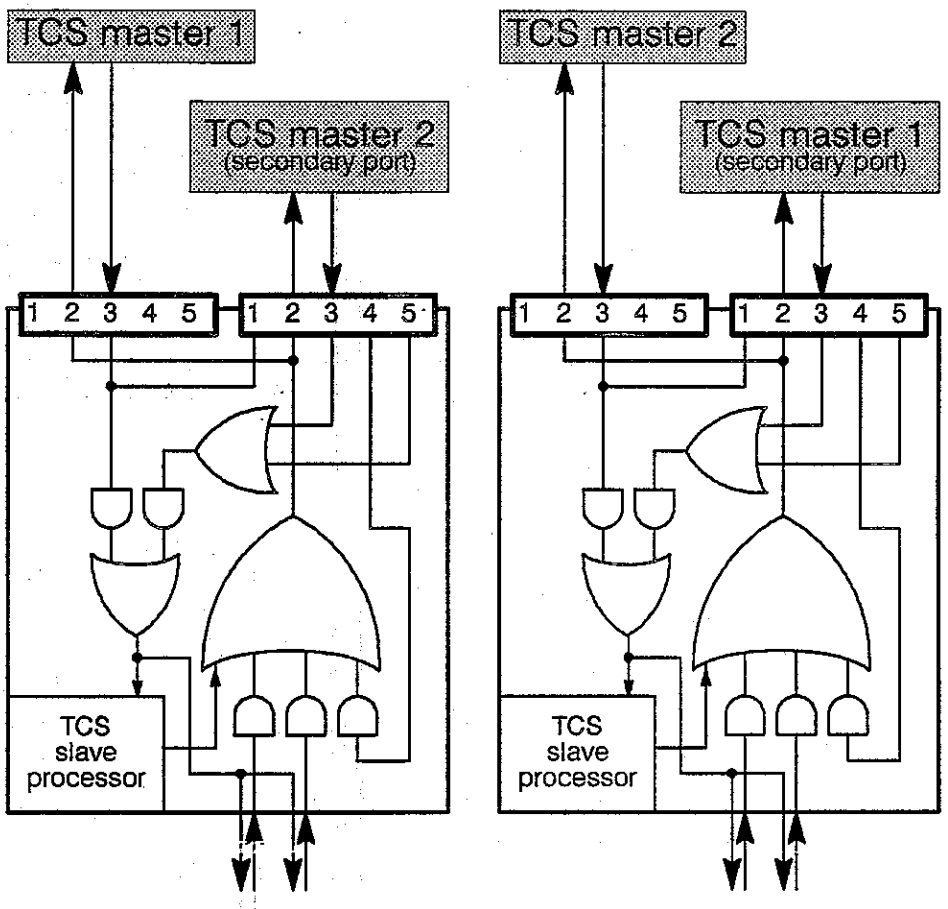


Figure 9–42 shows the same fully redundant system built with later revision B2TCS boards.

Figure 9–42 **Interconnection of redundant B2CLKs with later revision B2TCS cards.**



9.4.11 Power

The power requirements of the B2CLK are unusual because it combines analog and four kinds of digital circuitry. A list of voltages would include +8V for the SAA1057 supply, +5V for the FAST logic and ECL/TTL translators, TCS +5V for the TCS circuitry, –2.0V ECL termination voltage, –4.5V for the 100KECL logic, and –5.2V supplying the 10KECL. Each of these supplies is less than two or three amps.

A pair of four–conductor cables supply +24V, –24V, TCS +5V and ground to the B2CLK. Both cables are functionally identical, but receive their power from two different midplanes, so that if either midplane is powered down, the B2CLK may draw power from another, and the machine may continue to function. Clearly, in an eight–slot machine, there is no need for the second cable except as a "hook" for future expansion. The DC–DC converters which constitute the first level of the B2CLK power supply section use the ±24V to produce all of the on–board supply voltages. The TCS +5V powers the TCSSP circuitry, independent of the DC–DC converters.

A triple-output DC-DC converter supplies +5V, +12V and -12V, while a second, single-output converter provides -4.5V. A pair of 1.5A adjustable linear regulators produce -5.2V (from the -12V supply) and -2.0V (from the -4.5V supply). A low-power adjustable positive linear regulator produces +8V from the +12V output. All supplies except the +8V regulator are trimmed with potentiometers and may be margined using TCS commands. The +8V regulator should be viewed as a power filter with a voltage drop, since its function is to provide a clean supply to the SAA1057, rather than to produce an exact voltage.

9.4.12 Routing and Printed Circuit Board Considerations

Critical Areas Routed by Hand

Much of the routing of the B2CLK is critical, especially that in the timing interface circuitry section. All of the ECL has been placed and routed by hand, and the clock fanout traces, 32 lines of equal length, have been matched through manual successive approximation. It is unfortunate that the precision required from the clock card forces the layout to be so labor-intensive, and further, it forbids certain types of rework which would change the wire delay of critical nets.

Overlay 100113 and 100114 with ECLiPS 100E111

The first revision of the B2CLK features an *overlay* for the main clock buffer, a set of holes for both of two possible parts that might be used for the buffer function. The center of both possible packages are roughly coincident, hence the label "overlay"; the pads for one are *laid over* those of the other. This pattern supports either driver so that the decision is made at stuffing time — no revision of the printed circuit board is required. These patterns were placed on the B2CLK and B2SR/S (switch cards) where the ECLiPS 100E111 was desired. The overlay allowed machines to be built if the sole-sourced ECLiPS part were to become unavailable. Later revisions of the Butterfly II boards will omit the overlay since system timing has become dependent enough upon the unequalled performance of ECLiPS that standard 100KECL buffers (100113 on the B2CLK; 100114 on the B2SR/S) cannot be considered as alternate parts.

The overlay does not contribute to skew under either stuffing condition, and conforms to design rules published by Fairchild for routing ECL test fixtures. The combined pattern consumes little more area than the 100113 or 100114, and is no more aggressive than the Butterfly II midplane with respect to pad size, routing or drilling.

Controlled Impedance Printed Circuit Board

The 100KECL family has one of the fastest edge rates and shortest typical clock periods of any commercial logic family. Every ECL trace on the B2CLK, as well as the ECL signals passed through cables, are treated as full-blown transmission lines, complete with characteristic impedance-matched parallel termination. This level of care is required to ensure reliable operation with this fast and noise-sensitive family. Noise due to transmission line faults would quickly overcome all care if the printed circuit board trace characteristic impedance were not carefully controlled. Signals going onto or coming from cables would be particularly affected. For these reasons, the B2CLK has two controlled impedance signal layers, specified to be 50Ω ±10%.

9.4.13 Summary of TCS Sensors and Effectors

All of the high-level TCS monitor and control functions are discussed elsewhere in this chapter, and are summarized in Figure 9–43 for convenience. Note that "ADC port" refers to an analog-to-digital converter port, eight of which are supplied in the 68HC11 package.

Figure 9-43 B2CLK TCS sensors and effectors.

Sensor Functions

VCO control voltage	1 ADC port
48V and regulated voltages	8 ADC ports
B2CLK board temperature	1 ADC port
Each clock cable's connection to A or B connector on the midplane	8 parallel bits
PLL lock condition (and trigger internal time-to-lock)	1 parallel bit
Phase jumper block positions	3 parallel bits
Termination jumper block positions	8 parallel bits
Net time monitor	1 parallel bit
Monitor 65 milliseconds and hold	2 parallel bits
Corresponding TCS master is A or B master	1 parallel bit

Effector Functions

Net time select	3 parallel bits
Select fixed/variable clock source	1 parallel bit
Set SAA1057 frequency and configuration	3 direct bits
TCS slave-to-master serial interface disable	9 parallel bits
Sixty-five milliseconds prescaler value	5 parallel bits
Hold period divider value	8 parallel bits
Power supply enable	1 parallel bit
Voltage margin control (adjusts all levels together)	3 parallel bits
ADC port external analog multiplexer	2 parallel bits
Attend (listen to) corresponding or other TCS Master	1 parallel bit
TCS signal light	1 parallel bit

The eight ADC ports provided on the MC68HC11 are expanded to the required ten with a single external 4:1 analog multiplexer. Input and output parallel I/O also requires external circuitry as it has on other Butterfly II cards.

11
TC/FPV
Function Board with Processor and VMEbus Interface

11.1 Introduction

This is a functional specification of the TC2000 processor and VMEbus interface function board (TC/FPV). This specification assumes familiarity with the VMEbus specification, the T-bus specification, the TC2000 architecture, and the Motorola 88000 documentation.

The TC/FPV is the first in a family of function boards for the TC2000 machine. It provides a combination of processing, memory and I/O resources. The CPU is based on the Motorola 88100/88200 chip set. The memory array is optionally 4 or 16 megabytes. The I/O channel provides a path between the TC2000 machine and industry standard VMEbus devices.

This description of the TC/FPV function board covers the following four configurations:

TC/FPV4	4 megabytes, 2 cache chips
TC/FPV4-1	4 megabytes, 3 cache chips
TC/FPV16	16 megabytes, 2 cache chips
TC/FPV16-1	16 megabytes, 3 cache chips

For a configuration with only 4 megabytes of memory or only two cache chips, reference to the additional 12 megabytes or to the third cache chip (code CMMU #2) should be ignored.

11.2 Major Functional Blocks

Figure 11-1 is a block diagram of the TC/FPV, and Figure 11-2 shows its physical layout. Board areas implementing the major functional blocks of Figure 11-1 are shown shaded in Figure 11-2.

Figure 11-1 TC/FPV block diagram.

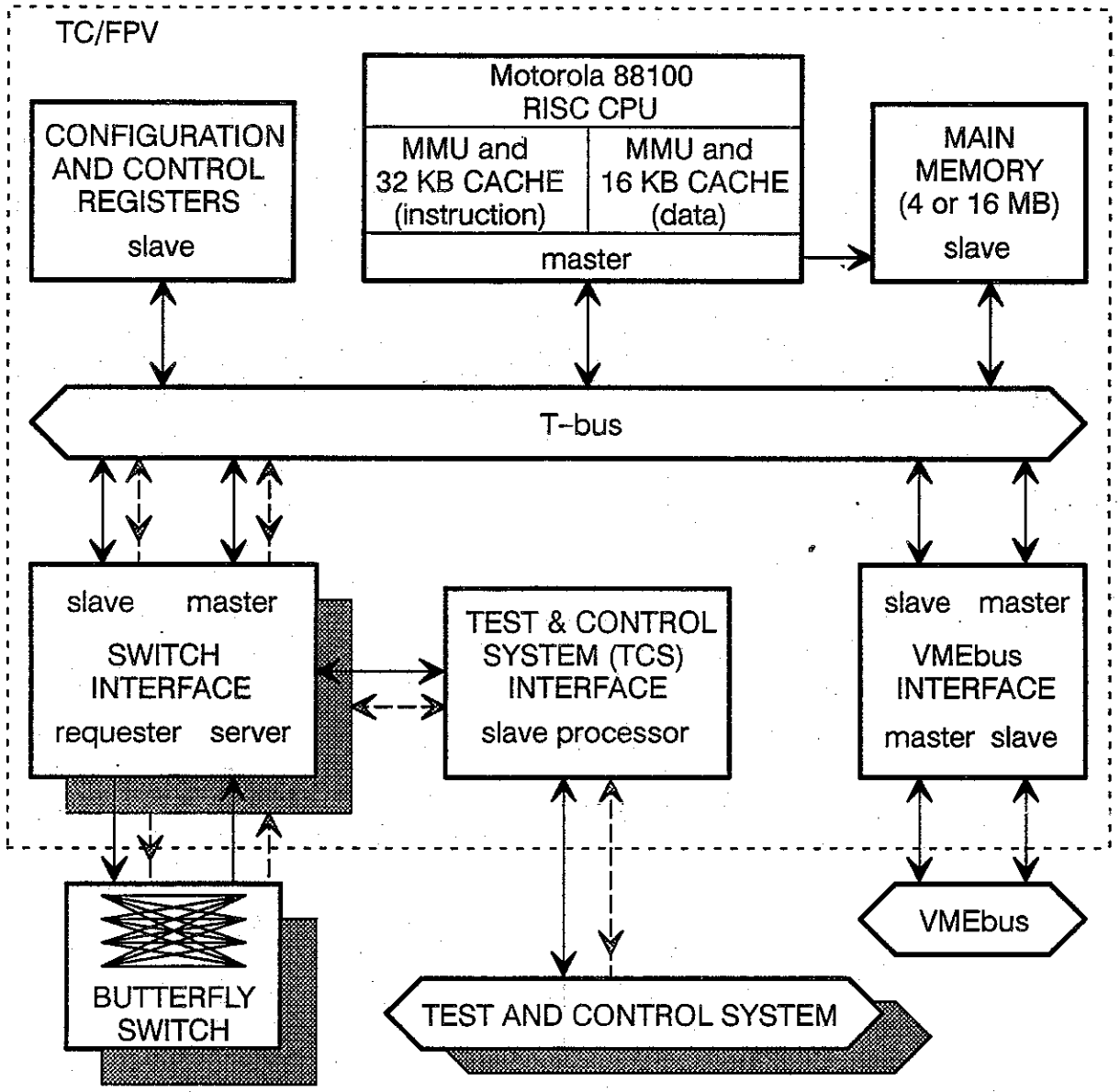
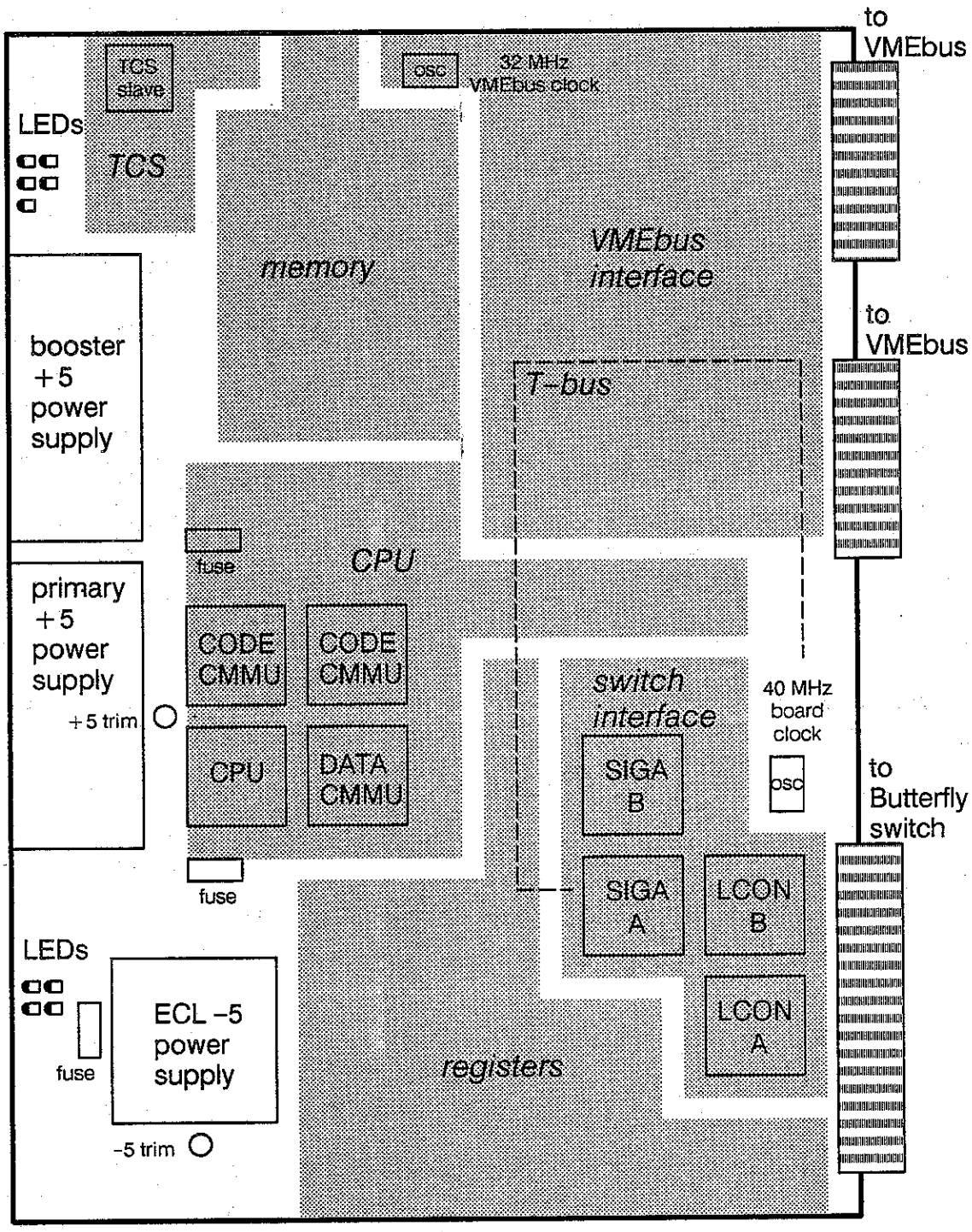


Figure 11-2 TC/FPV physical layout.



viewed from component side

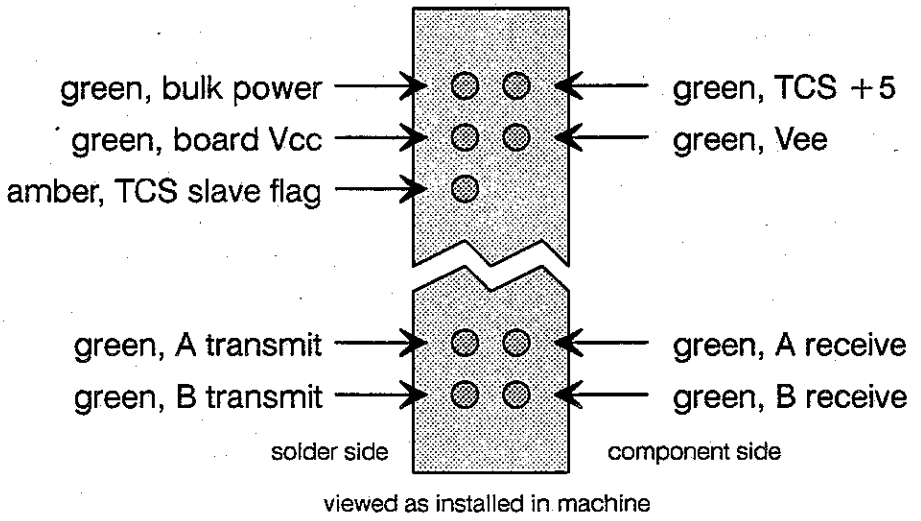
(T-bus circuitry is in the dashed line area)

The TC/FPV consists of six modules: the CPU, the memory, the switch interface, the Test and Control System (TCS) interface, the VMEbus interface, and TC/FPV configuration and control registers. These modules communicate with each other across the T-bus. The T-bus is a synchronous, multiplexed address/data bus that supports a variety of transactions. The T-bus transaction protocols are designed to avoid deadlock situations while obtaining good performance in communication among devices on the bus and throughout the system. The operation of the T-bus is documented in the *T-bus Specification*. Dedicated paths between the CPU and the memory, and between the TCS and the Switch Interface Gate Array (SIGA), have been added to enhance performance and simplify the design.

This TC/FPV specification describes each module. However, the description of the switch interface, in section 11.6, emphasizes aspects important to TC/FPV functions and omits aspects common to the design of any TC2000 function board.

The TC/FPV circuit board is 15 × 20 inches and has 10 conductive layers. There are nine LED indicators mounted along the outer edge of the board. Four green indicators at the upper end of the board show power supply status. A fifth (amber) indicator is used by the Test and Control System to show the status of the board (inactive, loading software, running, etc.). Four green indicators at the lower end of the board show switch interface activity. The only manual controls on the TC/FPV are voltage trim pots and jumpers, described in section 11.18.

Figure 11-3 TC/FPV indicators.



11.3 CPU

The CPU includes a Motorola 88100 microprocessor, three Motorola 88200 Cache/MMU chips (one data CMMU and two instruction (code) CMMUs), and an interface to the T-bus.

The CPU interface includes a T-bus master finite state machine that generates single cycle read and write requests and cache burst read and write requests on the T-bus. The CPU interface also contains a mapping RAM and other circuitry.

11.3.1 CPU Fast Path to Memory

Because the operation most frequently performed by the CPU is fetching instructions and data from local memory, the CPU also has a specialized **fast path** that bypasses the T-bus and allows the CPU to communicate directly with the memory to optimize performance. This path has circuitry to detect when the CPU will be given ownership of the T-bus in advance of the T-bus arbitration circuitry, and a dedicated address path. The response to a memory fetch is returned over the T-bus, whether the access used the fast path or not. Both instruction and data fetches use the fast path. The fast path is never used if the page is interleaved. (Only local accesses may use the fast path, and only remote accesses may use the interleaver.)

A reference from the CPU will use the fast path only if all of the following conditions are met:

— The operation is a read (either code or data), not a write.

— The Machine Configuration register "fast path enable" bit is 1.

— The "fast path disable" bit of the CPU Mapping RAM entry selected by the current reference is 0.

— The "local" bit of the CPU Mapping RAM entry selected by the current reference is 1.

For further details of the fast path and its interaction with other CPU interface mechanisms, please refer to the CPU Mapping RAM description in section 11.17.

11.3.2 Local and Remote Accesses

The TC/FPV uses three criteria to decide whether references issued by the CPU should be steered to local or remote resources.

1. CPU references to interleaved memory are always controlled by the *path* bits in the TC/FPV's Process Configuration register (PCR). The intent is that if interleaving is in use, software will set these PCR bits to specify use of the Butterfly switch, so interleaved references will always go out over the switch, even if the target of the reference is on the local board. Section 11.3.6 discusses detecting that a reference is to interleaved memory, and consequent generation and use of the signal T_INTERLEAVED.
2. If not interleaved, CPU references are directed to local memory if the *local* bit of the CPU Mapping RAM (CMR) entry used is "1". When the CMR *local* bit is "1", the TC/FPV forces the T-bus T_PATH control bits to "11", directing the reference to a local T-bus slave (memory, configuration and control registers, or VMEbus interface).
3. Otherwise, the default choice is specified by the *path* bits in the TC/FPV's Process Configuration register.

11.3.3 Code and Page Tables on Remote Nodes

The CPU can execute code stored on a remote node. Fetching each instruction over the switch, however, slows execution appreciably, so code is almost always stored locally. With the instruction cache turned on, the remote access cost may be amortized over many executions of the code, such as iterations of a loop. In this case, or for very short sections of code, execution across the switch may be appropriate. The same considerations apply to page tables. Access will be slower for remote page tables, but the penalty is small if the descriptors stay in the address translation cache.

Three complications arise when code or page tables are kept in remote memory, all concerning the TC2000 locking protocol. One concerns the effect of encountering memory locked by someone else, and the others concern a constraint on actions while you hold a lock yourself.

1. If the remote memory is locked (by its local CPU or VMEbus interface), then instruction or page descriptor fetches could be delayed, slowing execution. This problem is eliminated by the hardware. All instruction fetches automatically bypass the TC2000 locking protocol. Similarly, all page descriptor fetches made by the CMMU automatically bypass locks.

   (An analogous case arises when the instruction or page descriptor is in local memory, but the local memory is temporarily locked by another CPU (or by a VMEbus interface). Again, such fetches are automatically made bypassed, so execution is not delayed.)

NOTE

IN-USE SWITCH PORT IS NOT BYPASSED
If the remote memory is locked by a CPU or VMEbus interface on a *different* board, then not only is the memory locked but also the switch path to the memory is locked. The bypass mechanism does not help in this case, because it cannot get around the switch port in use by the locked switch path.

2. If the local CPU is making a locked reference to a remote memory module, then instruction or page descriptor fetches to either *that* remote memory or to the local memory are OK. The locked remote memory can be referenced because the switch path to it is established, and the local memory can be referenced (for instructions or page descriptors) because of the hardware automatic bypassing described above. (Without the automatic bypassing, the local reference would claim to be "maintaining" a lock, and the local memory would not accept the access because this CPU doesn't have a lock "open" to it.)

NOTE

Note that during a locked transaction to a remote memory, *data* accesses to local memory must be mapped bypassed in the CMR, or they will be refused as described above.

3. If the local CPU holds a lock on a remote memory module, the switch path can't be temporarily disrupted to fetch an instruction or page descriptor from a *different remote memory module*. This is enforced by the local SIGA, and is a constraint on the design and implementation of software.

11.3.4 The XMEM Instruction

The 88100 instruction set includes the "exchange register with memory" (XMEM) instruction to support atomic operations. For a description of this instruction, refer to the *88100 User's Manual*. When the CPU executes the atomic instruction XMEM, the $\overline{\text{DLOCK}}$ output pin is asserted. The TC/FPV hardware maintains the atomicity presumed by the CPU by opening and maintaining a T-bus lock when $\overline{\text{DLOCK}}$ is asserted. The T-bus locking protocol ensures atomicity for both local and remote references. Consistent with the T-bus protocol, the XMEM operation is *not atomic* if the reference is to a bypassed page. (The atomicity is lost if the target location of the XMEM is referenced bypassed *either* by the CPU that executes the XMEM *or* by another CPU that writes to the location.) The T-bus FREE_LOCKS operation is executed when the $\overline{\text{DLOCK}}$ pin goes from asserted to de-asserted.

NOTE

RESTRICTION OF USE
XMEM instructions cannot be included in sequences that use the TC2000 locking protocol, unless they access the memory module that is locked, or access bypassed memory (in which case the reference is not atomic). If this restriction is violated, the instruction will terminate in a bus error. The locking protocol is controlled by the Augmentation register, described in detail later.

11.3.5 Interrupts to the CPU

The TC/FPV CPU receives interrupts from three kinds of sources: other processors, VMEbus devices, and on-board timers. The TC/FPV registers used by the CPU to control and service interrupts are described in detail in the register summary section. There are three types of interrupts:

- Interrupts from processors (typically other processors, but the CPU may generate an interrupt to itself if desired)
  - Maskable interprocessor interrupt
  - Non-maskable interprocessor interrupt
- Interrupts from VMEbus devices — seven levels are distinguished
- Interrupts from on-board timers
  - Interrupts Disabled Timer
  - Time Of Next Interrupt (TONI) registers, two programmable timers associated with the Real Time Clock

The 88100 microprocessor has only one interrupt level, so all interrupts are OR'ed together to generate the interrupt signal to the processor chip. When an interrupt is detected, the interrupt handler reads the Interrupt Source register to find out which type of interrupt has occurred. The Interrupt Source register has a field for each interrupt type.

A TC2000 processor can generate a remote interrupt request by setting a bit in the Interprocessor Interrupt register at the destination node. The Interprocessor Interrupt register for each processor node is accessible to every processor in the machine. The operating system controls access to the interprocessor interrupt by giving the page on which it resides appropriate protection attributes.

The destination processor further controls remote interrupts with a bit in the Interrupt Enable Mask register. If the Interrupt Enable Mask register bit for the interprocessor interrupt is zero, then the interrupt request will not be generated. Attempts to set the Interprocessor Interrupt register when it is masked do change the register's contents, but no interrupt occurs. The interrupt system is described in detail in section 11.13.

A register for generating a non-maskable interrupt resides in a supervisor page of configuration and control registers. The non-maskable interrupt will interrupt the CPU unless the CPU has disabled its entire interrupt system by manipulating its internal processor status register.

In addition to interprocessor interrupt requests, VMEbus interrupt requests can interrupt the TC/FPV processor. Like interprocessor interrupts, VMEbus interrupts to the processor are subject to the Interrupt Enable Mask. If the mask bit for a VMEbus level is zero, then the interrupt to the processor will not be generated, even if the VMEbus interrupt request signal at that level is asserted. When a VMEbus interrupt occurs, the level of the highest priority pending interrupt is indicated by a field in the Interrupt Source register. The TC/FPV can also generate interrupts onto the VMEbus, as described later.

Finally, interrupts are generated to the TC/FPV processor, and to each TC2000 processor, by the expiration of the Interrupts Disabled Timer and by the Time Of Next Interrupt register. These, and other timers, are discussed in section 11.12.

11.3.6 Interleave Decision RAM

Like all T-bus masters, the CPU must drive the T_INTERLEAVED bit whenever it initiates a T-bus access. The value of the T_INTERLEAVED bit indicates whether the CPU wishes to access interleaved or non-interleaved memory. The TC2000 memory interleaving scheme is associated with the memory architecture in general, rather than with the TC/FPV (or any specific function board), and therefore is described in detail in conjunction with the memory rather than here.

The T_INTERLEAVED bit is sourced by the Interleave Decision RAM that takes in 14 bits (31..26 and 22..15) of the Physical Address generated by the 88200 CMMU. For diagnostics and initialization, the contents of the Interleave Decision RAM are read and written using the interleaver loader described in section 11.17. The output of the Interleave Decision RAM is gated by the *interleave enable* bit of the CPU Mapping RAM.

As noted earlier, all interleaved references are routed through the switch, regardless of the target address.

11.3.7 What Path an Access Takes

This section describes the path taken by a memory access from the TC/FPV CPU. VMEbus accesses are covered in the VMEbus interface section. This description is both to clarify operation of the hardware, and to set context for memory access timing in section 11.11. More detailed information about the 88000 chip set can be found in the *MC88100 User's Manual* and the *MC88200 User's Manual*.

The possible outcomes of an access are:
— a cache hit
— a fast path access to local memory
— a T-bus access to local memory
— a switch access to remote memory (including back to this function board)
— a bus error
— a transaction fault detected during address translation in the CMMU.

The CPU's normal processing can be disrupted by any of the following exceptions:

- Internally, the CPU can detect and assert various exception conditions (see the *88100 User's Manual*).
- Externally:
  - Assertion of the reset pin resets the CPU.
  - The CPU may be interrupted by assertion of its interrupt (INT) pin.
  - A data transaction (read or write) may encounter an error, signalled to the CPU by a code on its Data Reply pins (11 = transaction fault).
  - A code transaction (instruction fetch) may encounter an error, signalled to the CPU by a code on its Code Reply pins (11 = transaction fault).

The four external causes have different exception vectors assigned to them. Reset is not of concern here, where we assume processing is in progress. Interrupts are signalled only by the interrupt logic on the TC/FPV described in section 11.3.5. The data and code transaction faults are signalled only by the CMMUs.

START:

- The 88100 generates a reference. It supplies Process Logical Address bits, supervisor/user mode bit, read/write bit, and instruction/data bit.
- One of the three CMMUs will respond, based on the instruction/data bit and, if the reference is an instruction fetch, the code cache selection logic, as follows:
  - If the Machine Configuration register *cache selection scheme* bit is zero, then Process Logical Address bit 12 selects the code cache.
  - If the Machine Configuration register *cache selection scheme* bit is one, then the CPU's supervisor/user mode bit selects the code cache.

  From here on, the CMMU that responds is called "the" CMMU.
- The CMMU attempts to locate a mapping for the Process Logical Address. The possible outcomes of this process are:

- The CMMU may find a valid entry in its ATC with protection attributes that match those of the process making the reference.
- The CMMU may have to search its translation tables to find the mapping. This involves memory references initiated by the CMMU. If a memory error occurs on one of these references, the CMMU returns a transaction fault to the CPU. Otherwise, it loads the appropriate translation information into its Address Translation Cache (ATC) and translates the address.
- The CMMU may find that the transaction violates the protection specified in the selected translation register, either write protection or supervisor mode protection. A transaction fault is returned to the CPU.
- The CMMU may find that a segment descriptor or page descriptor that it needs is invalid. A transaction fault is returned to the CPU.

- The CMMU attempts to make the requested access, using the Physical Address generated in the translation step. The possible outcomes are:
  - The page may be marked "cache inhibit", that is, non-cacheable. The access is passed on through the CMMU to the memory bus. If a bus error occurs on the memory bus, the CMMU passes the error back to the CPU as a transaction fault.
  - The CMMU may have the data cached (a cache hit). For a read, the data is returned quickly to the CPU. For a write, the cached data is overwritten with the new data. If the page being referenced is in copyback mode and the cache line (four 32-bit words) had previously been modified, the transaction is done. If the page is in writethrough mode, or if it is in copyback mode and this is the first write since the line was loaded into the cache, the CMMU writes the cache line back to memory. If a bus error occurs on this write, the CMMU returns a transaction fault to the CPU.
  - The CMMU may not have the data cached (a cache miss). The CMMU will select a cache line to replace with the desired line. If the selected line is modified and is in copyback mode, the CMMU must write it back to memory before replacing it. If a bus error occurs on this write, the CMMU returns a transaction fault to the CPU. If the selected line is not modified, or is in writethrough mode, it is simply discarded. The CMMU then reads the new cache line from memory. If a bus error occurs on this read, the CMMU returns a transaction fault to the CPU. Having filled the cache line, the CMMU performs the requested access, as described above for a cache hit.

Whenever the CMMU initiates a memory reference, the CPU interface on the TC/FPV translates the CMMU address, data and control signals, which obey the M-bus protocol defined by Motorola, into signals that obey the TC2000

T-bus protocol. In addition to accounting for various timing differences, the CPU interface logic performs several operations:

- Decide whether the reference can be handled by the fast path, and issue the necessary control signals if it can.
- Decide whether the reference should be intercepted. If the *intercept access** bit (the * is part of the name of the bit, indicating it is low true) in the CMR entry selected by this reference is zero, the access is intercepted. That is, the CPU is acknowledged as if the reference has completed, but no T-bus cycle occurs. If the reference is a read, the data returned is undefined.

  The intercept access mechanism is used for speeding up certain block transfer operations involving the cache. It is described further in sections 11.10.3 and 11.11.1.
- Generate the T-bus control signals needed to specify the path to be taken by the address and data for this reference (if not the fast path). These are:
  - T_INTERLEAVED — Generated by the Interleave Decision RAM, subject to the CMR *interleave enable* bit. If this is a remote reference, this signal indicates that the transformed address generated by the interleaver should be used by the SIGA. The CMR *local* bit is ignored if the T_INTERLEAVED bit is asserted.
  - T_PATH<1..0> — Generated by the Process Configuration register, subject to the CPU Mapping RAM *local* bit and interleaving. Indicates whether this reference should be handled by switch interface A, switch interface B, or local memory. See section 11.9.2.
- Generate T-bus control signals that specify other attributes of the reference:
  - T_PRIORITY<1..0> — Generated by the priority scheme mechanism. Indicates the priority of this reference in the switch.
  - BYPASS — Derived from the CPU Mapping RAM *bypass* bit. When this signal is asserted, the TC2000 locking protocol is inhibited and T_LOCKOP is set to "bypass".
  - T_LOCKOP<1..0> — Derived from the state of the Augmentation register *lock* bit, the $\overline{\text{DLOCK}}$ signal asserted by the 88200 CMMU during XMEM operations, and the BYPASS signal. Opens, maintains and frees locked transactions, or bypasses locks, according to the TC2000 locking protocol.
  - T_SYNC — Generated by the *synchronized access* bit in the Process Configuration register. If this is a remote reference, this signal influences the time at which the request message is allowed to enter the switch, as described in the SIGA specification.

- T_AD<33..23> — Generated by the CPU Mapping RAM. These bits select an 8-megabyte "bank" of memory within the System Physical Address space global to the machine. As part of that selection, they specify the switch port that is addressed. The switch port, however, is subject to possible further modification by the interleaver. If the reference falls in a window to VMEbus memory, the address is subject to further modification by the VMEbus master mapper at the addressed switch port.

- If the request is not serviced by the fast path or by the intercept access mechanism, the CPU interface issues a T-bus request, where it may be serviced by any of several T-bus slaves, depending on the value of T_PATH<1..0>.
  - T_PATH = 11: The local memory, the VMEbus master, or the configuration and control registers will respond.
  - T_PATH = 10: The switch interface serving the "A" switch will issue a request message, retransmitting until the request reaches its destination or is timed out.
  - T_PATH = 01: The switch interface serving the "B" switch will issue a request message, retransmitting until the request reaches its destination or is timed out.
  - T_PATH = 00: Illegal value.
- If the reference is non-local, the SIGA at the remote end makes a request on its local T-bus. The parameters of this request are as follows:
  - The T_PATH<1..0> bits are always 11.
  - The T_AD<33..25> bits are driven to zero. For a local access (T_PATH = 11), the *T-bus Specification* requires that T_AD<33..25> be driven to valid binary levels, but their value is undefined.
  - The remaining address and control bits are specified by the switch request message.

  The result of the T_PATH bits being 11 is that the the request cannot go back out either SIGA on the remote function board, but must be served (if at all) by a device local to that board. In the TC/FPV, this can be memory, local configuration and control registers, or the VMEbus master interface. In fact, the VMEbus interface can be set up so the request goes out onto the VMEbus and comes back in the VMEbus slave interface, so the request could proceed further. However, use of this facility (for other than testing) is highly contrived and risks having timers (that help control latency) go off, aborting the connection and returning a bus error.
- When the switch reply message comes back, the SIGA acquires the T-bus and places the requested data on it. The CPU interface takes the data from the T-bus and hands it to the CMMU, which retains a copy

in its cache if it is a cacheable reference. The CMMU hands the data on to the CPU, and execution resumes.

11.4 Memory Subsystem

The TC/FPV memory subsystem features either four or sixteen megabytes of dynamic RAM arranged as 36-bit words. Each word has 32 data bits and four parity bits, one for each byte. The memory array is implemented using 80-nanosecond, 1M or 4M × 9-bit single in-line memory modules (SIMMs). The memory array can be populated with 1-megabit SIMMs, yielding a 4-megabyte TC/FPV, or with 4-megabit SIMMs, yielding the 16-megabyte version of the board. Writes are acknowledged immediately without waiting for the write to complete. The memory cannot process new requests while the write is completing, so it will refuse all new accesses during the clock tick immediately following a write. Memory accesses are never split on the T-bus. (A T-bus "split cycle" allows other use of the T-bus while a request is being processed.) Accesses are refused when the memory is refreshing. The memory subsystem supports the T-bus locking protocol.

The memory subsystem supports byte, halfword, and word reads and writes. Multiple-word burst reads and writes are also supported. New data is supplied every 100 nanoseconds during burst reads. The T-bus "PAUSE, MORE" protocol is used to throttle data requests during bursts.

Byte-wide parity generation and checking is performed on every memory access. Parity errors are reported on the M_PARITY signal.

The memory subsystem performs a "hidden" refresh cycle once every 12.8 microseconds. Hidden refresh utilizes internal counters to generate the refresh address, and thus requires minimal external support circuitry. The refresh cycle is included in the memory control finite state machine. A refresh cycle requires five T-bus clock cycles (250 nanoseconds). During a refresh cycle, all T-bus requests to memory are responded to with "REFUSED". Refresh requests have the highest priority of any request to the memory.

If a parity error occurs on a memory access, the error is given to the CPU on that access. Note, however, that if an access from the CPU results in a burst read by one of the CMMUs (that is, the access is to a cacheable location which is not currently in the cache), a parity error on any of the four words in the burst will result in a parity error given to the CPU (as a transaction fault). Therefore, the CPU may receive a parity error due to a word that it didn't explicitly reference.

11.5 TCS Slave Interface

Like other TC2000 boards, the TC/FPV includes a Test and Control System slave processor to support bootstrapping, servicing, and monitoring of various environmental conditions.

The TC/FPV TCS slave has access to every addressable location in the TC/FPV via the T-bus. Thus, it can access memory, VMEbus devices, and configuration registers. It can also make references across the switch. To reduce hardware cost, the TCS slave shares the T-bus interface used by the SIGA. When the TCS master requests a data transfer into or out of the TC/FPV, the TCS slave uses a dedicated four-wire communication channel to the SIGA to fulfill the request. The four wires are data in, data out, clock and frame. The protocol for this communication channel is defined in the SIGA specification.

The TC/FPV TCS slave processor has several functions. It monitors the ambient temperature in one location and three power supply voltage levels (±24 volt input, +5 volt output and −5 volt output) on the board. The slave supplies this information to the TCS master when requested. It controls the on-board power supply. The slave can reset the TC/FPV. Finally, the TC/FPV TCS slave can inform the TCS master of the TC/FPV's board type. The capabilities of the TCS slave and the mechanisms for invoking them are described in detail in the TC/FPV TCS slave software specification.

The TC/FPV TCS slave is implemented using a Motorola 68HC11. This device has on-chip EEPROM, general purpose parallel I/O and serial I/O, A/D channels, a priority interrupt structure, a number of timers, and a 6800-type instruction set.

The TCS slave interface is powered by a supply located centrally in the TC2000 chassis. The power is distributed to the slave through the TC/FPV midplane connector.

11.5.1 Power-up and Resetting

The TC/FPV is reset by application of power and by TCS action, as described below. There is no reset button on the board. There is no way to explicitly force a reset across the switch. No VMEbus device can explicitly reset the TC/FPV, and when the TC/FPV resets the VMEbus that does not reset the TC/FPV board.

A power-on reset circuit in the TCS slave leaves the TC/FPV in a reset state after power-on. Before the TC/FPV can run, the TCS master must load the appropriate bootstrap code, set several configuration registers, then clear the power-on reset.

The TCS clears the power-on reset in two steps. First, the TCS clears the reset for all TC/FPV components except the CPU. With the CPU still held reset, the TCS can perform appropriate start-up tests, configure the hardware by loading control and configuration registers, and load bootstrap code into memory. The second step is to clear the reset for the CPU, so it starts executing. CPU actions after reset are described in detail in the Motorola *88100 User's Manual*.

11.6 Switch Interface

The interface between the TC/FPV and the Butterfly switch is implemented by the Switch Interface Gate Array (SIGA), described in detail in the *SIGA Specification*, and the Level CONverter (LCON), described in the *LCON Specification*. Besides the SIGA's direct connection to the T-bus, the switch interface includes an interleaver. The interleaver optionally maps references (if they go over the switch) from the switch port given in the address on the T-bus to a new switch port.

The TC/FPV contains board space and wiring to support a second switch interface. Normally only one interface is populated, since only one switch is normally supplied per machine. In a two-switch machine, the two interfaces are functionally equivalent. Only one switch would be enabled for carrying traffic at any given time; the other would be available as a standby.

NOTE

DETAIL OF TWO SWITCHES — SIMULTANEOUS USE IS NOT OK
The simultaneous use of both switches to pass traffic not only would be tricky to orchestrate in software, but also is prohibited because a temporary deadlock can occur. The deadlock involves the priorities for use of the T-bus on the TC/FPV. In the TC/FPV, these priorities are:

highest:	SIGA B requester	(a T-bus slave)
	SIGA A requester	(a T-bus slave)
	VMEbus master interface	(a T-bus slave)
	SIGA B server	(a T-bus master)
	SIGA A server	(a T-bus master)
	VMEbus slave interface	(a T-bus master)
lowest:	CPU	(a T-bus master)

When a T-bus slave device operates in split-cycle mode, it releases the T-bus after receiving a request, allowing other use of the T-bus while the request is processed. When the response is ready, the device, still called a T-bus "slave", acquires the T-bus and supplies the response. Thus such slaves are assigned a priority. The *T-bus Specification* requires that such slaves must have higher priority than any requesting T-bus master. Otherwise, a master repeatedly re-trying an access to the lower-priority slave could prevent the split-cycle slave from ever supplying its response.

However, there is still a temporary deadlock possible. Both sides of the VMEbus interface must be busy; the VMEbus master interface must be busy so that it will refuse requests, and the VMEbus slave interface must be busy so that the VMEbus itself is busy, preventing the VMEbus master interface from completing its access. To make both sides busy, either of two scenarios can take place. A VMEbus device can begin an access to TC2000 address space at the same time as the CPU begins an access to VMEbus address space. Alternatively, the VMEbus interface can be set up to loop accesses from the CPU out onto the VMEbus and back into the TC2000; then a CPU reference busies both sides at once.

Once both sides of the VMEbus are busy, suppose both SIGA servers try to access the VMEbus. Their priority is higher than that of the VMEbus slave interface. The two SIGAs take turns on the T-bus, getting refused by the VMEbus master interface, often enough to saturate the T-bus and exclude the VMEbus slave interface. Eventually, a timeout (switch connection or VMEbus TC/FPV master) will occur and break the deadlock. The resulting bus error, however, can be very hard for software to interpret properly. To avoid this, the two switches should not be used to pass traffic simultaneously.

This deadlock is an example of a general concern whenever enough T-bus masters try to access a split-cycle slave. It is the only example that arises in the TC/FPV. The memory module does not split cycles.

NOTE

DETAIL OF TWO SWITCHES — TWO RTCs AND THEIR TONIs
The Real Time Clock in the two SIGAs, and their TONI timers, could all be operated simultaneously. However, because SIGA A's RTC is driven by the switch A master clock, and SIGA B's RTC by the switch B master clock, the two RTCs may count at very slightly different rates. The oscillator used in January 1990 is rated at $\pm 0.005\%$ long-term accuracy, yielding up to one minute per week relative drift between the two clocks.

11.7 VMEbus Interface

This section briefly describes the VMEbus, and then discusses the operation of each VMEbus feature implemented by the TC/FPV. A detailed description of the VMEbus can be found in *The VMEbus Specification*, a Motorola publication available from BBN ACI. The TC/FPV is designed to revision C.1 of the specification.

The VMEbus interface on the TC/FPV is a general-purpose bus coupling mechanism that provides a path between the T-bus and a VMEbus. Through the T-bus connection, VMEbus devices can access local or remote memory, perform locked operations, and interrupt any processor on the machine. Simi-

larly, any TC2000 processor can access VMEbus memory, perform indivisible operations on VMEbus memory, and interrupt VMEbus devices.

11.7.1 Brief Description of the VMEbus

The VMEbus Specification defines an interface, called the VMEbus, used to connect data processing, data storage and peripheral control devices. The VMEbus is an asynchronous bus defined in broad terms that allow a wide range in performance while still conforming to the specification. This results in some difficulty in characterizing maximum cycle times on the VMEbus, because cycles can be extremely long without violating the specification.

The VMEbus Specification defines a functional module as "a collection of electronic circuitry that works together to accomplish a task". The types of functional modules defined in *The VMEbus Specification* are listed below. The TC/FPV implements all but those marked with ♠.

- VMEbus master
- VMEbus requester (associated with master or interrupt handler)
- VMEbus slave
- Interrupt handler
- Interrupter (interrupt generator)
- System controller
- VMEbus arbiter
- IACK daisy chain driver
- System clock driver
- Bus timer
- Backplane interface logic
- Power monitor ♠
- Location monitor ♠
- Serial clock driver (for VMS bus only) ♠

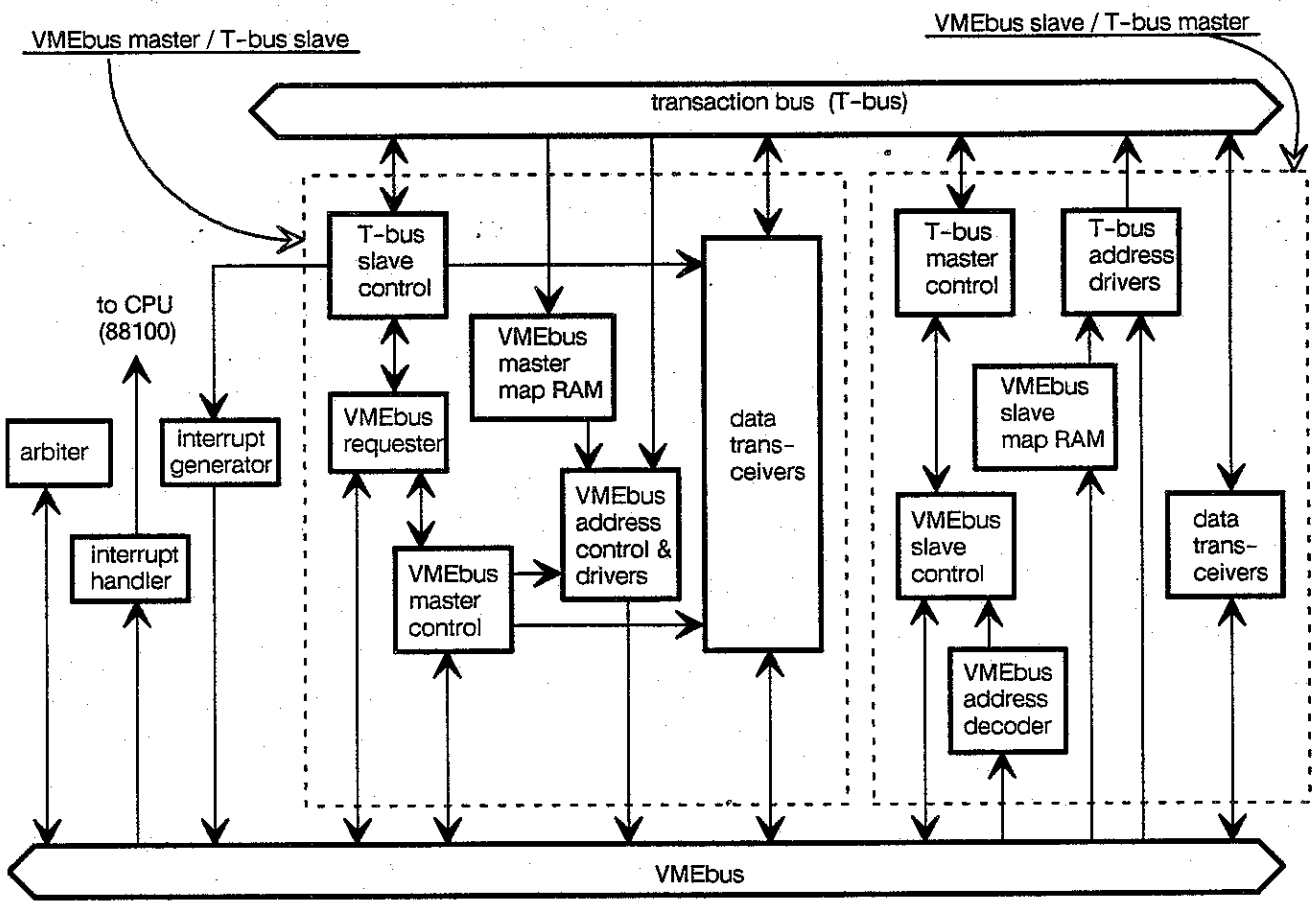
A VMEbus device implements a subset of this list to achieve the functionality required of the device. For example, a CPU board might consist of a VMEbus master, an interrupter, an interrupt handler, and a requester. A disk controller might include a VMEbus master, slave, interrupter, interrupt handler, etc. A memory board might consist of only a VMEbus slave.

A device called a *system controller* is required in each VMEbus system. It physically resides in slot one of a VMEbus, and performs the functions of arbiter, system clock generator, bus timer, and IACK daisy chain driver.

11.7.2 TC/FPV Functionality

Figure 11–4 is a block diagram of the TC/FPV VMEbus interface.

Figure 11–4 VMEbus interface block diagram.



The TC/FPV can be used in several different configurations. In some environments, the TC/FPV may be the only VMEbus master in the system. When this is the case, the TC/FPV may also be required to perform the VMEbus system controller task. In other environments, the TC/FPV will provide a path that a VMEbus master will use as a DMA (direct memory access) target, from the VMEbus into TC2000 memory. In this case, the TC/FPV will function primarily as a VMEbus slave. Because the TC/FPV must perform a variety of tasks on the VMEbus, it implements all but three (marked with ♠ in the list above) of the functional modules defined in *The VMEbus Specification*. The system controller functions (arbiter, system clock driver, bus timer, and IACK daisy chain driver) can be enabled or disabled by setting a jumper on the TC/FPV. Note that when the TC/FPV is functioning as a VMEbus master, it is also functioning as a T-bus slave. Similarly, if the TC/FPV is functioning as a VMEbus slave, it is also a T-bus master.

11.7.3 VMEbus Master

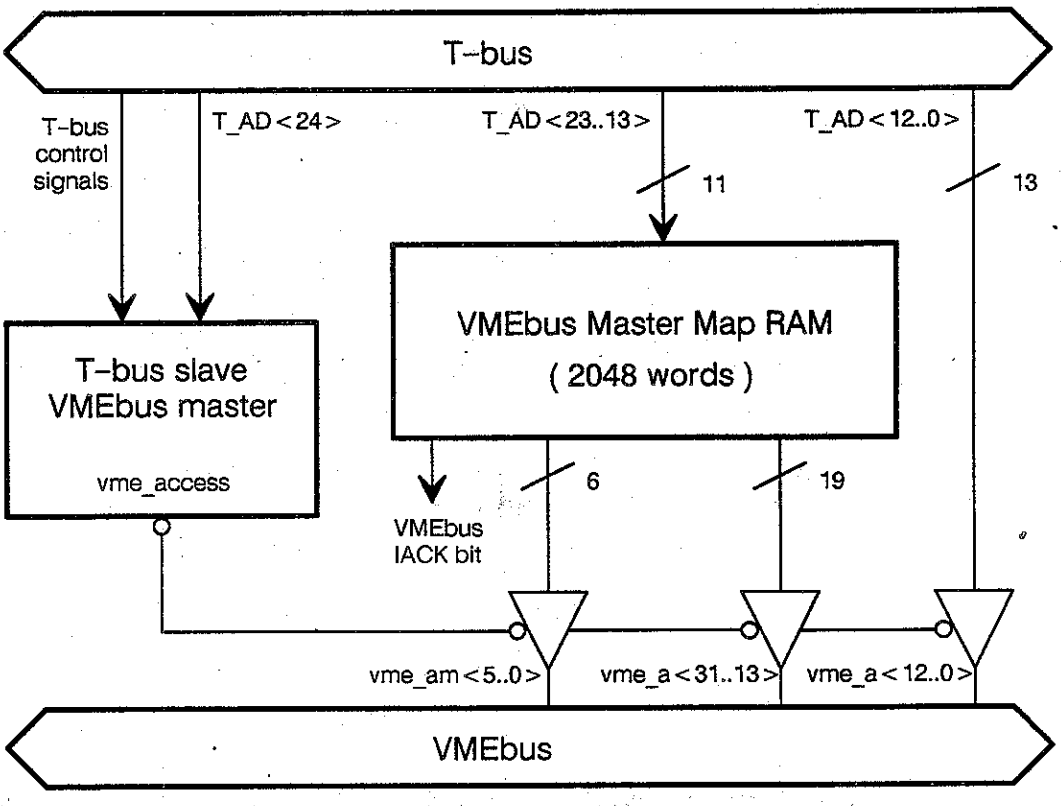
The TC/FPV VMEbus master interface accepts 1–, 2– or 4–byte read and write requests from the T–bus, and converts them into requests on the VMEbus. Halfword (2–byte) and word (4–byte) requests must be halfword and word aligned, respectively. The VMEbus master interface splits cycles on the T–bus, letting other T–bus masters use the T–bus while the VMEbus transaction is completing. This is necessary because the access time on the VMEbus can be relatively long, and therefore an access to the VMEbus is not guaranteed to complete in a reasonable amount of time from the perspective of the TC2000 design.

The T–bus locking protocol is observed by the VMEbus master interface. A lock request on the T–bus results in holding the VMEbus until the lock is released. This prevents any other VMEbus master from accessing any VMEbus resource, thus preserving the atomicity of the T–bus lock. Note that if the VMEbus resource is dual–ported, then atomicity is no longer guaranteed. An example of a dual–ported VMEbus resource is the Motorola memory board that connects to both a VMEbus and a faster auxiliary bus such as VMX or VSB.

The VMEbus is accessed through a 15–megabyte window in the System Physical Address space of the TC/FPV. This window is part of the global System Physical Address space of the machine. Thus, the VMEbus is accessible to every function card in a TC2000 machine. The addresses in this window are translated to VMEbus addresses by the the VMEbus master mapper, which maps 8–kilobyte segments of TC/FPV System Physical Address space to 8–kilobyte segments of VMEbus address space. The VMEbus master mapper is implemented by a mapping RAM that is loaded by the interleaver loader described in section 11.17.

Figure 11–5 illustrates the operation of the VMEbus master mapper. If bit 24 of the TC/FPV System Physical Address (thus, the high order bit of the address offset field) is "1", then it lies within the VMEbus window and the TC/FPV's VMEbus master passes the transaction on to the VMEbus. Bits 23..13 of the TC/FPV System Physical Address are used to generate bits 31..13 of the VMEbus address plus the six address modifier bits that must be driven on every VMEbus transaction. The 13 low order bits of the TC/FPV System Physical Address pass directly through to the VMEbus. The address modifier bits are described in detail in *The VMEbus Specification*.

Figure 11–5 VMEbus master mapper.



NOTE

USE OF TC2000 → VMEbus WINDOW BY SOFTWARE
The window from TC2000 System Physical Address space into VMEbus address space is at a fixed location, **15** megabytes in every TC/FPV, 0x1000000 to 0x1EFFFFF. If a T–bus access falls within this window, an access to some VMEbus location will be made; there is no way to disable the mapping altogether, nor to map any of the 8–kilobyte segments to not make a VMEbus access. Therefore, the smallest amount of VMEbus address space that the TC2000 machine can see is 8 kilobytes, obtained by mapping all segments to one value.

NOTE

TOP MEGABYTE IS NOT IN VMEbus WINDOW
The top one megabyte of TC/FPV address space, 0x1F00000 to 0x1FFFFFF, is occupied by configuration and control registers.

11.7.4 VMEbus Requester

Associated with the VMEbus master on the TC/FPV is a VMEbus requester. Before the master initiates a transfer, the requester must communicate with the VMEbus arbiter to acquire ownership of the bus. *The VMEbus Specification* defines two types of requester: release–on–request and release–when–done. The requester on the TC/FPV can be configured to be either type by using the VMEbus Configuration register.

Release–on–request requesters do not let go of the bus after the transaction is completed. Instead, they wait until another master requests the bus before releasing it. If another transaction is requested by the the release–on–request requester's master, then arbitration does not take place again, because ownership has not been relinquished. Release–on–request requesters are useful when the associated master is the primary, or most frequent, owner of the bus.

Release–when–done requesters release the bus during the transaction or after it has completed. Releasing the bus during the last transaction allows arbitration to take place while the transaction completes, and reduces the overhead of changing masters on the bus. Release–when–done requesters are useful when the associated master is not the primary master on the VMEbus.

11.7.5 VMEbus Slave

The VMEbus slave interface responds to 1–, 2– or 4–byte read and write requests on the VMEbus. Halfword (2–byte) and word (4–byte) requests must be halfword and word aligned, respectively. It translates each such request into a T–bus transaction that can access local memory, remote memory, or the configuration and control registers on the TC/FPV (not the CPU's internal registers). Note that the timeouts on VMEbus devices that access TC2000 *remote* memory must be longer than the worst case switch latency for the system in which they are intended to operate. The VMEbus slave interface on the TC/FPV does not respond to sequential transfer requests (also called "block transfers" in *The VMEbus Specification*) from VMEbus masters.

A VMEbus master accesses TC2000 memory through a 4– or 16–megabyte window in VMEbus address space, depending on how the VMEbus slave interface is configured. The addresses in this window are translated to TC2000 System Physical Addresses by the VMEbus slave mapper, which maps 8–kilobyte segments of VMEbus address space to 8–kilobyte segments of TC2000 System Physical Address space. Unlike the master mapper, the VMEbus slave mapper is implemented by a mapping RAM that can be read and written by any T–bus master. Thus, the slave mapper RAM can be manipulated by the CPU, by the the TCS slave on the TC/FPV, or by a remote processor.

The VMEbus slave responds to the VMEbus master on the TC/FPV in the same way that it responds to any other VMEbus master. This is useful for diagnostic purposes.

Figure 11–6 VMEbus slave mapper.

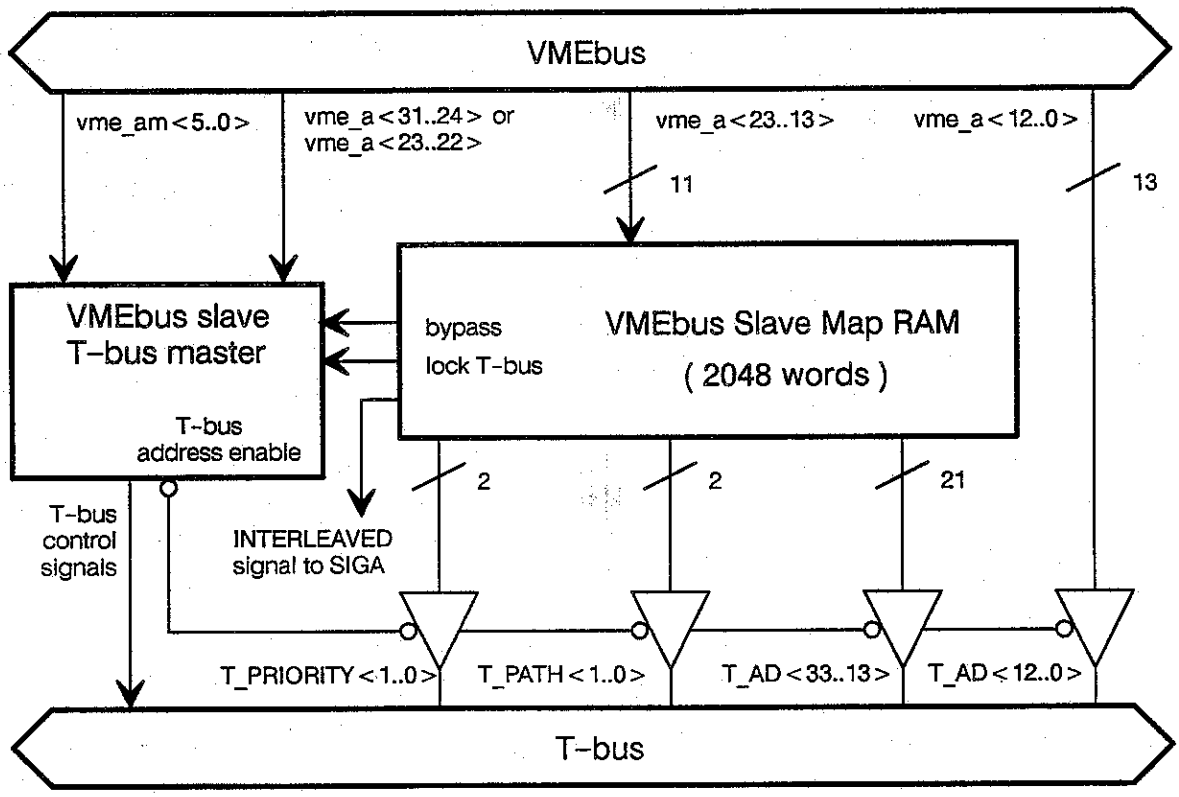


Figure 11–6 illustrates the operation of the VMEbus slave mapper. Whenever a VMEbus master initiates a VMEbus operation, the VMEbus slave must first determine whether the target address lies inside the window that maps to TC2000 System Physical Space. The way that this is done depends on whether the VMEbus slave on the TC/FPV has been configured to respond to "standard" (24–bit) or "extended" (32–bit) VMEbus addresses. This is determined by the value of the *standard / extended addressing* bit in the VMEbus Configuration register. If the slave is configured for extended addressing, then the window size is 16 megabytes, and the slave mapper uses bits 31..24 of the VMEbus address to determine whether the transaction should be passed through to the TC/FPV. If the the slave is configured for standard addressing, then the window size is 4 megabytes, and the slave mapper uses bits 23..22 of the VMEbus address for the determination. The window is smaller for standard addressing so that it occupies a reasonable fraction of the 16–megabyte space accessible to devices that use standard addressing. The VMEbus address modifier bits indicate the addressing mode for each transaction. The VMEbus slave responds only when the type of the address on the VMEbus (standard or extended) matches the type that the slave is configured to handle. The appropriate VMEbus address bits are compared to *VMEbus slave window* bits in the VMEbus Configuration register. If the comparison results in a match,

the access falls within the TC/FPV's window and the TC/FPV's VMEbus slave will pass it on to the TC/FPV.

The window from VMEbus address space into TC2000 System Physical Address space starts at a location determined by the VMEbus Configuration register. If a VMEbus access falls within this window and has the appropriate addressing mode, an access to some T-bus location will be made; there is no way to disable the mapping altogether, nor to map any of the 8-kilobyte segments to not make a T-bus access. Therefore, the smallest amount of TC2000 address space that the VMEbus can see is 8 kilobytes, obtained by mapping all segments to one value.

When set to respond to VMEbus standard addressing, the VMEbus slave mapper uses only 512 of its 2048 mapping registers. This is discussed further in the description of the VMEbus Slave Map RAM registers in the register definitions section.

If the transaction is to be passed on to the TC/FPV, the slave mapper uses bits 23..13 of the VMEbus address to generate bits 33..13 of the TC2000 System Physical Address plus several T-bus control bits. The 13 low order bits of the VMEbus address are passed directly through to the T-bus.

In addition to generating the necessary T-bus address bits, the VMEbus slave interface uses bits 23..13 of the VMEbus address to select values for several T-bus control signals. Thus, the attributes of an access can vary depending on which 8-kilobyte block the address resides in. The control bits whose values are selected are:

- T_PRIORITY — determines the priority of a transaction
- T_PATH — specifies a local (on-board) access or a remote (over the Butterfly switch) access, and selects one of two possible switches if remote
- INTERLEAVED — indicates that the target of the transaction is in interleaved memory
- bypass — when asserted, the transaction proceeds regardless of whether or not the target memory module is locked by the TC2000 locking protocol
- lock T-bus — intended to support VMEbus read-modify-write cycles; when this bit is asserted, the accessed memory is locked for the duration of the transaction

The effect of these control bits is discussed in detail in the *T-bus Specification*. In the following section, we present a few examples that show how they can be used. In addition to specifying values for T-bus control signals, the VMEbus slave can be configured to allow conversion of VMEbus read-modify-write cycles into locked operations within the TC2000 machine. This is shown in one of the examples in section 11.7.15.

The bypass control bit is intended for transactions to local memory that can safely ignore the TC2000 locking protocol. For example, locked transactions are never used to modify locations in code space, so instruction fetches by the CPU always bypass locks. This improves the execution speed of the machine, by letting the CPU fetch code even when local memory has been locked by another CPU performing an indivisible operation. The performance of a simple DMA device, such as a disk interface, can be improved in the same way. As long as the device driver does not use locked operations to synchronize with the DMA device, the TC2000 memory accessed by the controller can be mapped bypassed by the VMEbus slave mapper, and DMA transfers will not be held up when TC/FPV memory is locked.

NOTE

USE OF ADDRESS MODIFIER BITS
The TC/FPV VMEbus slave mapper receives the six bits of VMEbus address modifier. In the initial implementation, these bits are used only to determine whether the reference is using standard or extended addressing. This processing is performed in a programmable logic array (PLA) chip. If a high volume application required a more sophisticated interpretation, it might be possible to accommodate this need by creating a new version of the board with a modified PLA.

11.7.6 VMEbus Interrupt Handling and Generation

The TC/FPV generates interrupts to the VMEbus, and handles interrupts from the VMEbus. A Signetics SCB68154 chip implements the interrupt generator. This device has two internal read/write registers, the VMEbus Interrupt Request register and the VMEbus Interrupt Vector / Control register. The protocol for generating an interrupt on the VMEbus is discussed in detail in the SCB68154 data sheet.

Interrupts received from the VMEbus result in interrupts to the 88100 on the TC/FPV as described in the section on interrupts to the CPU.

Briefly, the following steps are involved in receiving and servicing an interrupt from the VMEbus.

1. The TC/FPV interrupt system must be set up. This includes:

   Set the CPU's exception vector to point to the handler code.

   The CPU's Processor Status Register *interrupt disable* (IND) bit must be zero.

   The Augmentation register *disable interrupts* bit must be zero.

Other sources of interrupts on the TC/FPV must be initialized appropriately: the Interrupts Disabled Timer register, the Interprocessor Interrupt register, and the TONI Configuration registers are relevant here.

Set one of the VMEbus Master Map RAM registers ready to generate VMEbus IACK cycles.

Set the Interrupt Enable Mask register to accept VMEbus interrupts on the desired level(s).

2. A VMEbus device asserts an interrupt request on an enabled level.
3. The CPU traps to the exception handler, where code examines the Interrupt Source register, and finds that the interrupt was generated by a VMEbus device, and at which level.
4. The interrupt handler software generates an IACK cycle on the VMEbus, by performing a half-word read to an address mapped by a VMEbus Master Map RAM register that has been set up to trigger an IACK cycle when read. Details can be found in the description of the VMEbus Master Map RAM in the register definitions section.
5. The software continues with normal 88100 interrupt processing, including saving registers, servicing the VMEbus device, and at some point re-enabling interrupts.

Generating interrupts onto the VMEbus is considerably simpler, since it involves only the manipulation of two registers — the VMEbus Interrupt Request register and the VMEbus Interrupt Vector / Control register. The contents of these registers are set up to request the desired interrupt, and the TC/FPV hardware does the rest. When a VMEbus device acknowledges a requested interrupt, the corresponding bit in the VMEbus Interrupt Request register is cleared. By testing the bit, the TC2000 code can determine that the request has been acknowledged.

11.7.7 VMEbus System Controller

The TC/FPV can function as the VMEbus system controller when necessary. This option is jumper selectable. As the system controller, the TC/FPV will perform the following VMEbus functions: arbiter, bus timer, IACK daisy chain driver and system clock driver. To perform these functions correctly, a VMEbus system controller must reside in slot 1 of the VMEbus system that it controls.

VMEbus Arbitration

Three types of arbitration can be implemented on the VMEbus: prioritized, round-robin-select and single level. The simplest form of arbitration, single level, only monitors requests and issues grants on bus level 3. Because bus grants are daisy chained within a single level on the VMEbus, priority is as-

signed only by slot order in the VMEbus card cage. The TC/FPV VMEbus interface implements only single level arbitration. Refer to *The VMEbus Specification* for descriptions of the other arbitration schemes.

The VMEbus arbiter is implemented as a finite state machine clocked at 32 megahertz. The TC/FPV pulls the bus grant 2, bus grant 1 and bus grant 0 signals to a high state.

IACK Daisy Chain Driver

The VMEbus interrupt acknowledge (IACK) signal is carried on two bits. One bit is common to all devices and announces that an interrupt acknowledgment is present. The other bit is daisy–chained from device to device, and each device passes that signal onto the next link of the chain, unless that device is processing the acknowledgment. The VMEbus system controller always resides at the head of the daisy chain. When a VMEbus device wishes to acknowledge an interrupt, it asserts the single–wire signal. The VMEbus system controller takes the single–wire signal and drives it onto the daisy chain. In the TC/FPV, the IACK daisy chain is driven by the SCB68154 (interrupt generator). For further details, see a Signetics SCB68154 data sheet.

VMEbus System Clock Driver

The VMEbus system controller must drive a 16 megahertz clock signal on the VMEbus. When the TC/FPV is operating as system controller, this signal is generated by dividing the arbiter's 32 megahertz finite state machine clock by two and driving the result onto the bus.

11.7.8 VMEbus Timers

The VMEbus interface contains three timers. The VMEbus Arbiter Timer is used only when the TC/FPV is system controller on the VMEbus. The operation and expiration of all three timers is confined to the VMEbus, and does not affect the TC/FPV unless it is involved in the VMEbus transaction that is timed out.

VMEbus Arbiter Timer

When a VMEbus device requests the bus, the arbiter grants the bus to the device by asserting the *bus grant* signal, and starts the VMEbus Arbiter Timer. If the device is behaving as required by *The VMEbus Specification*, it will respond by asserting the *bus busy* signal, becoming bus master. If the device has failed to assert *bus busy* when the VMEbus Arbiter Timer expires, the arbiter and removes *bus grant*. This prevents a malfunctioning device from hanging the VMEbus.

VMEbus System Bus Timer

Once a device has become VMEbus master, it accesses a VMEbus slave device by asserting the *address strobe* signal. If a slave device responds, it does so by asserting the *DTACK* signal. It may be that no slave will respond, in which case the VMEbus System Bus Timer expires, and the VMEbus interface declares an error by asserting the *BERR* signal.

VMEbus TC/FPV Master Bus Timer

Unlike the other two VMEbus-related timers, the VMEbus TC/FPV Master Bus Timer specifically, although indirectly, affects operation of the TC2000. When the TC/FPV itself is master of the VMEbus, this timer limits how long the TC/FPV will await a response from a VMEbus slave, to control the latency of TC2000 references. Upon expiration of the timer, the bus error signal *BERR* is asserted on the VMEbus. This in turn causes removal of *address strobe* from the VMEbus, and return of a bus error to the T-bus master making the access.

The behavior of this timer is identical to that of the VMEbus System Bus Timer except that this timer runs only when the TC/FPV itself asserts *address strobe*, and the value of this timer's timeout is normally shorter.

11.7.9 Resets to and from the VMEbus

The TC2000 generates a reset on the VMEbus by clearing a bit in the VMEbus Configuration register. Resetting the TC/FPV board resets the VMEbus. However, resetting the VMEbus does not reset the TC/FPV. VMEbus resets can be monitored by reading the VMEbus Configuration register. If the VMEbus is turned off, all signals are terminated correctly such that no spurious signals occur and the VMEbus is held reset.

11.7.10 Communication Between Separate VMEbuses

This section and the following section describe TC2000 communication with VMEbus devices, and in particular two types of temporary deadlock that may arise if the machine is not used as intended. These deadlocks limit use of the TC2000 as a completely general interconnect among multiple VMEbus systems.

The TC2000 hardware supports direct communication between devices on separate VMEbuses. However, some care must be exercised in order to avoid deadlocks, because the VMEbus protocol was not designed with this kind of communication in mind. A deadlock situation arises when a device on VMEbus A initiates a read or write to a location on VMEbus B, while at the same time a device on VMEbus B initiates a read or write to a location on VMEbus

A. The request from VMEbus A cannot complete until VMEbus B becomes free, but VMEbus B is occupied by the request to VMEbus A.

Since one or both of the transactions will eventually time out, the deadlock will not hang any part of the system forever, but this arrangement will not lead to satisfactory performance. The timeout leads to a bus error being returned to one or both VMEbus devices, and a VMEbus device is unable to distinguish a bus error meaning "something is broken" from one meaning "couldn't service your request now, try again". This ambiguity would make it hard to program the VMEbus device correctly.

This deadlock can also arise in groups of more than two VMEbus systems. For example, VMEbus A may be waiting for completion of its access to VMEbus B, B is waiting for completion of its access to VMEbus C, and C in turn is waiting for completion of its access to A.

WARNING

VMEbus ⇆ VMEbus SIMULTANEOUS ACCESS
Mutual communication between separate VMEbuses through the TC2000 should be avoided in the design of TC2000 system software and applications. Communication *in one direction only* between such VMEbuses is not subject to the difficulty described above, and will work from a hardware point of view. Because of the various timers involved in such communication, however, the operation of one-way communication should be analyzed and tested before assuming that it will perform acceptably in a given application.

11.7.11 Communication Between CPU and Remote VMEbus

This section describes two forms of communication between a CPU and a remote VMEbus. Each form of communication works properly in isolation, but in combination they lead to temporary deadlock, timeouts, and bus errors. The deadlock is similar to that described in section 11.7.10, but more subtle. As in that case, one of the resources under contention is mastership of a VMEbus. In this case, however, the other resource is a switch cable.

Suppose a CPU makes a reference to a device on a remote VMEbus. The request goes out the switch port of the local function board, into the server SIGA on the remote board, onto the remote T-bus to the remote VMEbus master interface. That VMEbus interface responds to the server SIGA with "promise", meaning a reply will be supplied later, allowing other use of the T-bus while the VMEbus is accessed. For speed, the VMEbus master interface responds "promise" before it knows it can get mastership of the VMEbus.

Also suppose that, at about the same time, a device on that VMEbus makes a reference to memory on a function board remote from itself, so the reference must go out over the switch. The VMEbus device gains mastership of its

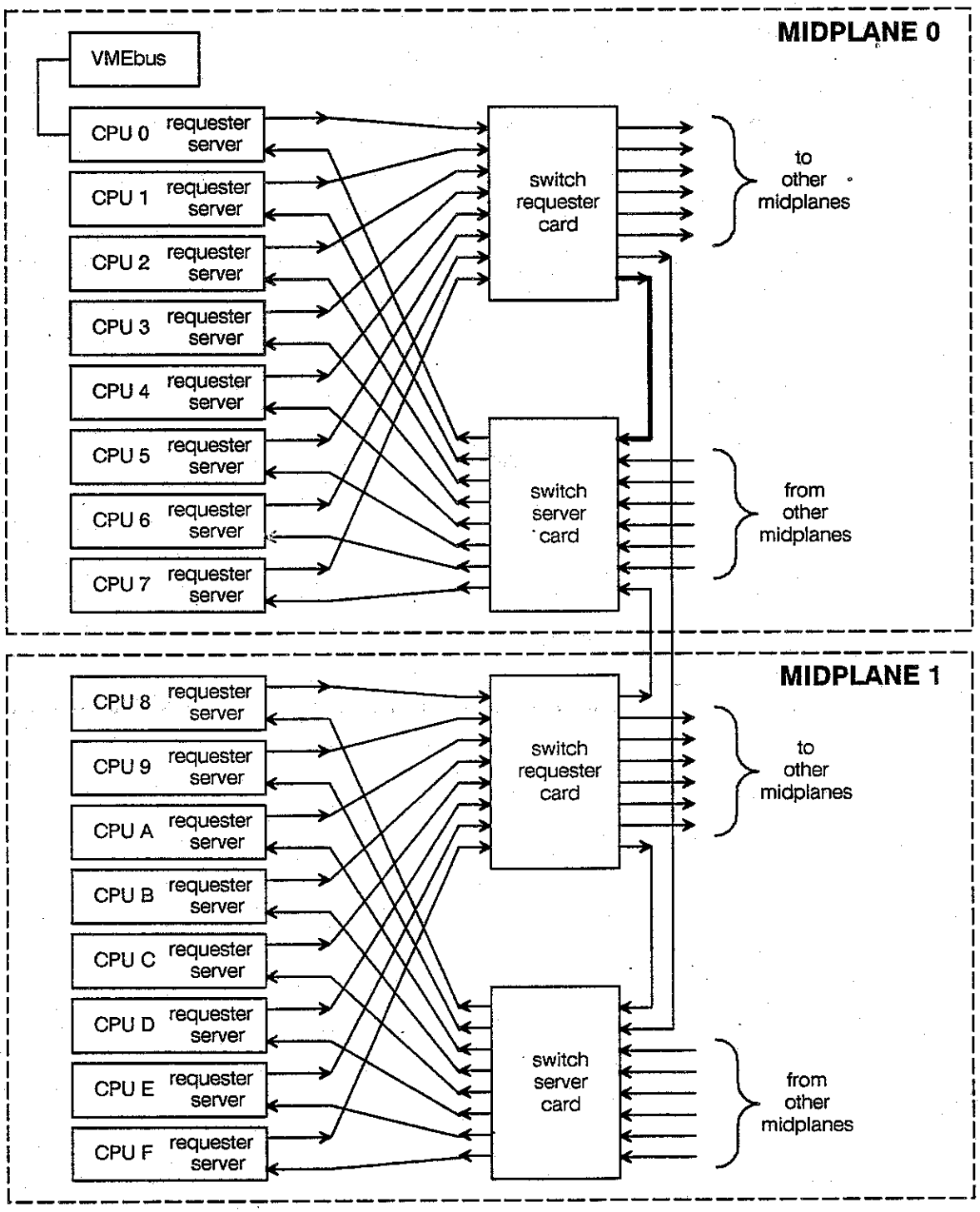
VMEbus and the request goes into the TC/FPV VMEbus slave interface, onto that function board's T-bus, and into the requester SIGA of that board's switch interface. The requester SIGA tries to set up a connection to the target function board.

Finally, suppose that the two switch paths involved have a Butterfly switch cable in common. The CPU's connection, including that cable, is held open waiting for the CPU's reference to complete. Therefore, the remote requester SIGA trying to set up a connection will repeatedly be rejected and retry. This is the temporary deadlock situation.

The deadlock situation is always broken by some timeout. Usually the switch connection timer breaks the CPU's connection. The switch path then becomes available, and the next retry of the VMEbus device's reference will typically succeed. Thus a typical outcome is that the VMEbus device's reference is merely delayed, but the CPU receives a bus error. The CPU is unable to determine whether its reference was completed or not. Depending on timing, the remote VMEbus master interface may give up before the VMEbus becomes free, or may eventually obtain bus mastership and complete the reference requested by the CPU. This ambiguity makes it hard to program the CPU to correctly handle the bus error, especially if the reference was a write (or a read with side effects).

One situation in which this temporary deadlock arises is shown in Figure 11–7. This is a 2–column, 64–function board machine; in such a configuration there is just one switch cable from any midplane to itself. That cable becomes a resource for which the two references contend. If the VMEbus device references any of the seven non–local function boards on that midplane, simultaneously with any of those seven CPUs referencing the VMEbus, the deadlock may occur. The heavy line shows the part of the CPU's switch path to the VMEbus that the VMEbus reference also needs.

Figure 11-7 CPU — VMEbus deadlock example.



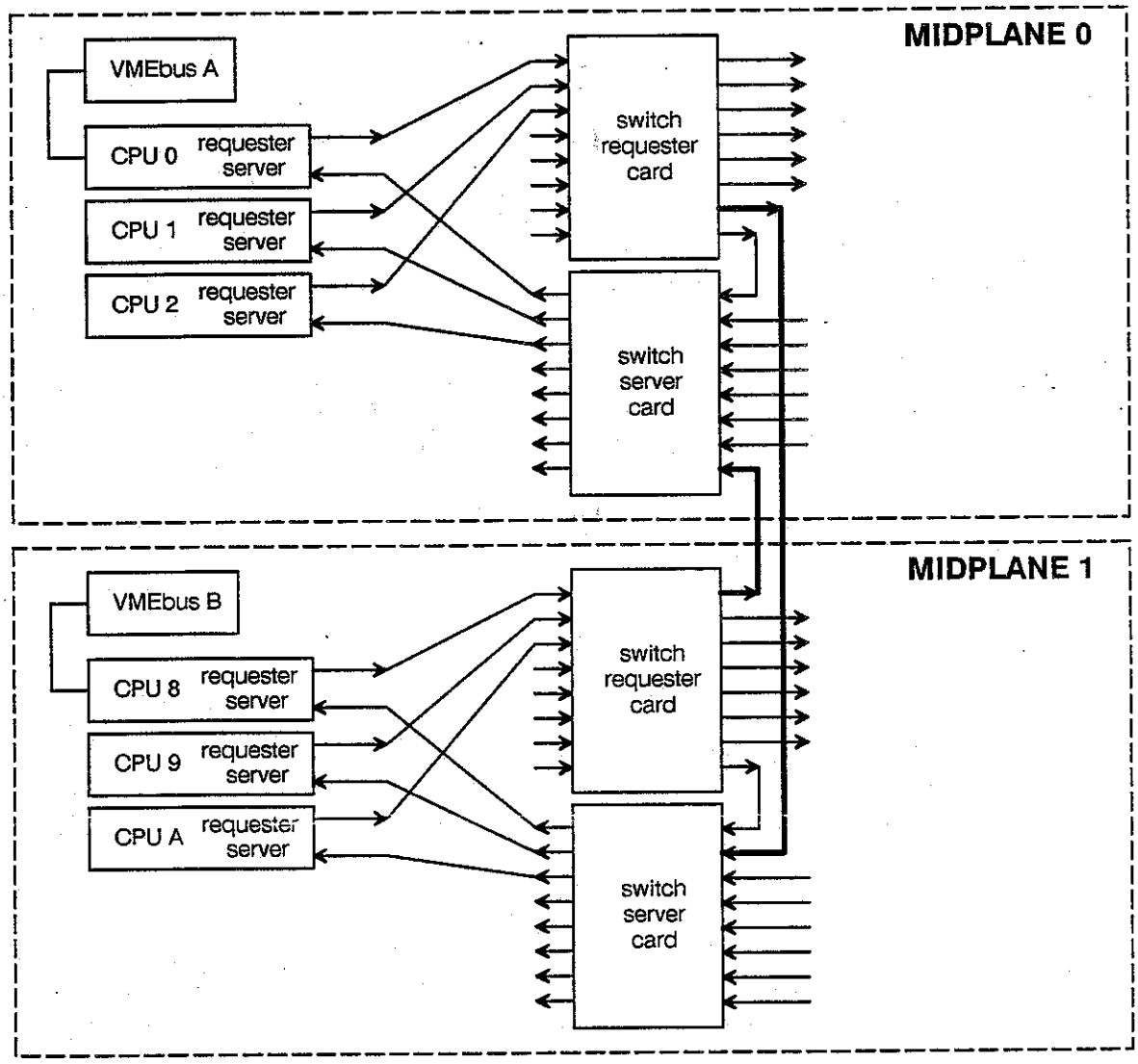
As can be seen from Figure 11-7, this deadlock arises in a 2-column machine only when the CPU, the VMEbus, and the function board target of the VMEbus device's reference are all on the same midplane. If any of the three are on a different midplane, the two switch paths have no cable in common, and

this deadlock does not arise. In a 3-column switch, the deadlock can arise if all three components are in the same bay (of eight midplanes).

The temporary deadlock can occur even if the machine has multiple cables between midplanes. Such cables provide *alternate paths* through the switch. However, there is a chance that the requester SIGA servicing the VMEbus device's reference will choose the same alternate path that is serving the CPU's request. For example, in a machine with two alternate paths, the temporary deadlock will arise half as often as on the 64-board machine described earlier. Note that when a requester SIGA first attempts to make a connection, it chooses at random one of the alternate paths, and continues to use that path for all retries of that request. It does not cycle through the alternate paths on successive retries. Trying the same path is necessary for the express message (priority promotion) mechanism to work.

Also, the temporary deadlock can occur in more complex situations involving multiple VMEbus systems. For example, Figure 11-8 shows a deadlock in which CPU 1 is waiting for completion of its access to VMEbus B, VMEbus B is waiting for a switch path to memory on CPU 2's board, that switch path is in use by CPU 9 which is waiting for completion of its access to VMEbus A, and VMEbus A is waiting for a switch path to memory on CPU A's board, but that path is in use by CPU 1. Note that multiple midplanes are involved here, not just one as in the example of Figure 11-7.

Figure 11–8 Multiple CPU and VMEbus deadlock example.



WARNING

CPU → VMEbus, VMEbus → MEMORY SIMULTANEOUS ACCESS
A CPU may reference devices on a remote VMEbus, or devices on a VMEbus may reference remote function boards, but no VMEbus should be used both as a target for references from a remote CPU and as a source of references to a remote function board, in any way that the two references might be made nearly simultaneously. This warning may be disregarded if the software is constructed to ensure either of two conditions:

1. Access to the VMEbus may be time multiplexed, operated in half-duplex fashion. A small guard interval must occur whenever the access is turned around, to prevent the race condition hazard of instantaneous reversal.

2. The location of the CPUs, VMEbuses and memories involved may be arranged so that the relevant switch paths never have any piece in common.

11.7.12 Locked Operations from VMEbus Devices

For applications that use intelligent VMEbus devices such as CPUs or array processors, the TC2000 locking protocol provides a useful mechanism for synchronization with TC2000 processors and with devices on other VMEbuses. Ideally, the VMEbus slave mapper would translate every VMEbus read–modify–write cycle to a locked operation within the TC2000 machine. Unfortunately, the VMEbus protocol is structured so that this cannot be done without slowing down ordinary VMEbus transactions. Instead of paying this penalty on every transaction, support for read–modify–write operations can be enabled or disabled for each 8–kilobyte segment of VMEbus address space that maps to TC2000 memory. All references in an enabled 8–kilobyte segment will invoke the TC2000 locking protocol.

11.7.13 Performance

The VMEbus slave interface sustains a measured data rate of 8 megabytes per second, either reading or writing. This is also the peak (burst) rate, since the TC/FPV does not perform block transfers.

The VMEbus master interface performs single transfers at a rate of 5 megabytes per second, either reading or writing.

Note that *The VMEbus Specification* permits a wide range of performance. Therefore, the performance obtained with the TC/FPV depends significantly on the performance of the other device(s) on the VMEbus. For example, the VMEbus master interface performance was measured using a 200–nanosecond memory card on the VMEbus.

11.7.14 VMEbus Termination

The VMEbus Specification requires that the VMEbus be terminated at both ends by 194 ohms to +2.94 volts, or equivalent. To ensure signal integrity, a set of VMEbus terminating resistors resides on the TC/FPV. This leads to two constraints.

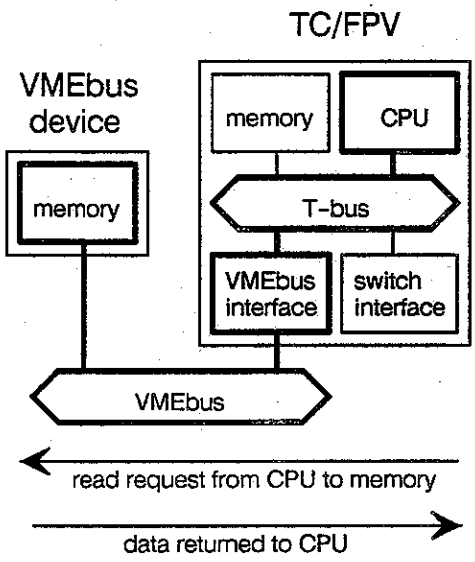
- First, the TC/FPV must reside in the first slot or the last slot of a VMEbus system.
- Second, there can be at most two TC/FPV boards in any one VMEbus system.

11.7.15 VMEbus Transaction Examples

This section explains the operation of the TC/FPV in more detail by describing four different types of transaction, step by step. The first is a read operation initiated by the 88100 CPU on the TC/FPV. The next two are write operations initiated by a VMEbus master, and the last is a read–modify–write operation initiated by a VMEbus master.

EXAMPLE 1: A read operation initiated by the CPU on the TC/FPV.

Figure 11–9 VMEbus example 1.



From the point of view of the CPU on the TC/FPV, reading from the VMEbus is no different from reading an ordinary memory location. The hardware interactions proceed as follows.

1. The virtual address emitted by the CPU is checked for validity and translated to a TC2000 Physical Address by the CMMU. The CMMU passes the translated address on to the CPU interface to the T–bus, where the 32–bit Physical Address is translated to the 34–bit System Physical Address used on the T–bus.
2. The CPU interface gains T–bus mastership and drives the T–bus address and control lines to indicate a read operation.
3. The VMEbus master interface notices that the address is within the VMEbus window, and responds with a "promise" signal, freeing the T–bus for other transactions while the VMEbus read operation is in progress. ("promise" signals cannot be queued. The *T–bus Specification* says that if a "promise" is already pending, the master must respond to further requests with "refused".)
4. The VMEbus requester module obtains mastership of the VMEbus.
5. The VMEbus master interface drives the VMEbus address and control lines to indicate a read operation.

   At this point, two timers begin running — the VMEbus TC/FPV Master Bus Timer and the VMEbus System Bus Timer. When the VMEbus slave later acknowledges the read operation (by asserting the VMEbus

signal DTACK), both timers halt. It is intended that the VMEbus TC/FPV Master Bus Timer's setting be shorter than the VMEbus System Bus Timer's setting.

6. The target of the read operation acknowledges the read and returns a value.

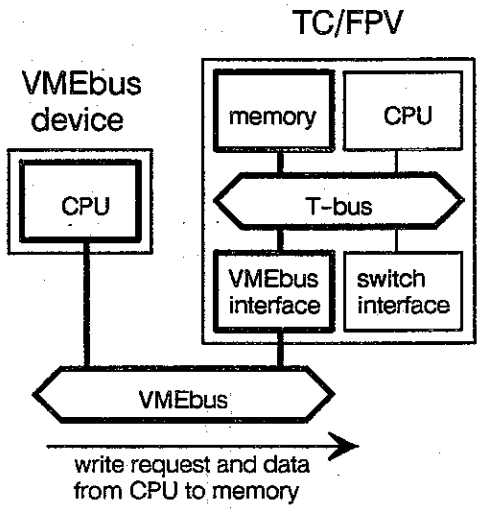
   If either of the two timers expires before the VMEbus slave responds, the VMEbus interface declares a bus error on the VMEbus by asserting the VMEbus signal *BERR*. The error condition is sensed and conveyed by the VMEbus master interface from the VMEbus over the T-bus to the CPU.

7. The VMEbus master gains T-bus mastership and drives the T-bus address and control lines to indicate a "function response" operation.

8. The CPU interface accepts the data and passes it on to the CMMU.

EXAMPLE 2: A write operation initiated by a VMEbus master, addressing memory on *this* TC/FPV.

Figure 11–10 VMEbus example 2.

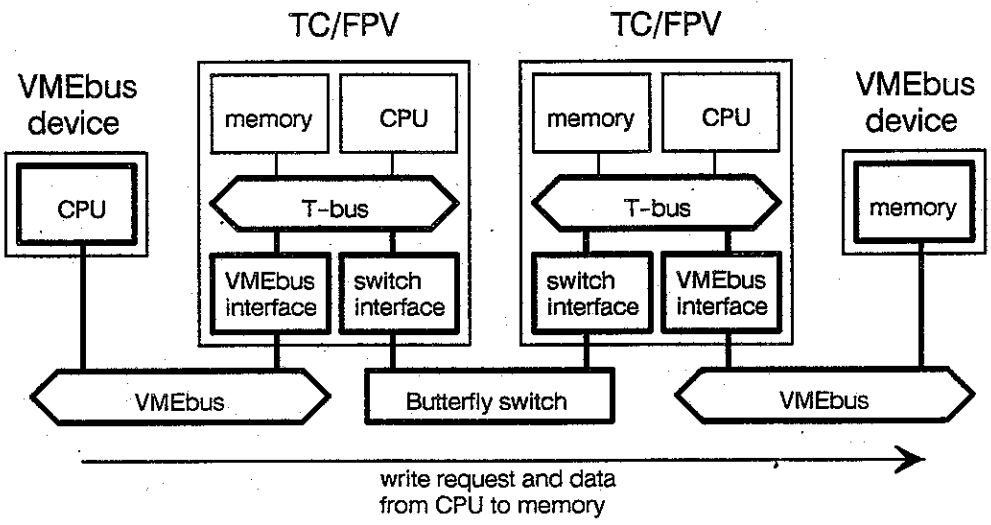


From the point of view of a VMEbus master, writing to TC/FPV memory is no different from writing to an ordinary VMEbus location.

1. The VMEbus requester module on the VMEbus device obtains mastership of the VMEbus.
2. The VMEbus slave on the TC/FPV recognizes that the VMEbus address falls within the slave's window, transforms the address, gains T-bus mastership, and drives the T-bus address and control lines to indicate a write operation. If the VMEbus slave on the TC/FPV does not respond soon enough, the VMEbus System Bus Timer will expire, generating a VMEbus error, that will cause the master to abort the transaction.
3. The TC/FPV memory accepts the data and acknowledges the write operation.
4. The VMEbus slave on the TC/FPV completes the transaction by acknowledging the write operation to the VMEbus master.

EXAMPLE 3: A write operation initiated by a VMEbus master, addressing VMEbus memory connected to a *remote* TC/FPV.

Figure 11-11 VMEbus example 3.



Among these examples, the interaction that involves the largest number of components is triggered by writing from a VMEbus master to a memory location on a VMEbus connected to a remote TC/FPV.

1. The VMEbus requester module on the VMEbus device obtains mastership of the VMEbus.
2. The VMEbus slave on the TC/FPV sees its window accessed, transforms the address, gains T-bus mastership, and drives the T-bus address and control lines to indicate a write operation.
3. The target is a remote processor node, so the Switch Interface Gate Array (SIGA) accepts the request and responds with a "promise" signal, freeing the T-bus for other transactions while the VMEbus write operation is in progress. See the *T-bus Specification* for the response of the VMEbus slave to other possible replies, such as "refused_locked". In general, the VMEbus slave either waits or aborts the transaction by signalling a bus error to the VMEbus master.
4. The SIGA sends a message containing the write address and data across the switch to the target TC/FPV.
5. The SIGA on the remote TC/FPV receives the message, gains T-bus mastership and drives the T-bus address and control lines to indicate a write operation.

6. The VMEbus master interface notices that the address is within its window. It accepts the write data and responds with a "promise" signal, freeing the T-bus for other transactions while the VMEbus write operation is in progress.

7. The VMEbus requester module obtains mastership of the VMEbus.

8. The VMEbus master interface drives the VMEbus address and control lines to indicate a write operation.

9. The target of the write operation accepts the data and acknowledges the write.

   If the VMEbus slave does not respond before the VMEbus TC/FPV Master Bus Timer expires, the resulting bus error on the VMEbus causes the VMEbus interface to abort the transaction and signal a bus error. The error is passed back through the local T-bus, the switch, and the remote T-bus to the remote VMEbus slave, which signals a bus error to the requesting VMEbus device.

   In this example, several timers are running: a VMEbus System Bus Timer on each end, a VMEbus TC/FPV Master Bus Timer in the second TC/FPV, and a connection timer in the first TC/FPV's requester SIGA. Expiration of any of these timers will be seen by the requesting VMEbus device as a bus error.

10. The VMEbus master gains T-bus mastership and drives the T-bus address and control lines to indicate a "function response" operation.

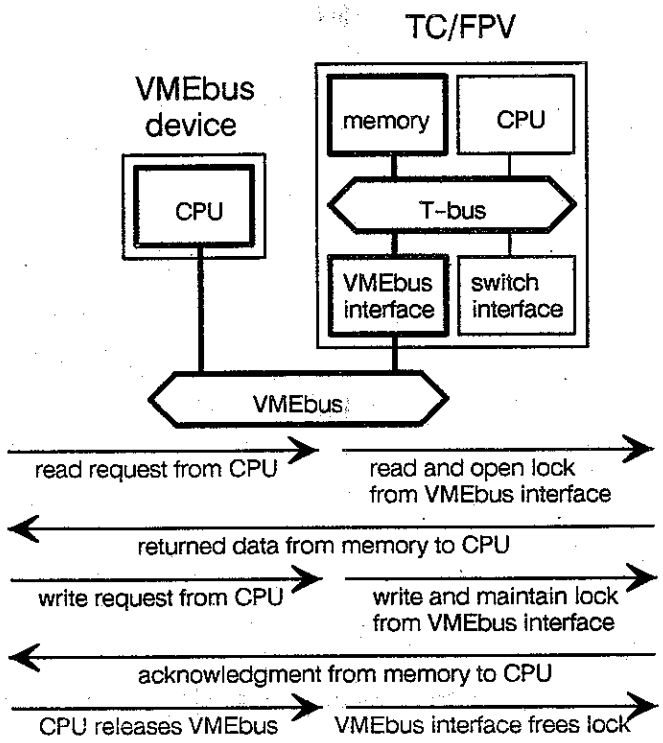
11. The SIGA passes the acknowledgement back to the SIGA on the requesting TC/FPV and closes the switch connection.

12. The SIGA on the requesting TC/FPV gains T-bus mastership and drives the T-bus address and control lines to indicate a "function response" operation.

13. The VMEbus slave completes the transaction by acknowledging the write operation.

EXAMPLE 4: A read–modify–write operation initiated by a VMEbus master.

Figure 11–12 VMEbus example 4.



Support for read–modify–write transactions is enabled by setting the VMEbus slave mapper so that every transaction locks the target by using the TC2000 locking protocol. Thus, a read–modify–write transaction proceeds as follows.

1. The VMEbus device initiates the read operation.
2. The VMEbus slave on the TC/FPV transforms the address, gains T–bus mastership, and drives the T–bus address and control lines to indicate a "read and open lock" operation.
3. The target of the transaction returns a value and puts itself into the locked state.
4. When the read completes, the VMEbus slave on the TC/FPV returns the value to the VMEbus master.

   If the read does not complete, a bus error will be signalled on the VMEbus. This will tell the VMEbus master to terminate the request by releasing *address strobe*.

5. When the VMEbus master acknowledges receipt of the value, it also indicates that this is a read–modify–write operation by holding the VMEbus and asserting the value to be written along with the necessary control bits. Up to this point, the TC/FPV has no way of telling whether this is an ordinary read or the beginning of a read–modify–write. The TC/FPV must lock the target memory and keep it locked until this point in the transaction.

6. The VMEbus device initiates the write operation.

7. The VMEbus slave on the TC/FPV gains T–bus mastership and drives the T–bus address and control lines to indicate a "write and maintain lock" operation.

8. When the write completes, the VMEbus slave on the TC/FPV acknowledges the VMEbus master.

   If the write does not complete, a bus error will be signalled on the VMEbus. This will tell the VMEbus master to terminate the request by releasing *address strobe*.

9. The VMEbus master terminates the transaction by releasing the VMEbus *address strobe* signal. When the VMEbus slave interface detects the release of *address strobe*, it generates a T–bus FREE_LOCKS cycle, thus unlocking the target memory module. If either the read or the write failed, the FREE_LOCKS cycle is still produced by the release of *address strobe* as noted above.

If the transaction is an ordinary read or write, and not a read–modify–write, steps 1–3 are the same. In step 4, the VMEbus slave completes the transaction by asserting the T–bus FREE_LOCKS signal as soon as it discovers, by seeing the VMEbus address strobe signal fall, that the transaction is not a read–modify–write.

NOTE

VMEbus DEVICE CANNOT HOLD LOCK LONGER

This example showed a simple read–modify–write operation. A VMEbus device cannot keep the memory locked for additional cycles by holding the VMEbus (asserting *address strobe*), for two reasons. First and most basically, the *address strobe* signal must be de–asserted between cycles, and this automatically frees the lock. The TC/FPV VMEbus slave interface does not respond to sequential ("block") transfer requests. Second, VMEbus timers and switch connection timers (if any are involved) limit how long the transaction may last.

11.8 Augmentation

Augmentation is a feature of the TC2000 architecture that allows each CPU in the system to perform some operations auxiliary to normal functionality. The primary goal of augmentation is to provide a mechanism for performing atomic operations in the multiprocessor environment of the TC2000, where many processors can access the same memory. In addition, augmentation permits the implementation of some features, unique to the TC2000 machine, that are not supported by the 88100.

An augmented memory reference results in the occurrence of some operation in addition to the reference. In particular, augmenting a read or write operation with the TC2000 locking protocol gives the CPU exclusive access to the target memory module until it releases the memory by clearing the lock augmentation. A mechanism of this type would not be required in a uniprocessor environment, where only one CPU initiates transfers to the memory.

The Augmentation Register (AR) controls augmentations. The AR entry in the register summary section shows the layout of the Augmentation Register and discusses different methods for accessing it.

11.8.1 Disabling Interrupts

Interrupts to the processor are disabled by setting the disable interrupts bit in the AR. Setting this bit prevents the interrupt request input to the processor from being asserted. Disabling interrupts is a key step toward maintaining atomicity. This is discussed further below.

Interrupts are serviced by the processor when it is between instructions. Unfortunately, disabling the interrupt request input to the CPU does not guarantee that previously pending interrupts will not be serviced, because there is an internal interrupt request pipeline. For example, if a device is requesting an interrupt, and interrupts become disabled, the interrupt request will no longer be asserted at the processor interrupt request pin. However, the processor may still see and service the interrupt because the request was latched internally before the pin was de-asserted. If interrupts are disabled with a read instruction, and an interrupt has been latched internally, it will be serviced immediately following the read and before any subsequent instructions can be executed.

Thus, disabling interrupts via augmentation works in concert with an atomic sequence as follows.

1. The program sets the AR *disable interrupts* bit to "1".
2. Now, before the next instruction, any interrupt request latched in the CPU's internal pipeline will be serviced.

3. The program now performs whatever atomic sequence is needed. (The AR *lock* bit must be set here and not in step 1, to avoid interactions between locking and the possible handling of interrupts in step 2.)
4. When the atomic sequence is completed, the program loads the AR with a new value that terminates the TC2000 locking protocol and re-enables interrupts.

There are two important differences between the disable interrupts *augmentation* and the disable interrupts bit in the 88100 processor status word. First, the disable interrupts augmentation does not affect the function of the Non-maskable Interprocessor Interrupt register. This provides a mechanism for a debugger or error handler to gain the attention of a CPU without resetting it, even when it is stuck with interrupts disabled by augmentation. Second, the disable interrupts augmentation is subject to the Interrupts Disabled Timer. This timer helps guarantee an upper bound on interrupt latency, and is therefore a useful tool in time-critical applications.

11.8.2 Lock Augmentation

Setting the *lock* bit in the AR causes the memory module that is next accessed to be held dedicated to this CPU after the access has completed. And if the switch is used to make the access, the switch path to that memory module is also held open.

The first augmentable reference with the lock augmentation enabled turns into an OPEN operation on the T-bus. Subsequent references turn into MAINTAIN T-bus accesses.

A transition from one to zero of the AR *lock* bit automatically generates a T-bus FREE_LOCKS cycle. This allows clearing the AR *lock* bit and generating the FREE_LOCKS to be performed in a single CPU instruction.

Refer to the *T-bus Specification* for a complete description of the TC2000 locking protocol.

NOTE

HOW LOCKS ARE FREED
If the *lock* bit in the AR is "1", and the CPU locks a resource, the CPU interface remembers this. Then, when the *lock* bit is cleared to "0", the CPU interface automatically generates a FREE_LOCKS cycle on the local T-bus. If a remote resource was locked through a switch connection, the local SIGA knows this and, seeing the FREE_LOCKS on the local T-bus, propagates the drop-lock condition over the switch to the remote node, where the remote SIGA generates a FREE_LOCKS cycle on the remote T-bus, freeing the locked resource. The CPU cannot explicitly generate a FREE_LOCKS cycle, only implicitly as a side effect of clearing the *lock* bit in the AR after having locked a resource, or as a side effect of the XMEM instruction (described above). If no resource is locked by the CPU, setting and clearing the *lock* bit in the AR does not generate a FREE_LOCKS cycle.

11.8.3 Bypassing Locks

The TC2000 locking protocol affects only a subset of the references generated by the CPU. Since it is never useful to lock out instruction fetches, these accesses ignore the locking protocol. In addition, augmenting certain data references, such as references associated with exception processing in the CPU, would result in incorrect system behavior and must be prevented. The architecture therefore supports a mechanism that suppresses the locking protocol when data resides in *bypassed* address space. A data reference is to bypassed memory if the *bypass* bit in its CPU Mapping RAM entry is set. Locations in the 8-megabyte block specified by such CPU Mapping RAM entries are said to be in bypassed address space.

Figure 11-13 lists the types of data that should reside in bypassed address space. The hardware automatically bypasses the lock augmentation on instruction fetches, and on page descriptor fetches made by a CMMU. The segment descriptor fetch, however, is not automatically bypassed; system software must map the memory containing segment descriptors, as fetched by the CMMUs, bypassed.

Figure 11-13 Types of data references that should be mapped bypassed.

Instruction fetches (automatically bypassed)
MMU page table walks (partly automatically bypassed)
Exception vector fetches
Supervisor stack references
Configuration and control registers

The primary purpose of the bypass mechanism is to let the CPU access instructions and page tables in its local memory, even if someone has the local memory locked. Normally, references to remote memory do not bypass locks.

If the *bypass* bit in the selected CMR entry is "1" (indicating a bypassed reference), then the T-bus signal T_LOCKOP = *bypassed* is asserted (regardless of the setting of the AR). Otherwise, the AR is used. In particular, the AR *lock* bit is OR'ed with the CPU's $\overline{\text{DLOCK}}$ bit (meaning an XMEM instruction is occurring), and if the OR is true, T_LOCKOP = *open* or *maintain* is asserted, making it a locked reference. If the OR is false, the reference is a normal one and T_LOCKOP = *normal* is asserted.

While our discussion of bypassing locks focuses on the CPU interface, the VMEbus can also bypass locks, when the *bypass* bit in the VMEbus Slave Map RAM is asserted.

11.9 The Process Configuration Register

The Process Configuration Register (PCR) contains information associated with the software process currently running on the processor. The fields of the PCR control priority, path and "synchronized access" mechanisms described in the sections below.

11.9.1 Switch Priority and Priority Schemes

The intent of the priority mechanism is to bound switch latency by ensuring that each message eventually gets through the switch, despite contention from other switch traffic. The paragraph below describes this mechanism in general terms. The SIGA and SGA specifications describe the operation of this mechanism in detail. Although two priority bits are supplied on the T-bus, manipulated by the SIGA, and contained in switch message headers, the SGA implementation uses only one of the two. Consequently, there are functionally just two priorities: low (normal), and high. Messages with high priority are called ***express*** messages and receive preferential treatment in the switch.

The SIGA copies the T_PRIORITY bits from the T-bus, and places them in the switch message header. The SIGA, during a regularly occurring interval called its priority time slot, promotes (modifies) the priority bits in retransmitted messages, forcing them to high priority. A message encountering contention is eventually promoted to high priority, contends more successfully for SGA output ports on its path, and reaches its destination. This mechanism places an upper bound on the time that a message takes to establish a connection and traverse the switch.

The definition of T-bus T_PRIORITY bits is:

0 0 = high
0 1 = low (normal)

1 0 = illegal
1 1 = illegal

The priority of messages can be modified and used automatically by the switch hardware. **The recommended, simplest use of the priority mechanism by software is to send all messages at low priority.** To do otherwise can interfere with the hardware's use of priority to bound latency. However, the TC/FPV is designed to permit software specification of message priority. This flexibility permits other, application-driven uses of the priority mechanism.

Each master on the T-bus must drive the priority bits. The initial priority of the messages sent can be set independently in each master interface, as described below.

In the VMEbus slave/T-bus master interface, bits in each VMEbus Slave Map RAM register specify the priority bits placed on the T-bus when that register is used to map an access from the VMEbus into the TC2000 address space.

The switch server interface generates only local T-bus transactions, never ones that go back out to the switch, so its setting of T-bus priority bits is of no concern.

In the CPU interface, the priority bits are computed from fields of the Process Configuration register and from properties of the access being made. The PCR *priority scheme* field selects one of four possible algorithms for this computation. Each algorithm may make use of the PCR *default priority* field and properties of the access (such as read/write and single-word/burst). **In accordance with the recommendation above, the PCR *priority scheme* field should be set to "00", which selects low priority for all types of message.** With this setting, the PCR *default priority* field is irrelevant, but using "00" is suggested.

11.9.2 The Path Bits

The two T_PATH bits on the T-bus differentiate local accesses from switch accesses; this permits local T-bus slaves to detect accesses to themselves without decoding the upper bits of the address. The T_PATH bits also select between the primary and backup switches for remote references. "11" indicates local access, "10" means SIGA A, and "01" means SIGA B. ("00" is illegal.)

The CPU interface computes the value of the T_PATH bits from four sources:

- The Process Configuration register (PCR) two *path* bits
- The selected CPU Mapping RAM (CMR) entry's *local* bit
- The selected CPU Mapping RAM (CMR) entry's *interleave enable* bit
- The selected Interleave Decision RAM entry's single bit

The computation is as follows:

If ((CMR *local* bit is "1")
AND NOT ((CMR *interleave enable* bit is "1")
AND (Interleave Decision RAM bit is "1")))
then T_PATH = "11";
otherwise T_PATH = PCR *path* bits.

In words, T_PATH is specified by the PCR *path* bits unless the reference is local and not interleaved, in which case T_PATH is forced to "11" (local).

The PCR *path* bits are used by the CPU interface only. The switch server interface generates only local T-bus transactions (T_PATH = 11). In the VMEbus slave/T-bus master interface, bits in each VMEbus Slave Map RAM register specify the T_PATH bits placed on the T-bus when that register is used to map an access from the VMEbus into the TC2000 address space.

11.9.3 Synchronized Access

Remote accesses can be attempted less frequently ("backed off") after the switch interface has received a reject. This throttles the retry rate and achieves a better pattern of switch traffic than would be attained if retries were allowed at any time. The SIGA has a very general mechanism for specifying the initial transmission and retry criterion for each message class. This mechanism is described in detail in the SIGA specification.

Setting the *synchronized access* bit of the PCR causes subsequent references to assert the T-bus signal T_SYNC. This indicates to the SIGA that the current access should use the "Slot0" transmission strategy. When the SIGA is appropriately set up by system software, selecting "Slot0" can produce the effect of attempting the current access using the backoff criterion on the *initial* transmission, rather than waiting until a reject has occurred to back off.

11.10 The CPU Mapping RAM

The CPU interface contains the CPU Mapping RAM (CMR), a set of registers whose main purpose is to translate addresses as described in section 11.10.1. The CMR also generates access control signals listed in section 11.10.2.

11.10.1 CMR Address Translation

The series of translations that convert an 88100 virtual address (Process Logical Address) to a System Physical Address is shown in Figure 11-14. The purpose of the CPU interface address translation is to convert the 32-bit Physical Address generated by the 88200 CMMU to a 34-bit System Physical Address. The TC/FPV uses the CPU Mapping RAM to implement this translation, shown in Figure 11-15.

Figure 11–14 Address translation.

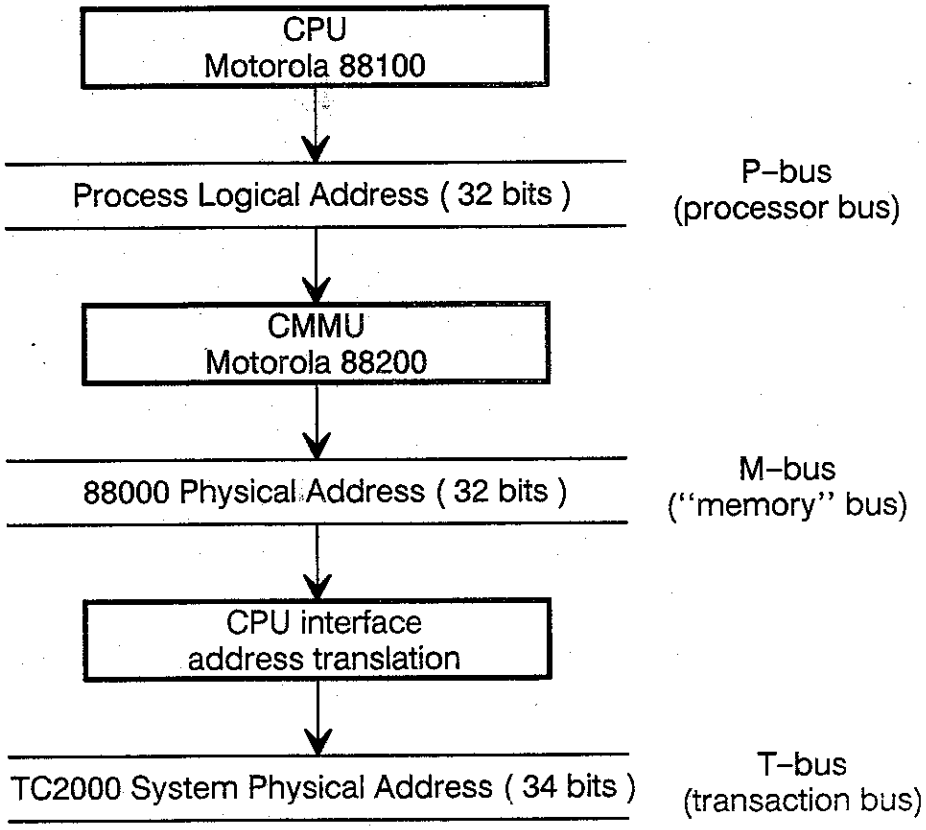
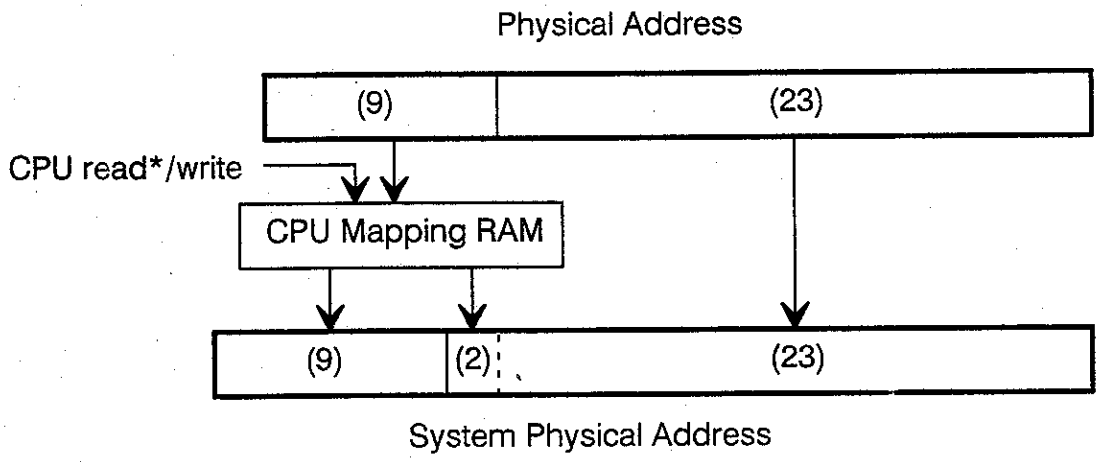


Figure 11–15 Physical Address to System Physical Address.



The CMR takes in the high nine bits of the Physical Address generated by the CMMU (and one other bit) and uses them to generate the high eleven bits of the System Physical Address. These 11 bits select one of the 2048 8–megabyte "banks" of address space within the machine.

The additional bit used to select a CMR entry is whether the CPU is performing a read or write operation. This allows separate mapping for read access versus write access to the same Physical Address. (Certain CMR bits do not distinguish between reads and writes, as noted in the CMR description in section 11.17.)

Note that it is possible to build systems with more memory than the 88000 chip set can directly address. The following observations regard the memory that lies outside the 4–gigabyte range that is directly accessible to any given TC/FPV CPU.

1. There is no hardware restriction against setting the CPU Mapping RAM differently on different function boards. Thus it can include mapping of private copies of code and data. nX kernel code that must be replicated on every processor function board is one example. Programs with supervisor privileges (or access to appropriate system calls) can manipulate the mapping RAM to access private memory on remote function boards for diagnostics, initialization, debugging, and other purposes. This flexibility could also be used to allocate different subsets of physical memory to different clusters.
2. Other function boards than the TC/FPV need not have the 32–bit limitation of the 88000. For example, an array processor or high–throughput I/O function board could be built to access a 16–gigabyte memory space.

11.10.2 Control Bits in the CMR

In addition to translating addresses, the CMR also produces certain control signals that describe the kind of access being made. The bits that control these signals are:

- *local* — Indicates that the reference is to local rather than remote memory (or other resource). Discussed in section 11.3.2.
- *bypass* — Indicates that the reference bypasses the TC2000 locking protocol. Discussed in section 11.8.3.
- *interleave enable* — Indicates that the reference is to interleaved memory. Discussed in section 11.3.6.
- *fast path disable* — Indicates that the reference will not use the CPU's fast path to memory. Discussed in section 11.3.1.
- *intercept access** — Indicates that the reference will be terminated and acknowledged as if it had succeeded. (The * is part of the name of the bit, indicating it is low true.) Discussed in section 11.10.3.

Besides the sections referenced above, the description of the CMR in section 11.17 discusses all aspects of the CMR in detail.

11.10.3 CMR Intercept Access Mechanism

A mechanism called **intercept access** is controlled by the CPU Mapping RAM. The intended use of this mechanism is to reduce the number of switch accesses required when transferring blocks of data from one function board to another. Such use may enhance performance and reduce the asymmetrical nature of pushing versus pulling blocks of data around the TC2000 machine.

The 88200 CMMU behavior around cache write misses when the page is marked copyback cacheable is as follows. When the processor misses the cache on a cacheable write (in copyback mode), the 88200 selects a cache line. (If none are available, it selects one for replacement and if necessary, copies it back into memory.) It then reads in the new line with the intent–to–modify bit set. (This read is to force any snooping master with dirty data to flush it, a non–issue in the case of the TC/FPV.) Next, it writes the datum to memory, and finally it writes the datum to the newly–read cache line. See the *MC88200 User's Manual* for more detail.

The effect of the 88200's behavior in the TC2000 computer is as follows. For simplicity, the description assumes that the data is quad–word aligned. In one common model for copying blocks of data around the TC2000 machine, a block of data is read into the cache, and then written to another cacheable location. The process of cache line replacement, or explicit flushing, causes the data to be written to memory. In the case of copying data from local memory to remote memory (**pushing**), the behavior described above results in three ***switch transactions*** for each line of data to be copied:

1. The initial read with intent–to–modify of the cache line
2. The write–once
3. The ultimate burst write when the line is flushed from the cache

On the other hand, when copying data from a remote function board to local memory (**pulling**), the data is read into the cache across the switch, and the burst read, write–once, and burst write are performed to ***local*** memory. Hence, only one switch reference is required. In either case, the burst read and the write–once are extraneous. New data is immediately copied over the data that has been read in, and the line is written to memory when it is selected for replacement, or when it is explicitly flushed, so the write–once is redundant.

When data is being pulled into local memory from remote memory, the performance penalty is small. However, when data is being pushed, the performance penalty is large enough to warrant a hardware assist. Using the intercept access mechanism, it is possible to suppress the two extraneous references when pushing data across the switch.

When the *intercept access** bit is asserted (cleared to "0"; the * indicates low true) in a CMR entry, support circuitry in the CPU interface causes any cycle mapped through the entry to get intercepted. The processor is acknowledged as if the cycle had completed, but no T-bus cycle is actually generated. The CMR has separate entries for read and write cycles, so reads and writes can be intercepted independently of each other.

When pushing data to remote memory, the *intercept access** bit should be asserted in the read entry for the destination function board. This reduces the number of switch references from three to two for each line in the pushing case, by intercepting the read-with-intent-to-modify cycle when the cached write occurs. In addition, if the destination line is read in (and intercepted locally) before the data is copied to it, then a cache miss will not occur when the data is written to the destination. This will prevent the write-once from occurring, reducing the number of switch references to one per cache line, regardless of whether the data is pushed or pulled through the switch.

11.11 CPU Memory Access Timing

Figure 11-16 shows the memory access time from a TC/FPV CPU under a variety of conditions. The accompanying notes are essential. Each value shown applies throughout its connected white space. These are generally best case (minimum) times, as detailed further in the notes.

Figure 11–16 Memory access time (microseconds).

Cache		Access Made by CPU			
Mode	**Activity**	**Read from Local**	**Write to Local**	**Read from Remote**	**Write to Remote**
inhibited	none	0.550	0.600	1.913	1.889
writethrough	hit	0.150			
copyback	hit				
writethrough	miss	0.850	1.200	2.529	4.168
copyback	miss with no writeback				
	miss with writeback to local	1.500	1.850	3.179	4.818
	miss with writeback to remote	2.905	3.255	4.534	6.173

Notes for Figure 11–16.

1. The operating frequency of the TC/FPV characterized here is 20.0 MHz.
2. The timing shown is the full latency including the CPU's "execute" phase (a Motorola term for instruction decoding) and "address" phase through completion of the access. For example, a read with a cache hit takes three cycles (execute, address, and successful reply), a total of 0.150 microseconds. The *bandwidth* may be greater than the reciprocal of the timing shown, since the execute and/or address phases may be pipelined with the processing of other instructions, depending on the mix of instructions. The maximum bandwidth is one access per cycle. Pipeline stalls caused by recent instructions that have not yet finished are possible during the execute and/or address phases; the timing shown assumes such stalls are absent. For further details, see the Motorola *MC88100 User's Manual.*
3. The fast path is assumed used where possible, namely in reads (either a burst or a single word) from non–interleaved local memory. Using the fast path reduces the access time by 3 cycles (0.150 microseconds).

4. The timing shown assumes that no page table walking is performed. If the CMMU must load memory mapping information to service the access, the time seen by the CPU is increased.

5. The intercept access mechanism is assumed not used. Its effect on timing is discussed later.

6. If the access is augmented with the TC2000 locking protocol, the access time in some cases may be reduced because, if a remote function board is referenced, locking holds the switch path to it open. Therefore, if switch transmissions were not immediate or if there was contention in the switch or at the destination port, only the initial locked reference would be delayed and not the subsequent references during the sequence. However, the timing shown assumes immediate transmission and no contention, so there is no effect due to locking on the timing shown. (The timing of VMEbus accesses, not discussed here, is affected by locking.)

7. The chart is intended to show data access timing, although the entries relevant to an instruction fetch are also valid for that. (In an instruction fetch, the times shown include the 1-cycle "prefetch" phase rather than the 1-cycle "execute" phase.) The M-bus ("memory" bus, the common output of the CMMUs) is assumed not occupied by another CMMU. (That is, occupied by one of the two instruction CMMUs when a data access is made, or by the data CMMU when an instruction fetch is made.) If the M-bus is occupied, the access time shown is increased by the time needed for that access to release the M-bus. (Note that there are separate instruction and data P-buses, so instruction and data accesses do not contend for the P-bus.)

8. The T-bus is assumed not occupied. If the T-bus is occupied, the access time shown is increased by the time needed for that access (and any other accesses with higher T-bus priority than the CPU) to complete. In the TC/FPV, the CPU has the lowest T-bus priority. Note that accesses to the VMEbus master interface and to the switch interface split cycles on the T-bus, releasing the T-bus for other use while the requested operation is performed.

9. Any memory module used in servicing the access is assumed not locked and idle. If it is locked (via the TC2000 locking protocol), the access time shown is increased by the time needed for the locked transaction to complete and the access to be retried. The hardware performs the retry automatically. (If it is a bypassed access, the timing is unaffected by the locked status of the memory module.) If the memory module is busy — with a refresh cycle or completing a write cycle — the access time shown is increased by the time needed for a T-bus REFUSED reply and a new T-bus arbitration. This is typically 2 cycles (0.100 microseconds).

10. When the access includes one or more references to a remote function board, the timing shown assumes there is no contention for the T-bus and memory on the remote function board.

11. The timing shown assumes the access is to non-interleaved memory. If any reference involved in serving the access is to interleaved memory, that reference is forced to go over the switch. If the reference would have gone over the switch anyway, the access time is not changed. If the reference would have been serviced entirely on-board, the access time is increased.

12. The Butterfly switch included in this characterization is a 2-column switch and operates at a clock frequency of 38.0 MHz.

13. The TC/FPV contribution to access timing was calculated by accounting for individual cycles of the board clock. The Butterfly switch contribution was calculated by a program that models the switch parameters set to their fastest settings. In particular, immediate transmission strategy is assumed. The timing shown would be increased, for example, by a strategy that delays before the first transmission. Such a strategy may be used to pace accesses made to a software spin lock.

14. The Butterfly switch timing assumes no switch contention. That is, there is no contention for the local SIGA, for switch ports within the switch, or for the switch port at the destination function board. If contention is present, the access time shown is increased.

15. The times for specific switch transactions included in the access timing chart are as follows:

transaction	microseconds
1-word read	1.337
1-word write	1.363
4-word read	1.953
4-word write	1.729
synchronizer uncertainty	0.152 (see note)

16. Synchronizer uncertainty:
The TC/FPV is clocked at a different frequency than the Butterfly switch. Therefore, each time data enters or leaves the switch, it passes through a synchronizer, a circuit that re-clocks it to the new environment. The delay at a synchronizer varies from no delay up to one cycle at the new clock frequency. Each normal switch access passes through four synchronizers — the request goes into and out of the switch, and the reply does likewise. The "synchronizer uncertainty" shown above is the maximum total delay for all four synchronizations. The computation of access timing includes one half the maximum delay, assuming that the delay is uniformly distributed and therefore is, on the average, half the maximum. Each switch reference incurs this synchronizer delay. For example, if the CPU's access requires three switch references, the access time shown includes three times half the synchronizer uncertainty.

11.11.1 Intercept Access and Timing

The intended use of the intercept access mechanism is to speed up the copying of blocks of data. In this application, one typical case of its use is as follows:

- CPU access is a write to remote memory
- Cache mode is copyback
- Cache activity is a miss with writeback to remote memory
- CMR is set up to intercept the read (The data read in will be immediately overwritten, so it can safely be intercepted.)

In this case, the intercept access mechanism reduces the access time seen by the CPU from 6.173 microseconds to 4.344 microseconds, a savings of about 30 percent.

The intended use of the intercept access mechanism is to intercept only reads. Nevertheless, writes also may be intercepted. In each CMR entry, the bit that controls the intercept access mechanism does distinguish between read and write references. Further, the writeback of a cache line may use a separate CMR entry than that used by the location the CPU accesses. Each of up to three memory references (writeback, read, write) resulting from one CPU access may be independently intercepted or not. Therefore, a complete analysis of the mechanism's effect on all cases of timing is somewhat tedious, is not particularly enlightening, and is omitted here in favor of the example above.

11.12 Timers

A variety of timers are available in the TC/FPV. The timers are provided to guarantee maximum interrupt latency, to prevent inappropriate use of the lock and the disable interrupts augmentations, to be used as a software tool, and to detect faulty hardware. Figure 11-17 is a table of the timers available on the TC/FPV, the range of each timer, a brief description of its function, and the action taken if the timer expires. Each timer is described below.

Figure 11-17 Timers in the TC/FPV.

Name and Range	Purpose	Action on Expiration
	LOCK AND INTERRUPT TIMERS	
CPU Lock Timer 1 - 255 microseconds	Limit how long the CPU may hold a lock.	Generate a FREE_LOCKS cycle. CPU will later get a "maintain present" error.
Interrupts Disabled Timer 1 - 255 microseconds	Help guarantee maximum interrupt service latency.	Interrupt.
Interrupts Pending / Abort Retries Timer 1 - 255 microseconds *	Help guarantee maximum interrupt service latency.	Signal SIGA to abort retries in case connection establishment is in progress. CPU gets bus error if retries are aborted.
	SWITCH PROTOCOL TIMERS	
Reject Timer 1 microsecond - 0.49 seconds *	Prevent SIGA from trying too long to establish a connection.	Bus error.
Connection Timer 1 - 255 microseconds *	Prevent switch connection from being held open too long.	Tear down connection. CPU gets bus error — code and timing depend on when timer expires.
	REAL TIME CLOCK TIMERS	
Time Of Next Interrupt - A (TONI-A) 1 microsecond - 1 hour *	Allow software to ask for an interrupt at a specified time.	Interrupt.
Time Of Next Interrupt - B (TONI-B) 1 microsecond - 1 hour *	Allow software to ask for an interrupt at a specified time.	Interrupt.
	VMEbus INTERFACE TIMERS	
VMEbus Arbiter Timer 4 - 1020 microseconds	Limit how long VMEbus bus grant may be asserted without bus busy.	Arbiter removes bus grant.
VMEbus TC/FPV Master Bus Timer 1 - 255 microseconds	Limit how long the TC/FPV as VMEbus master may await a response from a slave.	Assert VMEbus signal BERR.
VMEbus System Bus Timer 4 - 1020 microseconds	Limit how long any VMEbus master may await a response from a slave.	Assert VMEbus signal BERR.

* Also, these timers can be disabled by software.

11.12.1 Switch Reject Timer

The Switch Interface Gate Array (SIGA) implements a timer, called the switch reject timer, that limits the maximum time spent trying to establish a path through the Butterfly switch. The switch reject timer is enabled whenever a remote memory access is attempted. The SIGA samples the state of this timer whenever it receives a reject. If the timer has expired, the SIGA stops trying to establish the connection, and a bus error is returned to the CPU. In normal operation, the switch reject timer should not time out. Its expiration may indicate broken hardware, or attempting to access a non–existent or disconnected switch port.

11.12.2 Switch Connection Timer

The SIGA contains a switch connection timer that limits how long a connection may be held. Each time the SIGA sends the initial message to establish a connection, the switch connection timer is enabled. If that initial message is rejected, the SIGA retries the message later and restarts the timer upon each retry. Expiration of the switch connection timer results in a bus error. Two different error codes are returned, indicating whether the SIGA was waiting for a response from the server end of the connection when the timer expired (wait timeout) or not (idle timeout).

The switch connection timer helps enforce an upper bound on maximum switch latency by limiting the amount of time that a switch path can be held.

11.12.3 Lock Timer

This timer oversees the TC2000 locking protocol. Every T–bus master that generates a lock requires a timer to prevent it from keeping a slave locked too long. The TC/FPV has three masters that can generate locks: the CPU interface, the SIGA server, and the VMEbus slave/T–bus master interface. The CPU interface has a timer that is enabled when the slave is locked and disabled when the lock is freed. If the timer expires, a FREE_LOCKS cycle is generated, telling the slave it is freed.

However, expiration of the lock timer does **not** clear the lock bit in the T–bus master of the CPU interface. Thus, the CPU still considers itself to hold the slave locked. If the CPU does not clear the lock bit before making another reference to that slave, the CPU interface will make a MAINTAIN access, because it appears to be locked. The slave will return an error because it is no longer locked. To avoid this, the programmer must clear the *lock* bit in the Augmentation register in conjunction with servicing the lock timer timeout.

The SIGA server does not have a lock timer because the switch connection timer serves the same purpose.

The VMEbus slave does not have a lock timer because VMEbus locked accesses should always complete in two cycles, as follows. The VMEbus slave interface produces a locked access if the *lock T–bus* bit is "1" in the VMEbus Slave Map RAM entry used by the access. The locked sequence lasts as long as the VMEbus *address strobe* signal is asserted. That signal may be asserted for more than a single access in two cases. First, *address strobe* is asserted during a read–modify–write operation to achieve atomicity, and that atomicity is preserved by using the TC2000 locking protocol. Second, *address strobe* is asserted during a VMEbus sequential transfer (block transfer), but the TC/FPV VMEbus slave does not respond to these.

11.12.4 Interrupts Pending / Abort Retries Timer

The time to establish a path through the Butterfly switch to a remote memory can occasionally be large. When a CPU executes an instruction that makes a reference to remote memory, the instruction does not complete until the path has been established through the switch, and the data has been read or written. Since interrupts are serviced only between CPU instructions, they could be ignored for longer than the maximum permitted delay to service interrupts, if the time to establish the switch path was too long. The switch reject timer could address this problem, but it would impose conflicting constraints on the duration of this timer. For detecting broken or missing hardware, the switch reject timer should be set to a long timeout period. To guarantee maximum interrupt latency, however, the switch reject timer would have to be set to a relatively short (microseconds range) time. The "siga_abort_retries" input to the SIGA is used to address this problem.

When an interrupt is being requested of the CPU, the interrupts pending / abort retries timer starts running. This timer waits for a programmable period of time, and then asserts the "siga_abort_retries" signal if any interrupt is still pending. The siga_abort_retries signal is asserted whenever the timer expires (even if the CPU is not currently waiting for an access to complete). The time period can be specified from 1 to 255 microseconds, or infinite. Setting the time to infinite disables the siga_abort_retries mechanism.

If the siga_abort_retries signal is asserted when the SIGA has received a reject, the attempt to open the switch path (retry) is aborted and a bus error is returned to the CPU. The interrupt can then be serviced, as the CPU is no longer in the middle of an instruction. Thus, this mechanism decouples the maximum switch latency from the maximum latency to service an interrupt.

11.12.5 Interrupts Disabled Timer

The interrupts disabled timer is enabled when interrupts to the processor are disabled by the disable interrupts augmentation. This timer can be set by software from 1 to 255 microseconds. This timer is used to ensure that interrupts

are re-enabled soon enough that the maximum interrupt latency can be guaranteed.

When the interrupts disabled timer expires, an interrupt is generated to the CPU. When this happens, the interrupt handler should clear the AR. Any locks associated with the augmentation are freed by the T-bus FREE_LOCKS cycle that automatically occurs when the *lock* bit changes from one to zero. This timer has no effect when the IND (interrupt disable) bit in the 88100 Processor Status Register is set, and setting the IND bit does not enable this timer.

11.12.6 Real Time Clock Interrupt

Interrupts based on the Real Time Clock are generated by a mechanism described in section 11.13.

11.12.7 VMEbus Interface Timers

The three timers that are part of the VMEbus interface are described in section 11.7.8.

11.13 Interrupts

The 88100 CPU has only one interrupt line, so the TC/FPV circuitry combines the various sources of interrupt into one signal applied to the 88100. Further, the TC/FPV interrupt system provides information to the CPU allowing it to determine what interrupt source(s) are currently asserted, and mechanisms for dismissing and/or disabling certain interrupts.

The sources of interrupt to the TC/FPV CPU are listed below. Those marked with ✣ cannot be disabled except by the CPU disabling its internal interrupt system.

- Interrupts disabled too long (asserted by Interrupts Disabled Timer register) ✣
- Non-maskable interprocessor interrupt (asserted by Non-maskable Interprocessor Interrupt register) ✣
- VMEbus interrupts (any one of seven levels; asserted by VMEbus devices)
- Maskable interprocessor interrupt (asserted by Interprocessor Interrupt register)
- Two real-time timers (asserted by Time Of Next Interrupt registers in SIGAs)

The generation and servicing of VMEbus interrupts is covered in section 11.7.6.

Figure 11–18 shows the collection and processing of interrupts, and their visibility in the Interrupt Source register. The gating illustrated is conceptual, and does not necessarily reflect the gate level implementation in the hardware. Most of this processing is discussed fully in connection with the associated registers (see section 11.17). The real–time timers, however, deserve further discussion here.

Figure 11–18 TC/FPV interrupt derivation.

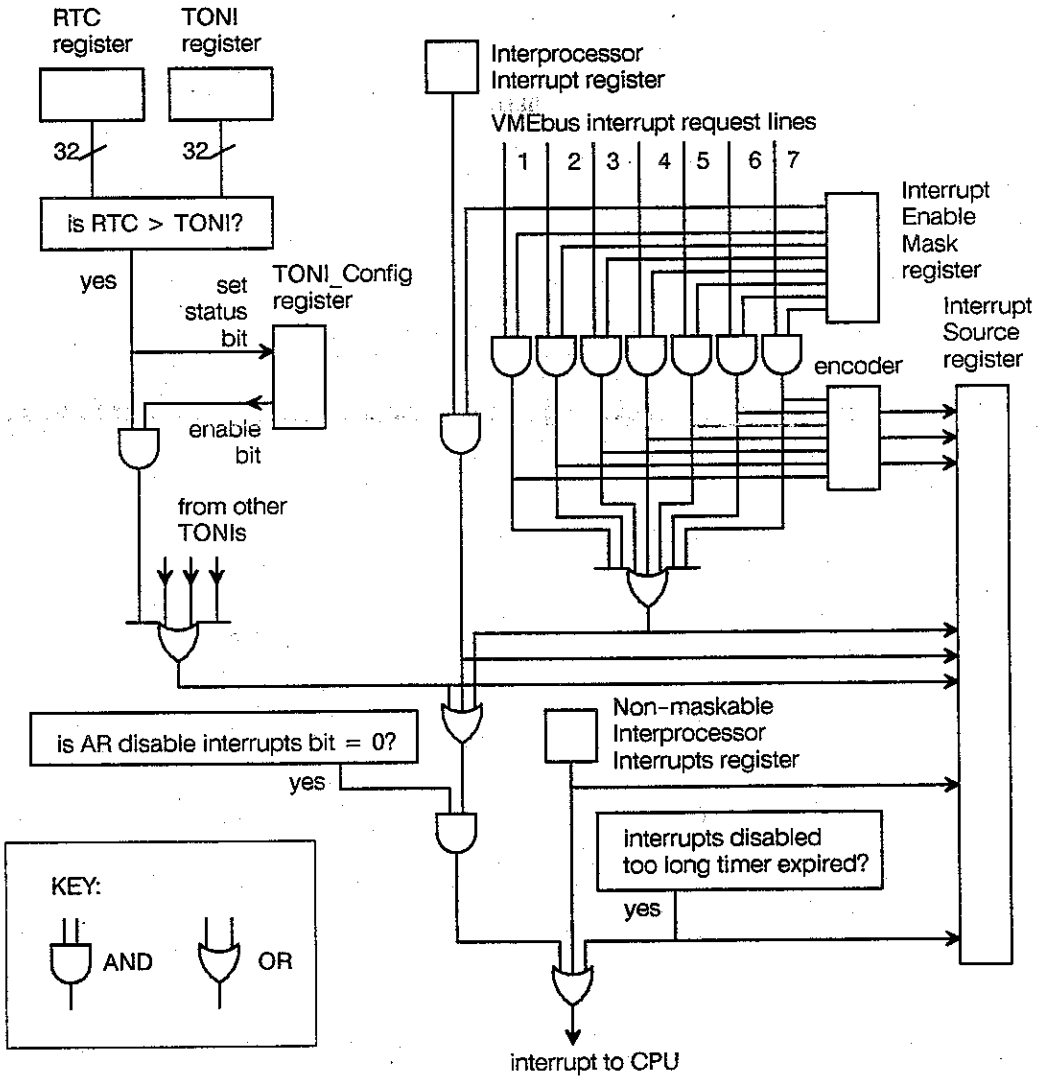
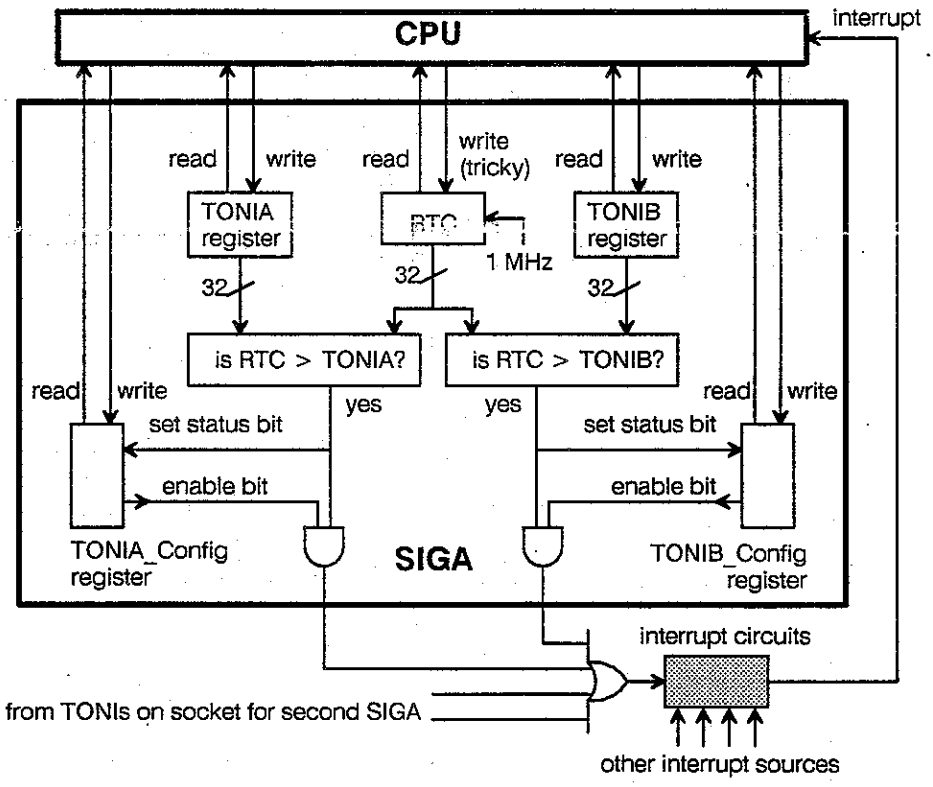


Figure 11–18 shows the derivation of the real-time timer interrupt signal for one of the four TONI registers in the TC/FPV. Figure 11–19 shows this programmable timer mechanism in greater detail. The TC/FPV has board space for two SIGAs, one for each of two possible switch interfaces. (Normally, only one interface is populated with chips, and if both were populated, only one would be in use at a time.) Within the SIGA are two independent Time Of Next Interrupt (TONI) systems. Each system consists of a TONI register, a comparator, a configuration register and an output signal. The SIGA compares the TONI register to its Real Time Clock (RTC) every microsecond. The result of the comparison sets the status bit in the associated TONI configuration register, and also is presented on a SIGA output pin if that configuration register's enable bit is set. The two signals, one from each TONI (and also the signals from the two TONIs on the normally unpopulated second SIGA), are OR'ed together to produce a signal that is true when at least one TONI timer has "gone off".

Figure 11–19 TC/FPV TONI mechanism.



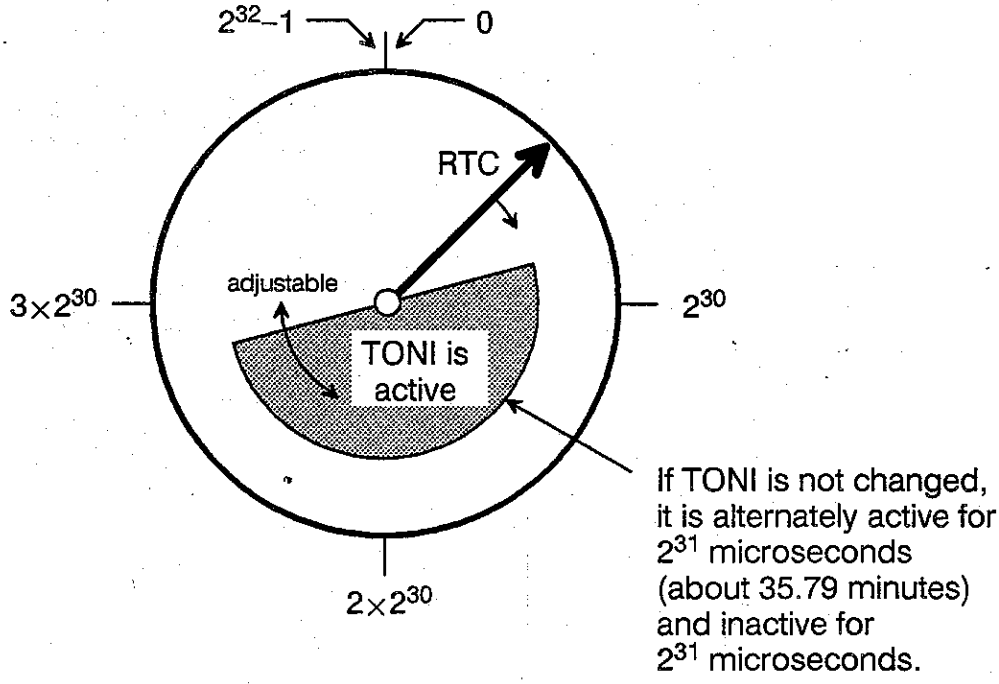
The RTC always counts at 1 megahertz, so some care is required in writing the RTC to avoid a race condition between the CPU write operation and the count. This and other details of the RTC and TONI system are discussed in the *SIGA Specification*.

The TONI mechanism is based on the ***real time clock***, and works ***like an alarm clock***, not like an interval timer. No periodic interrupts are generated, only an interrupt when the requested time is reached. For example, if an interrupt is desired in one second from the present time, the software can read the RTC, add one million (1–microsecond ticks), and write the sum into a TONI register. This operation, based on real time, may seem unusual to programmers familiar with interval–based timers. However, the TONI mechanism is much more convenient in the TC2000 multiprocessor environment, because all RTCs throughout the machine are synchronized. Therefore, a command to perform some action at time T can be passed from processor to processor, and acted upon by any and all processors easily, without concern for the passage of time while the command is distributed. In a purely interval–based environment difficult adjustments are required to even approximate this capability of coordinated action.

The TONI system asserts an interrupt request when the RTC is "greater than" the value in a TONI. In making this comparison, the RTC and TONI values are treated as 32–bit unsigned integers, and the subtraction is performed modulo 2^{32}. Therefore, once the TONI "alarm clock rings" — or more technically, "becomes active" — it stays active for half the wrap–around period of the RTC, regardless of the absolute values of the TONI and RTC. The RTC, being 32 bits and counting at 1 megahertz, wraps around in about 71 minutes. Therefore, the TONI stays active for about 35 minutes. This long active period insures that the active condition will be seen by a service routine, even if the RTC wraps around before the condition is seen. Another benefit of this design is that software computation of a desired TONI setting is simple: use 32–bit unsigned arithmetic and ignore overflow. For example, if the RTC is half a second short of wrapping around, and an interrupt is desired in one second, the TONI should be set for 500,000. Although the RTC will wrap around before the next desired TONI interrupt, there is no need to take an additional interrupt or to perform any other special operation.

Figure 11–20 shows the operation of the TONI graphically. The RTC moves constantly around the dial. Underneath it is a half–circular disk representing the TONI setting. The software can adjust the disk's position by writing into the TONI register. Whenever the RTC "clock hand" is over the TONI disk, the TONI is active and will be seen as such by reading its configuration register. While it is active, it will assert an interrupt request if so enabled by the interrupt enable bit in its configuration register.

Figure 11-20 TONI activity period.



NOTE

PROGRAMMING HINT
When an interrupt occurs, to be sure that a TONI caused the interrupt, the software must check both that the TONI is active and that it is enabled. Neither condition alone is sufficient.

11.14 Bus Errors

There are a variety of conditions in the TC2000 machine that terminate CPU cycles with a bus error. Because the number of bus error conditions is rather large, all bus error causes are prioritized and encoded. The CPU can read the encoded information and can use it as an offset into a dispatch table in the bus error handler. This mechanism is included to improve the bus error service latency.

The encoding of the bus error information is given in the description of the Bus Error Vector register, in section 11.17.

11.15 TC/FPV Registers

The operation of the TC/FPV is controlled by a collection of configuration and control registers. These registers can be divided into nine functional groups:

user registers
configuration registers
interrupt system registers
bus error registers
latency control registers
Switch Interface Gate Array registers
VMEbus interface registers
interleaver control registers
CPU and CMMU internal registers

This section outlines some of the top level characteristics of the TC/FPV register set, and summarizes the characteristics of each functional group. The next section describes the layout and characteristics of each register in detail.

11.15.1 Register Access

Many of the registers reside in the System Physical Address space of the TC2000 machine, and are therefore accessible to any T-bus master — the 88100 CPU, a remote processor node, a VMEbus device, or the Test and Control System slave processor. This keeps the programming model simple, and provides flexible support for bootstrapping, diagnostics and low level system software. The registers are placed in the address space of the TC/FPV so that access can be controlled by the 88200 CMMU and the VMEbus mapping RAMs. The only registers not in System Physical Address space are as follows:

- CPU and CMMU internal registers
- The five RAMs that are read and written via the interleaver loader mechanism, described in section 11.15.3:
  - CPU Mapping RAM in the CPU interface
  - Interleave Decision RAM in the CPU interface
  - Modulus and pool RAMs in the interleaver
  - VMEbus Master Map RAM in the VMEbus interface

All TC/FPV configuration and control registers should be accessed as word (32-bit) quantities, aligned on word boundaries, even if the only meaningful bits lie in an upper byte of the word. The values of unused bits are irrelevant (don't care) on a write, and are unspecified (unless otherwise noted) on a read.

The TC/FPV registers described here are intended for write access by privileged processes only, typically supervisor mode only, with three exceptions.

The three user–writable registers are the Augmentation register (accessed as the AR block), the Interprocessor Interrupt register, and the Process Configuration register. Writing to the PCR is controlled by mask bits in a separate control register. **The hardware does not enforce any register access restrictions explicitly based on the CPU mode (supervisor, user).** Instead, it is up to the software to map the registers to pages with appropriate access protection. Thus, to access a register, the register must be mapped to permit access in the current mode; if the register is protected by a mask and the operation is a write, that mask must be set to permit access also. Registers that may need different access attributes are placed on separate pages.

The register access mechanisms are summarized below.

Name of register(s)	**Access protection mechanism**
Augmentation Register block (AR block)	resides on its own two 8–kilobyte pages
Process Configuration register (PCR)	PCR Disable Mask register
Interprocessor Interrupt register	resides on its own page
all other registers, including AR read–and–clear and AR read/write	intended to be mapped to permit supervisor mode access only

Protection by a mask register is summarized in these points:

- Only the PCR is protected by a mask register. "1" bits in the PCR Disable Mask register *prohibit* access to fields in the PCR.
- Masking applies to both user mode and supervisor mode.
- Masking never applies to reads, and always applies to all writes, regardless of source: CPU, VMEbus slave, switch interface, or TCS.
- Writing a "1" bit into any masked–off field results in a bus error, and the target register is not changed.
- Writing a "0" bit into a masked–off field does *not* result in a bus error. The bit is left unchanged.

11.15.2 Process Context

The process context must be saved and restored upon servicing interrupts and process switching. In addition to the normally required 88100 context, the TC/FPV contains registers that are part of the process context and must be saved, cleared and restored appropriately. These are listed below.

- **Augmentation register** (AR)
- **Process Configuration register** (PCR)
- **PCR Disable Mask register**, if its contents vary from process to process

11.15.3 The Interleaver Loader

The interleaver and the VMEbus master mapper perform similar functions for the Butterfly switch and the VMEbus, respectively. Both of these modules look at the System Physical Address on the T-bus, and alter some subset of the address bits before the address is shipped out over its respective I/O port (the switch or the VMEbus). The address translation must happen in an expedient fashion in both cases, because any time used to calculate the altered address can directly impact the transfer speed. Static memories efficiently solve this problem, and are employed in the TC/FPV. Part of the T-bus address is presented on the address lines of the RAM, and one access time later the altered address bits appear on the data lines. The altered address bits are simply the contents of the memory at the location indicated by the T-bus address. Because these memories are used to map one address to another, they are referred to here as mapping RAMs.

To keep the mapping operations as fast as possible, entries in these mapping RAMs are read and written using a somewhat circuitous mechanism called the *interleaver loader*. Conceptually, the interleaver loader could be a separate piece of hardware. To reduce hardware cost, the control logic for the interleaver loader has been incorporated into the SIGA.

The interleaver loader is also used to read and write the Interleave Decision RAM and the CPU Mapping RAM (CMR), both in the CPU interface. The Interleave Decision RAM, together with the *interleave enable* bit in the selected CMR entry, determines whether an access generated by the CPU is to an interleaved page, and thus will assert the signal T_INTERLEAVED, used by the SIGA. The CPU Mapping RAM is used to translate Physical Addresses into System Physical Addresses, and also to supply various control signals describing references made by the CPU.

Thus, the interleaver loader is used to access five RAMs:

- the modulus RAM in the interleaver
- the pool RAM in the interleaver
- the VMEbus master mapper RAM in the VMEbus interface
- the Interleave Decision RAM in the CPU interface
- the CPU Mapping RAM in the CPU interface

Use of the interleaver loader is described in the Interleaver Loader registers entry in section 11.17.

11.15.4 Registers by Functional Group

User Registers

Three registers in the TC/FPV are intended to be user-accessible: the *Interprocessor Interrupt* register, the *Process Configuration* register, and the *Augmentation register block*. Each of these registers resides on its own page to provide maximum flexibility for the operating system in permitting each individual process to access these registers. Further, certain fields of the PCR are protected by mask bits; attempting to set a bit that is masked returns a bus error. The PCR and Interprocessor Interrupt registers are read/write. The AR Block is read only, although reads of the AR Block result in the setting and clearing of Augmentation Register bits.

Configuration Registers

In addition to the user-accessible registers, two configuration registers directly control the execution environment. These two are intended for access by privileged processes only, so the page where these registers reside should be mapped in supervisor address space. The *PCR Disable Mask* register controls modification of individual fields in the PCR. This allows precise control over permissions granted to each user process. The *Machine Configuration* register is tied to low level TC/FPV board configuration parameters used during testing and initialization.

Interrupt System Registers

The interrupt system registers provide control over the TC/FPV's CPU interrupt facility that extends the single-level interrupt capability of the 88100. The *Interprocessor Interrupt* register and the *Non-maskable Interprocessor Interrupt* register permit any T-bus master to interrupt the CPU. In particular, the CPU on another TC/FPV can interrupt the CPU by using these registers. Therefore, the Interprocessor Interrupt register may be considered a user register also, because typically it is mapped to be accessible in user mode. The *Interrupt Source* register informs the CPU which of several possible events is requesting interrupt service. The *Interrupt Enable Mask* register allows the CPU to selectively disable interrupts from certain sources. Interrupts to the VMEbus are generated by VMEbus interface registers described below.

Bus Error Register

The bus error signal presented to the CPU can be asserted for a variety of reasons. The *Bus Error Vector* register, the only register in this functional group, indicates the reason that a bus error was generated. This allows the software to take appropriate action based on the location and nature of the error. For

example, a bus error may be intentionally provoked in testing whether a given memory module is installed in the system.

Latency Control Registers

The latency of interrupt servicing and access to a memory module may be controlled by use of these three registers. Each holds a value used by an associated timer; expiration of the timer invokes procedures to relinquish held resources. The *Interrupts Disabled Timer* register limits how long the CPU may disable certain interrupts. The *Interrupts Pending / Abort Retries Timer* register limits how long the CPU will wait for establishment of a switch connection. The *CPU Lock Timer* register controls how long the CPU may hold a memory module locked. Latency is also controlled by timer registers in the VMEbus interface for VMEbus transactions, and in the SIGA for switch transactions.

VMEbus Interface Registers

The TC/FPV VMEbus interface contains three general types of registers. The first type controls the role played by the TC/FPV as a device on the VMEbus. Aspects of this role are the location and size of the window from TC2000 address space into VMEbus address space, the location and size of the reverse window, whether the TC/FPV is system controller, and related parameters. This includes the *VMEbus Configuration* register, the *VMEbus Master Map RAM* registers and the *VMEbus Slave Map RAM* registers. Second, the generation of interrupts on the VMEbus is controlled by two registers, the *VMEbus Interrupt Request* register and the *VMEbus Interrupt Vector / Control* register. The third type of register is timers that detect and abort conditions that persist too long. The *VMEbus Arbiter Timer* register allows the TC/FPV to perform services as VMEbus system controller. The *VMEbus System Bus Timer* register provides a recovery mechanism if no VMEbus slave responds to an access from any VMEbus master. The *VMEbus TC/FPV Master Bus Timer* register prevents an unduly slow VMEbus transaction from hanging the TC/FPV's VMEbus master interface, thus helping to bound access latency.

SIGA Registers

The Switch Interface Gate Array contains several registers accessible from the T-bus. One group of SIGA registers controls the transmission of switch messages by the requester portion of the SIGA. This includes the *Message Classification, Protocol Timer Configuration, Transmit Time Configuration,* and *Priority Time Configuration* registers, two *Requester Configuration* registers, and a *Requester Test* register.

A second group of SIGA registers controls the reception of switch messages by the server portion of the SIGA. This includes two *Server Configuration* registers and a *Server Test* register.

A third group of SIGA registers concerns timekeeping. This includes the *Real Time Clock* (RTC) register, two *Time Of Next Interrupt* registers (TONIA, TONIB), and two *TONI Configuration* registers.

The fourth group of SIGA registers implements an interface between the T-bus and various high speed RAMs. The SIGA registers comprising this interface, called the interleaver loader, are the *Interleave Address* register and the *Interleave Data* register. The RAMs that are read and written via the interleaver loader are the interleaver Modulus and Pool RAMs, the Interleave Decision RAM and the CPU Mapping RAM (both in the CPU interface), and the VMEbus Master Map RAM (in the VMEbus interface).

Interleaver Control Registers

The TC/FPV has two kinds of registers controlling interleaving — registers that determine whether a given access is to interleaved memory, and the registers that produce the modified bits of the interleaved address. Each T-bus master contains its own mechanism to produce the T_INTERLEAVED signal that indicates to the SIGA that the access is interleaved. Accesses from the VMEbus slave drive T_INTERLEAVED from a bit in the *VMEbus Slave Map RAM* register being used. Accesses from the CPU drive T_INTERLEAVED from the *Interleave Decision RAM* register being used, enabled by a bit in the *CPU Mapping RAM* register being used. The SIGA server makes only local accesses, so it never drives T_INTERLEAVED.

The interleaver *Modulus RAM* and *Pool RAM* registers produce the modified address bits used in an interleaved access. The translation takes bits from the T-bus and supplies modified bits to the SIGA, for inclusion in the outgoing switch message.

CPU and CMMU Internal Registers

The Motorola 88100 CPU and 88200 CMMU contain several internal registers, for control of data processing and of caching and memory management, respectively. Occasionally, we refer to those internal registers in this document. For example, the CPU Processor Status register permits disabling interrupts, and doing so makes the CPU immune to all the TC/FPV bus error interrupts. A more complete discussion of CPU and CMMU internal registers is beyond the scope of this document. For a full description of CPU and CMMU registers, please consult the Motorola *User's Manual* for each of these devices.

11.15.5 Register Name Summary List

Figure 11-21 lists all registers on the TC/FPV except the CPU and CMMU registers. (Also excluded are the TCS registers, which are accessible only to the Test and Control System.)

Figure 11-21 List of registers by functional group.

- user registers
  - Process Configuration register (PCR)
  - Augmentation register (AR)
  - (also, the Interprocessor Interrupt register is user-accessible)
- configuration registers
  - CPU Mapping RAM registers (no T-bus access)
  - Machine Configuration register
  - PCR Disable Mask register
- interrupt system registers
  - Interprocessor Interrupt register
  - Non-maskable Interprocessor Interrupt register
  - Interrupt Enable Mask register
  - Interrupt Source register
- bus error registers
  - Bus Error Vector register
- latency control registers
  - CPU Lock Timer register
  - Interrupts Disabled Timer register
  - Interrupts Pending / Abort Retries Timer register
  - (also, the VMEbus TC/FPV Master Bus Timer register, and switch protocol registers in the SIGA, help control latency)
- VMEbus interface registers
  - VMEbus Arbiter Timer register
  - VMEbus Configuration register
  - VMEbus Interrupt Request register
  - VMEbus Interrupt Vector / Control register
  - VMEbus Master Map RAM registers (no T-bus access)
  - VMEbus Slave Map RAM registers
  - VMEbus System Bus Timer register
  - VMEbus TC/FPV Master Bus Timer register

- (also, the Interrupt Enable Mask register and the Interrupt Source register contain VMEbus-related fields)

- Switch Interface Gate Array registers
  - Real_Time_Clock (hi and lo) register(s)
    Time_Of_Next_InterruptA register
    Time_Of_Next_InterruptB register
    TONIA_Config register
    TONIB_Config register
  - Priority_Time_Config register
    Protocol_Timer_Config and Message_Class register
    Requestor_ConfigA register
    Requestor_ConfigB register
    Requestor_TestA register
    Transmit_Time_Config register
  - Server_ConfigA register
    Server_ConfigB register
    Server_TestA register
  - Interleave_Address_Reg register
    Interleave_Data_Reg register
- interleaver control registers
  - CPU Mapping RAM registers (no T-bus access)
  - Interleave Decision RAM registers (no T-bus access)
  - Interleaver Modulus and Pool RAM registers (no T-bus access)
  - VMEbus Slave Map RAM registers
- CPU and CMMU internal registers
  - Refer to *88100 User's Guide* (CPU) and *88200 User's Guide* (CMMU)

11.15.6 Register Addresses – CPU and T-bus

The formal specification of TC/FPV local configuration and control register addresses is their **System Physical Address**, that by which they are referenced on the T-bus. The **Process Logical Address** used by a program may be transformed by the CMMU memory management mapping and/or by the CPU Mapping RAM mapping before it is presented to the T-bus. By setting up the CMMU and CMR appropriately, software can adjust the apparent location of a register to any desired page in Process Logical Address space. Only the register's address on the T-bus remains fixed.

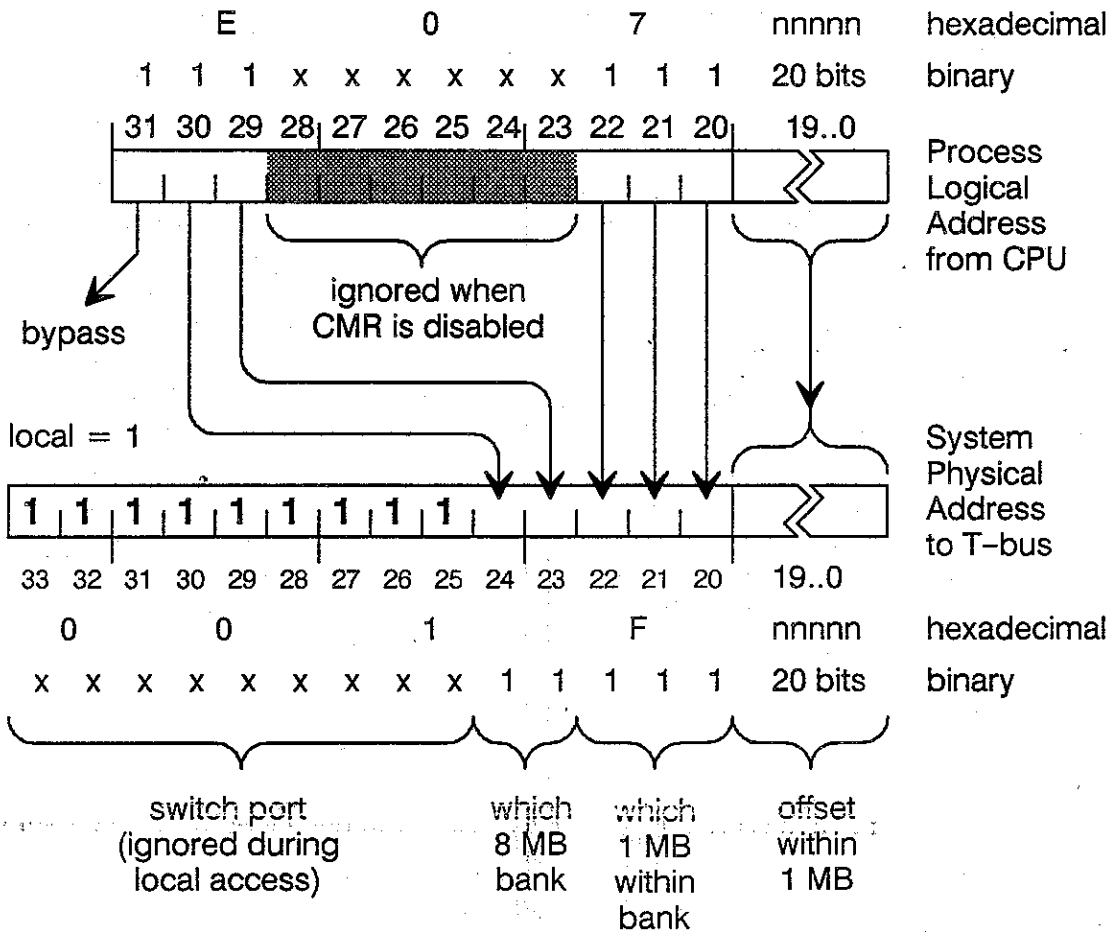
However, when the CMMU and CMR are disabled, the hardware performs a default mapping from Process Logical Addresses to System Physical Ad-

dresses. Operating systems usually set up translation so that the CMMU and CMR do not change local register addresses from their default mapping. Therefore, an understanding of this default mapping is often applicable to operation with mapping enabled, and is always applicable when mapping is off.

Figure 11–22 shows the default mapping ***when both the CMMU and CMR mappings are disabled***, with annotations for local register accesses. When the CMMU is disabled, it copies the Process Logical Address directly to the Physical Address without modification. The complexities shown are due to the CPU interface that maps the Physical Address to the System Physical Address.

The figure should be examined from bottom to top. At the bottom, the System Physical Address is interpreted as four fields. The switch port is ignored during a local access, so the value of its bits is irrelevant. T–bus address bits 24..23 determine which of the board's four 8–megabyte banks of address space is being referenced. Within that bank, bits 22..20 select which megabyte of the eight is being referenced. Since local registers reside in the top megabyte of local address space on every TC/FPV, bits 24..20 must be "11111". T–bus address bits 19..0 specify an offset within the selected megabyte; in this example, they select the register to be referenced. Thus we see that local registers have System Physical Addresses of the form 0x001Fnnnnn.

Figure 11–22 Addresses of local registers, CMMU and CMR disabled.



As noted above, T-bus address bits 33..25 are ignored for a local access. For the sake of simplicity in writing System Physical Addresses, we think of those bits as zeros. The hardware is required to drive all T-bus bits to valid logic levels, however, and the design happens to drive each of these bits to "1".

When the CMR is disabled, all CPU accesses are forced to be local. The circuitry normally driven by the CMR *local* bit is instead driven by a "1".

The remaining System Physical Address fields are derived from the Process Logical Address as Figure 11–22 shows. The bank is selected by Process Logical Address bits 30..29. The megabyte within the bank is selected by Process Logical Address bits 22..20. Therefore, these five bits must be ones to address a local register. The offset within the megabyte comes from Process Logical Address bits 19..0. The two other fields of the Process Logical address are bits 28..23 and bit 31. Bits 28..23 are ignored when the CMR is disabled, and for

simplicity we consider them to be "000000". Bit 31 drives the bypass circuitry, driven by the CMR *bypass* bit when the CMR is enabled.

Local registers should be referenced bypassed. The reason for making any reference bypassed is so that the TC2000 locking protocol will not interfere with the reference. But the concern is different for local registers than for memory. When memory references are bypassed, it is so they will succeed even if the target memory module is locked. The TC/FPV local registers, on the other hand, cannot be locked; the interface to them does not support locking. In fact, a locked access to local registers results in a bus error. It would work perfectly well to always access local registers NORMAL (as opposed to OPEN lock, MAINTAIN a lock, or BYPASS a lock). However, this may be hard to arrange.

- One example is accessing a local register with the XMEM instruction; it automatically invokes the TC2000 locking protocol.
- Another example is handling a bus error that occurs during a locked sequence. The bus error handler typically needs to save and clear the Augmentation register right away, so that locking will not be in effect while the bus error is handled. This is done by reading the AR read–and–clear address. And since the AR has not yet been cleared, locking is still in effect, so the read of the AR will be an OPEN or MAINTAIN access, and will get a bus error.

These problems are avoided by reading the AR (or other local register) with a BYPASSED access. The bypass overrides the fact of being in a locked sequence, so the local register interface gets a bypassed reference and services it. To be sure that the local registers are referenced bypassed when needed, it is best to simply *always* reference *all* local registers bypassed.

NOTE

REFERENCE LOCAL REGISTERS BYPASSED
As explained above, it is advisable that all accesses to local registers be bypassed references.

Thus, the CMR *bypass* bit should be "1" on any entry used to access the local registers. If the CMR and the CMMU are disabled, then bit 31 of the Process Logical Address should be "1". If the CMR is *disabled* but the CMMU is *enabled*, the CMMU's mapping of local registers should produce a "1" in bit 31 of the Physical Address.

This explains all the fields of the Process Logical Address to reference local registers. As shown in Figure 11-22, the form of these addresses is 0xE07nnnnn. The "nnnnn" offset is the same as the offset in the System Physical Address. This leads to the following simple rule.

NOTE

LOCAL REGISTER ADDRESSES
When the CMMU and CMR are disabled, the following addresses are used to access local configuration and control registers on the TC/FPV:

Process Logical Address (from CPU):	0xE07nnnnn
System Physical Address (on T-bus):	0x01Fnnnnn

Thus one can be converted into the other simply by exchanging the leading "E07" or "01F". ***Depending on software use***, the Process Logical Address used by programs when the CMMU and/or CMR are enabled is often the same as that when they are disabled. The System Physical Address, in contrast, never changes.

EXCEPTION
The local Interprocessor Interrupt register's normal address has a different form. Its Process Logical Address is 0x80002000 and its System Physical Address is 0x2000.

11.15.7 Register Address Summary List

Figure 11–23 shows the location of each register in the address space of the TC/FPV.

The addresses in the list below are **Process Logical Addresses**, or "CPU addresses", by which a program references the registers ***when the CMMU and CMR are disabled***. When translation by the (data) CMMU and the CMR is enabled, the addresses shown may be mapped to any page. Operating systems usually set up translation so that these addresses are unchanged by the CMMU and CMR, and therefore the addresses given below are typically the addresses a program will use to access the registers.

Occasionally, the System Physical Address (T-bus address) of registers is useful to know, such as in TCS programming. System Physical Address bits 33..25 specify switch routing, and bits 24..0 are the address offset at that switch port. The correspondence between Process Logical Addresses and System Physical Addresses is discussed further in section 11.15.6.

The SIGA registers have several aliases within the address space listed for the SIGA; see the *SIGA Specification* for these details. The *SIGA Specification* should also be consulted for details of the SIGA registers' structure and operation. Some SIGA registers are also mentioned elsewhere: the TONI registers and the RTC in section 11.13, and the IAR and IDR in section 11.17 under "Interleaver Loader registers".

The TC/FPV configuration and control registers other than the SIGA are *not* aliased; all bits (24..0) of their address are decoded. A read or write to a location in the top megabyte of System Physical Address space (where these regis-

ters reside) but not to a specific register's address (or one of the SIGA alias addresses, as noted above) will have no effect and will not cause a bus error.

Figure 11-23 Addresses of TC/FPV T-bus registers.

— Process Logical Addresses with CMMU and CMR disabled —

0x80002000 Interprocessor Interrupt register (normal address)

0xE0720000 to 0xE0721FFC
VMEbus Slave Map RAM registers

0xE0740000 Process Configuration register (PCR)

0xE0740004 Interprocessor Interrupt register (alternate address)

0xE0758000 to 0xE075BFFC
Augmentation register, as AR block

0xE0760000 Machine Configuration register

0xE0760004 Non-maskable Interprocessor Interrupt register

0xE0760008 Interrupts Pending / Abort Retries Timer register

0xE076000C Interrupts Disabled Timer register

0xE0760200 VMEbus TC/FPV Master Bus Timer register

0xE0760204 VMEbus Configuration register

0xE0760208 VMEbus System Bus Timer register

0xE076020C VMEbus Arbiter Timer register

0xE0760800 VMEbus Interrupt Vector / Control register

0xE0760804 VMEbus Interrupt Request register

0xE0780000 CPU Lock Timer register

0xE0780004 Augmentation register, as AR read/write

0xE078000C Bus Error Vector register

0xE0780010 PCR Disable Mask register

0xE0780014 Interrupt Enable Mask register

0xE0780018 Interrupt Source register

0xE078001C Augmentation register, as AR read-and-clear

——————————— SIGA B registers begin here ———————————

0xE07D0000 to 0xE07DBFFC
SIGA B registers — same as SIGA A registers below, but with base address 0xE07D0000 instead of 0xE07E0000

SIGA A registers begin here

0xE07E0000 to 0xE07EBFFC
SIGA A registers as follows (and aliases; see the *SIGA Specification* for details of aliased addresses)

0xE07E0000 TONI A (Time Of Next Interrupt, A) Configuration register

0xE07E0004 TONI A register

0xE07E2000 TONI B Configuration register

0xE07E2004 TONI B register

0xE07E8000 Protocol Timer Configuration and Message Class register

0xE07E8004 Transmit Timer Configuration register

0xE07E8008 Priority Timer Configuration register

0xE07E800C Requester Configuration A register

0xE07E8010 Requester Configuration B register

0xE07E8014 Requester Test A register

0xE07E8018 Real Time Clock (RTC) register

0xE07E9000 Server Configuration A register

0xE07E9004 Server Configuration B / Server Test A register

0xE07EA000 and following
causes Interleaver Data register transfers

0xE07EA00C Interleaver Address register

0xE07EB000 Interleaver Data register

end of SIGA registers

0xFFF7D000 code CMMU #2 base address (ID register) after reset

0xFFF7E000 data CMMU base address (ID register) after reset

0xFFF7F000 code CMMU #1 base address (ID register) after reset

NOTE

CACHE CHIP ADDRESSING DETAILS
In a TC/FPV configured with only two cache chips, code CMMU #2 is omitted. Internal registers in the cache chips are accessible only from the CPU, not from the T-bus. Motorola's design does permit access from both the P-bus (processor bus) and M-bus (memory bus), but the TC/FPV configuration and control register interface does not support T-bus access to the CMMU registers.

11.16 TC/FPV Address Map

The TC/FPV address map below gives the layout of **System Physical Address** space on each TC/FPV. The low 25 bits of the System Physical Address describe resources on the function board; the high 9 bits, shown as zeros below, specify the switch port at which the given function board resides.

0x0000 0000 to 0x003F FFFF	4 megabytes of local memory (always installed)
0x0040 0000 to 0x007F FFFF	12 more megabytes of local memory (optional)
0x0100 0000 to 0x01EF FFFF	15-megabyte window from TC2000 address space into VMEbus address space
0x01F0 0000 to 0x01FF FFFF	1-megabyte configuration and control register space

11.17 TC/FPV Register Definitions

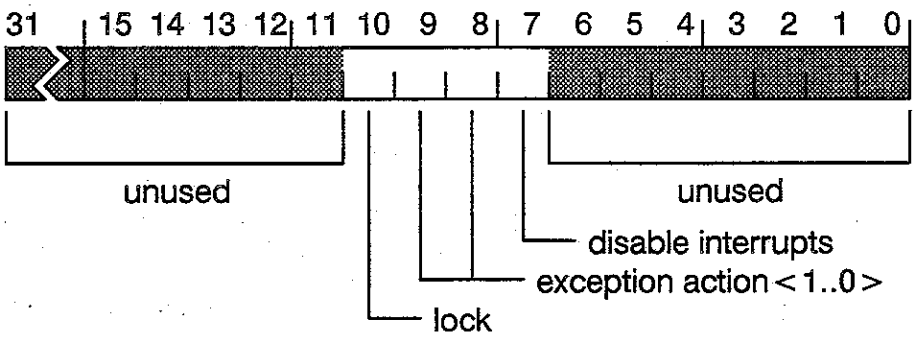
Certain registers are present in TC/FPV function board hardware, but are described elsewhere. These are:

- CPU and CMMU internal registers. These are described in Motorola literature.
- SIGA control and configuration registers. These are described in the *SIGA Specification*.

Each of the registers, in alphabetical order, is summarized below.

The description of each register includes its contents after power-on. The TCS power-up operation modifies the contents of several registers, some of which depend on the configuration of the system. See TCS software and operations documentation for those details.

Figure 11-24 Augmentation register (AR).



FUNCTION Enable augmentations to 88100 CPU instructions.

ADDRESS
0xE0758000 to 0xE075BFFC (inclusive) — AR block
0xE078001C — AR read-and-clear
0xE0780004 — AR read/write

ACCESS
AR block — read only, with side effects
AR read-and-clear — read only, with side effects
AR read/write — read/write

There is just one Augmentation register, but it may be referenced in three ways. It is intended that the operating system permit user-mode access to the AR only via the AR block.

- Reading a (word-aligned) location in the AR block causes bits 13..2 of the address to be loaded into bits 13..2 of the AR. The data returned in bits 31..16 is unspecified, and in bits 15..0 is all ones. Writing to the AR block has the same effect as reading it; the write data is ignored.
- Reading the AR read-and-clear location returns the contents of the AR and clears the AR to zero.
- Accessed at the AR read/write location, the AR may be read without clearing it, and may be written.

DESCRIPTION The *disable interrupts* bit, when set to one, prevents assertion of the CPU's interrupt request pin by certain sources (see section 11.13). Requests already in the CPU's internal pipeline are not disabled. Interrupts from the VMEbus, from the real time clock timer(s), and from the maskable interprocessor interrupt are disabled by this bit. Two interrupts are **not** disabled by this bit: the interrupts disabled timer interrupt, and the non-maskable interprocessor interrupt.

The *exception action* bits describe what action to take at the end of exception processing. The hardware does not use these bits; they are provided for use by software, to tell the exception handler software what to do when an exception occurs during an instruction sequence that uses the lock bit. Their meaning is only a programming convention, although "00" should mean "continue" because the AR is cleared to zero upon power-up.

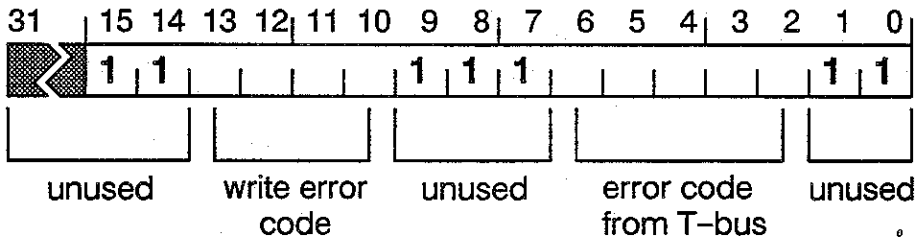
00	continue (proceed normally)
01	restart (resume instruction at restart address)
10	abort (getting any exception is fatal to the process)
11	undefined

The *lock* bit, when set to one, causes the switch path (if any) to the resource that will be accessed, and the resource itself, to be held open after the access has completed. Changing the lock bit from "1" to "0" generates a T-bus FREE_LOCKS cycle, but only if a resource has been locked (by making a locked reference).

On any access, the effect of the *lock* bit is overridden if the *bypass* bit in the CPU Mapping RAM entry selected for the access is "1", forcing a bypassed reference. If the CPU interface is not currently holding a lock, such an access does not make it think it is. Thus a subsequent access with *lock* "1" but *bypass* "0" is OPEN (not MAINTAIN). Or a subsequent clearing of the AR *lock* bit will not generate a FREE_LOCKS cycle.

Power-on clears all defined bits of the Augmentation register to zero.

Figure 11–25 Bus Error Vector register.



FUNCTION Indicate the cause of a bus error.

ADDRESS 0xE078000C

ACCESS read only

DESCRIPTION An operation initiated by the CPU may result in an error detected by a T–bus slave. In the TC/FPV, these slaves are the memory, the SIGAs, the VMEbus master interface, and the status and configuration registers. The slave indicates the error by responding with an error reply (T–bus control bits T_RR<2..0> are 000) and, when appropriate, placing a code describing the error on the T–bus data wires. (The code is placed on the T–bus data wires if the T–bus is not being driven by the master — namely, in a read or a response from a split cycle. Otherwise, placing the code on the T–bus is impossible because it is already being driven.) The CPU interface conveys this to the CMMU as an M–bus (memory bus) error, and the CMMU in turn conveys the error to the CPU, which receives it as an M–bus error exception. The error description code placed on the T–bus is captured in the Bus Error Vector register, for use by software handling the exception. Thus, the Bus Error Vector register contains the error code for the most recent bus error to the CPU. If there has been no such error, the contents of the register are undefined.

If the error regards a write operation for which the CPU is currently driving data onto the T–bus (that is, a non–split cycle write), then the error description code is captured in the *write error code* field, and the *error code from T–bus* field indicates, with the value CPU_write_error, that the real error code is in the *write error code* field. Otherwise, the contents of the *write error code* field are undefined.

The error codes in the *error code from T–bus* field are given in Figure 11–26, and the error codes in the *write error code* field are given in Figure 11–27.

Unused bits in the low two bytes of the Bus Error Vector always read as "1".

After power–on, the contents of the Bus Error Vector register are unspecified.

Figure 11-26 Bus error encoding.

6 5 4 3 2 — bit positions in Bus Error Vector register

SWITCH REQUESTER / CSU ERRORS
These errors come from the requester or the Configuration/Status Unit (CSU) in the local SIGA. The errors from the requester are prioritized, so that if two or more errors occur simultaneously, only the highest priority error is returned. Priority 1 is highest, 8 is lowest. Within a given priority, errors are mutually exclusive. For example, Wait Timeout (4a) and Idle Timeout (4b) cannot both occur at once. These errors are not detected unless enabled by appropriate setting of the SIGA's Requester Configuration B register.

0 0 0 0 0 **Maintain Absent** (priority 1a)
A locked switch connection has been established and has not been released, and the requester was asked to make an access that was not MAINTAIN or BYPASS. In other words, a NORMAL or OPEN access was attempted (by the master that holds the lock) while the requester was locked. (Technically, the SIGA permits OPENs during a locked connection, unlike the memory and the VMEbus master interface. However, no master on the TC/FPV can generate such a second OPEN, and its use would be discouraged anyway.)

0 0 0 0 1 **Maintain Present** (priority 1b)
The requester was asked to make a MAINTAIN access when no locked switch connection was opened.

(switch requester error priority 2 is unused and reserved)

0 0 0 1 1 **Lock Address** (priority 3)
Within a locked sequence, the requester was asked to access a switch port (function board) other than the one to which the connection was open.

0 0 1 0 0 **Wait Timeout** (priority 4a)
The switch connection timer expired while the requester was waiting for a response from the SIGA server on the remote function board.

0 0 1 0 1 **Idle Timeout** (priority 4b)
The switch connection timer expired while the requester was in its idle locked state, not waiting for a response. This condition arises during a locked sequence, when the SIGA requester has done all that was asked of it so far, and is waiting for further access requests from the T-bus. If the timer expires now, the error is not returned immediately, but is remembered by the SIGA requester. If another T-bus master tries to access the requester, it will get a REFUSED-LOCKED response as usual. If the T-bus master that created the locked sequence tries another switch access, it will get the Idle Timeout error. If that master does a FREE LOCKS cycle before making another switch access, the timer expiration condition is forgotten and no error is returned. The SIGA requester cannot be hung indefinitely, because the master will do a FREE LOCKS cycle eventually (the VMEbus slave interface almost immediately, and the CPU as enforced by the CPU Lock Timer).

00110 **Reject Abort** (priority 5)
The timer loaded from the Interrupts Pending / Abort Retries register expired, asserting the siga_abort_retries signal on the REJ_ABORT input pin of the SIGA, forcing the switch reject timer to expire. Interrupts were disabled for too long.

00111 **Reject Timeout** (priority 6)
The switch reject timer expired while the requester was trying to open a connection. This error is usually caused by addressing a failed board or an empty board slot. If the switch priority promotion mechanism is not enabled or is using inappropriate parameters, excessive contention for the destination switch port may also cause this error.

01000 **Reverse** (priority 7)
The requester detected an incorrect polarity of the *reverse* signal while receiving a response from the remote function board. This error indicates a hardware problem.

01001 **Check** (priority 8)
The requester detected an incorrect checksum in a response from the remote function board. This error indicates a hardware problem.

01010 **Miscellaneous Configuration/Status Unit Error**
An error was made by the T-bus master (here, the CPU) in accessing the SIGA's CSU. All access to internal SIGA registers and SIGA parameter setting is done through the CSU. (The only exception is setting the base address at which the CSU responds. That is set via the Test and Control System interface.) The error could be either or both of the following:

1. An OPEN lock was requested. The CSU is not a lockable resource.
2. A multiple-word transfer was requested, caused by trying to cache the location. The CSU supports only byte and word accesses.

SWITCH SERVER ERRORS
These errors are sensed by the SIGA server on the remote ("downstream") function board, reported back over the switch connection, and handed through the local SIGA requester to the T-bus master (here, the CPU) that opened the connection. They correspond to the analogous errors that could arise on the local function board. The server is always enabled to detect these errors during operation.

01100 **Downstream Refused**
A downstream T-bus slave responded with REFUSED-LOCKED when the server thought itself locked. A hardware fault can cause this error, but more likely the software design has allowed another master on that T-bus to lock the given slave.

01101 **Downstream Write**
A downstream write error was detected while the SIGA server was sourcing data on the T-bus of the remote function board. Because the SIGA was driving the T-bus data wires, the actual error code cannot be returned. If such a write error is detected locally, the error code is returned in the *write error code* field of the Bus Error Vector register, but that is impossible here.

01110 **Downstream Memory Parity Error**
An access to memory on the remote function board resulted in a parity error.

01111 **Downstream Out To Lunch**
A downstream T-bus slave did not respond to the SIGA server's request. The slave must respond on the T-bus cycle following the request.

VMEbus MASTER / T-bus SLAVE ERRORS
These errors are detected by the VMEbus master interface.

10000 **VMEbus Maintain Absent**
A locked sequence has been established to the VMEbus and has not been released, and the VMEbus master interface was asked to make an access that was not MAINTAIN or BYPASS. In other words, a NORMAL or OPEN access was attempted (by the master that holds the lock) while the interface was locked.

10001 **VMEbus Maintain Present**
The VMEbus master interface was asked to make a MAINTAIN access when no locked sequence was in progress — the lock was never opened.

10010 **VMEbus Burst Attempt**
A multiple-word access was requested, caused by trying to cache the location. The VMEbus master interface supports only byte and word accesses. (The VMEbus *slave* interface supports only byte and word accesses, and does not respond to sequential transfer requests, also called "block transfers", from the VMEbus. Such non-response is not itself a bus error on the VMEbus, although the VMEbus System Bus Timer eventually declares a bus error and asserts the VMEbus BERR signal.)

10011 **VMEbus Bus Error**
An access to the VMEbus resulted in a VMEbus bus error. This may be asserted by the addressed VMEbus slave device or by expiration of either of two timers — the VMEbus TC/FPV Master Bus Timer or the VMEbus System Bus Timer. No further information about the nature of the error is supplied by the VMEbus.

TC/FPV CONFIGURATION AND CONTROL REGISTER ERRORS
These errors arise in accessing the configuration and control registers on the local TC/FPV.

10100 **Local Register Lock Attempt**
An access attempted to open a lock to a configuration and control register. The interface to these registers does not support the TC2000 locking protocol.

10101 **Local Register Burst Attempt**
A multiple–word access was requested, caused by trying to cache a register. The configuration and control register interface supports only word accesses. If addressed as a byte or halfword, no error is generated but the data returned is undefined.

10110 **Local Register Masked Error**
A write to the Process Configuration register attempted to set to "1" one or more bits that were masked by the PCR Disable Mask register.

LOCAL MEMORY ERRORS
These errors arise in accessing the memory module on the local function board. For errors in accessing remote memory, see "switch server errors".

11000 **Memory Parity Error**
An access to local memory resulted in a parity error.

11100 **Memory Maintain Absent**
A locked sequence has been established to the local memory and has not been released, and the memory interface was asked to make an access that was not MAINTAIN or BYPASS. In other words, a NORMAL or OPEN access was attempted (by the master that holds the lock) while the memory was locked.

11101 **Memory Maintain Present**
The local memory interface was asked to make a MAINTAIN access when no locked sequence was in progress — the lock was never opened.

CPU INTERFACE ERRORS
These errors are reported by the CPU interface.

11011 **CPU Write Error**
A write operation from the CPU encountered an error while the CPU interface was driving data onto the T–bus. Further description of the error is in the *write error code* field of the Bus Error Vector register. (The numerical value of the CPU Write Error is 11011, out of sequence with the other codes listed here.)

11110 **CPU No Response**
A local T–bus slave did not respond to the CPU interface's request. The slave must respond on the T–bus cycle following the request.

RESERVED ERROR CODES
These error codes are reserved and should not arise in normal operation.

00010 reserved

01011 reserved

10111 reserved

11001 reserved

11010 reserved

11111 reserved

Figure 11–27 Write error encoding.

13 12 11 10 — bit positions in Bus Error Vector register

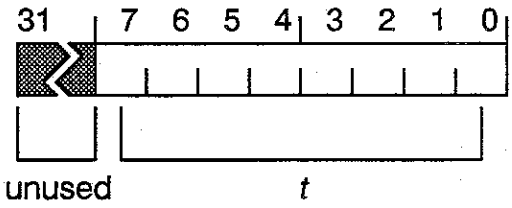
0 0 0 0	reserved
0 0 0 1	**Memory Maintain Absent** A locked sequence has been established to the local memory and has not been released, and the memory interface was asked to make an access that was not MAINTAIN or BYPASS. In other words, a NORMAL or OPEN access was attempted (by the master that holds the lock) while the memory was locked.
0 0 1 0	reserved
0 0 1 1	**Memory Maintain Present** The local memory interface was asked to make a MAINTAIN access when no locked sequence was in progress — the lock was never opened.
0 1 0 0	**VMEbus Maintain Absent** A locked sequence has been established to the VMEbus and has not been released, and the VMEbus master interface was asked to make an access that was not MAINTAIN or BYPASS. In other words, a NORMAL or OPEN access was attempted (by the master that holds the lock) while the interface was locked.
0 1 0 1	**VMEbus Maintain Present** The VMEbus master interface was asked to make a MAINTAIN access when no locked sequence was in progress — the lock was never opened.
0 1 1 0	**VMEbus Burst Attempt** A multiple–word access was requested, caused by trying to cache the location. The VMEbus master interface supports only byte and word accesses. (For the VMEbus *slave* interface action, see the comment under VMEbus Burst Attempt in Figure 11–26 above.)
0 1 1 1	reserved
1 0 0 0	**Local Register Lock Attempt** An access attempted to open a lock to a configuration and control register. The interface to these registers does not support the TC2000 locking protocol.
1 0 0 1	**Local Register Masked Error** A write to the Process Configuration register attempted to set to "1" one or more bits that were masked by the PCR Disable Mask register.
1 0 1 0	**Local Register Burst Attempt** A multiple–word access was requested, caused by trying to cache a register. The configuration and control register interface supports only word accesses. If addressed as a byte or halfword, no error is generated but the data returned is undefined.
1 0 1 1	reserved
1 1 0 0	reserved

1 1 0 1	reserved
1 1 1 0	reserved
1 1 1 1	reserved

The following page contains a summary of the error codes, for easy reference.

6 5 4 3 2 (error code from T–bus)	bit positions in Bus Error Vector register		**13 12 11 10** (write error code)
	SWITCH REQUESTER / CSU ERRORS		
0 0 0 0 0	**Maintain Absent** (priority 1a)		
0 0 0 0 1	**Maintain Present** (priority 1b)		
	(switch requester error priority 2 is unused and reserved)		
0 0 0 1 1	**Lock Address** (priority 3)		
0 0 1 0 0	**Wait Timeout** (priority 4a)		
0 0 1 0 1	**Idle Timeout** (priority 4b)		
0 0 1 1 0	**Reject Abort** (priority 5)		
0 0 1 1 1	**Reject Timeout** (priority 6)		
0 1 0 0 0	**Reverse** (priority 7)		
0 1 0 0 1	**Check** (priority 8)		
0 1 0 1 0	**Miscellaneous Configuration/Status Unit Error** (OPEN or multiple–word)		
	SWITCH SERVER ERRORS		
0 1 1 0 0	**Downstream Refused**		
0 1 1 0 1	**Downstream Write**		
0 1 1 1 0	**Downstream Memory Parity Error**		
0 1 1 1 1	**Downstream Out To Lunch**		
	VMEbus MASTER / T–bus SLAVE ERRORS		
1 0 0 0 0	**VMEbus Maintain Absent**		0 1 0 0
1 0 0 0 1	**VMEbus Maintain Present**		0 1 0 1
1 0 0 1 0	**VMEbus Burst Attempt**		0 1 1 0
1 0 0 1 1	**VMEbus Bus Error**		
	TC/FPV CONFIGURATION AND CONTROL REGISTER ERRORS		
1 0 1 0 0	**Local Register Lock Attempt**		1 0 0 0
1 0 1 0 1	**Local Register Burst Attempt**		1 0 1 0
1 0 1 1 0	**Local Register Masked Error**		1 0 0 1
	LOCAL MEMORY ERRORS		
1 1 0 0 0	**Memory Parity Error**		
1 1 1 0 0	**Memory Maintain Absent**		0 0 0 1
1 1 1 0 1	**Memory Maintain Present**		0 0 1 1
	CPU INTERFACE ERRORS		
1 1 0 1 1	**CPU Write Error**		
1 1 1 1 0	**CPU No Response**		
	RESERVED ERROR CODES		
0 0 0 1 0	reserved	reserved	0 0 0 0
0 1 0 1 1	reserved	reserved	0 0 1 0
1 0 1 1 1	reserved	reserved	0 1 1 1
1 1 0 0 1	reserved	reserved	1 0 1 1
1 1 0 1 0	reserved	reserved	1 1 0 0
1 1 1 1 1	reserved	reserved	1 1 0 1
		reserved	1 1 1 0
		reserved	1 1 1 1

Figure 11-28 CPU Lock Timer register.



FUNCTION Limit how long the CPU may hold a lock.

ADDRESS 0xE0780000

ACCESS read (timer counter) / write (preset register)

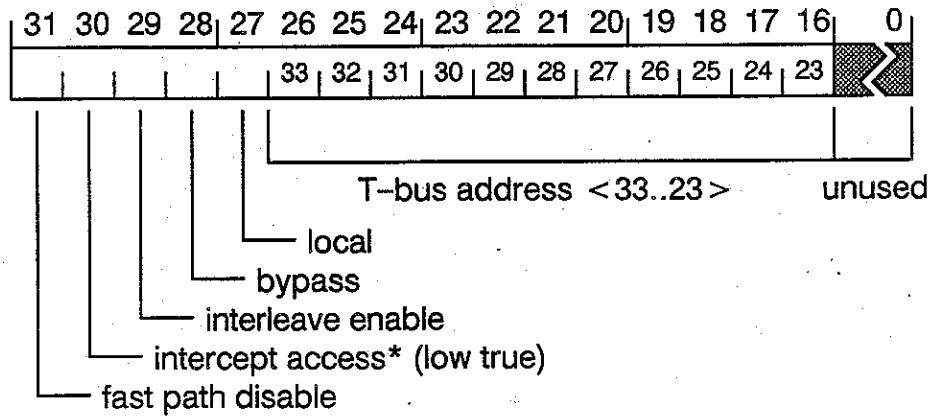
DESCRIPTION When the CPU interface to the T-bus initiates a locked operation by using the *lock* bit in the Augmentation register, the CPU Lock Timer is loaded from this register and begins counting. It stops when the CPU interface frees the lock. If the timer expires, the CPU is not explicitly notified, but a FREE_LOCKS cycle is automatically generated. The CPU's T-bus master interface, however, still believes the lock is held. If the CPU references the locked resource without first terminating the locked transaction (by clearing the AR *lock* bit to "0"), the reference will be a MAINTAIN access, to which the slave will return a "maintain present" bus error.

A CPU lock is timed out after 0xFF-*t* microseconds, where *t* is the contents of the CPU Lock Timer register.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter.

After power-on, the contents of the CPU Lock Timer register are unspecified.

Figure 11-29 CPU Mapping RAM registers (CMR).



FUNCTION Map Physical Addresses to System Physical Addresses, and supply additional signals controlling references made by the CPU.

ADDRESS (accessed via the interleaver loader)

ACCESS read/write

DESCRIPTION Whenever the 88000 chip set performs an operation that results in a T-bus request, the CPU Mapping RAM translates the (M-bus) Physical Address from the CMMU into a (T-bus) System Physical Address for the access, and produces other signals controlling the access. The *CMR enable* bit in the Machine Configuration register controls whether the CMR performs these functions. If the *CMR enable* bit is "1", the functions described below are performed on all CPU accesses that proceed beyond the CMMU. If the *CMR enable* bit is "0", the CMR is disabled as described at the end below.

Whenever the CPU Mapping RAM is involved, one of 1024 registers is selected by ten bits. Nine of these bits are Physical Address bits 31..23, and the tenth bit is the write signal. Therefore, each CMR register controls the mapping and access characteristics of an 8-megabyte bank (in both the Physical Address space and the System Physical Address space). Further, one CMR entry governs read operations in that bank, and another CMR entry governs write access in the bank. (However, a few of the bits are common to read and write, as discussed later and marked with ‡ below.)

Section 11.10 gives a higher level description of the CPU Mapping RAM.

The *T-bus address <33..23>* bits of the selected CMR register supply bits 33..23 of the System Physical Address; bits 22..0 are copied directly from the Physical Address to the System Physical Address. The resulting System Physical Address is placed on the T-bus or, if the fast path is used, sent directly to the local memory module.

‡ *T-bus address <23>* has no read/write distinction, as described below.

The *local* bit, when set to "1", says that the reference should not go out over the switch, but instead should be serviced by the local memory module, the control and configuration registers, or the VMEbus interface.

The intent of the *local* bit is to identify accesses that address resources on the local function board — memory, control and configuration registers, or the VMEbus interface — rather than remote resources. The local resources can be accessed more quickly without going over the switch. When the *local* bit in the selected CMR register is set to "1", the hardware "shortcuts" the switch by requesting a response locally. When the bit is "0", the access will go out over the switch even if it addresses a resource on the local board. The *local* bit interacts with other signals as follows.

- When the CMR *local* bit is set to "1", the *path* bits in the PCR are ignored and the T_PATH bits on the T-bus are driven with "11", inhibiting the use of either switch interface. (Therefore, the value of T-bus address bits T_AD<33..25> is irrelevant, although they are driven from the CMR onto the T-bus to provide valid logic levels as required by the T-bus specification.)
- Setting both the *local* bit and the *interleave enable* bit in the same CMR register is legal and plausible, since interleaving is further controlled on a page-by-page basis by the Interleave Decision RAM. If an access has interleaving enabled in both the CMR and the Interleave Decision RAM, then the T-bus signal T_INTERLEAVED is asserted — this identifies an interleaved access. The intent is that any interleaved access go over the switch (as it must, to use the modified address bits from the interleaver). If the CMR *local* bit is also asserted, ***its effect is suppressed by interleaving and the PCR path bits are used despite the CMR local bit.***

‡ The *bypass* bit, when set to "1" in the selected CMR register, causes references to bypass the TC2000 locking protocol. The CPU interface accomplishes this by asserting the T-bus signal T_LOCKOP<1..0> = "01", meaning "bypassed" rather than "normal", "open" or "maintain". The VMEbus interface has a similar capability, controlled by the VMEbus Slave Map RAM. When the *bypass* bit is asserted, it overrides the AR *lock* bit. Thus, a bypassed reference not only circumvents a lock held by another device, but also suppresses the CPU's own in-progress locked sequence (if any).

‡ The *interleave enable* bit, when set to "1" in the selected CMR register, permits interleaving in the associated 8-megabyte bank. Interleaving is further controlled by the Interleave Decision RAM registers; see their description for further details.

The *intercept access** bit, when cleared to "0" in the selected CMR register, causes the access to be acknowledged as if it has completed successfully, but in fact no T-bus access will be made. On a write, the data is discarded; on a read, the returned data is undefined. This mechanism is useful in speeding up certain block transfer operations using the CMMU's cache, as described in section 11.10.3. When the *intercept access** bit is set to "1", the access proceeds normally. Note that this bit is ***low true***. The intercept access mechanism is intended for use only on reads.

‡ The *fast path disable* bit, when cleared to "0" in the selected CMR register, causes the CPU's read references to local memory to use the fast path. The fast path applies the request directly to the memory module, without using the T-bus, and is therefore three clock ticks faster than using the T-bus. The data is returned over the T-bus as usual. The fast path benefits only ***local memory read*** operations. The fast path is not employed if the operation is a memory write. Control and configuration register accesses, VMEbus accesses, and switch references do not use the fast path either. When the fast path disable bit is set to "1", no references use the fast path. The fast path mechanism interacts with other signals and with VMEbus access, as noted in the caution below. The fast path will be used only if all the following conditions are met:

- Reference must be a read (not a write) to local memory (not to configuration and control registers, to VMEbus address space, or to resources over the switch)
- Machine Configuration register *fast path enable* bit is "1"
- *fast path disable* bit in the selected CPU Mapping RAM entry is "0"
- The reference cannot be to interleaved memory; that is, either the CMR *interleave enable* bit in the selected entry must be "0", or the single bit in the selected entry of the Interleave Decision RAM must be "0" (if both of these bits are "1", the reference will go out over the switch as an interleaved access and not use the fast path)

CAUTION

INTERACTIONS
Since the CMR mapping RAM controls features that interact, a few combinations of settings are either illegal (cause indeterminate behavior) or undesirable (yield correct results, but lower performance). These are as follows:

ILLEGAL: **fast path** and **VMEbus access**
It is *illegal* to access VMEbus address space using a CMR register whose *fast path disable* bit is "0". If this rule is violated, the results are indeterminate and depend on the T-bus traffic at the time. If the fast path is not available, a correct VMEbus reference is made. However, if the fast path is available, the contents of local memory at the offset addressed will be returned.

ILLEGAL: **fast path** and **read local Interprocessor Interrupt register**
If the *fast path disable* bit in the selected CMR register is "0" during a read

of the *local* Interprocessor Interrupt register at its *normal* address, unspecified data will be returned. To read the local Interprocessor Interrupt register, the alternate address should be used as described under the Interprocessor Interrupt register.

UNDESIRABLE / ILLEGAL: **fast path** and **not local**
This combination of CMR bits — allowing fast path but saying not local — is inconsistent because the fast path mechanism applies only to local memory accesses. If the *fast path disable* bit in the selected CMR register is "0", and the access qualifies to use the fast path (it is a read to local memory), but the *local* bit in the CMR is "0", three different outcomes are possible. Depending on the traffic conditions on the local T–bus, the access may use the fast path or may instead go out over the switch. If the fast path is used, the correct data is returned with no delay. If the access goes out over the switch ***and*** the CMR entry used maps the access ***back to this function board***, the correct data is returned but the access is slower than usual. If the access goes out over the switch and the CMR entry used maps it ***to another function board***, that function board is accessed and different data is returned. Thus, depending on the CMR mapping, the combination of fast path and not local may be viewed either as undesirable or as illegal.

UNDESIRABLE: **fast path** and **intercept access**
If the selected CMR register attempts to do both intercept access and fast path (that is, the *intercept access** bit = "0" and the *fast path disable* bit = "0", the correct data will be returned but the time to complete the access will vary by four clock ticks or more, depending on the availability of the fast path. This combination of bits is legal, but would be unusual because the intercept access mechanism is intended to terminate *remote* references in certain special cases, whereas the fast path applies only to *local* memory references.

(END OF "INTERACTIONS" CAUTION)

‡ Most bits in the CMR are duplicated, one bit controlling read accesses and the other bit controlling write accesses for the same 8–megabyte bank. However, four of the bits do not have this read/write distinction; the same one bit controls accesses to the given 8–megabyte bank, regardless of whether it is a read or a write. Similarly, when loading the CMR or copying out its contents via the interleaver loader, each of these four bits has no read/write distinction. For example, setting "bank 5 for reading" to be bypass, and then setting "bank 5 for writing" to be not bypass, results in any access to bank 5 being not bypass.

Normally, CMR registers are set up in pairs, with identical values in the registers selected by read and write operations to the same bank. The intended purpose of the read/write distinction on some of the CMR fields is to support the block copy operation described in section 11.10.3. To meet the performance goals for the fast path, the read/write distinction was eliminated for four bits that affect the critical timing paths on the board. The four bits with ***no read/write distinction*** are flagged with ‡ in the descriptions above, and are:

fast path disable bit

interleave enable bit

bypass bit

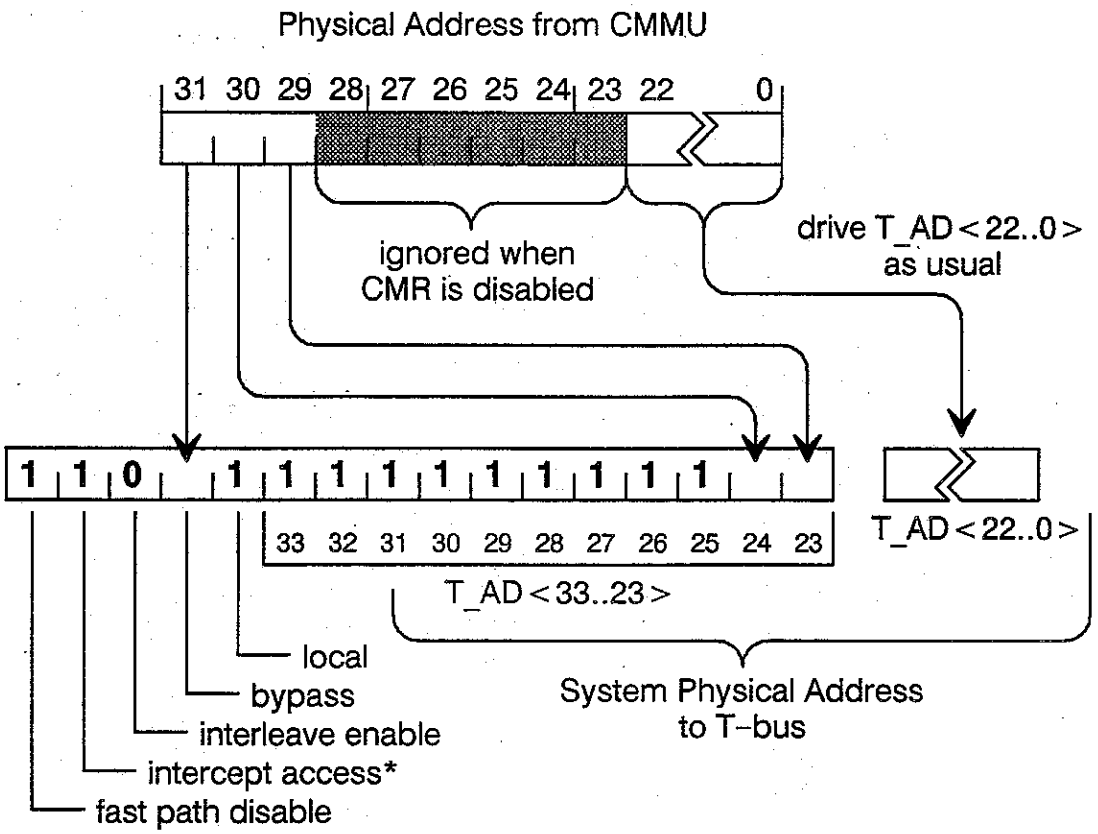
T–bus address <23> bit (but T–bus address bits 33..24 do have separate CMR bits to distinguish read and write)

The effect of having no read/write distinction on the *fast path enable*, *interleave enable* and *bypass* bits is straightforward — within an 8–megabyte bank, both reads and writes have the same behavior. (The *fast path disable* bit doesn't matter on a write, since writes never use the fast path anyway.) The effect of the *T–bus address <23>* bit having no read/write distinction, however, is slightly more subtle. If all T–bus address bits produced by the CMR did have read/write distinction, then any given 8–megabyte Physical Address bank could be mapped to any System Physical Address bank on read, and *independently* to any System Physical Address bank on write. However, since the *T–bus address <23>* bit does not have read/write distinction, those two System Physical Address banks must agree in bit 23 — they must either both be even–numbered 8–megabyte banks, or both be odd–numbered 8–megabyte banks. This constraint should be minor in practice, because the mapping of addresses is usually identical for read and write.

The CPU Mapping RAM is read and written by the same technique used to read and write the interleaver, described and illustrated in the Interleaver Loader registers entry.

After power–on, the contents of the CPU Mapping RAM registers are unspecified. However, since the *CMR enable* bit in the Machine Configuration register is "0" after power–on, ***the CMR is disabled after power–on.*** Whenever the CMR is disabled (by the *CMR enable* bit in the MCR being cleared to "0"), default values supercede the functions normally controlled by the CMR. These disabled–CMR defaults are shown in Figure 11–30. They give the processor simple access to local memory so it can execute bootstrap code that initializes the CMR and then enables it (by setting the *CMR enable* bit in the MCR to "1").

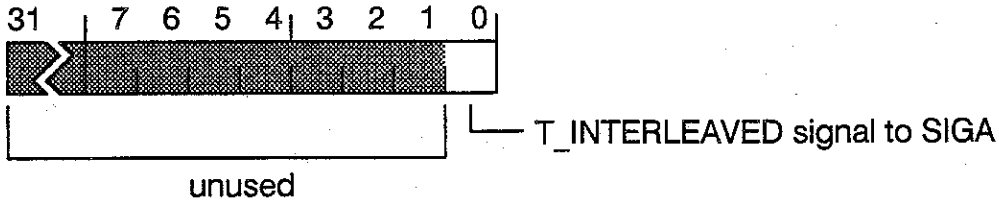
Figure 11-30 Power-up and disabled CMR operation.



In summary, when the CMR is disabled:

- The System Physical Address (T_AD) is derived as follows:
  - T_AD<22..0> come from Physical Address <22..0>, as usual
  - T_AD<24..23> come from Physical Address <30..29>
  - T_AD<33..25> are set to all "1"s
- Physical Address <28..23> are ignored (the T_AD mapping above permits access to the entire 32-megabyte function board address space)
- *local* is "1", so all references are local
- *bypass* comes from Physical Address <31>, so the CPU can bypass locks by using a Physical Address with its high bit set to "1"
- *interleave enable* is "0", so interleaving is not performed
- *intercept access** is "1", so accesses are normal (they are not intercepted)
- *fast path disable* is "1", so the fast path is not used

Figure 11-31 Interleave Decision RAM registers.



FUNCTION Tell SIGA whether the CPU's reference is to interleaved memory.

ADDRESS (accessed via the interleaver loader)

ACCESS read/write

DESCRIPTION The CPU interface in the TC/FPV contains the Interleave Decision RAM. This, in conjunction with the CPU Mapping RAM *interleave enable* bit, tells the SIGA whether a switch access made by the CPU is to interleaved or non-interleaved memory. Section 11.3.6 discusses the Interleave Decision RAM.

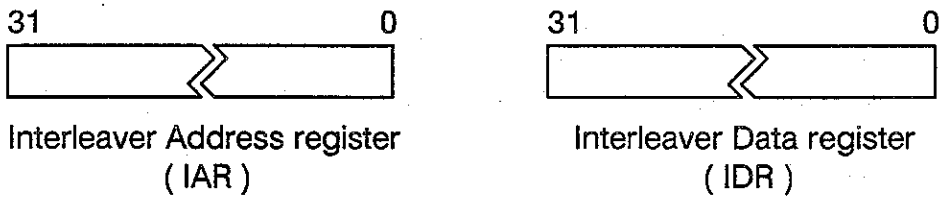
When the CPU makes a reference on the T-bus, Physical Address bits 31..26 and 22..15 select one of the 16,384 one-bit Interleave Decision RAM registers. If the selected Interleave Decision RAM register contains a "1", the reference is to interleaved memory; a "0" indicates the reference is to non-interleaved memory. However, if the *interleave enable* bit of the CPU Mapping RAM is "0", the contents of the Interleave Decision RAM are ignored and interleaving is not allowed. (For the SIGA to use the interleaver, the *Ena_Interleaver* bit in the SIGA's Requester_ConfigB register must also be set to "1".)

The Interleave Decision RAM is read and written by the same special technique used to read and write the interleaver, described and illustrated in the Interleaver Loader registers entry.

(Note that both the Interleaver Data register and the Interleave Decision RAM have initials "IDR". In this document, IDR always means the Interleaver Data register; the name Interleave Decision RAM is always written out in full.)

After power-on, the contents of the Interleave Decision RAM registers are unspecified.

Figure 11–32 Interleaver Loader registers.



FUNCTION Give the CPU read/write access to the interleaver mapping RAMs (the modulus RAM and the pool RAM). Also used for access to the Interleave Decision RAM and the CPU Mapping RAM in the CPU interface, and to the VMEbus Master Map RAM in the VMEbus interface.

ADDRESS 0xE07EA000 and following — see Figure 11–33

ACCESS IAR and IDR are read/write; other addresses are read only with side effects

DESCRIPTION When the TC/FPV generates a remote reference (and therefore a switch request), the System Physical Address from the T-bus is optionally modified by replacing the bits T_AD<33..25> with nine modified bits generated by the interleaver. The T_INTERLEAVED signal to the SIGA controls whether the SIGA uses the unmodified address straight from the T-bus or the address as modified by bits from the interleaver. The interleaver contains two RAMs — the modulus RAM and the pool RAM. The ***interleaver loader*** is used to read and write these RAMs. It is also used to read and write three other RAMs: the Interleave Decision RAM, the CPU Mapping RAM and the VMEbus master mapper RAM.

Figure 11–33 lists the SIGA addresses used to operate the interleaver loader. Most of these addresses hold no data, but trigger an operation when accessed. The two registers that do hold data are:

- Interleaver Address register (IAR)
  The IAR contains the mapping RAM address to be read or written. For example, if the IAR contains "10", then location 10 in one of the RAMs will be read or written.
- Interleaver Data register (IDR)
  The IDR contains the data to be written, or receives the data that is read.

  (Note that both the Interleaver Data register and the Interleave Decision RAM have initials "IDR". In this document, IDR always means the Interleaver Data register; the name Interleave Decision RAM is always written out in full.)

Figure 11-33 Interleaver loader registers and commands.

Physical Address	Access	Function
E07EB000	R/W	IDR (Interleaver Data register)
E07EA00C	R/W	IAR (Interleaver Address register)
E07EA000	Read only	IDR ⇒ pool RAM
E07EA004	Read only	IDR ⇒ modulus RAM
E07EA008	Read only	IDR ⇒ VMEbus master map RAM
E07EA010	Read only	IDR ⇒ CPU Mapping RAM
E07EA014	Read only	IDR ⇒ Interleave Decision RAM
E07EA020	Read only	IDR ⇐ pool RAM
E07EA024	Read only	IDR ⇐ modulus RAM
E07EA028	Read only	IDR ⇐ VMEbus master map RAM
E07EA030	Read only	IDR ⇐ CPU Mapping RAM
E07EA034	Read only	IDR ⇐ Interleave Decision RAM

To access a mapping RAM, the following steps are used.

1. Write the address of the RAM location to be accessed into the IAR. The IAR bits are the same as the T-bus address/data (T_AD) bits that drive the RAM's address lines in normal operation, with three exceptions:
   - Bits IAR<1..0> take the place of T_AD<33..32>.
   - When accessing the interleaver modulus RAM, bits IAR<15..13> are substituted for the pool number that normally comes from the pool RAM. Thus, the contents of the interleaver's pool RAM don't matter while accessing the modulus RAM.
   - When accessing the Interleave Decision RAM, bits from the IAR substitute for Physical Address bits from the CMMU, not for System Physical Address bits from the T-bus.
2. If the RAM location is to be written, write the data to be written into the IDR.
3. Issue the interleaver loader command, by reading from the appropriate address in Figure 11-33.
4. If the RAM location was read, obtain the data by reading the IDR.

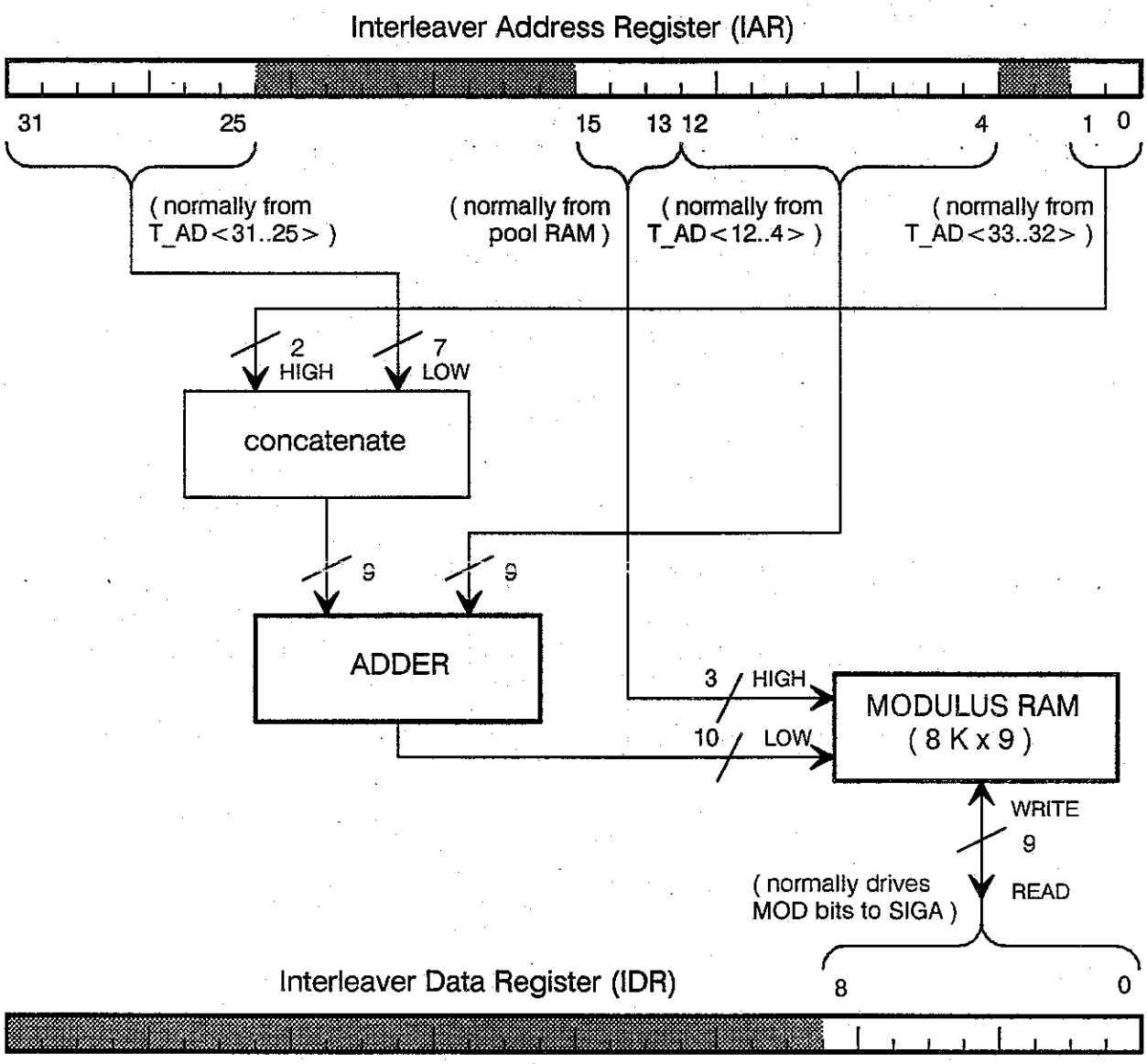
NOTE

NO READ-BACK ON INTERLEAVER DATA REGISTER
The IDR acts as two registers, one that is written by the CPU and whose contents go into RAM, and one that is loaded from RAM and whose contents are read by the CPU. Data written into the IDR cannot be read directly from the IDR. Therefore, to test operation of the IDR, data written to it must be transferred into a RAM and back from RAM into the IDR, from where the CPU can then read it. CPU ⇒ IDR ⇒ RAM ⇒ IDR ⇒ CPU.

The format of the data that the CPU must deposit in the IAR and the IDR when accessing a given RAM via the interleaver loader is illustrated below.

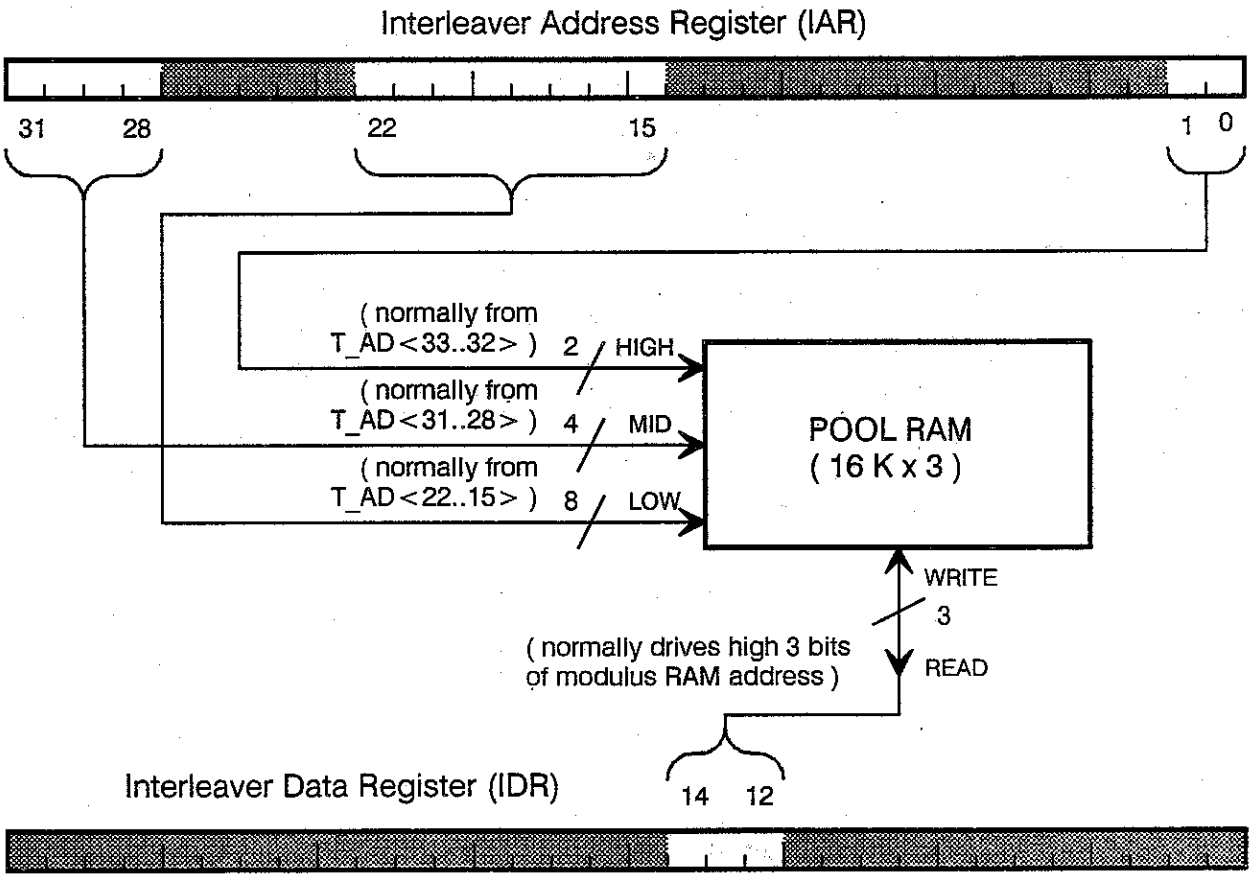
After power-on, the contents of the interleaver modulus and pool RAM registers are unspecified. For the power-on contents of other RAMs loaded by the interleaver loader, see the description of those RAMs.

Figure 11-34 Interleaver loader access to modulus RAM.



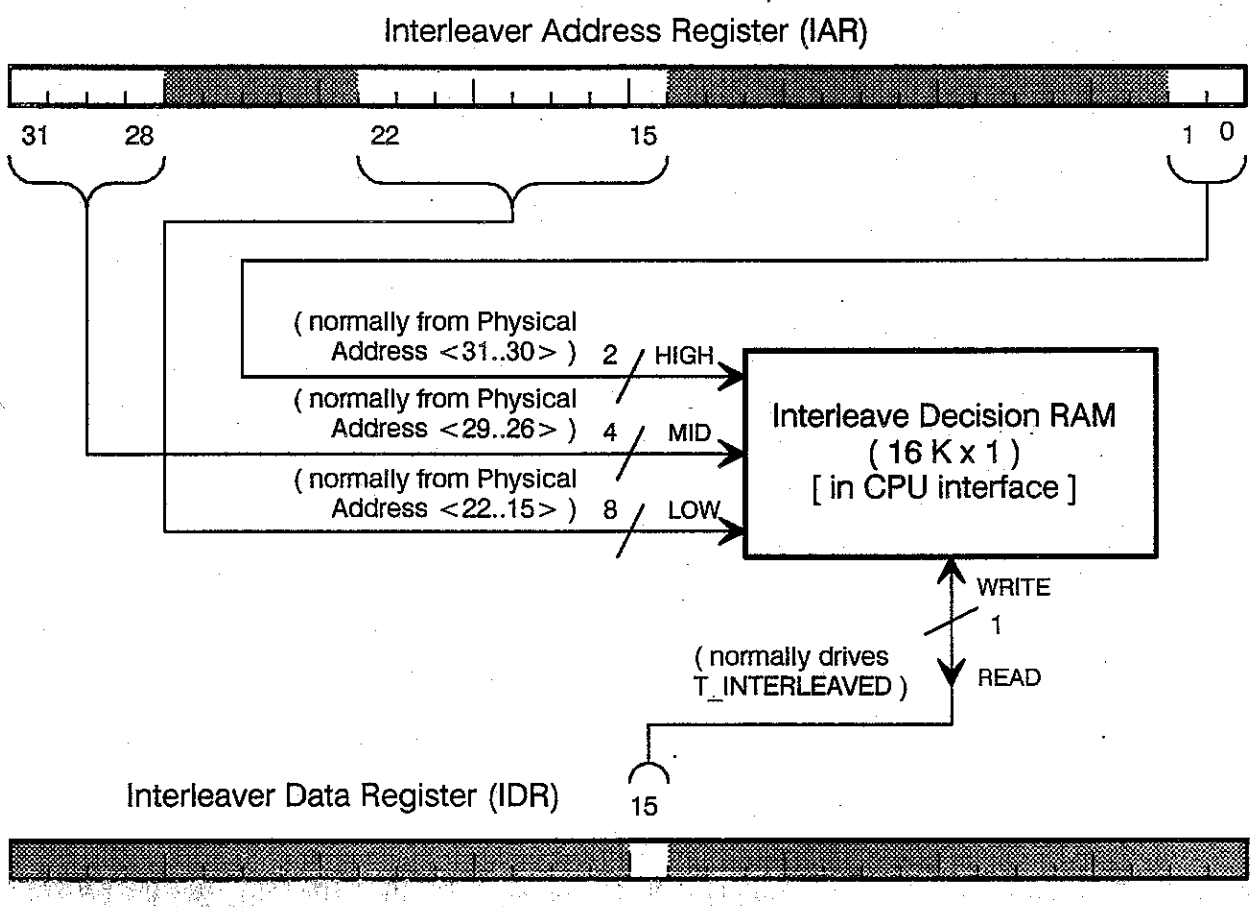
NOTE: the Modulus RAM is normally driven from the **System Physical (T-bus) Address**, the result after translation by the CPU Mapping RAM.

Figure 11-35 Interleaver loader access to pool RAM.



NOTE: the Pool RAM is normally driven from the **System Physical (T-bus) Address**, the result after translation by the CPU Mapping RAM.

Figure 11-36 Interleaver loader access to Interleave Decision RAM.



NOTE: the Interleave Decision RAM is normally driven from the **Physical Address**, the output of the CMMU before translation by the CPU Mapping RAM.

Figure 11-37 Interleaver loader access to CPU Mapping RAM.

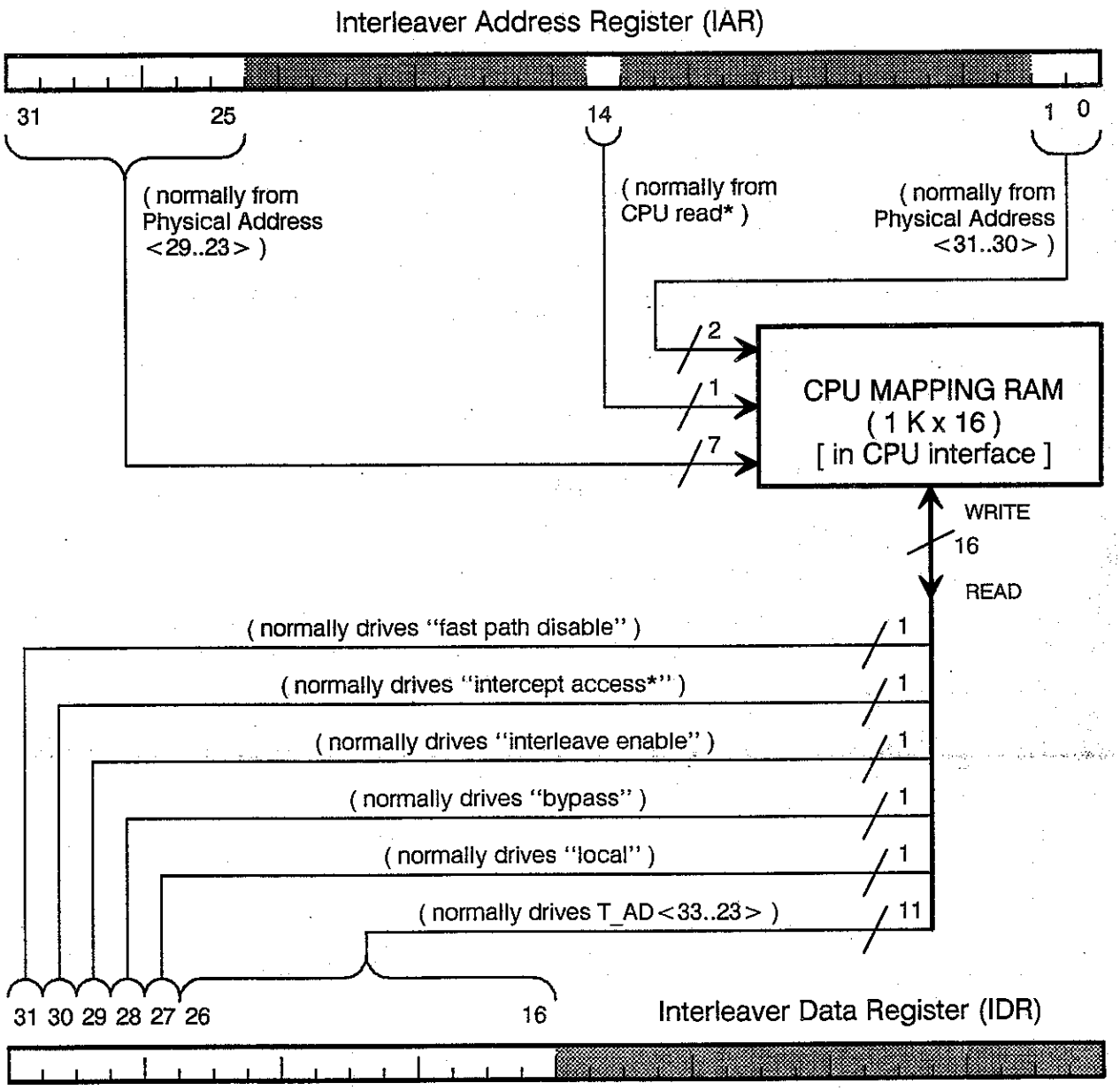
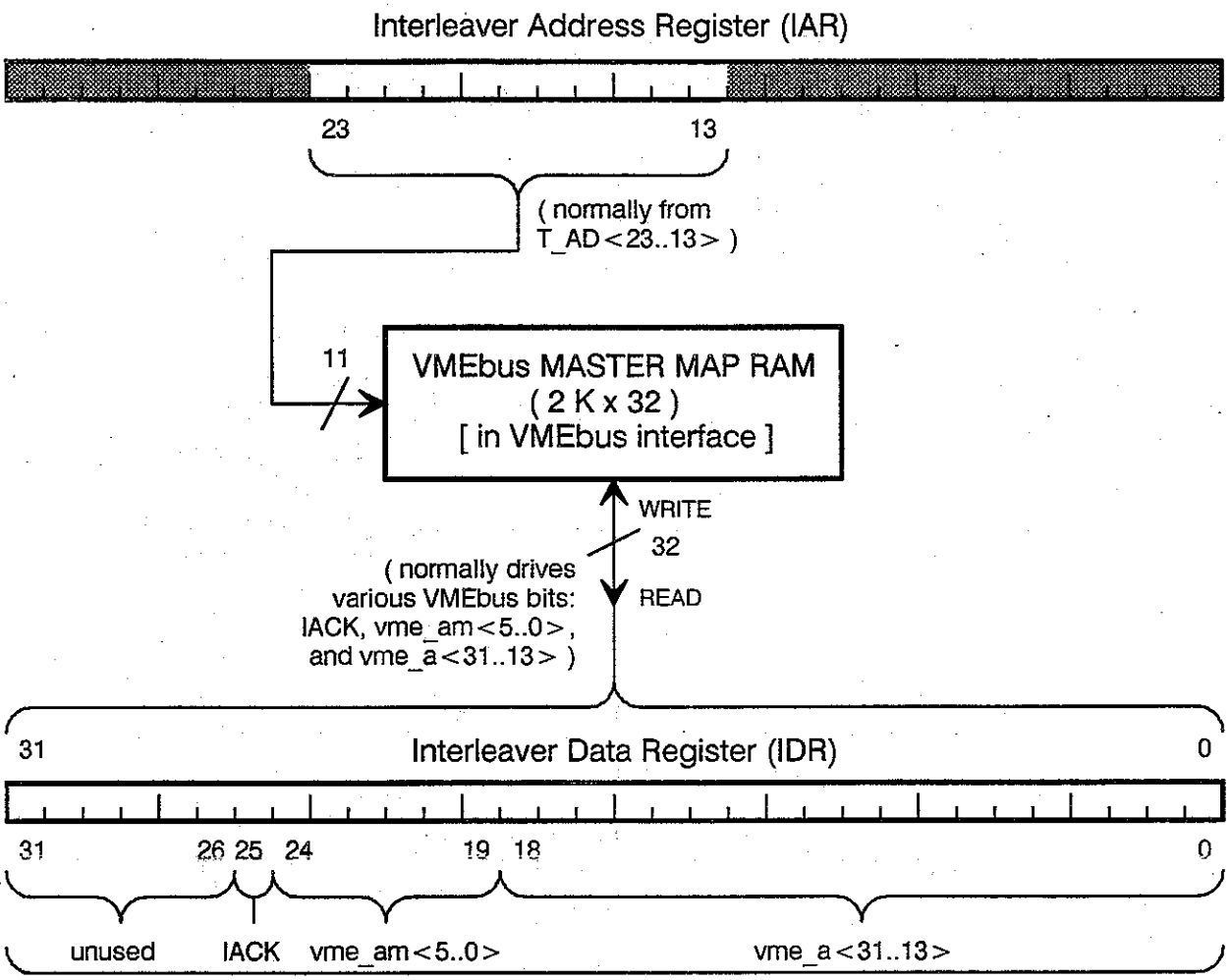
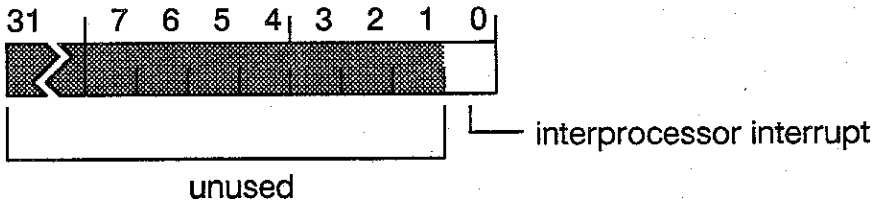


Figure 11-38 Interleaver loader access to VMEbus Master Map RAM.



function of fields in VMEbus Master Map RAM

Figure 11–39 Interprocessor Interrupt register.



FUNCTION Interrupt the CPU.

ADDRESS 0x80002000 — normal address
0xE0740004 — alternate address, for local reading

ACCESS normal address — read (except locally with fast path) / write
alternate address — read/write

DESCRIPTION Setting the *interprocessor interrupt* bit in this register requests an interrupt to the CPU. Any T-bus master, including the VMEbus slave interface and the switch interface, may write this register and thereby interrupt this CPU, subject to the Interrupt Enable Mask register. It is expected that the operating system will limit access to this register by setting the protection attributes of the page that contains it. This bit is cleared either by writing a "0" into it, or by reading the Interrupt Source register while the Interrupt Enable Mask register *interprocessor interrupt* bit is "1".

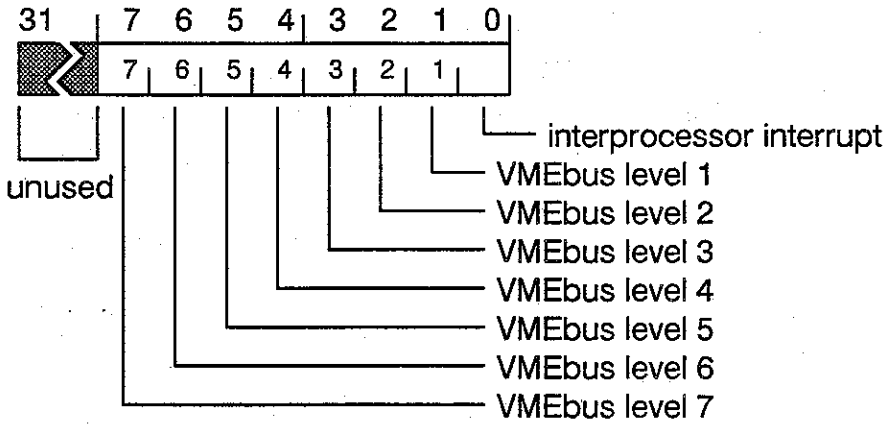
If interprocessor interrupts are disabled by the Interrupt Enable Mask register, the requested interrupt will not be seen by the CPU until interprocessor interrupts are re–enabled.

The Interprocessor Interrupt register may be read and written at any time. Reading the Interprocessor Interrupt register always returns its contents, regardless of whether interprocessor interrupts are disabled by the Interrupt Enable Mask register.

Reading the Interprocessor Interrupt register at its normal address with the ***fast path*** enabled returns an ***unspecified value***. To read the register locally when the fast path is enabled, the ***alternate address*** must be used. Since writes do not use the fast path, the local Interprocessor Interrupt register can be written at either address, regardless of the state of the *fast path disable* bit in the CMR. Note that read access to the Interprocessor Interrupt register is provided largely for diagnostic capability; operating system functions would usually read the state of this register by reading the Interrupt Source register.

Power–on clears the one defined bit of the Interprocessor Interrupt register to zero.

Figure 11–40 Interrupt Enable Mask register.



FUNCTION Enable interprocessor and VMEbus interrupts to the CPU.

ADDRESS 0xE0780014

ACCESS read/write

DESCRIPTION If a *VMEbus* mask bit in this register is "1", then the corresponding interrupt is recognized by the CPU. If a VMEbus mask bit is "0", then the CPU is insensitive to assertion of the VMEbus interrupt request on that level. The VMEbus interrupt request state is not latched in the TC/FPV; if the enable bit for a level is "1", the state seen by the CPU is the state of the corresponding interrupt line on the VMEbus.

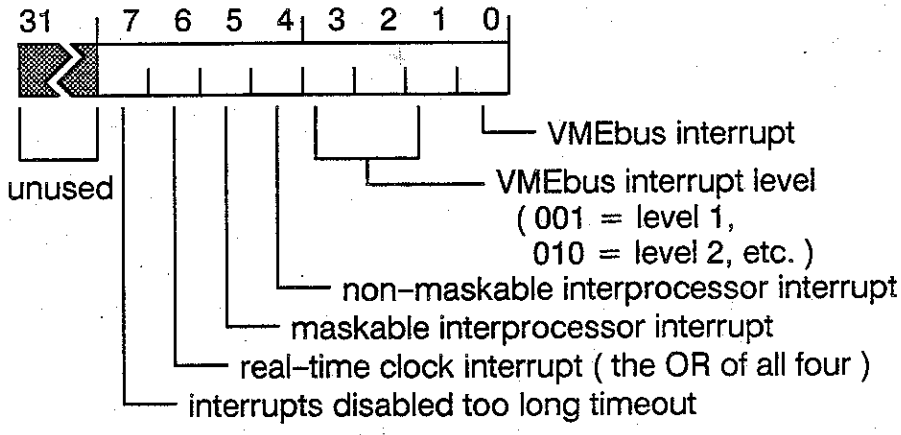
If the *interprocessor interrupt* mask bit is "1", a "1" in the Interprocessor Interrupt register asserts an interrupt to the CPU. If the bit is "0", the CPU is insensitive to the contents of the Interprocessor Interrupt register. Clearing the mask bit in the Interrupt Enable Mask register to "0" prevents (maskable) interprocessor interrupts to the CPU, but it does *not* clear or disable access to the Interprocessor Interrupt register.

The *interprocessor interrupt* bit also controls whether reading the Interrupt Source register clears the Interprocessor Interrupt register. If the bit is "1", reading the Interrupt Source register clears the Interprocessor Interrupt register; if the bit is "0", reading the Interrupt Source register does not clear the Interprocessor Interrupt register.

The Interrupt Enable Mask register bits also control whether the corresponding interrupt requests are visible in the Interrupt Source register. If the mask bit is "0", the corresponding bit in the Interrupt Source register reads as "0" regardless of whether that interrupt request is present.

Power-on clears all defined bits of the Interrupt Enable Mask register to zero, disabling the interrupts masked by this register.

Figure 11–41 Interrupt Source register.



FUNCTION Indicate the cause of a CPU interrupt.

ADDRESS 0xE0780018

ACCESS read only

DESCRIPTION There are fourteen sources of interrupts on the TC/FPV. The 88100 has only one interrupt level, so all interrupt sources are OR'ed together to generate a CPU interrupt. More than one type of interrupt may be asserted. Only the VMEbus interrupts are prioritized.

The *VMEbus interrupt* bit, if "1", indicates that at least one VMEbus interrupt request is pending (and that its bit in the Interrupt Enable Mask register is "1"). These levels are prioritized, and the *VMEbus interrupt level* bits indicate the level of the highest priority request currently pending. (The highest level is 7, the lowest is 1. A value of 0 should never be seen when the *VMEbus interrupt* bit is "1".) If the VMEbus interrupt bit is "0", the contents of the VMEbus interrupt level field is unspecified.

The *non–maskable interprocessor interrupt* bit, if "1", indicates that the interrupt bit of the Non–maskable Interprocessor Interrupt register has been set to "1" (and that its bit in the Interrupt Enable Mask register is "1").

The *maskable interprocessor interrupt* bit, if "1", indicates that the interrupt bit of the Interprocessor Interrupt register is "1" and the interprocessor interrupt bit of the Interrupt Enable Mask register is "1". If either bit is "0", this bit of the Interrupt Source register is "0".

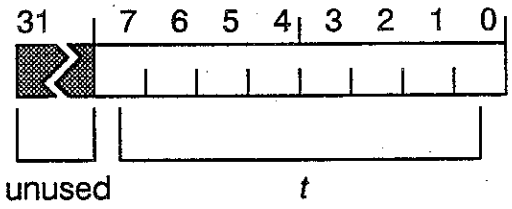
The *real–time clock interrupt* bit, if "1", indicates that one or both of the Time Of Next Interrupt (TONI) registers, in one or both of the SIGAs, has expired. Each SIGA has two real–time timers, but in normal operation, only one SIGA will be producing such interrupts.

The *interrupts disabled too long timeout* bit, if "1", indicates that the Interrupts Disabled Timer has expired.

PROGRAMMING ADVICE: Reading the Interrupt Source register clears the non-maskable interprocessor interrupts, and clears the maskable interprocessor interrupts if the Interrupt Enable Mask register's interprocessor interrupt bit is "1". If the Augmentation Register is being used in such a way that interrupts can be taken during locked sequences, the AR must be saved and cleared whenever an interrupt service routine is entered. This can be done by reading the AR read-and-clear address. The interrupts disabled too long timeout will be cleared within 300 nanoseconds of clearing the AR. The VMEbus interrupt gets cleared by doing an IACK cycle on the VMEbus. Each request must be given a separate IACK response. The real time clock interrupt is cleared by writing a value greater than the current time to the SIGA's TONI register that interrupted, or by changing that SIGA's configuration register to disable TONI interrupts.

After power-on, the contents of the Interrupt Source register are unspecified.

Figure 11–42 Interrupts Disabled Timer register.



FUNCTION Help guarantee maximum latency of interrupt servicing, by detecting protracted use of the *disable interrupts* augmentation.

ADDRESS 0xE076000C

ACCESS read (timer counter) / write (preset register)

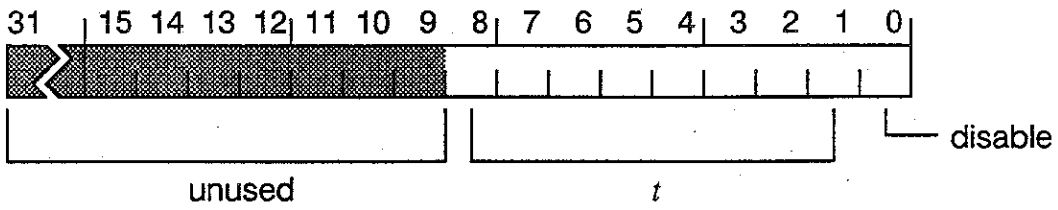
DESCRIPTION When the *disable interrupts* bit in the Augmentation register is set to one, the Interrupts Disabled Timer is loaded from this register and begins counting. It stops when the disable interrupts bit is cleared to zero. If the timer expires, an "interrupts disabled too long" interrupt is generated. That interrupt is cleared by clearing the disable interrupts bit in the AR.

The interrupts disabled augmentation is timed out after 0xFF–*t* microseconds, where *t* is the contents of the Interrupts Disabled Timer register.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter.

After power–on, the contents of the Interrupts Disabled Timer register are unspecified. The CPU must initialize this register before setting the *disable interrupts* bit of the AR. Further, it may be desirable to initialize this register before setting any bits in the Interrupt Enable Mask register.

Figure 11–43 Interrupts Pending / Abort Retries register.



FUNCTION Help guarantee maximum latency of interrupt servicing, by aborting switch transactions that take too long to establish a connection.

ADDRESS 0xE0760008

ACCESS read (timer counter and *disable* bit) / write (preset register and *disable* bit)

DESCRIPTION When a CPU interrupt request is asserted, a timer is loaded from the Interrupts Pending / Abort Retries register and begins counting. It stops when **all** interrupt requests are gone. If the timer expires, the signal SIGA_ABORT_RETRIES is asserted until **all** interrupt requests are gone. This signal has no effect unless the SIGA is trying to establish a switch connection and has been rejected at least once. If the SIGA has been rejected and is about to retry, it will abort the attempt and return a bus error.

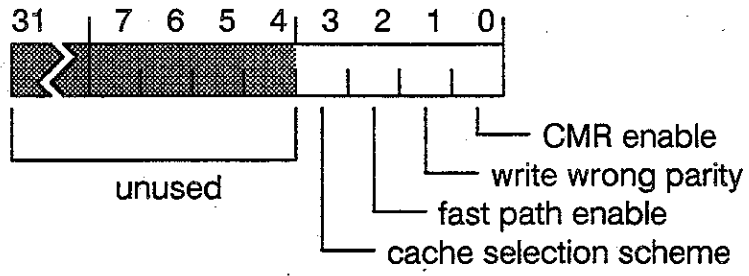
When this timer expires and the SIGA consequently returns a bus error, that error is returned to whatever T-bus master made the switch request. Typically this is the CPU, but it could be the VMEbus interface. This reflects the design choice that having the CPU service interrupts is more important than any switch traffic, including VMEbus-related switch traffic. There is a danger, however, of handing this bus error to a VMEbus device that is not prepared to handle it. Therefore, if the Interrupts Pending / Abort Retries register is used to control switch latency, the user may wish to choose to never generate references across the switch from VMEbus devices. Alternatively, the user may ensure that any VMEbus devices making such references are prepared to handle a bus error.

A pending interrupt is timed out, if a remote reference is being attempted, after 0xFF–*t* microseconds, where *t* is the contents of the Interrupts Pending / Abort Retries register. If bit 0 of the register is set to "1", the timer is disabled and the value of *t* is irrelevant; bit 0 must be "0" for the timer to operate.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter. The *disable* bit is also read or written on any access.

After power-on, the contents of the Interrupts Pending / Abort Retries register are unspecified. The CPU must initialize this register before setting any bits in the Interrupt Enable Mask register.

Figure 11-44 Machine Configuration register.



FUNCTION Control certain basic operating characteristics.

ADDRESS 0xE0760000

ACCESS read/write

DESCRIPTION When the *CMR enable* bit in this register is set to "1", the CPU Mapping RAM operates on references made by the CPU. When the CMR enable bit is cleared to "0", the CPU Mapping RAM has no effect, and default values are supplied for the signals driven by the CPU Mapping RAM when it is enabled.

The *write wrong parity* bit, when set to "1", causes incorrect parity to be written, into local memory only, on any size of write, by any T-bus master (CPU, switch or VMEbus). This bit is intended for use in testing the memory. When this bit is cleared to "0", correct parity is written.

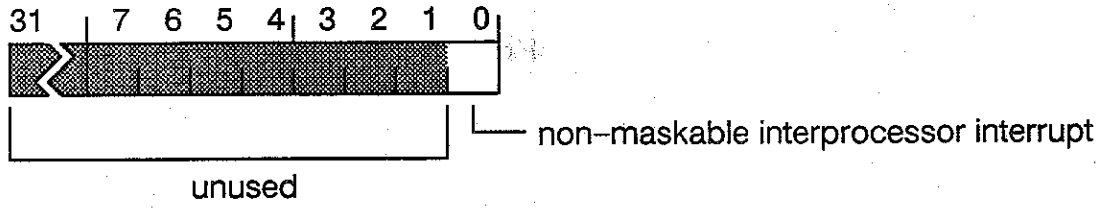
The *fast path enable* bit, when set to "1", enables the fast path. When the enable fast path bit is cleared to "0", the fast path to memory is never used. For the fast path to be used, several conditions must all be met, as discussed in the description of the CPU Mapping RAM.

The *cache selection scheme* bit determines which of two schemes is used to share the two code CMMUs. The scheme, in turn, determines which code CMMU is addressed on an instruction fetch, as follows:

- If *cache selection scheme* bit = "0", use Process Logical Address bit 12
  - If Process Logical Address bit 12 = "0", use code CMMU #1
  - If Process Logical Address bit 12 = "1", use code CMMU #2
- If *cache selection scheme* bit = "1", use CPU supervisor/user mode bit
  - If supervisor/user bit = "0" (user), use code CMMU #1
  - If supervisor/user bit = "1" (supervisor), use code CMMU #2

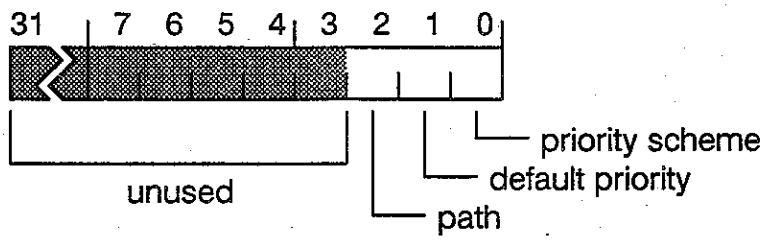
After power-on, the contents of the Machine Configuration register are unspecified. It is recommended that the CPU (or TCS) initialize this register early in the initialization code.

Figure 11–45 Non–maskable Interprocessor Interrupt register.



FUNCTION	Interrupt the CPU, unless the CPU itself has interrupts disabled.
ADDRESS	0xE0760004
ACCESS	read/write
DESCRIPTION	Setting the *non–maskable interprocessor interrupt* bit of this register generates a non–maskable interrupt to the CPU. This interrupts the CPU unless the CPU has all interrupts disabled via its internal Processor Status register. This bit is cleared either by writing a "0" into it, or by reading the Interrupt Source register. Power–on clears the one defined bit of the Non–maskable Interprocessor Interrupt register to zero.

Figure 11–46 PCR Disable Mask register.



FUNCTION Prohibit setting bits in the Process Configuration register.

ADDRESS 0xE0780010

ACCESS read/write

DESCRIPTION Setting a bit in this register disables write access to the corresponding *field* (not necessarily a single bit) in the Process Configuration register.

- Masking applies both in user mode and in supervisor mode.
- Masking applies only to writes, never to reads. Masking applies to all sources of a write: the CPU, the VMEbus slave, the switch interface, and the TCS.
- If the write attempts to set to a "1" any bit of any PCR field, while the mask bit corresponding to that field is "1", then the entire PCR is left unchanged and a bus error is generated. Otherwise, the resulting PCR bit is:

field's mask bit	data to be written	resulting bit in PCR
0	0	0
0	1	1
1	0	unchanged (no bus error)
1	1	bus error — see above

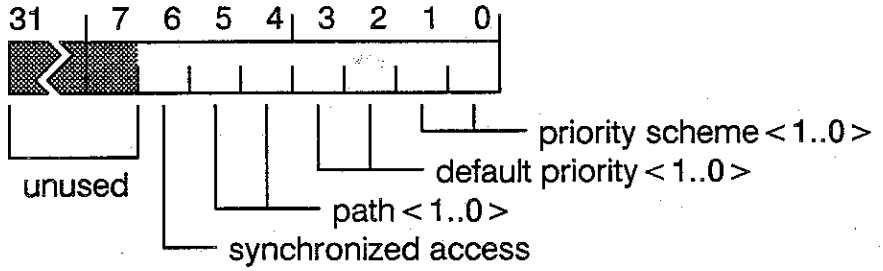
  Writing a "0" to a masked bit leaves the bit unchanged and incurs no bus error so the user may change other PCR bits that are not masked.

- Changing the PCR Disable Mask register does not affect the contents of the PCR.

Access to the PCR *synchronized access* bit is not protected by a mask bit. The reasoning behind this is that if a user program wishes to apply synchronized access to all its remote references, it will not hurt the performance of other user or system programs.

Power-on clears all defined bits of the PCR Disable Mask register to zero, so all bits of the PCR are initially writable.

Figure 11-47 Process Configuration register (PCR).



FUNCTION Hold certain information associated with the software process currently running on the processor.

ADDRESS 0xE0740000

ACCESS read/write, subject to PCR Disable Mask register

DESCRIPTION The PCR holds information about the way remote accesses are presented to the switch while the current process is executing.

The CPU interface uses the *priority scheme* bits to select one of four priority schemes. Each priority scheme uses an independent set of criteria for determining the value of the T_PRIORITY bits for every T-bus request. The T_PRIORITY bits are used by the SIGA to determine the priority when it converts a T-bus request into a switch request message. This mechanism affects references to remote memory only. The *default priority* bits may be used in any of the schemes. The priority mechanism and computation are described in section 11.9.1.

The *path* bits are used in computation of the T_PATH bits placed on the T-bus, that in turn determine the access path used. The computation of T_PATH is described in section 11.9.2. The meaning of T_PATH values, and therefore the path that the PCR *path* bits will specify if not overridden by the computation, is as follows:

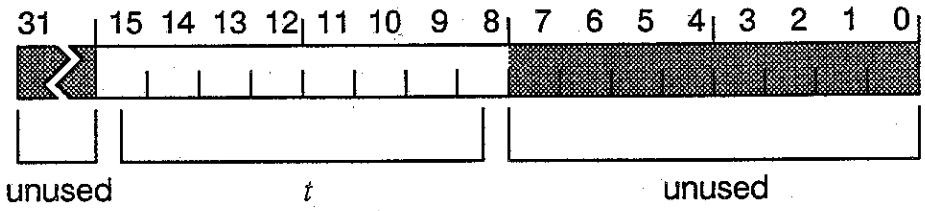
00	illegal (both SIGAs, if installed and enabled, will respond, causing indeterminate results)
01	use SIGA B
10	use SIGA A
11	local access (prohibits use of either switch; access can be served only by resources on the local function board — memory, configuration and control registers, or VMEbus interface)

The computation of T_PATH consists of a choice either to use the *path* bits from the PCR, or to use "11". If the access is local and not interleaved, then "11" is used.

The *synchronized access* bit, when set to one, asserts the T-bus bit T_SYNC on switch transactions, which tells the SIGA to use the "Slot0" transmission strategy. The intent is that the SIGA will be set up so this causes initial switch transmissions to use a delay similar to the backoff applied to retransmissions. Then, if several CPUs are all accessing the same server (such as memory at the same switch port), congestion is less than if initial transmissions were not delayed.

After power-on, the contents of the Process Configuration register are unspecified.

Figure 11–48 VMEbus Arbiter Timer register.



FUNCTION Detect when no VMEbus device responds to bus grant.

ADDRESS 0xE076020C

ACCESS read (timer counter) / write (preset register)

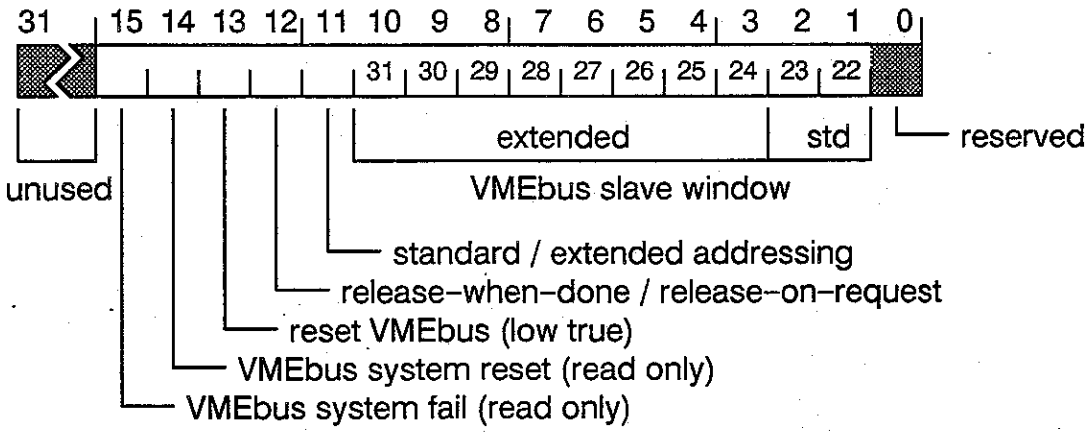
DESCRIPTION If the TC/FPV is VMEbus system controller, one of its duties as bus arbiter is to time out the *bus grant* signal. When the arbiter asserts bus grant, the VMEbus Arbiter Timer begins counting, and stops when a bus master acknowledges the grant by asserting the *bus busy* signal. If no device responds, expiration of this timer causes the arbiter to remove bus grant. This occurs entirely within the VMEbus, and is not explicitly visible to the TC/FPV CPU or to any other T–bus device. This timeout never occurs if all devices obey *The VMEbus Specification*; but timing out bus grant allows forward progress if a violation does occur.

The arbitration cycle is timed out after $4\times(0xFF-t)$ microseconds, where t is the contents of the VMEbus Arbiter Timer register. If the TC/FPV is system controller, the timer is initialized from the register each time the TC/FPV asserts bus grant.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter.

After power–on, the contents of the VMEbus Arbiter Timer register are unspecified.

Figure 11–49 VMEbus Configuration register.



FUNCTION Control the configuration of the TC/FPV as a device on the VMEbus.

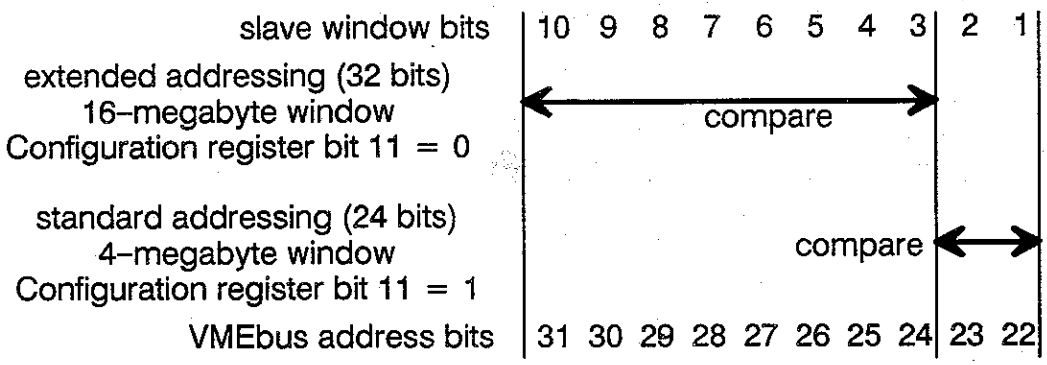
ADDRESS 0xE0760204

ACCESS read (all bits) / partial write (all but the *VMEbus system reset* and *fail* bits)

DESCRIPTION The VMEbus slave mapper compares the *VMEbus slave window* to the high bits of the address from the VMEbus, to determine whether the VMEbus is referencing TC2000 address space and therefore the TC/FPV should respond. See below for the number of bits compared.

The *standard / extended addressing* bit determines the type of VMEbus addressing to which the TC/FPV responds as a VMEbus slave. The type of addressing being used is encoded in the VMEbus address modifier bits for every VMEbus transaction. The TC/FPV VMEbus slave does not respond unless the addressing mode of the transaction matches the mode specified by this bit of the VMEbus Configuration register. This bit also determines how many bits of the VMEbus address are compared to the VMEbus slave window, and therefore the size of the window from VMEbus address space into TC2000 address space. Note that the addressing used when the TC/FPV is VMEbus master is not determined by this bit, but rather by the vme_am<5..0> bits of the VMEbus Master Map RAM register.

value	**addressing**
0	extended (32–bit); compare vme_a<31..24> against VMEbus Configuration register <10..3> (eight bits); window is 16 megabytes
1	standard (24–bit); compare vme_a<23..22> against VMEbus Configuration register <2..1> (two bits); window is 4 megabytes



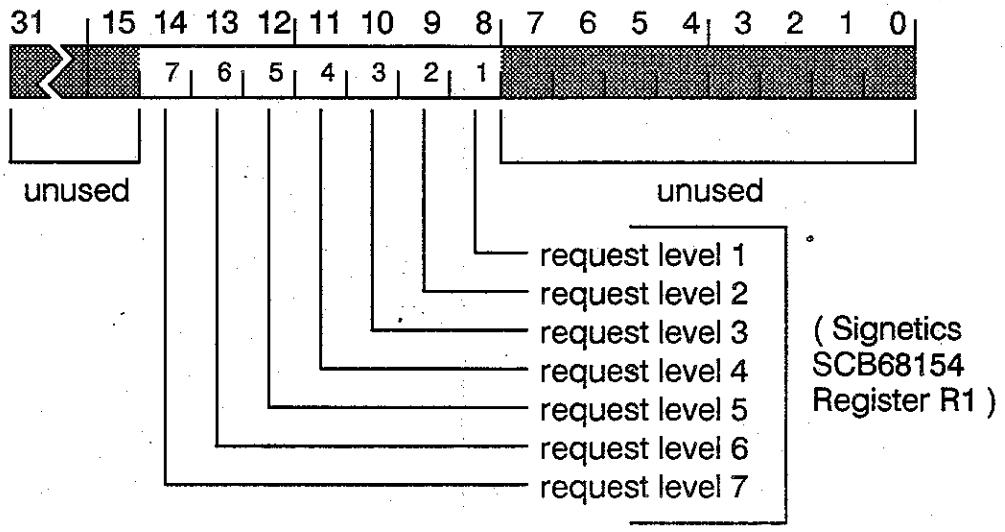
The *release–when–done / release–on–request* bit controls when the TC/FPV relinquishes VMEbus mastership. A "1" indicates release–when–done, a "0" release–on–request. The difference between the two settings is discussed in section 11.7.4 and more technically in *The VMEbus Specification*.

The *reset VMEbus* bit, when cleared to "0", generates a reset on the VMEbus as long as it remains "0". The bit must be set to "1" to de–assert the reset signal on the VMEbus. The low–true sense of this bit ensures that upon power–on, which clears the bit to zero, the VMEbus is held reset. Resetting the TC/FPV does not generate a VMEbus reset, nor does resetting the VMEbus reset the TC/FPV.

The *VMEbus system reset* and *VMEbus system fail* bits read as "1" if the VMEbus has been reset or suffered a system failure (not necessarily a power failure) since this register was last read; otherwise, these bits are "0". Reading this register clears these bits to "0". The setting and clearing of these bits is controlled by devices on the VMEbus; see *The VMEbus Specification* for details.

After power–on, the contents of the VMEbus Configuration register are as follows. Bits 9..0 (most of the VMEbus slave window) are unspecified. Bits 13..10 are zero. Bit 14 will be "1" because the TC/FPV itself is holding the VMEbus reset. Bit 15 depends on the state of other VMEbus devices.

Figure 11-50 VMEbus Interrupt Request register.



FUNCTION Select the level(s) on which the TC/FPV generates a VMEbus interrupt request.

ADDRESS 0xE0760804

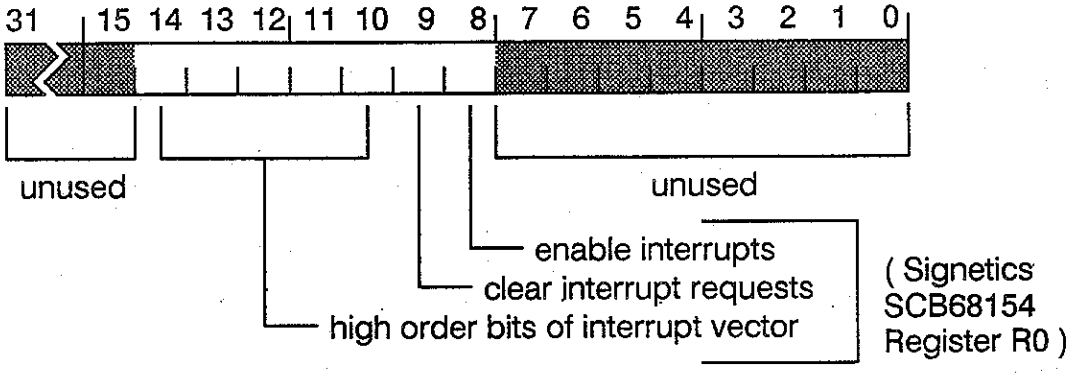
ACCESS read/write

DESCRIPTION Writing to this register generates an interrupt request(s) to the VMEbus, on each level whose corresponding bit in the data written is a "1". However, placing the request onto the VMEbus is under control of the *enable interrupts* bit in the VMEbus Interrupt Vector / Control register. If the TC/FPV is already requesting VMEbus interrupts on some levels, the levels requested by writing to this register are added to those already present, like an inclusive-OR function. The bits of this register are cleared individually by IACK cycles on the VMEbus, or all together by the *clear interrupts* bit of the VMEbus Interrupt Vector / Control register.

Refer to a Signetics SCB68154 data sheet for further details on this register, and on the associated VMEbus Interrupt Vector / Control register.

Power-on clears all defined bits of the VMEbus Interrupt Request register to zero.

Figure 11-51 VMEbus Interrupt Vector / Control register.



FUNCTION Control interrupts the TC/FPV generates onto the VMEbus.

ADDRESS 0xE0760800

ACCESS read/write

DESCRIPTION The *enable interrupts* bit, when set to "1", allows the TC/FPV to generate interrupt requests onto the VMEbus.

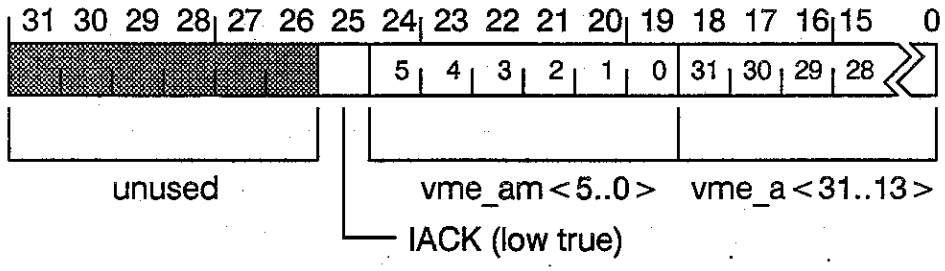
The *clear interrupt requests* bit, when set to "1", forces off all VMEbus interrupt requests that the TC/FPV is asserting. This clears the VMEbus Interrupt Request register.

The *high order bits of interrupt vector* field specifies the highest five bits of the interrupt vector that the TC/FPV will place on the VMEbus. The interrupt vector is placed on the bus when the TC/FPV has requested an interrupt and some VMEbus device has responded with IACK to our requested level. A VMEbus interrupt vector is eight bits. *The VMEbus Specification* defines that bit 0 is a "1". Bits 1 and 2 come from the SCB68154 chip, and its specification is that these bits are copied from VMEbus system bus address bits A2 and A3, respectively. (The VMEbus device acknowledging the TC/FPV's interrupt request is driving A1, A2 and A3 with the 3-bit code of the level being acknowledged.)

Refer to a Signetics SCB68154 data sheet for further details on this register, and on the associated VMEbus Interrupt Request register.

Power-on clears all defined bits of the VMEbus Interrupt Vector / Control register to zero.

Figure 11–52 VMEbus Master Map RAM registers.



FUNCTION Map TC2000 addresses into VMEbus addresses.

ADDRESS (accessed via the interleaver loader)

ACCESS read/write

DESCRIPTION When the TC/FPV is VMEbus master, system physical addresses from the T-bus are translated into VMEbus addresses by the master mapper, using these registers. The eleven bits T_AD<23..13> select one of 2048 VMEbus Master Map RAM registers, whose bits then supply parts of the VMEbus address.

When the IACK bit is not asserted (that is, when it is "1"), the VMEbus address bits vme_a<31..13>, and the VMEbus address modifier bits vme_am<5..0>, are supplied from the selected register, and a VMEbus read or write takes place. When the IACK bit is asserted (cleared to "0"), the other two fields of the register are ignored and a VMEbus interrupt acknowledge (IACK) cycle is generated, as follows. (The boldface terms **interrupter** and **interrupt handler** are VMEbus functions defined in *The VMEbus Specification*.)

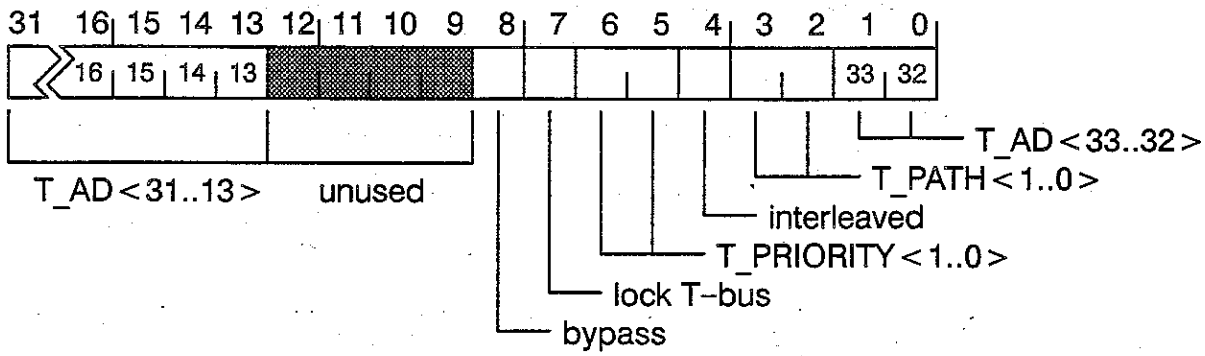
The *IACK* bit is used when the TC/FPV is responding as a VMEbus **interrupt handler** to an interrupt request generated by a VMEbus **interrupter** device. When an **interrupt handler** receives an interrupt, it must generate an interrupt acknowledge (IACK) cycle to get the interrupt vector from the **interrupter**. To generate the IACK cycle, a T-bus master on the TC/FPV (normally the CPU) must perform a halfword (16-bit) read from a specially-mapped address. The address is mapped by the VMEbus Master Map RAM registers, and the particular map register used has its IACK bit asserted (cleared to "0"). Bits 3..1 of the System Physical Address on the T-bus are the interrupt level (1-7) the software obtains from bits 3..1 of the Interrupt Source register. The VMEbus master interface on the TC/FPV translates the read operation into a VMEbus IACK cycle, encoding the interrupt level in VMEbus address bits A03, A02 and A01 (driven from T-bus address bits 3..1). The IACK cycle causes the **interrupter** to respond with an interrupt vector, which is returned to the **interrupt handler** as the result of its halfword read.

Typical software may allocate one 8–kilobyte segment (thus one VMEbus Master Map RAM register) just for generating interrupt acknowledgements on the VMEbus.

The interleaver loader mechanism is used to read and write these registers. See the Interleaver Loader registers entry for discussion and illustration of reading and writing the VMEbus Master Map RAM registers.

After power–on, the contents of the VMEbus Master Map RAM registers are unspecified.

Figure 11-53 VMEbus Slave Map RAM registers.



FUNCTION Map VMEbus addresses into TC2000 addresses.

ADDRESS 0xE0720000 to 0xE0721FFC (inclusive)
(Unlike the VMEbus Master Map RAM, which is accessed via the interleaver loader, the VMEbus Slave Map RAM is on the T-bus and is accessed just as main memory is.)

ACCESS read/write

DESCRIPTION When the TC/FPV is a VMEbus slave, responding to some other VMEbus device, addresses from the VMEbus are translated into TC2000 System Physical Addresses by the slave mapper, using these registers. When a VMEbus master references an address that falls within the window set by the VMEbus Configuration register, the VMEbus slave interface responds by passing the request through to the T-bus. The eleven bits vme_a<23..13> select one of 2048 VMEbus Slave Map RAM registers, whose bits then supply parts of the address and other control information placed on the T-bus as indicated.

See section 11.9.1 for a discussion of T_PRIORITY, and section 11.9.2 for a discussion of T_PATH.

The three bits named *bypass*, *lock T-bus* and *interleaved* describe the type of reference to make. Bypass tells the VMEbus slave (T-bus master) to make an operation that will bypass the TC2000 locking protocol; lock T-bus tells it to make a locked operation using that protocol. The interleaved bit tells the SIGA whether to take the high nine address bits from the T-bus (T_AD<33..25>) or from the interleaver (MOD<8..0>).

When the VMEbus slave is configured to respond to VMEbus *standard* addressing, only 512 of the 2048 VMEbus Slave Map RAM registers are used. Which quarter of the 2048 is used depends on the window specified in the VMEbus Configuration register, as follows. The size of block mapped is always 8 kilobytes. With *extended* addressing, the window is 16 megabytes, divided into 2048 blocks, each controlled by one map register. With *standard* addressing, the window is 4 megabytes, divided into 512 blocks, each controlled by one map register. The VMEbus Configuration register specifies the 4–megabyte window by specifying VMEbus address bits 23..22. These two bits, however, are also among the bits (23..13) used to select a VMEbus Slave Map RAM register. Therefore, the location of the 4–megabyte window in VMEbus address space, and the determination of which 512 map registers are used, are linked.

In summary, *when using VMEbus standard addressing*:

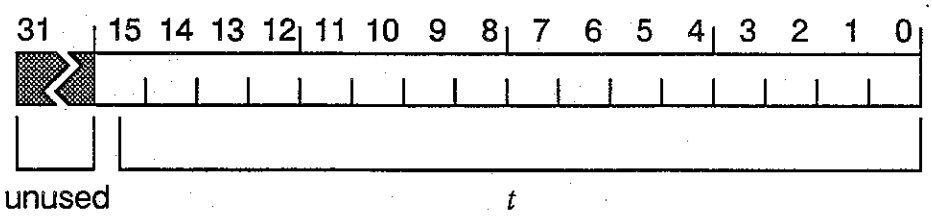
IF using VMEbus standard addressing and VMEbus Configuration register bits 2..1 are...	THEN the System Physical Address presented to the T–bus is generated by VMEbus Slave Map RAM registers...
0 0	0 – 511
0 1	512 – 1023
1 0	1024 – 1535
1 1	1536 – 2047

After power–on, the contents of the VMEbus Slave Map RAM registers are unspecified.

WARNING

T_PATH = 00 IS ILLEGAL
VMEbus Slave Map RAM registers must not be used with their bits T_PATH<1..0> cleared to "00". This value is illegal. It can provoke both SIGAs to initiate switch cycles.

Figure 11–54 VMEbus System Bus Timer register.



FUNCTION Detect lack of response from a slave to any VMEbus master.

ADDRESS 0xE0760208

ACCESS read (timer counter) / write (preset register)

DESCRIPTION When any VMEbus master asserts the *address strobe* signal, the VMEbus System Bus Timer begins counting, and stops when a slave device responds with the signal *DTACK* ("data acknowledge"). If no slave responds, expiration of this timer causes the TC/FPV hardware to assert the VMEbus signal *BERR* ("bus error"). This occurs entirely within the TC/FPV VMEbus interface, and is not explicitly visible to the TC/FPV CPU or to any other T-bus device. This situation may occur in a properly operating system. For example, on power-up, a program may poll the VMEbus to find available memory.

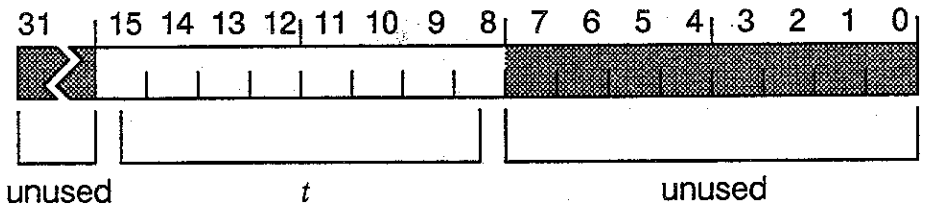
Note that the VMEbus System Bus Timer operates regardless of whether the TC/FPV is acting as VMEbus system controller. If the application has another VMEbus device to time out address strobe, the VMEbus System Bus Timer register may be set for a long timeout, effectively disabling the timer. Then the other VMEbus device will always time out first, and the TC/FPV's timer will never expire.

A VMEbus access is timed out after $4\times(0xFFFF-t)$ microseconds, where t is the contents of the VMEbus System Bus Timer register. The timer is initialized from the register each time address strobe is asserted by any VMEbus master.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter.

After power-on, the contents of the VMEbus System Bus Timer register are unspecified.

Figure 11–55 VMEbus TC/FPV Master Bus Timer register.



FUNCTION Detect lack of response from a slave to the TC/FPV as bus master.

ADDRESS 0xE0760200

ACCESS read (timer counter) / write (preset register)

DESCRIPTION When the TC/FPV is VMEbus master, it asserts the VMEbus signal *address strobe* and waits for a slave to respond with *DTACK* ("data acknowledge"). The VMEbus TC/FPV Master Bus Timer starts counting when the TC/FPV asserts address strobe, and stops when DTACK is received. If no response is received, expiration of this timer causes the TC/FPV to assert the VMEbus signal *BERR* ("bus error"). This occurs entirely within the TC/FPV VMEbus interface. Thus, operation when this timer expires is like that of the VMEbus System Bus Timer but for a shorter time interval.

However, in this case the TC/FPV itself is the VMEbus master, so it reacts to the BERR signal by removing address strobe and returning a bus error to the T–bus master that made the reference. Without this timeout, access to a non–responding VMEbus device would be timed out only by the VMEbus system controller. The system controller's bus timer is typically set much longer than the VMEbus TC/FPV Master Bus Timer, so relying on the system timer would compromise responsiveness. The longer timeout would be especially serious for VMEbus access across the Butterfly switch, where maximum switch latency would be affected.

Any VMEbus reference initiated by the TC/FPV is timed out after 0xFF–*t* microseconds, where *t* is the contents of the VMEbus TC/FPV Master Bus Timer register. The timer is initialized from the register each time the TC/FPV asserts address strobe.

Writing to this address modifies the "preset" value loaded into the timer counter whenever the timer begins running; reading this address returns the current (instantaneous) contents of the timer counter. Software cannot directly read the preset register, nor directly write the timer counter.

After power–on, the contents of the VMEbus TC/FPV Master Bus Timer register are unspecified.

11.18 TC/FPV Controls and Jumpers

The only manually alterable controls on the TC/FPV are two voltage trim pots, and jumpers E1 through E29. The function of the trim pots and jumpers is described below. LED indicators on the TC/FPV are described in section 11.2.

11.18.1 Connectors

The TC/FPV contains three connectors, whose positions are shown in Figure 11-2. Two connectors are the standard P1 and P2 connectors defined in *The VMEbus Specification* (Motorola). The third connector is the TC/FPV-to-midplane connector described in conjunction with the midplane (TC/MP) specification.

11.18.2 Voltage Trim Pots

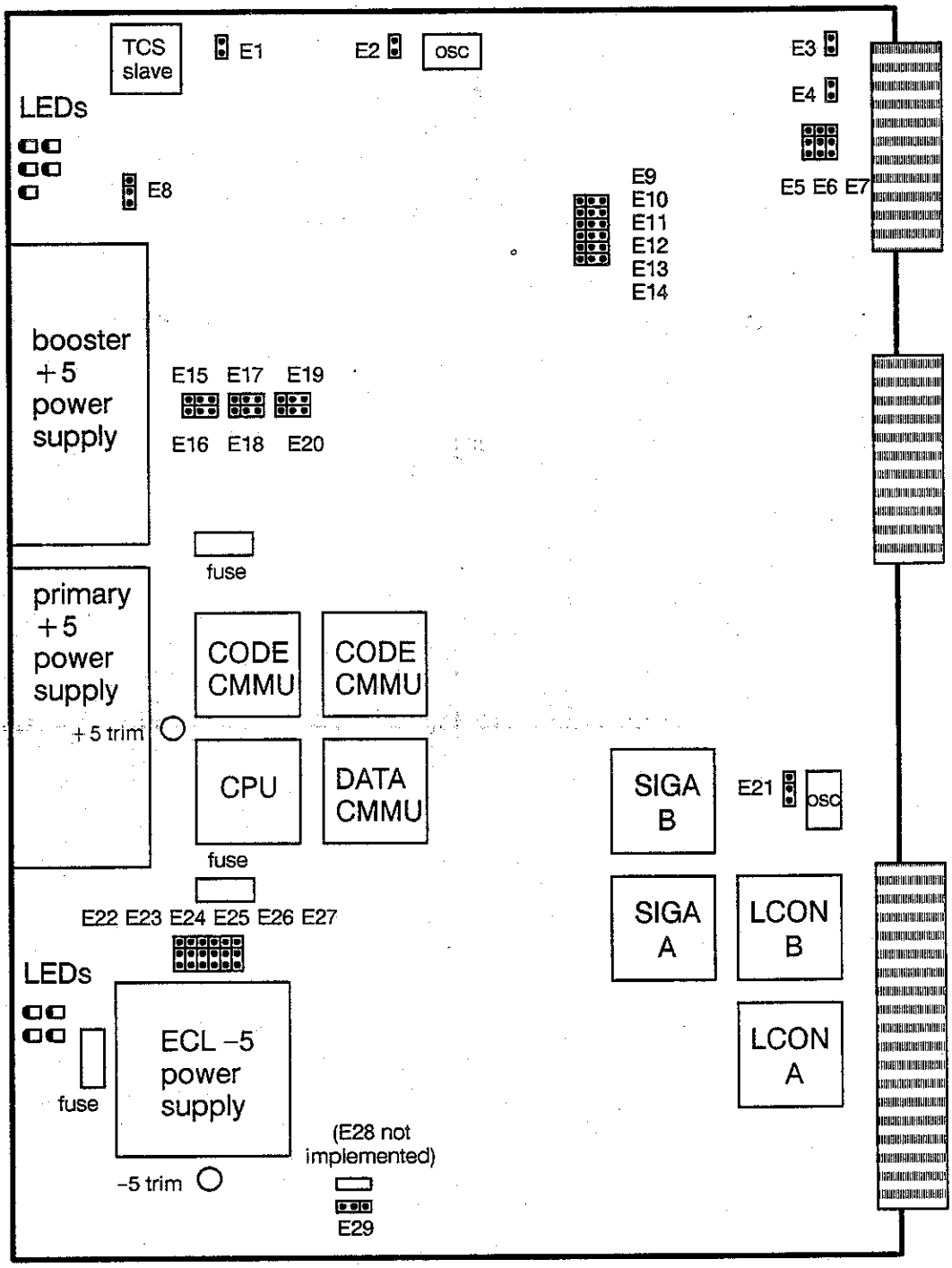
+5 trim — located between the MC88100 CPU chip and the primary +5 power supply, this pot adjusts the +5 volt supply for the board. The pot adjusts the primary +5 supply directly, and the booster +5 supply tracks the primary +5.

−5 trim — located near the -5 power supply, this pot adjusts the -5 volt supply for the board. This (actually -5.2 volts) is used in the ECL circuits.

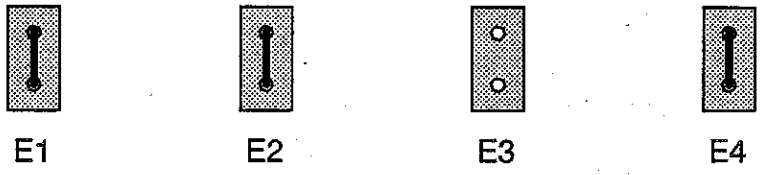
11.18.3 Jumpers

The TC/FPV contains 2-pin jumper positions and 3-pin jumper positions, as shown in Figure 11-56. On 3-pin positions, pin 1 is marked with a square in Figure 11-56 and in the individual jumper diagrams below. (It is also so marked in the silk screen, but the pin base covers this up on an assembled board.) The jumper positions are shown in the same orientation in Figure 11-56 as in the individual diagrams. With the board oriented as in Figure 11-56, the numbering of jumper pin positions is the same as how you read a page — left to right, top to bottom.

Figure 11-56 TC/FPV jumper locations.



viewed from component side



E1 Disables the 8 MHz crystal used to generate the clock signal for the 68HC11 TCS slave. Always installed except for testing.

E2 Disables the 32 MHz oscillator used to generate the VMEbus system clock and to time parts of the VMEbus interface. Always installed except for testing.

E3 Connects the VMEbus interface arbiter enable signal to an unused pin on the VMEbus connector. Used for testing during manufacture. Normally removed.

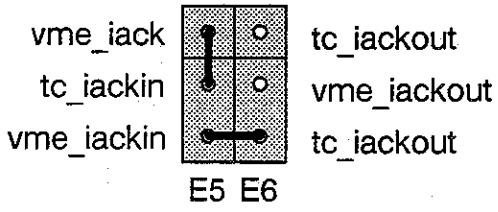
E4 Connects the 16 MHz VMEbus system clock to the VMEbus connector. Jumper is installed if the TC/FPV is VMEbus system controller, removed if not system controller.

E5, E6 VMEbus IACK daisy chain jumpers

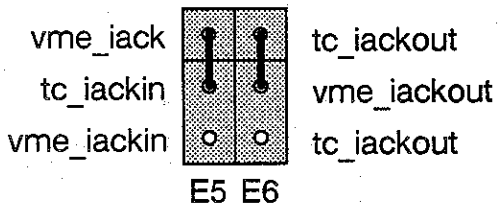
These jumpers are designed to accommodate several VMEbus configurations. The signals connected to the E5 and E6 jumper pins are shown in each configuration diagram below. *vme_iack* is the VMEbus interrupt acknowledge signal, used in conjunction with the interrupt acknowledge daisy chain (*vme_iackin* and *vme_iackout*) to acknowledge interrupts. For details, refer to *The VMEbus Specification*.

The setting of these jumpers depends on whether the TC/FPV is serving as VMEbus system controller, whether the TC/FPV is in the last slot of the VMEbus, and whether a VMEbus board is plugged into the other side of the VMEbus midplane (in the VMEbus slot opposite the TC/FPV). The five legal combinations are described below; other combinations are illegal.

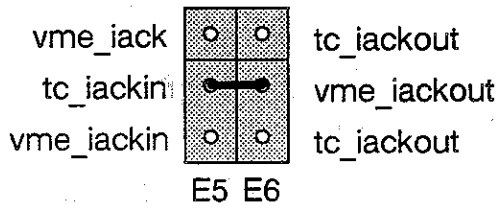
If the TC/FPV is serving as the VMEbus system controller (that is, installed in slot 1), and there ***is*** a VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows. This is the default jumper setting.



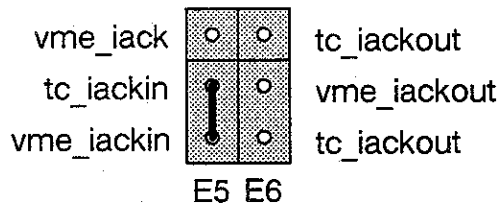
If the TC/FPV is serving as the VMEbus system controller and there is ***no*** VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.



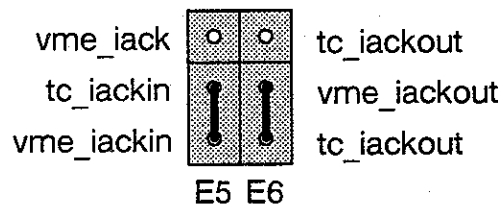
If the TC/FPV is ***not*** serving as the VMEbus system controller, and it is the ***last*** slot in the VMEbus, and there ***is*** a board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.

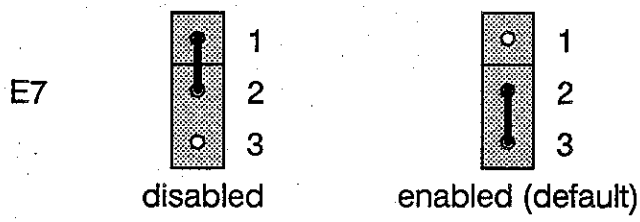


If the TC/FPV is ***not*** serving as the VMEbus system controller, and it is the ***last*** slot in the VMEbus, and there is ***no*** VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.

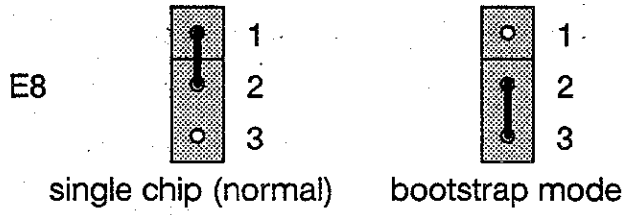


If the TC/FPV is ***not*** serving as the VMEbus system controller, and it is ***neither*** the beginning ***nor*** the end of the VMEbus, then there ***must not*** be a board plugged into the VMEbus slot opposite the TC/FPV, and the jumpers should be connected as follows.

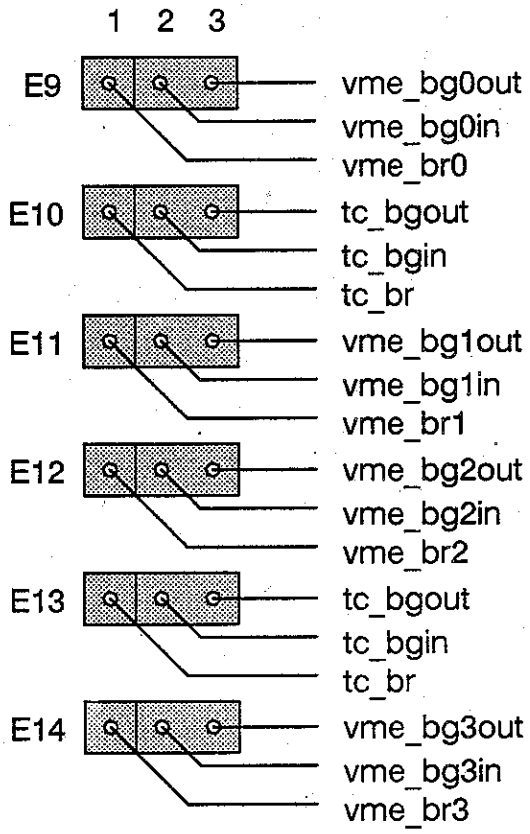




E7 VMEbus arbiter enable. Jumpering pins 1 and 2 disables the board VMEbus arbiter. Jumpering pins 2 and 3 enables the board VMEbus arbiter, required when the board acts as VMEbus system controller. The default is enabled.



E8 TCS slave mode select. Jumpering pins 1 and 2 puts the 68HC11 in normal single chip mode. Jumpering pins 2 and 3 puts the 68HC11 in special bootstrap mode. For details, refer to Motorola documentation (such as *MC68HC11E9 HCMOS Single-Chip Microcontroller Advance Information*). Normally in single chip mode.

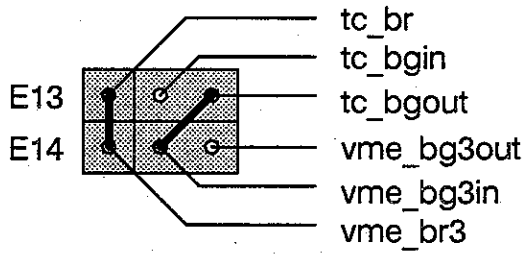


E9 — E14 VMEbus bus grant jumpers

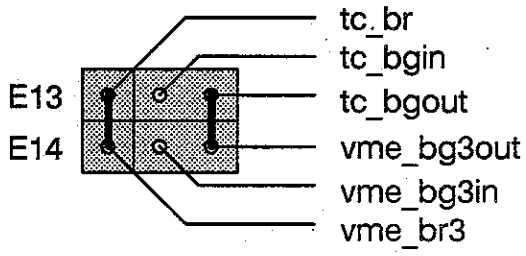
These jumpers are designed to support a variety of configurations. The signals connected to E9 through E14 are shown above. In addition to the various midplane positions it can occupy, the TC/FPV can participate in VMEbus bus arbitration on any one of the four priority levels. (*The VMEbus Specification* defines four levels. The TC/FPV performs single level arbitration only.) The following examples give the jumper settings for participating in priority level 3 arbitration, which is the default. Those pins not associated with the level being used are left with no jumpers. Therefore, only pin positions E13 and E14 are involved in these examples. The jumper settings for use of other levels is analogous to that shown.

The setting of these jumpers depends on whether the TC/FPV is serving as VMEbus system controller, whether the TC/FPV is in the last slot of the VMEbus, and whether a VMEbus board is plugged into the other side of the VMEbus midplane (in the VMEbus slot opposite the TC/FPV). The five legal combinations are described below; other combinations are illegal.

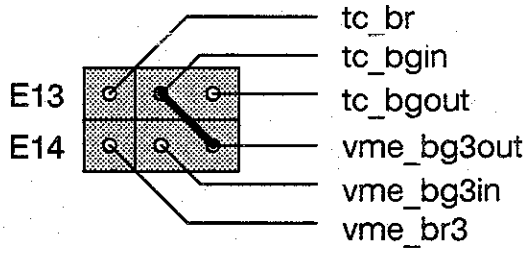
If the TC/FPV is serving as the VMEbus system controller (that is, installed in slot 1), and there ***is*** a VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows. This is the default jumper setting.



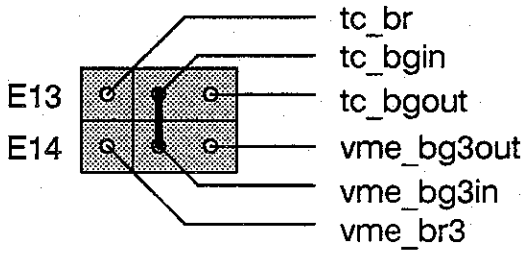
If the TC/FPV is serving as the VMEbus system controller and there is ***no*** VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.



If the TC/FPV is ***not*** serving as VMEbus system controller, and it is the ***last*** slot in the VMEbus, and there ***is*** a VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.

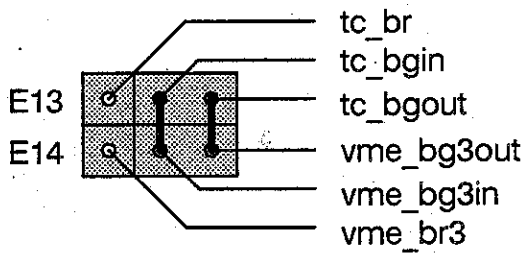


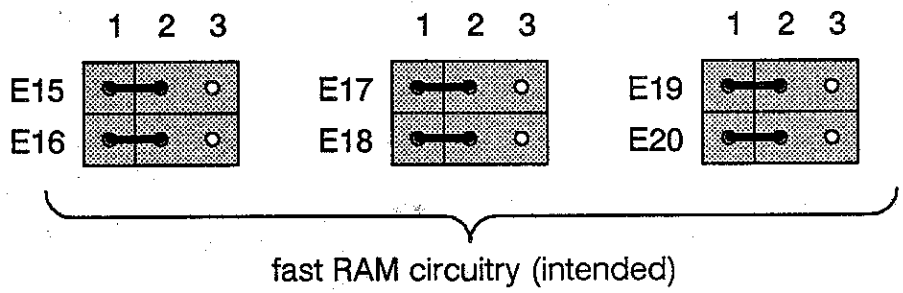
If the TC/FPV is ***not*** serving as the VMEbus system controller, and it is the ***last*** slot in the VMEbus, and there is ***no*** VMEbus board plugged into the VMEbus slot opposite the TC/FPV, then the jumpers should be connected as follows.

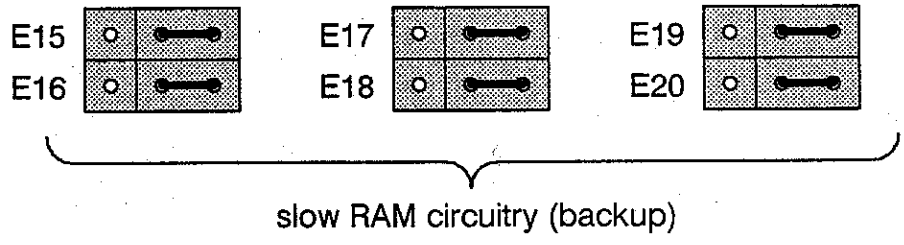


If the TC/FPV is ***not*** serving as the VMEbus system controller, and it is ***neither*** the beginning ***nor*** the end of the VMEbus, then there ***must not***

be a VMEbus board plugged into the VMEbus slot opposite the TC/FPV, and the jumpers should be connected as follows.

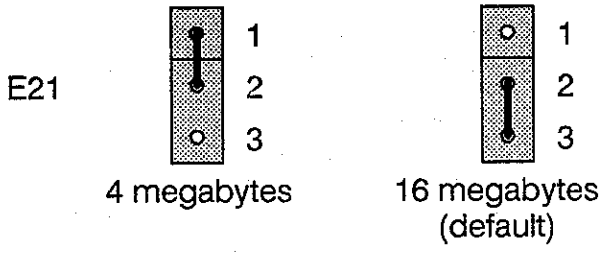




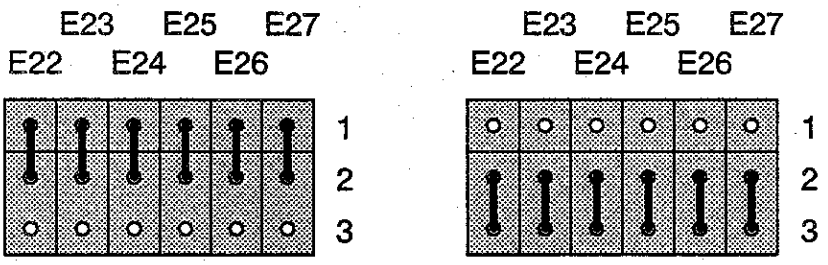


E15 — E20 fast/slow RAM circuitry select

A very new static RAM is used in the TC/FPV to implement the CPU fast path circuitry. To alleviate the chance of performance or availability problems with these parts, an alternate circuit is implemented using slower RAMs. The alternate circuit does not permit use of the fast path, but does allow the board to function. Jumpers E15 through E20 select between the slow and fast RAM circuits. Jumpering pins 1 and 2 selects the fast RAM circuitry. Jumpering pins 2 and 3 selects the slow RAM circuitry.



E21 Memory size select. Jumpering pins 1 and 2 configures the TC/FPV for 4 megabytes of memory. Jumpering pins 2 and 3 configures the board for 16 megabytes. The default is 16 megabytes.



RIFA or ERIPOWER circuitry Power General circuitry

E22 — E27 Two types of module that convert 48 volts to -5 volts are qualified for the TC/FPV, but they require slightly different trim circuits. Both trim circuits are included on the board, and jumpers E22 through E27 select between them. Jumpering pins 1 and 2 on all six sets of pins configures the board for RIFA or ERIPOWER modules. Jumpering pins 2 and 3 on all six sets of pins configures the board for Power General modules.

E28 No jumper pins are implemented with this designation.

16 megahertz 20 megahertz (intended)

E29 Clock speed indicator. Indicates the speed at which the board is running to the 1 MHz clock generator. Jumpering pins 1 and 2 configures the board for 20 megahertz operation. Jumpering pins 2 and 3 configures the board for 16 megahertz operation. The intended operation is 20 megahertz.

11.19 References

This section lists the documents referenced in this document, or otherwise directly relevant to the material herein.

VMEbus spec *The VMEbus Specification* (cover title), aka *VMEbus Specification Manual* (title page title), revision C.1, October 1985, Motorola Series in Solid-State Electronics, Motorola, Inc., 261 pages, document number HB212.

CPU manual	*MC88100 RISC Microprocessor User's Manual*, 1988, Motorola, Inc., 290 pages (numbered by section), document number MC88100UM/AD. Includes *Errata to MC88100 User's Manual First Edition MC88100UM/AD*, November 11, 1988, Motorola, Inc., 2 pages, document number MC88100UMAD/AD.
CMMU manual	*MC88200 Cache/Memory Management Unit User's Manual*, 1988, Motorola, Inc., 254 pages (numbered by section), document number MC88200UM/AD. Includes *Errata to MC8200 User's Manual First Edition MC88200UM/AD*, December 8, 1988, Motorola, Inc., 2 pages, document number MC88200UMAD/AD. Also defines the **P–bus** and **M–bus** protocols.
TCS slave	hardware manual: *Advance Information, MC68HC11E9 HCMOS Single–Chip Microcontroller*, 1988, Motorola Semiconductor Technical Data, Motorola, Inc., 140 pages (numbered by section), document number MC68HC11E9/D. programming manual: *M68HC11 HCMOS Single–Chip Microcontroller Programmer's Reference Manual*, First Edition, no date, Motorola, Inc., 180 pages (numbered by section), document number M68HC11PM/AD (not MC68...).
SIGA spec	*Switch Interface Gate Array (SIGA) Functional Specification*, April 20, 1989, BBN ACI, 111 pages, document number FS14160–01.
LCON spec	*Butterfly II Level Converter Array*, October 5, 1988, BBN ACI, 39 pages, document number FS14161–00.
SGA spec	*BF2 Switch Gate Array: Functional Description*, March 22, 1989, BBN ACI, document number FS14159–00.
T–bus spec	*T–bus Specification*, September 30, 1988, BBN ACI, 48 pages. Internal ACI document.
TC/FPV spec	*TC/FPV Functional Specification*, January 17, 1990, BBN ACI, 144 pages, document number FS17522G01. Content is the same as this document, just reformatted.
TC/MP spec	*TC/MP Functional Specification*, January 4, 1990, BBN ACI, 29 pages, document number FS18512G01. The TC/MP is the TC2000 midplane.
TC/SR,SS spec	*TC/SR and TC/SS Functional Specification*, July 19, 1989, 23 pages, document number FS17810G00. The TC/SR and TC/SS are the switch requester card and the switch server card, respectively, for the TC2000.
SCB68154 data sheet	In the *Signetics Microprocessor Data Manual*, 1987, Signetics Microprocessor Products, Signetics Corporation.
TCS operation	In the *TC2000 System Administration Guide*, software revision 1.0, document revision A, August 1989, BBN ACI, 565 pages, document number A310007G10. An on–line, more detailed description of the current version of TEX, the TCS master executive program, is normally kept in the following file: /usr/bfly2/src/tcs–m/tex/*version–name*/tex.doc

A

Switch Interface Gate Array (SIGA) Specification

A.1 Introduction

The SIGA is a gate array device which serves as the bidirectional interface between a Computational Node and the Switch network of the Butterfly II Parallel Processor. As such, the SIGA provides devices on each Computational Node with virtually transparent read and write data access to similar devices on physically remote nodes. The SIGA accomplishes this by accepting/presenting data via the standard interface that these devices support — namely the T–Bus — and then presenting/accepting this same data to the Butterfly Switch interface for transport.

This document will present both a detailed functional and operational description of the SIGA. It is intended to be used as a design guide for both hardware and software system engineers. This specification is necessarily limited in its scope and thus will touch upon other Butterfly II–related subjects only when it is necessary for completeness. Therefore, it is assumed that the reader of this document has a general knowledge of the concepts of the Butterfly II architecture and operation. The reference documents are as follows:

Figure A–1 **Reference documents.**

- T–Bus Specification (Ward Harriman)
- Switch Gate Array Design Specification (Ward Harriman)
- Butterfly II Level Converter Array Specification (Mike Sollins)
- Switch Protocol Specification (Ward Harriman)

A.2 Terminology

The following terms will be used throughout this document:

Byte Refers to an 8-bit quantity.

Anticipation A feature of the SIGA design that allows the SIGA to take advantage of certain parallel optimizations.

Downstream Node The node which services a switch transaction.

Drop-Lock When the Requestor negates Frame during a locked sequence, causing the Server to issue a FREE-LOCK.

Function Response A generic term for the various incarnations of a response to a function request from some downstream T-Bus slave to an upstream T-Bus slave. This includes the transformations that the response undergoes as it travels from the downstream T-Bus, downstream SIGA, Switch, upstream SIGA, and finally the upstream T-Bus (see Function Request).

Function Request A generic term for the various incarnations of a request from some upstream T-Bus master to a downstream T-Bus slave. This includes the transformations that the request undergoes as it travels through the upstream T-Bus, upstream SIGA, Switch, downstream SIGA, and finally the downstream T-Bus (see Function Response).

Final Locked message The same as a Locked message except that the Switch path is released by letting Frame = 0 for at least two Switch Intervals after the operation has been acknowledged.

Halfword Refers to a 16-bit quantity (see Word).

Initial Locked message Occurs under the same circumstances as the Unlocked message except that the Switch path is held open once the operation has been acknowledged without errors.

Local Errors Errors which originate in the Requestor.

Logical Route Address A 9-bit Switch node address generated from either the Interleaver or the T-Bus. This address is then transformed, possibly by randomizing some of the bits, into the Physical Route Address.

Locked message A message which occurs when the Switch path was already locked and causes it continue to be locked after the operation has been acknowledged.

Message With the exception of Reject, a Message is the assertion of Frame (downstream message) or Reverse (upstream message) possibly with associated data on the data lines.

Message Acknowledgment Also known as M_ACK. This refers to the assertion of Reverse for at least two Switch Intervals during a function response. It indicates that the downstream Server has Acknowledged the receipt of a Function Request.

Message Header The part of a downstream Switch message that carries routing information. That part is stripped off by the Switch and thus never reaches the downstream Server. The message header for an upstream Switch message is null.

Message Body The part of the downstream Switch message that carries the command, address, data, and checksum bytes.

Multi-Word Transfer Refers to a read or write function request that involves more than one word (32 bit) of data.

Operational State A SIGA initialization state which allows full operation of the SIGA.

Physical Route Address The transformation of the Logical Route Address after some of its bits have been randomized. The Physical Route Address is placed into the downstream Message Header.

Quick-Drop This is an optimization in the Requestor where the R_FRAME signal is negated as soon as possible after an R_REVERSE is received.

Quiescent State A SIGA initialization state which allows partial operation of the SIGA.

Remote Errors Errors which originate in the Server.

Reject An assertion of Reverse for one Switch Interval. Indicates that a message was rejected at either a Server or an SGA.

Sequence A function request followed by a function response.

Split-Cycle A T-Bus Read transaction where the Master releases the bus while the Slave is completing the transaction.

Switch Interval The 25 ns period in which Switch data is propagated.

Switch Modulus The number of ports that a basic switching element can handle. That number is currently eight.

Transaction Another word for a Sequence.

Unlocked Message Occurs when the Switch path had previously been "torn-down". This occurs whenever Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn-down again.

Upstream Node The node which initiates a switch transaction.

Valid Message A downstream message which carries a read or write request and does not violate switch protocol.

Word Refers to a 32–bit quantity (see Halfword).

A.3 Document Standards

The following describes some of the standard syntax and expressions used in this document.

A.3.1 Register Definition Syntax

A typical register definition is shown in Figure A–2. Referring to Figure A–2, the "–" in any bit indicates that this bit is a "don't care" on a write and indeterminate on a read. If "–" totally fills a field of eight bits, that field should NEVER be written to but of course, can be read from. The entire register may be referred to in any one of the following ways:

(1) Protocol_Timer_Config<15..0>
(2) Protocol_Timer_Config
(3) PTC<15..0>
(4) PTC

The subfields, shown in Figure A–2 within "[]", can be referred to in various ways. For instance, the "Cnt" subfield could be referred to as:

(1) Protocol_Timer_Config<15..12>
(2) Protocol_Timer_Config.Cnt<3..0>
(3) Protocol_Timer_Config.Cnt
(4) PTC.Cnt

Figure A–2 Register syntax definition.

```
Register: Protocol_Timer_Config<15..0>

15                      0
|                       |
3..0    3..0    7......0
CCCC    PPPP    --NNNNNN
[Cnt]   [Pre]   [Con]
```

A.3.2 Logical Operators

Figure A–3 shows the standard operators used in this document.

Figure A–3 Example — logical operators.

OPERATOR	FUNCTION
&	logical "and"
#	logical "or"
$	logical "exor"
!	logical "not"
!$	logical "exnor"
|	concatenate

A.3.3 Timing Diagram Symbols

Timing diagrams use ASCII characters to represent signal states. Figure A–4 illustrates some of those symbols and their associated meanings. In addition, if no clock signal is present in a timing diagram, it is assumed that each character column represents an active transition of the appropriate clock.

Figure A–4 Example — signal symbols.

SYMBOL	MEANING
H	logical "1"
_	logical "0"
.....	continue previous state
?????	state unknown and unimportant

A.4 Functional Overview

The following describes the basic functionality of the SIGA at a conceptual level.

A.4.1 Functional Unit Description

The SIGA is composed of four basic elements, the Requestor, Server, Control Net Interface and the Config/Status Unit. Although these are physically colocated and share some common logic and control, they are functionally independent units and will be described separately.

A.4.1.1 Requestor

The Requestor is a T–Bus slave device which transparently couples physically remote T–Bus slave devices to the local T–Bus by interacting with both the Switch and the downstream Server. The Requestor appears to the current T–Bus master as a segment of memory which is out of the range of physical memory at the local node. Signals on the T–Bus alert the Requestor that the current access is for a remote location and the Requestor then initiates the switch transaction to comply with the master's read or write request.

Since the transaction is not completed immediately (indicated by the Requestor with a PROMISE response), the requesting T–Bus master follows the T–Bus protocol and releases the bus so that other devices may use it. The Requestor eventually regains control of the T–Bus, alerts the requesting master that the read or write operation has been completed, and returns data or an error indication. If the current sequence is locked, as requested by the T–Bus master, and no errors are encountered, the Requestor holds open the Switch path for the next transaction rather than rearbitrating for a new Switch path. Any errors that may have occurred during this operation are signaled to the T–Bus Master through the ERROR response.

A.4.1.2 Server

The Server acts as a master on the local T–Bus of the downstream node and services requests from the upstream node's Requestor. When a new valid message enters the Server from the Switch, the Server obtains the local T–Bus; locks the T–Bus slave, if desired; performs the read or write operation; and then returns the data and/or error byte to the Upstream Node's Requestor. The Server can also initiate other special operations independently of receiving a new Switch message. This operation, known as drop–locks, is described elsewhere in this document.

A.4.1.3 TCS Control Unit

The basic purpose of the TCS Control Unit (TCU) is to give the serial interface of the TCS Control Slave Processor access to the T–Bus interface — in essence, to act as a protocol converter. A secondary function is to allow the TCS Slave Processor DIRECT access to some of the internal functions of the SIGA, rather than forcing it to access via the T–Bus interface. This is useful for fault tolerance and "out–of–band" functions such as bootstrapping.

A.4.1.4 Configuration/Status Unit

The Config/Status Unit (CSU), acting as a T–Bus slave, allows read/write access to all programmable parameters of the Requestor, Server and TCS Con-

trol Unit. The CSU also provides convenient access to the internal state of certain nodes for testability.

A.4.2 System Operation

From a high–level view, the SIGA is one link in the chain of devices that allows a T–Bus device to fulfill a function request with a function response. The role of the SIGA in fulfilling both function requests and responses is now described.

A.4.2.1 Function Requests

A local T–Bus master in the upstream node, usually the CPU, initiates the sequence by placing an address on the T–Bus, which is detected by the SIGA Requestor as a remote access request. During the T–Bus request phase, the SIGA stores the address, produces and stores the bid, and command bytes. It then initiates the downstream message at the Switch interface by asserting Frame and placing the bid symbols on the Switch data lines. Normally, this message transmission is initiated by the SIGA immediately upon receiving the address from the T–Bus, but it can be programmed to start later. On a write, the SIGA loads its data registers during the response phase of the T–Bus cycle. All operations are split–cycle and thus the Server will release the bus while it processes the transaction request.

If there is no Switch contention, the assembled message continues to be transmitted and is ultimately appended with a checksum derived from the message data bytes. If there is Switch contention, a Reject is generated by the Switch and eventually makes its way upstream to the Requestor via the Reverse line. When this happens, the Requestor negates Frame, waits for a predetermined amount of time and then retries the message by asserting Frame and sending the message components stored from the first attempt.

Sometime after the beginning of the message reaches the Server at the downstream node (i.e., it is not Rejected by the Switch), that Server begins arbitration for its local bus to complete the transaction. If the device on the downstream node is locked to a remote bus master other than the Server, the Server issues a Reject which propagates upstream and is eventually detected at the upstream Requestor. This Reject is treated exactly the same by the Requestor as a Reject from the Switch. Note that this is the ONLY instance in which the Server will issue a Switch Reject — an Initial Message.

Assuming that the Requestor receives neither a Switch Reject nor a Server Reject, it deasserts Frame for one switch interval while it sends the checksum byte. Within the checksum byte, the "forward" bit is reset. This event would normally cause the forward drivers of the SGA's to turn off after they send the checksum byte. However, the current implementation of the SGA ignores this bit and turns on its forward drivers in response to the Frame profile. The Re-

questor then awaits a response from the Server. Note that the forward bit is not used by the current SGA's.

In the meanwhile, the downstream Server begins processing the request by arbitrating for the local T–Bus. Assuming that the target downstream bus slave was not locked to a downstream master other than the Server, the Server obtains the local bus and possibly opens the local memory lock. The Server will open the lock only if this action was requested in the downstream message. This would occur if the master on the upstream node's local bus requested an OPEN lock when it initiated a transaction through it's associated Requestor.

Once the downstream Server obtains the local bus, it makes a function request to perform the appropriate read or write operation. The only exception to this is when the Server detects a checksum error in the downstream message. If this occurs, instead of making a request, the Server releases control of the T–Bus, creating a "dead" bus cycle and thereby aborting the transaction. This action on an aborted transaction should eliminate any unwanted side effects if the switch message is corrupted.

A.4.2.2 Function Responses

Assuming that a read transaction was requested, the downstream Server completes the read as a normal local T–Bus master. As soon as the read data is obtained by the Server, a message is returned to the upstream Requestor. This happens (over the same data wires which the downstream message was sent) by asserting Reverse and applying data to the Switch data lines. The upstream message contains the read data, and possible error data; a checksum; and a message acknowledgment, or M_ACK which is implicit in the assertion of Reverse for at least two Switch intervals. If a write transaction was requested, the Server writes the data to the address specified in the downstream message and sends back an M_ACK with an error byte data and checksum after the data has been accepted by the slave on the local T–Bus. In short, a read returns data/errors and an acknowledgment whereas a write only returns possible errors and an acknowledgment.

In the case of a read transaction, the upstream Requestor detects the M_ACK and alerts the local split–cycle master which initiated the request that the requested data has been returned. That master then completes the operation by retrieving the data. In the case of a write transaction, the Requestor also alerts the initiating local bus master that the write was completed but returns only error information.

In the absence of errors, the Requestor will continue to hold the Switch path open by asserting Frame only if the sequence was initiated with an OPEN. If that master decides to release the lock, the Requestor will tear–down the switch connection by negating Frame and will enter its unlocked idle state. This is the state that it was in at the beginning of this discussion of function requests. If the upstream bus master does not release the lock, it may initiate

another read or write transaction. This and subsequent transactions are referred to as locked transactions. Outside of errors, locked transactions end only when the upstream T-Bus master which OPENed, MAINTAINed or BYPASSed the SIGA Requestor lock decides to release that lock with a FREE-LOCKS command.

Subsequent message transactions in a locked sequence differ from the initial transactions described above in three major ways. First, locked messages do not contain any bids because the path has already been established. Second, the Switch will never issue a Reject because the path has already been established and is being reserved for the Requestor. And third, the downstream Server will never issue a Reject because it will already have exclusive use of the local memory lock. Aside from these exceptions, subsequent locked transactions occur in exactly the same manner as unlocked transactions. As mentioned previously, the upstream T-Bus master owning the SIGA Requestor MUST release that lock explicitly with a FREE-LOCKS.

A.4.3 Basic Message Formats

Message formats differ mainly with the type of function request; read or write. Within a read or write message, the downstream and upstream messages corresponding to a function request and response also differ.

A.4.3.1 Read Messages

Read message formats differ mainly depending on whether or not they are downstream or upstream messages.

A.4.3.1.1 Downstream

Downstream Read messages are differentiated partly because of their data format and partly because of the state of Frame at the beginning and end of the message. The formats for three possible SIGA Requestor read operations are considered:

1. An Unlocked Read occurs when the Switch path had previously been "torn-down". This occurs whenever Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn-down again.
2. An Initial Locked Read occurs under the same circumstances as the Unlocked Read except that the Switch path is held open once the operation has been acknowledged.
3. A Locked Read is a read which occurs when the Switch path was already locked and it continues to be locked after the operation has been acknowledged.

In all cases, the Requestor waits for a Message Acknowledgment (M_ACK) from the downstream Server before completing the message. Figure A–5 illustrates the three read message types for a two column switch. In this figure, the "d" field indicates the direction of the LCON drivers which interface data with the LCON. When d = "P" (Output), the Requestor is sourcing data to the Requestor/LCON interface. When d = "I" (Input), the LCON drivers are sourcing data to the Requestor/LCON interface. The "f" field is the state of the Frame bit. Data is MSB at the left of the field.

Figure A–5 Read switch message format — downstream.

```
Unlocked              Initial               Locked
Read                  Locked Read           Read
============          ============          ============
d f   data            d f   data            d f   data
- - --------          - - --------          - - --------
P 0 xxxxxxxx          P 0 xxxxxxxx          P 1 xxxxxxxx
P 0 xxxxxxxx          P 0 xxxxxxxx          P 0 xxxxxxxx
P 1 -bid1---          P 1 -bid1---          P 1 -cmd----
P 1 -bid2---          P 1 -bid2---          P 1 -addr1--
P 1 -cmd----          P 1 -cmd----          P 1 -addr2--
P 1 -addr1--          P 1 -addr1--          P 1 -addr3--
P 1 -addr2--          P 1 -addr2--          P 0 -check--
P 1 -addr3--          P 1 -addr3--          I 1 00000000
P 0 -check--          P 0 -check--          I 1 00000000
I 1 xxxxxxxx          I 1 xxxxxxxx                "
I 1 xxxxxxxx          I 1 xxxxxxxx             M_ACK
      "                     "             and read data
    M_ACK                 M_ACK                 "
and read data         and read data         I 1 xxxxxxxx
      "                     "             P 1 xxxxxxxx
I 0 xxxxxxxx          I 1 xxxxxxxx
P 0 xxxxxxxx          P 1 xxxxxxxx
```

A.4.3.1.2 Upstream

When a downstream read message has been received and processed by a Server, an upstream message is returned to the initiating Requestor based on the operation requested. Under normal conditions, the Upstream Message is composed of two parts: the returned data (with checksum) and the M_ACK (Message Acknowledge). The returned data is the contents of the remote memory location read, which can be 1, 2 or 4 words in length. With the exception of rare error conditions, the actual message data field is almost always a multiple of four.

Figure A–6 illustrates the upstream message. The "r" field is the Reverse signal. Data is MSB at left of the field.

Figure A–6 Read switch message format — upstream.

```
1-word, 4-byte Read
==========
r   data
- --------
0 xxxxxxxx
1 -data a-
1 -data b-
1 -data c-
1 -data d-
1 -check--
0 xxxxxxxx
```

A.4.3.2 Write Messages

Write message formats differ mainly depending on whether or not they are downstream or upstream messages.

A.4.3.2.1 Downstream

Downstream Write messages are differentiated partly because of their data format and partly because of the state of Frame at the beginning and end of the message. The formats for three possible SIGA Requestor write operations are considered: In all cases, the Requestor waits for a Message Acknowledgment (M_ACK) from the downstream Server before completing the message. Figure A–7 illustrates the three write message types for a two column switch. In the figure, The "d" field is the direction of the LCON drivers which interface data with the SGA. When d = I, the Requestor is sourcing data to the Requestor/LCON interface. When d = P, the LCON drivers are sourcing data to the Requestor/LCON interface. The "f" field is the state of the Frame bit. Data is MSB at left of the field.

1. An Unlocked Write occurs when the Switch path had previously been "torn–down" by the fact that Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn–down again.
2. An Initial Locked Write occurs under the same circumstances as the Unlocked Write except that the Switch path is held open once the operation has been acknowledged.
3. A Locked Write is a write which occurs when the Switch path was already locked and it continues to be locked after the operation has been acknowledged.

Figure A-7 **Write switch message format — downstream.**

```
Unlocked                Initial                 Locked
Write                   Locked.Write            Write
============            ============            ============
d f   data              d f   data              d f   data
- - --------            - - --------            - - --------
I 0 xxxxxxxx            I 0 xxxxxxxx            P 1 xxxxxxxx
I 0 xxxxxxxx            I 0 xxxxxxxx            P 0 xxxxxxxx
P 1 -bid1---            P 1 -bid1---            P 1 -cmd----
P 1 -bid2---            P 1 -bid2---            P 1 -addr1--
P 1 -cmd----            P 1 -cmd----            P 1 -addr2--
P 1 -addr1--            P 1 -addr1--            P 1 -addr3--
P 1 -addr2--            P 1 -addr2--            P 1 -data a-
P 1 -addr3--            P 1 -addr3--            P 1 -data b-
P 1 -data a--           P 1 -data a-            P 1 -data c-
P 1 -data b-            P 1 -data b-            P 1 -data d-
P 1 -data c-            P 1 -data c-            P 0 -check--
P 1 -data d-            P 1 -data d-            I 1 xxxxxxxx
P 0 -check--            P 0 -check--            I 1 xxxxxxxx
I 1 xxxxxxxx            I 1 xxxxxxxx                  "
I 1 xxxxxxxx                  "                     M_ACK
      "                     M_ACK                     "
    M_ACK                     "                 I 1 xxxxxxxx
      "                 I 1 xxxxxxxx            P 1 xxxxxxxx
I 0 xxxxxxxx            P 1 xxxxxxxx
P 0 xxxxxxxx
```

A.4.3.2.2 Upstream

When a downstream write message has been received and processed by a Server, an upstream message is returned to the initiating Requestor based on the operation requested. Under some conditions, the Server will not act on the downstream message and will instead send a Reject back to the Requestor. Under normal conditions however, upstream messages contain an M_ACK, an error byte (normally all 0's) and a checksum.

The following illustrates the only possible return message for a write. The "r" field is the Reverse signal. Data is MSB at left of field.

Figure A-8 **Write switch message format — upstream.**

```
Any Write
==========
r   data
- --------
0 xxxxxxxx
1 -error--
1 -check--
0 xxxxxxxx
0 xxxxxxxx
```

A.5 Detailed Functional Description

The Requestor, Server, TCU and Configuration/Status Unit are now described in detail.

A.5.1 Requestor

The Requestor is described from the point of view of its overall operation and its two major interfaces: the T-Bus interface and the Switch Interface.

A.5.1.1 Operation

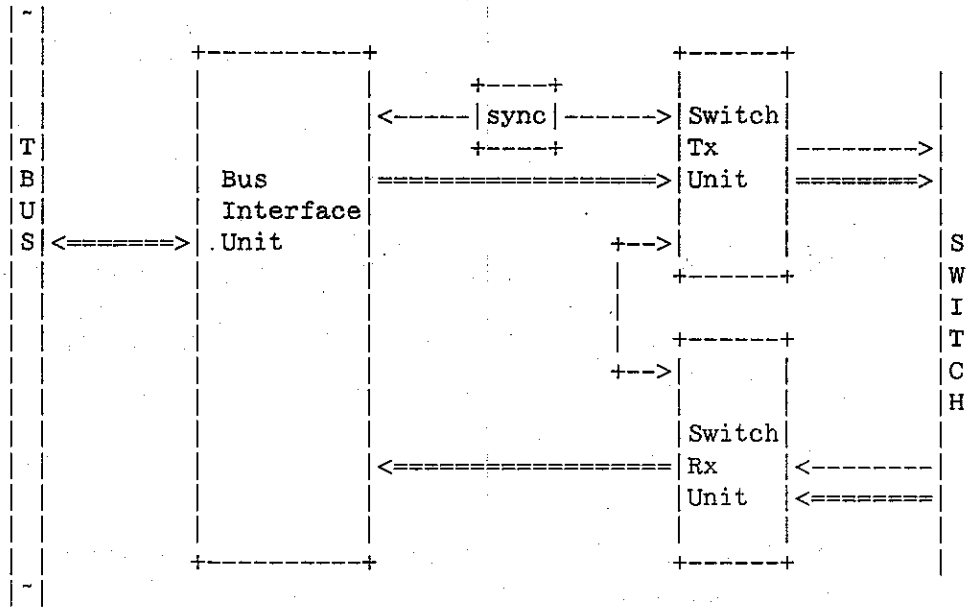
The operation of the Requestor is described by discussing its major functions.

A.5.1.1.1 Overview

The Requestor is a local T-Bus slave which creates a logical coupling to a physically remote T-Bus slave via the Switch. The Requestor acts as the "initiator" of this coupling on the Switch and thus can be thought of as a "slave" on the T-Bus but a "master" to the Switch. Referring to Figure A-9, the Requestor contains three major functional units: Bus Interface Unit (BIU), Switch Tx Unit (STU), and the Switch Rx Unit (SRU). The BIU is clocked by the T-Bus clock and both the STU and SRU are clocked by the Requestor Switch clock (R_CLK). Interfacing of control signals between these units is accomplished with handshake synchronizers, as shown. The BIU handles all of the T-Bus transactions of the Requestor. The STU translates function requests that it receives from the BIU into Switch transactions. The SRU receives reply messages from the Switch and passes their status, in the form of a status code, back to the STU and their data back to the BIU. The STU serves as the single interface for control information between the T-Bus side and Switch side of the Requestor and therefore control information in either direction must pass

through the STU. This is done to reduce the number of control interfaces that the BIU must deal with.

Figure A–9 Requestor block diagram.



The BIU/STU interface is a streamlined request/response type interface where for each BIU request there is an STU response. The BIU presents an encoded function request to the STU and sets an "execute" flag. When the STU is done operating on that request, it sets a "done" flag and returns a status code and data to the BIU. Both the BIU and STU are responsible for handling their own functions independently and they have very little real-time knowledge of each other's state. This approach simplifies the Requestor design and carries the request/response philosophy throughout the system.

The BIU has four major responsibilities: (1) screen T–Bus requests for correctness; (2) transfer screened T–Bus requests to the STU if a Switch transaction is indicated by that T–Bus request; (3) receive replies from the STU; and (4) pass replies, including any errors, as responses to the T–Bus. The BIU acts as a T–Bus slave which is always in split–cycle mode. In other words, it NEVER responds immediately to a function request from a T–Bus master except when a request error is detected. Outside of those exceptions, the BIU always responds with a PROMISE to T–Bus requests.

The BIU screens T–Bus requests for both T–Bus protocol violations and illegal function requests. Without exception, these conditions will prevent the BIU from ever activating the STU to complete an initial function request. The BIU can also initiate certain function requests to the STU independently of T–Bus

requests. An example of this is the drop-lock function which may under certain conditions be initiated by the BIU rather than the T-Bus.

The STU acts on a function request from the BIU and initiates the Switch transaction to carry out that request. The STU also is responsible for assembling and transmitting the data in an outgoing message. It also handles things such as the message start/retry and priority promotion algorithms and deals with various protocol timeout violations.

The SRU is fairly simple in function. It detects the return message of a function request initiated by the STU, verifies the checksum and alerts the STU of the incoming message and the checksum status. The SRU also detects Switch Rejects.

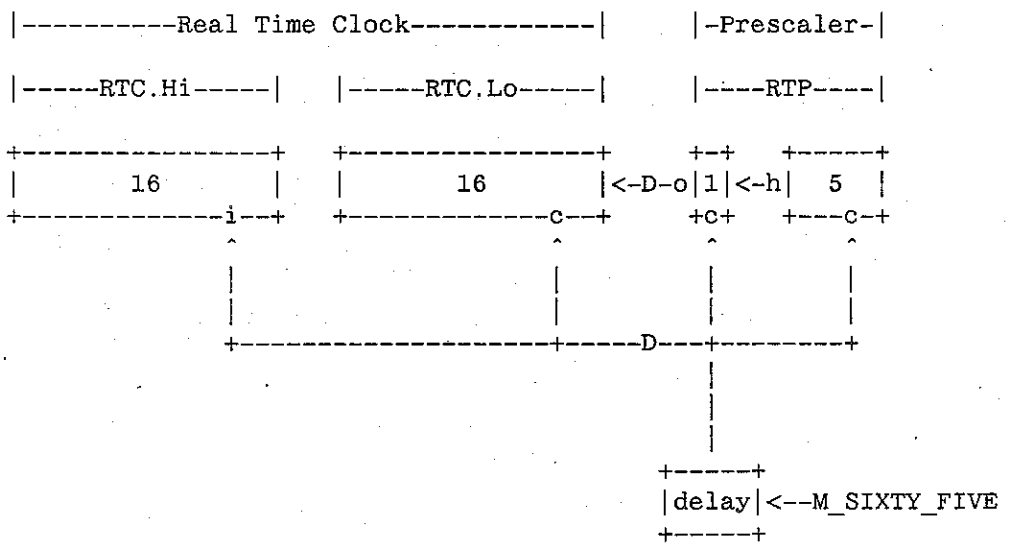
A.5.1.1.2 RTC and related functions

The Real Time Clock, besides being useful as a system timekeeper, is central to the operation of much of the Requestor. It is used to directly control the functions of the Time_Of_Next_Interrupt and the Priority_Time_Slot mechanisms. These mechanisms are described in this section. The RTC is also used, in a less direct manner, to control the Protocol Timers. Protocol timers are discussed elsewhere in this document.

A.5.1.1.2.1 Real Time Clock and Prescaler

The RTC is basically a large (32 bits) counter which is updated every one microsecond from a divided-down version of the Switch clock. Since the frequency of the Switch may vary in different applications, the Real Time Clock uses a programmable prescaler to divide the Switch frequency down to a one microsecond time base. A functional diagram of the Real Time Clock is shown in Figure A-10.

Figure A–10 Functional diagram — Real Time Clock.



...where,

h = increment pulse (period = 0.5 us)
o = increment pulse (period = 1.0 us)
s = increment pulse (period = 65536 us)
c = clear input
i = increment input
D = one D Flip-Flop for pipelining

Figure A–10 shows that prescaler is actually composed of two parts. The first part is a count-up prescale counter that has a programmable terminal count value. This 5-bit terminal value is supplied by the Real_Time_Prescale subfield of the ConfigA register (REQ_ConfigA.Real_Time_Prescale). The 5-bit counter drives the second part of the prescaler: a divide-by-two. The divide-by-two then generates the 1-microsecond time base used by the Real Time Clock. Figure A–10 also shows the presence of the M_SIXTY_FIVE signal. This signal is a system-wide pulse which occurs every 65 milliseconds and lasts for one Switch Interval. It is used to keep all the Real Time Clocks on all nodes in synchronization.

The M_SIXTY_FIVE resets the entire prescaler and the lower half of the Real Time Clock. In addition, it increments the upper half of the Real Time Clock. Figure A–10 also shows a "pipeline" delay for the M_SIXTY_FIVE signal. The Configuration bits, REQ_ConfigA.Sixty_Five_Delay<1..0>, allow the adjustment of this delay. The adjustment values and their effects are shown in Figure A–11.

Figure A–11 Sixty_Five_Delay settings.

```
DD   Delay
==   =====
00   none
01   1 Switch interval
10   2 Switch intervals
11   3 Switch intervals

...where,

D..D  = ConfigB.Sixty_Five_Delay<1..0>
```

WARNING

The setting DD = 00 is for test purposes only and must NOT be used in normal operation.

In actual operation, the prescaler RTP<4..0> counts up at the Switch frequency until it reaches the count stored in REQ_ConfigA.Real_Time_Prescale, where it generates an increment pulse lasting one Switch Interval. In the next Switch clock interval, the prescaler rolls over to zero. Thus, the ConfigB.Real_Time_Prescale must always be programmed to make RTP<5> have a period of 0.5 microseconds.

WARNING

Because of hardware speed considerations, the OMSP generated by the RTP is actually pipelined by one Switch Interval. Therefore, the RTP appears to be running "ahead" of the RTC by one Switch interval. This fact only becomes significant for the Slotted Start/Retry criterion. See that section for further details.

The Real Time Clock is basically, as mentioned previously, a large counter. The register definition of the Real Time Clock is shown in Figure A–12.

Figure A–12 Register definition — Real_Time_Clock.

```
Register: Real_Time_Clock<31..0>

31                                  0
|                                   |
15.............0 15.............0
HHHHHHHHHHHHHHHH LLLLLLLLLLLLLLLL (read)
HHHHHHHHHHHHHHHH ---------------- (write)
[Hi]             [Lo]

...where,

H..H  = high-order value (in 65,536 us)
L..L  = low-order value (in 1 us)
```

Referring to Figure A–12, both the upper and lower halves of the Real Time Clock (RTC.Hi) can be read from during actual operation. However, the lower half should not normally be written to because of unwanted side effects.

WARNING

Writing to RTC.Lo will cause the write data to override the count function but NOT override the clear function of that register. This means that the signal M_SIXTY_FIVE will clear the counter on its next occurrence. Therefore, there is not much meaning to initializing RTC.Lo.

WARNING

Any reads of the RTC must be taken as needed. This means that if the entire 32 bits must be read, it should be done in a single word mode operation. Performing this same function with two serial halfword operations will yield incorrect results. In addition, any reads of the Real Time Clock have an uncertainty of approximately one microsecond. For writes, ONLY the halfword mode is acceptable for loading a value into the RTC.Hi register. This operation should only be attempted after reading the RTC.Lo register and determining that it will not overflow when the write is being performed.

When performing reads of the Real Time Clock, the Configuration/Status Unit must take some special action to ensure that the read data is valid (stable). This is required because the Switch and T–Bus clocks are not always ensured to be synchronous and thus the Real Time Clock may be advancing as it is being read. The CSU accomplishes this goal in the following manner:

> When a read request for the Real Time Clock is detected by the CSU, the CSU immediately asserts the external SIGA pin: T_NSPAUSE_SIGA, and sends a request across a handshake synchronizer to the RTC control-

ler logic. The RTC controller logic then waits for the next occurrence of the one microsecond increment pulse from: Real_Time_Prescaler<5>. When this occurs, the CSU is ensured of having a stable reading from the Real Time Clock for at least one microsecond. The RTC controller logic then sends an acknowledgment back across the handshake synchronizer where the CSU, upon detecting this event, negates T_NSPAUSE_SIGA and allows the data to be read. This is what contributes to the one microsecond uncertainty mentioned above.

WARNING

The CSU relies on the fact that the requesting T-Bus master will ensure that the total time — from the next occurrence of the 1-microsecond increment pulse to the reading of data — will take no more than 1 us. This time includes the synchronizer delay from the RTC controller, the response time of the CSU, and time for any pauses that the T-Bus master may assert. Excluding the assertion of those pauses (T_NMPAUSE_xxxx) from the T-Bus master, the delay in the SIGA will be: 2*p(R_CLK) + 6*p(T_CLK) nanoseconds. The "p" represents the period of the indicated clock in nanoseconds. Therefore, the T-Bus master should use EXTREME caution when causing the assertion of T_NMPAUSE_xxxx. Beyond that, the CSU cannot guarantee the accuracy of the read data!

A.5.1.1.2.2 Time Of Next Interrupt

The Time Of Next Interrupt or TONI registers, are two 32-bit registers (A and B) which in combination with the Real Time Clock, are used to schedule an interrupt to occur at some moment in the future. Both registers, and their associated control logic, are completely independent from each other although they both interact with the Real Time Clock.

The TONI control logic performs a 32-bit subtraction between the current TONI_A (TONI_B) register values and the value of the entire Real Time Clock each time the OMSP is valid. Whenever this subtraction yields a negative (two's-complement form) number, the SIGA sets (=1) the bit: TONIA_Config.Status (TONIB_Config.Status).

Normally, whenever time the Status bit is asserted, an external pin, M_TONIA_INT (M_TONIB_INT), is also asserted (=1). This can be enabled/disabled — asynchronously to the OMSP — by setting the TONIA_Config.Enable (TONIB_Config.Enable) bit to a 1/0. Disabling will force ONLY the pin to a "0". The associated status bit will still reflect the result of the current subtraction. Figure A-13 illustrates the TONI register definition. Figure A-14 illustrates the TONIA(B) configuration register definition.

Figure A–13 Register definition — Time_Of_Next_Interrupt.

```
Register: Time_Of_Next_Interrupt

31.............................0
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT

...where,

T..T = interrupt value
```

Figure A–14 Register definition — TONIA(B)_Config.

```
Register: TONIA(B)_Config

31.............................0
------------------------------E (write)
------------------------------SE (read)

...where,

E = asynchronously enable external pin
  = 0 disable M_TONIA(B)_INT external pin
  = 1 enable M_TONIA(B)_INT external pin

S = status (raw output of the subtraction)
  = 0 TONIA(B) "interrupt" is not active
  = 1 TONIA(B) "interrupt" is active
```

The actual subtraction that is performed to initiate the interrupt is shown in Figure A–15. When performing writes to the TONI register, the Configuration/Status Unit must take some special action to ensure that the TONI register is not updated in the middle of the difference operation. The CSU accomplishes this goal in the following manner:

> When a write request for the TONI register is detected by the CSU, the CSU immediately asserts the external SIGA pin: T_NSPAUSE_SIGA and sends a request across a fixed-delay handshake synchronizer to the TONIA(B) controller logic. The TONIA(B) controller logic then waits for the next occurrence of the OMSP before it actually loads the TONIA(B) register. Because of pipelining, the TONIA(B) Subtraction Unit is ensured of having exactly one microsecond in which to complete the subtraction. The TONIA(B) controller then sends an acknowledgment back across the handshake synchronizer where the CSU, upon detecting this, negates T_NSPAUSE_SIGA, thus freeing up the T-Bus master. This means, of course, that the SIGA will assert T_NSPAUSE_SIGA for approximately one microsecond.

Figure A–15 Rule — time of next interrupt calculation.

```
TONIA(B)_Config<1> = 1 IFF,

    (TONIA(B)<31..0> - RTC<31..0>) < 0

...where TONIA(B) and RTC are treated as unsigned
   32-bit numbers and the difference is treated
   as a two's-complement number.
```

A.5.1.1.2.3 Priority Time Slot

The Switch protocol provides a mechanism by which initial messages may be transmitted at various levels of priority in order to place an upper bound on remote access time. Normally, this priority is set by the T–Bus bits, T_PRIORITY<1..0>, during the request phase of the T–Bus transaction. In this case, the initial message is transmitted/retransmitted with the priority set during the T–Bus transaction which initiated the message. However, the Requestor can also force these bits to their EXPRESS value independently of the T–Bus transaction via the Priority Time Slot mechanism.

This mechanism works by assigning each Requestor a particular active time slot which is based on the value of the Real Time Clock. When that time slot "arrives", any pending Initial Switch message in the Requestor will have its priority raised to the EXPRESS level (=00). The priority is "sticky" in that once raised to EXPRESS, it remains there until the T–Bus initiates a new Initial Switch message. This new Initial message updates the priority with the value of T_PRIORITY<1..0>, as normal.

The equation for determining the active Priority Time Slot is shown in Figure A–16.

Figure A–16 Rule — priority time slot promotion.

```
Priority Time Slot is active IFF the equation,

    (RTC.Lo<15..0> !$ PTC.Slot<15..0>) # PTC.Mask<15..0>

...is all 1's
```

This equation takes a slot value (PTC.Slot), compares it on a bit–by–bit basis with a portion of the Real Time Clock (RTC.Lo) and then logically "or's" the result with the priority slot mask (PTC.Mask). It then detects the result for

the presence of all "1's". Essentially, the RTC.Lo and the PTC.Slot are compared for equality on a word basis with some of the bits excluded, or "don't cared", in the comparison. A given bit position is excluded by setting the corresponding bit position in the Mask subfield to a "1". The Mask and Slot subfields, which are defined in Figure A-17, are programmable via the Configuration/Status Unit. The Priority Time Slot function can be disabled so that it NEVER promotes the priority of any message by negating (=0) the ConfigB.Ena_Priority_Promotion bit. The fully programmable capability of the Priority Time Slot allows the slot to be valid at different nodes in almost any order. It also allows the period of occurrence of the slot at a given node to be adjusted from constant up to 65 ms. Of course, the minimum time that a "slot" can be active at a given Requestor is one microsecond. Note that it is possible for the "slot" to arrive while the Requestor is sending out bids. This could result in one Bid being sent at lower priority and the remaining bid(s) sent at EXPRESS priority. However, logic in the Requestor ensures that no updating of priority occurs DURING Bid transmission. In addition, no updating will occur while the Requestor is either "idle" or "waiting". The "waiting" state is where the Requestor STU is waiting for a slotted/random start criterion to become valid.

Figure A-17 Register definition — Priority_Time_Config.

```
Register: Priority_Time_Config<31..0>

31                                0
|                                 |
15.............0 15.............0
SSSSSSSSSSSSSSSS MMMMMMMMMMMMMMMM
[Slot]           [Mask]

where,

S..S  = slot value
M..M  = mask value
```

Note that the purpose of the Priority Slot Value is NOT to ensure that a single high priority message be present in the Switch at any given time. Rather, the goal is to define the maximum bandwidth of priority messages to make the servicing of these messages as predictable as possible. In addition, the Priority Time Slot mechanism only applies to Initial Switch Messages (locked or not), which are always attempting to make a connection with some downstream node. Subsequent messages do not send Bids and thus are not affected by the Priority Time Slot mechanism.

A.5.1.1.3 Function Request Types

The Requestor handles various types of function requests from a T–Bus master. Those functions include read and writes of either bytes, words, or multiple words. Byte reads/writes may be of one to four bytes but must NOT wrap across word boundaries.

WARNING

It is important not to violate word wrapping because the Requestor does NOT check for this condition. Word reads/writes MUST be word aligned and multiple read/writes are limited to a maximum of four words.

A.5.1.1.4 T–Bus Request Screening

T–Bus requests to the BIU of the Requestor are screened for both context errors and T–Bus protocol errors before any action is taken on them. Protocol errors include such things as a T–Bus master requesting an illegal (=00) T_PRIORITY field or illegally wrapping across word boundaries. Currently, protocol errors are NOT detected. Context errors, mostly relating to errors in handling locking, are listed in Figure A–18.

Figure A–18 Requestor T–Bus screening errors.

1. Requestor was asked to access a node within a locked sequence which is different than the node which opened that sequence. (Lock Address Error)
2. Requestor was asked to MAINTAIN a remote lock when it was never opened. (Maintain Present Error)
3. Requestor was not asked to MAINTAIN, BYPASS or OPEN a lock that was not yet explicitly released with FREE–LOCK. In other words, a NORMAL was issued while the Requestor was locked. (Maintain Absent Error)

Any of these errors will cause the Requestor to return an ERROR response with the appropriate error code on the T–Bus (See: "Error Detection and Reporting"). In addition, no Switch message will leave the STU. If the Switch path happens to be locked, any of these errors will also cause the BIU to initiate a sequence which will tear–down the Switch path (drop–lock) providing certain conditions are met. See "Locked Sequences" for more details.

NOTE

The Requestor, if unlocked, will treat a BYPASS in the same manner as a NORMAL Function Request; that is, it will NOT open a lock.

A.5.1.1.5 Initial Message Start/Retry Criterion

The Requestor can use one of several different methods to decide when to first begin transmission of an Initial Message and when to retry that transmission if the Switch rejects it. These methods are referred to as: slotted, random and immediate. The start transmission time can be programmed to correspond to either one of two fixed time slots, one of two random numbers, or immediate transmission. The retry can correspond to either one of two fixed time slots or one of two random numbers. Only some combinations of these start and retry criterion are available for a given initial message.

The operation of random and slotted start and retry are described first. The process of selecting the various random/slotted start and retry criterion for a given message is then explained.

A.5.1.1.5.1 Random Start/Retry

There is a random number generator associated with the start/retry criterion. The generator is 12 bits long and is continuously updated at the Switch frequency. Each time an initial message start/retry occurs and the random backoff is selected, a new random number is transferred from the generator to a 12–bit count–down counter. This counter, known as the backoff counter, also runs at the Switch frequency. When the backoff counter reaches –1, the Requestor is released to start/retry the initial message transmission.

Before the backoff counter is actually loaded with the random number, that number is logically "anded" with a 12–bit backoff mask. When the Requestor first attempts the start/retry of an initial message, the backoff mask is initialized, forcing some number of most significant contiguous bits of the random number to zero as they are loaded into the backoff counter. After a certain number of Switch rejects for the same initial message, the mask is "shifted left" to allow an increase in the maximum allowable value of the next 12–bit random number loaded into the backoff counter. Thus, the random backoff limit, in terms of Switch intervals, is a binary number of length 12, or 4096. Each time a Switch reject is encountered, the Requestor makes a decision about whether or not to shift the backoff mask. That decision is made by adding a constant number to an accumulator after each Switch reject. Each time the accumulator overflows, the mask is shifted. Therefore, the mask may not change for several rejects.

In implementation, a register specifies randomization characteristics for the random start/retry criterion. This register is duplicated to allow for two sets

of characteristics to be stored simultaneously. The mechanism for choosing one set or the other is described in a subsequent section. Each register is 8 bits long and specifies the initial mask setting, the constant value for accumulator addition and whether or not immediate start transmission is requested. These registers, and the random specifications which they describe, are subfields of the Transmit_Time_Config Register known as "Random0" and "Random1". Figure A–19 illustrates the structure of the random registers. Referring to Figure A–19, the immediate field, "I", when "1", forces an initial random start to be immediate, ignoring any randomization parameters. For initial retries, the "I" field is ignored and the randomization parameters are always used. The constant value for accumulator addition is specified by the "EE" field. This number is added to a 3–bit accumulator, which is then tested for overflow. The initial backoff mask is specified by the 5–bit identifier, "MMMMM", which is loaded directly into a Johnson Counter. The output of the Johnson Counter is decoded to derive a 12–bit backoff mask as shown in Figure A–20. Figure A–20 also shows how the counter advances once loaded with an initial value. This advancement, of course, is governed by the overflow of the 3–bit accumulator. Also note that the LSB of the backoff mask can never be cleared.

Figure A–19 Register definition — Transmit_Time_Config.Random0,1.

```
Register: Transmit_Time_Config.Random0<7..0>,
          Transmit_Time_Config.Random1<7..0>

7......0
IMMMMMEE

where,

I     = immediate
EE    = accumulator addition constant
MMMMM = initial comparison mask
```

Figure A–20 Random start/retry bit mask encoding.

	mask identifier<5..0>	backoff mask<11..0>
increasing count	000000	000000000001
|	000001	000000000011
|	000011	000000000111
|	000111	000000001111
|	001111	000000011111
|	011111	000000111111
|	111111	000001111111
|	111110	000011111111
|	111100	000111111111
|	111000	001111111111
|	110000	011111111111
V	100000	111111111111

During the INITIAL start/retry, five of the mask identifier bits related to the initial message are specified by the "MMMMM" field in the random register. The sixth, most significant bit is ALWAYS initialized to "0". So, if MMMMM = "11111", the initial backoff identifier would be: "011111". In this case, the maximum possible random backoff is "1111110", or 128 Switch intervals (recalling that the backoff counter overflows at –1). Once the maximum identifier of "100000" has been reached, the counter "wraps around" and thus the next backoff mask will be "000000". The "multiply by two" effect of the left–shifting backoff mask is intended to implement an exponentially increasing random backoff. An equation summarizing the preceding discussion is shown in Figure A–21.

WARNING

The initial mask identifier MUST be a value which would result in a legal Johnson Counter value as shown in Figure A–20. Legal Values would be: "00011" or "01111" for example. Illegal values would be: "00100" or "10110", for example.

Figure A–21 Equation — maximum exponential random backoff.

$$\text{Maximum backoff (Switch intervals)} = 2^{[M + \text{int}(R*E/8)]}$$

...where,

M = initialized value of MMM bits
R = number of rejects
E = value of the EE bits

A.5.1.1.5.2 Slotted Start/Retry

Slotted start and retry involves holding off transmission based on the "arrival" of a pre-specified time slot. Once a slot has "arrived", a message assigned to that slot for starting can start transmission, and a message assigned to that slot for retry can retry transmission. The time slots are derived from the comparison of the Real Time Clock and a register used to specify the slot characteristics. This register is duplicated to allow for two sets of characteristics to be stored simultaneously. The mechanism for choosing one set over the other is described in a subsequent section. Each register is 8 bits long and specifies the comparison mask, the comparison value, and whether or not immediate start transmission is requested. These registers, and the slot specifications which they describe, are subfields of the Transmit_Time_Config Register known as "Slot0" and "Slot1". Figure A–22 shows the structure of the slot registers.

Figure A–22 Register definition — Transmit_Time_Config.Slot0,1.

```
Register: Transmit_Time_Config.Slot0<7..0>,
          Transmit_Time_Config.Slot1<7..0>

7..:...0
IMMDDDDD

...where,

I     = immediate
MM    = mask specification
          00  4.0 us slot period
          01  2.0 us slot period
          10  1.0 us slot period
          11  0.5 us slot period
DDDDD = phase specification (restricted, see text)
```

Referring to Figure A–22, the slot register contains three subfields: the compare mask field, specified by the two bit number, "MM"; the compare data field, specified by the five bit number, "DDDDD"; and immediate field, "I". The immediate field, when "1", forces an initial slotted start to be immediate, ignoring any slot parameters. For initial retries, the "I" field is ignored and the slot parameters are always used. The comparison for an active slot is made partially by comparing bits of the "D" subfield with bits of the of the Real Time Clock and Real Time Prescaler. The "M" subfield is used to either compare some of those bits with zeros or to ignore them in the comparison. This operation is shown in Figure A–23.

Figure A–23 Rule — start/retry valid slot comparison.

```
given, nnnnnnnn = RTC.Lo<1..0> | RTP<5..0>

mm    compare     with        cycle period
==    ========    ========    ============
00    OOODDDDD    nnnnnnnn       4 us
01    XOODDDDD    nnnnnnnn       2 us
10    XXODDDDD    nnnnnnnn       1 us
11    XXXDDDDD    nnnnnnnn      .5 us
```

Referring to Figure A–23, the D field can only take on values that are less than or equal to the setting of the Real_Time_Prescaler<4..0>.

WARNING

Values outside this range may cause the message to never be transmitted, and are therefore illegal.

Figure A–23 also demonstrates the two properties of the slots: frequency and phase. The D field allows setting a number of phases equal to the setting of REQ_ConfigA.Real_Time_Prescale<4..0> plus one. The M field allows the comparison to occur at varying time intervals.

Because of hardware limitations, the concatenated quantity, (RTC.Lo<1..0> | RTP<5..0>), does not act exactly like an eight bit counter. The RTP portion is actually running one switch interval "ahead" of the RTC.Lo<1..0> portion. This means that the RTC actually increments on the 000000–to–000001 transition of the RTP portion, rather than on the 111111–to–000000 portion. A sample transition would look like that in Figure A–24.

Figure A–24 Start/retry slot comparison count sequence.

```
RTC.Lo<1..0>      RTP<5..0>
============      ========
  10              11111100
  10              11111101
  10              11111110
  10              11111111
  10              00000000
  11              00000001
  11              00000010
  11              00000011
```

A.5.1.1.5.3 Start/Retry Criterion Selection

A function request from a master on the T-Bus is transformed into a Switch message by the Requestor. Depending on certain parameters of that function request, the Requestor categorizes the message into one of four Message Classes. Each of these classes will have a different start and retry criterion. The correspondence of start/retry criterion based on message classes is shown in Figure A-25.

Figure A-25 Start/retry criterion based on message classes.

Class	Start	Retry
00	Slot0/Immediate	Slot0
01	Slot1/Immediate	Slot1
10	Random0/Immediate	Random0
11	Random1/Immediate	Random1

A class is selected for each Switch message based on the state of three bits of T-Bus function request that initiated the message. Those bits are the T-Bus signals T_LOCKOP<1> and T_RR<1..0>. The Requestor uses the encoded state of those three bits to "look up" the class of the message. The look-up table itself is a 16-bit register known as the Message_Classification Register. This register is defined in Figure A-26.

Figure A–26 Register definition — Message_Classification.

```
Register: Message_Classification<15..0>

15                                          0
|                                           |
10   10   10   10   10   10   10   10
CC   CC   CC   CC   CC   CC   CC   CC
[A]  [B]  [C]  [D]  [E]  [F]  [G]  [H]

...where given that nnn = T_LOCKOP<1> | T_RR<1..0>,
   the subfields selected and the type of function
   request that selects them are,
```

nnn	Subfield	Function Request
000	MC.H	Unlocked Writes
001	MC.G	Unlocked Reads
010	MC.F	Auxiliary Unlocked Writes
011	MC.E	Auxiliary Unlocked Reads
100	MC.D	Locked Writes
101	MC.C	Locked Reads
110	MC.B	Auxiliary Locked Writes
111	MC.A	Auxiliary Locked Reads

To illustrate the Message Start/Retry Criterion selection with an example, suppose that a function request to the Requestor may have set, (T_LOCKOP<1> | T_RR<1..0>) = 100. From Figure A–26, this would cause the Requestor to look in the Message Classification register "D" subfield (for Locked Writes). In this subfield, the Requestor would find the "class of message" corresponding to the particular function request. If the "D" subfield were a "10", that particular message would have use the parameters in Random0 register for both message start and retry.

Both the Start/Retry Random and Start/Retry Slot registers are actually subfields of the Transmit_Time_Config Register. The bit definition for this register is illustrated in Figure A–27.

Figure A-27 Register definition — Transmit_Time_Config.

```
Register: Transmit_Time_Config<31..0>

31                                              0
|                                               |
7......0   7......0   7......0   7......0
IMMMMMEE   IMMMMMEE   IMMDDDDD   IMMDDDDD
[Random1]  [Random0]  [Slot1]    [Slot0]

...where, Random0, Random1, Slot0 and Slot1
   are previously defined
```

NOTE

Function requests can be forced to completely ignore the Message Classification register on a request–by–request basis. This occurs whenever a request is made and the T–Bus signal: T_SYNC is asserted (= 1). In this case, the message is automatically classed as "00" and both initial transmission and retry criterion is taken from the Transmit_Time_Config.Slot0 register.

A.5.1.1.6 Switch Tx Protocol Timers

The Requestor contains timers which monitor the progress of the transmitted message and alert the Requestor if they detect an error condition. Specifically, there are two timers, the Reject Timer and Connection Timer. The Reject Timer determines how long the Requestor will attempt to open a Switch path in the face of Switch rejects. The Connection Timer monitors how long the Requestor will keep a Switch path open once the rejection period is finished. Parameters for both the Reject Timer and the Connection Timer are contained in the Protocol_Timer_Config Register.

A.5.1.1.6.1 Reject Timer

The Reject Timer is enabled at the beginning of the first attempt to transmit an initial message. Each time the Requestor receives a reject, it first examines the Reject Timer. If the timer has underflowed (the underflow is latched), the Requestor halts the transmission attempt and returns the Rej_TO Error code to the T–Bus master. The Requestor also tears–down the Switch path whether or not it was locked. Parameters for the Reject Timer are located in the Protocol_Timer_Config Register.

The Reject Timer is structured as a 4–bit down–counter clocked by a selectable prescaled time base. The reload value for the counter is contained in Protocol_Timer_Config.Cnt<3..0>. A 4–bit prescale parameter, located in Protocol_Timer_Config.Pre<3..0>, is used to select the desired prescale time base

from one of sixteen possible frequencies. Those frequencies are derived from the low-to-high transition of bits of the Real Time Clock, Real_Time_Clock.Lo<15..0>, as illustrated in Figure A-28. The Reject Timer is continually loaded with PTC.Cnt<3..0> until it begins transmitting Bid #1. An equation for the maximum Reject timeout is shown in Figure A-29.

Figure A-28 Reject timer prescale selection.

PRE	Q	PRE	Q
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

...where,

PRE = Protocol_Timer_Config.Cnt<3..0>
Q = selection from Real_Time_Clock.Lo, bit Q

Figure A-29 Equation — reject timeout.

given,

CNT = Protocol_Timer_Config.Cnt<3..0>
PRE = Protocol_Timer_Config.Pre<3..0>

...then,

$$\text{Timeout} = CNT * 2^{(PRE+1)} \text{ microseconds}$$

...with an uncertainty of $2^{(PRE+1)}$ microseconds

A.5.1.1.6.2 Connection Timer

The Connection Timer is loaded each time the Requestor sends Bid 1. This means that it is reloaded both just before transmitting an initial message and after the Requestor receives each Switch reject. Like the Reject timer, its underflow condition is latched.

The Connection Timer's timeout has two different effects depending on when it occurs. If the timeout occurs while the Requestor is waiting for a message acknowledgment (M_ACK), the Switch path is torn-down (whether locked or not) and a Wait_TO Error is returned to the T-Bus master. If the timeout occurs while a Switch path is locked, but after the M_ACK was received, the Requestor will tear-down the Switch path but cannot return an error to the T-Bus master immediately. Rather, it waits until the next T-Bus master makes a request, and then returns an Idle_TO Error. In the "race condition" case where the M_ACK and connection timer underflow occur on the same clock edge, a Wait_TO Error is detected.

The Connection Timer is structured as an 8-bit down-counter clocked at 1 MHz by a bit from the Real Time Prescaler, Real_Time_Prescale<5>. The counter underflows at -1. The reload value for the counter is contained in, Protocol_Timer_Config.Con<7..0>. The equation for the maximum connection timeout is shown in Figure A-30.

Figure A-30 Equation — connection timeout.

```
given,

CON = Protocol_Timer_Config.Con<7..0>

then,

Timeout = CON + 1  microseconds

...with an uncertainty of 1 microsecond
```

A.5.1.1.6.3 Protocol Timer Programming

As previously mentioned, the parameters for the Protocol Timers are contained in subfields of the Protocol_Timer_Config Register as shown in Figure A-31.

Figure A-31 Register definition — Protocol_Timer_Config.

```
Register: Protocol_Timer_Config<15..0>

15                    0
|                     |
3..0   3..0   7......0
CCCC   PPPP   NNNNNNNN
[Cnt]  [Pre]  [Con]

...where, Cnt, Pre and Con have
   been previously defined.
```

A.5.1.1.7 Anticipation Support

The operation of the Requestor has two main goals: (1) to pass a T-Bus function request to the Switch as quickly and efficiently as possible, and (2) to return the corresponding function response from the upstream Switch message to the T-Bus master as quickly and efficiently as possible. Certain techniques can be used to take advantage of the expected operation of the logic in the function request and response path. These techniques are known collectively as "anticipation". The use of anticipation in achieving the two main goals of the Requestor are now discussed.

A.5.1.1.7.1 Function Requests

Maximizing downstream function request efficiency in the Requestor involves balancing the desire for speed with the desire to maintain a streamlined Switch protocol. These tradeoffs become apparent when considering a multi-word write sequence. Here, the Requestor could signal its Switch Transmit Unit to begin transmitting as soon as possible after receiving the T-Bus request. This would always work if the T-Bus were guaranteed to supply all words of a multi-word transfer at a bandwidth equivalent to the bandwidth of the Switch. However, this will not always be the case as the variations between the clock frequency of the T-Bus and the Switch, combined with the ability of the current T-Bus master to assert PAUSE, create the possibility of the STU "running out of data" in some circumstances.

To circumvent this problem, two immediate options are available. First, change the Switch protocol to allow the insertion of "null data word" fields when data is not available. Second, the Requestor could be programmed to signal the STU to start only after a specified number of words have been written during the data portion of the T-Bus transfer. The first alternative is unattractive because it increases Switch bandwidth and unnecessarily introduces complexity into the Switch message protocol. The second option is therefore

implemented in the Requestor. The programmed parameter is known as, FQ_Anticipation, and can be set to any of the thresholds listed in Figure A–32.

Figure A–32 Register definition — Requestor_ConfigA.FQ_Anticipation<2..0>.

```
Register: Requestor_ConfigA.FQ_Anticipation<2..0>

210     Anticipation
===     ============
000     after first data word transferred
001     after second data word transferred
010     after third data word transferred
011     after fourth data word transferred
1XX     immediately after T-Bus request
```

Since it is possible for the FQ_Anticipation to be set greater than the last word of a particular write, the Requestor will commit to transmission when either the last word has been written OR the Requestor FQ_Anticipation threshold has been reached — whichever occurs first. For example, if FQ_Anticipation were a "011" and a three word write occurred, anticipation would take place after the third word were written. In addition, an Interleaved request (I_INTERLEAVED = 1) will cause a "1XX" setting to signal the STU in the cycle AFTER the T-Bus request. The threshold should be set based on the T-Bus and Switch clock frequencies, the maximum number of PAUSE assertions expected during a write, and the handshake synchronizer delay setting.

WARNING

Because of current hardware restrictions, FQ_Anticipation MUST be set to 1XX. Therefore, any T-Bus master MUST be able to supply write data fast enough to prevent the Switch message from running out of data.

A.5.1.1.7.2 Function Responses

Anticipation during function responses would allow the Requestor to take advantage of the synchronizer settling time by beginning the T-Bus request BEFORE the message checksum has been verified. Unfortunately, the Requestor is limited in the amount of anticipation that it can provide. Whatever anticipation the Requestor can extract from an upstream message, that anticipation has to be constant over all messages. This is because the Requestor STU-to-BIU handshake synchronizer has to compensate for message anticipation and cannot have its setting varied according to the expected upstream message type. And of course, even if the anticipator could vary its setting, the return message profile is not always known.

In fact, the Requestor SRU must assume a minimum expected upstream message length before starting anticipation. That minimum message length is two bytes. And since the SRU cannot tell if the assertion of Reverse is a Reject until the second byte, the minimum anticipation of the Checksum byte is one Switch Interval (for a function response to a write request). This then limits anticipation of all messages to one byte. By comparison, the Server has a minimum message length of 5 bytes and can thus take greater advantage of anticipation techniques.

As previously mentioned, Switch to T-Bus anticipation usually requires some minimum setting on the receiving T-Bus synchronizer. However, it turns out that no MINIMUM setting of Req_ConfigA.BIU_Synch<3..0> is required to compensate for the small amount of Requestor SRU anticipation. This is because pipeline overhead already accounts for this anticipation. However, a minimum setting IS required to meet the minimum settling time for the synchronizer. For more details on this subject, see: "Special Topics/Synchronization".

A.5.1.1.8 Locked Sequences

Sometimes an upstream T-Bus master wishes to perform several consecutive function requests to a locked remote T-Bus slave without the overhead of opening the Switch connection before each request. A mechanism known as Switch locking allows such multiple accesses by keeping the Switch path open between function requests. All transactions that take place during locking are known as locked sequences. A locked sequence has three distinct events: opening, maintaining and closing. Each of these events has different characteristics and restrictions for the Requestor.

A.5.1.1.8.1 Opening and Maintaining Locks

Opening a Switch lock begins with an otherwise normal function request from a T-Bus master that carries with it a request for "opening a lock" to a remote T-Bus slave. The upstream Requestor transfers the OPEN lock request to the downstream Server via a bit in the message protocol. Since the Switch path has not yet been established, either the Switch or the downstream Server may reject the message. A Switch reject will occur because of normal Switch contention and the Server reject will occur if the downstream target was locked. The Requestor, not knowing the source of the Switch reject, will simply retry the message transmission within the constraints of the Protocol Timers.

Assuming that the message finally does "get through" to the downstream Server, that Server "opens a lock" to the target T-Bus slave in accordance to the T-Bus protocol. Meanwhile the upstream Requestor, recognizing that it has established the beginning of a locked sequence, does not normally tear-down the Switch connection upon receiving an M_ACK unless an error was detected. This is discussed in detail in the "Auto Drop" section.

Once a locked Switch path is established with OPEN lock, it must be explicitly instructed to remain open by the upstream T-Bus master. This is accomplished by following the OPEN function request with either: another OPEN, a MAINTAIN, or BYPASS function request. Essentially, the Requestor takes no special action on either of these requests but does demand their presence. If the OPEN/MAINTAIN/BYPASS protocol is violated by subsequently initiating a NORMAL function request, the Requestor will respond to the offending T_Bus master with an ERROR and tear-down the Switch path. This mechanism is described in the "T-Bus Request Screening" section.

A.5.1.1.8.2 Dropping Locks

The Requestor has a flag, known as the "drop-lock request" flag, which causes the Requestor to negate Frame and return to its unlocked Idle state. Although the flag does not cause this action until the Requestor BIU is in its Locked Idle state, it can be set at any time. Once set, a drop-lock condition is said to be active. There are three distinct scenarios under which a drop-lock condition may occur: (1) A T-Bus master which is locked to the Requestor may issue a FREE-LOCK, (2) The Requestor issues an ERROR response (under certain conditions), and (3) a Connection Timer timeout.

Whatever the cause of the drop-lock condition, the Requestor BIU waits until it returns naturally to its Locked Idle state before taking action. Once there, the Requestor BIU will then enter the "unlock" state in which it will fulfill the drop-lock request flag by commanding the Requestor STU to negate Frame. During this state, the Requestor BIU will issue a REFUSED response to ANY T-Bus Master that accesses it. Once the Requestor BIU has been signaled by the STU that Frame was negated, the BIU returns to its Unlocked Idle state. Of course, the drop-lock request flag is then also negated. The downstream Server, knowing that it was previously locked, interprets the subsequent loss of its incoming Frame to be a FREE-LOCKS. The Server, sensing an unexpected loss of Frame, then issues a FREE-LOCKS to the local T-Bus.

The first drop-lock scenario — a FREE-LOCKS issued by a T-Bus master — is the most conventional. The FREE-LOCKS request is the only function request that is NOT explicitly transmitted to the downstream Server in the form of a message. Instead, the Requestor responds to a FREE-LOCKS by negating Frame to the Switch interface. Because the drop-lock condition can be entered at any time, a T-Bus master can issue a FREE-LOCKS at any time — whether the Requestor is idle or acting on a current split-cycle. However, the Requestor must be already locked to the T-Bus master which made the request. If not, the BIU will ignore the FREE-LOCK request.

In the ERROR response scenario, the Requestor will NEVER enter the drop-lock condition when the ERROR response is due to a Remote Class Error. However, it MAY enter the drop-lock condition when the ERROR response is due to an FQ or Switch Class Error. This conditional action is described in the "Auto Drop" section. Error classes are discussed in the "Error Detec-

tion and Reporting" section. However, if those conditions ARE valid for a drop-lock, the Requestor processes the drop-lock in the same manner as the FREE-LOCKS scenario. Unlike the FREE-LOCKS however, drop-lock processing takes place almost immediately after the event which caused the drop-lock condition (responding with an ERROR). This is because the Requestor BIU always enters its Locked Idle state immediately after issuing an ERROR response.

The Connection Timer timeout scenario is slightly different from the previous two. When the Connection Timer times out, it indirectly causes the drop-lock condition by eventually causing an ERROR response (Wait_TO or Idle_TO) by the Requestor BIU. This normally would be sufficient because the BIU would then enter the drop-lock condition, which would then signal the Requestor STU to negate Frame. However, one of the reasons that the Connection Timer may have timed out was because the Requestor BIU had lost its T-Bus clock (T_CLK). In this case, Frame would never get negated. Therefore, the Requestor STU takes the initiative to negate Frame immediately after a Connection Timer timeout. For consistency, the drop-lock mechanism continues as normal. When the Requestor STU finally gets the request from the BIU to negate Frame, the STU simply ignores that request.

A.5.1.1.8.3 Auto Drop

Auto drop is a parameter set by the Req_ConfigA.Ena_Auto_Drop bit. When asserted (= 1) the Requestor will be permitted to enter the drop-lock condition whenever an ERROR response is generated because of an FQ or Switch Class error. Otherwise, the Requestor will NEVER enter the drop-lock condition due to an ERROR response. This is because the only other class of Requestor error — Remote Error — will NEVER cause the drop-lock condition.

A.5.1.1.9 Stolen Bit Support

Because of the structure of the Switch message format, only one bit of Stolen information can be transferred between upstream and downstream nodes during a given message. The Requestor records the state of the Stolen bit during the word transferred in a byte write operation. It is this state that is reflected in the Switch message. Normally, the Requestor expects the Stolen bit to be asserted only during a BYTE write operation. In fact, it is illegal to assert the Stolen bit to the Requestor during a multi-word operation.

NOTE

If the Stolen bit IS asserted during a multi-word write, the state of the first word written is recorded.

The Requestor provides a mechanism to verify that the Stolen bits of all words in a multi-word write are zero, and prevent the message from being transmitted if this is not the case. The Ena_Stolen_Verify bit in the Req_ConfigB register, when asserted, will enable this verification of Stolen bits in a multi-word write. There is however, a small price to pay for this feature: the FQ_Anticipation register must be set to its MAXIMUM value (=011). This is because the Requestor must load all words of a multi-word write and verify the Stolen bits before committing to transmission. The Requestor cannot "call back" the outgoing message. Figure A-33 summarizes the rules for verifying the Stolen bit. If the rules of Figure A-33 are adhered to and a particular multi-word write has some of the Stolen bits asserted, the Requestor will respond with an ERROR ("Stolen_Verify" error code) to the T-Bus master. The Requestor, of course, will NOT transmit the message in this case.

Figure A-33 Rules — Stolen bit verification — multi-word write.

```
To enable the verification of Stolen bits on a multi-word writes,

  1) Set FQ_Anticipation = 011, AND...

  2) Assert (=1) the Req_ConfigB.Ena_Stolen_Verify bit
```

For single-word reads, the Requestor presents to the T-Bus a Stolen bit (T_AD<32>) which is the same state as the Stolen bit in the upstream Checksum byte. For multi-word reads, the Requestor always assumes that the words of the transfer are NOT Stolen until it encounters an asserted Stolen bit in the Checksum byte. When this occurs, only the last word received by the Requestor is assumed to be Stolen. This fact is transmitted to the T-Bus by asserting T_AD<32> during the transfer of the last word on the T-Bus.

WARNING

Due to current hardware restrictions, Ena_Stolen_Verify must ALWAYS be negated (=0). Therefore, multi-word writes CANNOT be screened for stolen bits by the Requestor. This must be handled by the T-Bus Master.

A.5.1.1.10 Quick Drop

The Requestor STU has an option which enables it to negate Frame during an Initial Message as soon as the STU detects an asserted Reverse. This can be done without the STU actually waiting to see if Reverse is going to be a Reject or an actual message. This action is allowed only when the STU is transmitting an Initial Message (NOT an Initial Locked Message) because in this situation, the only possible responses are: Reject or an upstream Switch message. In either case, the Requestor will negate Frame immediately if the bit:

Requestor_ConfigB.Ena_Quick_Drop is asserted (=1). Essentially, Quick Drop is an optimization which will free up the Switch earlier — although only by one Switch Interval — than if Quick Drop were not enabled.

A.5.1.1.11 Reverse Profile Monitoring

The Requestor is enabled to monitor the profile of Reverse for errors asserting (=1) the Req_ConfigB.Ena_Rev_Err bit. Once enabled, the Requestor will report a Switch Class Error (Reverse_Error) whenever it observes an incorrect state for Reverse during an upstream message. Since there is more than one possible Reverse profile for a given Function Request, not every Switch Interval of Reverse can be checked for a given state (0/1) because either state may be valid. However, when the Reverse profile is incorrect in ANY place that is checked, a Reverse_Error is reported.

Figure A–34 illustrates how the Requestor checks the Reverse profile. The "x's" represent where either state is valid and is therefore not checked by the Requestor.

Figure A-34 Requestor Reverse profile monitoring.

```
TYPE     #WORDS              RETURN MSG FORMAT
=====    =========           ================

                             +--- first received
                             |
                             V
write    any                 xxL

read     non-multiple        xxHH,HL

         two words           xxHH,HL
            "                xxHH,HxHH,HL

         three words         xxHH,HL
            "                xxHH,HxHH,HL
            "                xxHH,HxHH,HxHH,HL

         four words          xxHH,HL
            "                xxHH,HxHH,HL
            "                xxHH,HxHH,HxHH,HL
            "                xxHH,HxHH,HxHH,HxHH,HL

...where,

   x = don't care
   H = check for Reverse = 1
   L = check for Reverse = 0
```

NOTE

The Requestor will NOT specifically check that Reverse was negated (=0) when the Function Request was initiated. However, it DOES begin looking for a 0-to-1 transition of Reverse in order to recognize the beginning of the upstream message. Therefore, if Reverse were to be "hung high" when the Requestor began its Function Request, the Requestor would eventually timeout the Connection Timer.

A.5.1.1.12 Error Detection and Reporting

Errors delivered by the Requestor to an initiating T-Bus master can be divided into three classes depending on which part of the SIGA detects them. The classes are: 1) FQ Errors — which are detected by the BIU from the original Function Request; 2) Switch Errors — which are detected by the STU and SRU because of Switch interactions and 3) Remote Errors — which are detected

by the downstream Server and are "reflected" up to the initiating T-Bus Master.

For a given Function Request/Response sequence, errors from different classes can occur simultaneously. Since only one error can be reported at a time, a sense of "priority" exists between error classes. If there is a FQ Error, it always be reported, regardless of the presence of Switch or Remote Errors. If there is no Local Error, than any Switch Errors will be reported, regardless of the presence of Remote Errors. If there is neither a Local nor a Switch Error, then and only then will any Remote Errors are reported.

Figure A–35 shows the Error Codes for the Requestor which include the FQ and Switch type errors. Note that WITHIN a given Error Class, the errors are again not all mutually exclusive, and are therefore given "within class" priorities. A more detailed description of the three Error Classes follows.

Figure A–35 Requestor error codes.

```
Requestor Error Codes:

7       0
|       |
PPPPdcba

d c b a  Requestor Error          Class
= = = =  ====================    =========
0 0 0 0  Maintain_Absent (1a)    FQ
0 0 0 1  Maintain_Present (1b)   FQ
0 0 1 0  Stolen_Verify (2)       FQ
0 0 1 1  Lock_Address (3)        FQ
0 1 0 0  Wait_TO (4a)            Switch
0 1 0 1  Idle_TO (4b)            Switch
0 1 1 0  Rej_Abort (5)           Switch
0 1 1 1  Rej_TO (6)              Switch
1 0 0 0  Reverse (7)             Switch
1 0 0 1  Check (8)               Switch

...where,

  P..P = Requestor_ConfigA.Error_Prefix<3..0>.
         Priority is from highest (1) to lowest (8).
         Within a given priority, errors are mutually
         exclusive (i.e., 4a,b...).
```

A.5.1.1.12.1 FQ Errors

FQ Errors are detected by the BIU during the original Function Request. Their detection, when enabled, will ALWAYS prevent the Function Request

from initiating a Switch access. If the Requestor is unlocked, it will NOT assert Frame after detecting an FQ Error. If the Requestor is locked, it MAY immediately tear-down the lock if certain conditions are met. See "Auto Drop" for more details.

FQ Error types and their definitions are illustrated in Figure A-36.

Figure A-36 FQ error definitions.

Lock Address Violation	Requestor was asked to access a node within a locked sequence which is different than the node which opened that sequence. (Only detected if configured to do so.)
Maintain Present	Requestor was asked to MAINTAIN a remote lock when it was never OPENed. (Only detected if configured to do so.)
Maintain Absent	Requestor was not asked to MAINTAIN, BYPASS or OPEN a lock that was not yet explicitly released with FREE-LOCK. (Only detected if configured to do so.)

A.5.1.1.12.2 Switch Errors

Switch Errors are caused by a variety of conditions that are detected by the logic which monitors the progress of the Switch message as it enters and returns from the Switch interface. Unlike FQ Errors, Switch Errors are detected once the Switch transaction is already underway. They are reported to the T-Bus Master only when the transaction is "finished", either normally or due to some timeout. Therefore, Switch Errors can only have a special effect on Frame during a locked sequence. In this case, the Requestor MAY immediately tear-down the lock if certain conditions are met. See "Auto Drop" for more details.

Switch Error types and their definitions are illustrated in Figure A-37.

Figure A-37 Switch error definitions.

Wait_TO	The Switch Transmit Connection Timer overflowed while the Requestor was waiting for a Function Response. (See: "Connection Timer")
Idle_TO	The Switch Transmit Connection Timer overflowed while the Requestor was in its idle state. (See: "Connection Timer")
Rej_Abort	The Switch Transmit Reject Timer was forced into overflow by the REJ_ABORT input pin. (See: "Reject Timer")

Rej_TO The Switch Transmit Reject Timer overflowed while the Requestor was attempting to open a connection. (See: "Reject Abort")

Reverse The Requestor detected an incorrect polarity of the Reverse signal during a Function Response. (See: "Reverse Profile Monitoring")

Check The Requestor detected an incorrect Checksum during a Function Response. (See: "Checksum Support")

A.5.1.1.12.3 Remote Errors

Remote Errors include: 1) errors which are detected within the Server logic itself, and 2) errors generated as T-Bus errors responses by a downstream T-Bus slave device. Both types of errors are simply passed through "as is" to the upstream Requestor. This Requestor simply "hands" them — without differentiation — to the initiating T-Bus Master. Remote Errors, unlike FQ and Switch Errors, can NEVER cause the Requestor to "drop" a lock.

For a summary of the "Server sourced" Remote errors, see: "Server/Operation/Error Reporting".

A.5.1.1.13 Disabled Operation

The Requestor can be disabled via a number of bits in the Requestor_ConfigB register. These include: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_SRU, and Ena_REQ_CNT. These bits reset the four major blocks of the Requestor.

WARNING

In normal operation, these bits SHOULD ALWAYS BE ASSERTED/NEGATED AT THE SAME TIME. Otherwise, erratic Requestor operation may result.

When these bits are disabled (= 0), the Requestor T-Bus interface will respond "REFUSED" to any T-Bus master that tries to access it. The Requestor will also ignore any assertions of REVERSE from the Switch interface.

A.5.1.1.14 Configuration Registers

The Requestor has two general Configuration Registers. They are: Requestor_ConfigA and Requestor_ConfigB. In general, both Configuration Registers are used to set miscellaneous parameters and enable/disable certain functions. Figure A-38 shows the structure of Requestor_ConfigA.

Figure A–38 Register definition — Requestor_ConfigA.

```
Register: Requestor_ConfigA<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    =====================
<31..29>   REQ_Slave_Num[3]
    <28>   Modulo_8
    <27>   Columns_2
    <26>   Ena_Auto_Drop
<25..23>   FQ_Anticipation[3]
<22..19>   STU_Synch[4]
<18..15>   BIU_Synch[4]
<14..11>   Error_Prefix[4]
 <10..9>   Sixty_Five_Delay[2]
  <8..6>   CSU_Slave_Number[3]
  <5..1>   Real_Time_Prescale[5]
     <0>   Columns_1
```

The bit definition of Requestor_ConfigA is shown in Figure A–39. This register contains mostly configuration bits that affect the run–time parameters of the Requestor. All bits are "high true" and are reset (low) upon system reset. The structure of Requestor_ConfigB is shown in Figure A–40. The bit definition of Requestor_ConfigB is shown in Figure A–41. This register contains mostly configuration bits that enable/disable different functions and error reports of the Requestor. All bits are "high true" and are reset (low) upon system reset.

Figure A–39 Bit definition — Requestor_ConfigA.

REQ_Slave_Num[3] Configures the T–Bus slave number that the Requestor will respond with (on the T_SOURCE<2..0> pins) when making a Function Response.

Modulo_8 Configures the Requestor to expect either a modulo–8 element (=1) or a modulo–16 (=0) Switch element.

Columns_2 Configures the Requestor to expect either a 2–column (=0) or a 3–column Switch.

Ena_Auto_Drop Enables the Requestor to tear–down a connection when a Function_Request or Switch class of error is detected (=1). Otherwise, these types of error will only be reported by the Requestor and no special action will be taken (=0).

FQ_Anticipation[3] Configures the Requestor for the desired Function Request Anticipation. (See: "Anticipation Support")

STU_Sync[4]	Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives an "execute" signal from the Bus Interface Unit (BIU). This signal is used to initiate a Function Request on the Switch. (See: "Synchronization")
BIU_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "completed" signal from the switch transmit unit (STU). This signal is used to indicate that a function response has been received by the SRU. (See: "Synchronization")
Error_Prefix[4]	Configures the Prefix (T–Bus bits: D7–D4) of the Error code response for Requestor errors. (See: "Error Handling")
Sixty_Five_Delay[2]	Configures the pipeline delay of M_SIXTY_FIVE pulse. Millisecond pulse as seen by the Requestor. WARNING: DO NOT USE THE "00" SETTING. (See: "Real Time Clock" for further details)
CSU_Slave_Number[3]	Configures the Slave number that the CSU will respond with (on the T_SOURCE<2..0> pins) when making a Function Response.
Real_Time_Prescale[5]	Configures the terminal count of the Real Time Prescaler. (See: "Real Time Clock" for further details)
Columns_1	Configures the SIGA for a 1–column switch. (See: "Real Time Clock" for further details)

Figure A-40 Register definition — Requestor_ConfigB.

```
Register: Requestor_ConfigB<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    =====================
 <31..23>    Route_Address_Mask[9]
     <22>    Ena_Stolen_Verify_Err
     <21>    Ena_Maintain_Absent_Err
     <20>    Ena_Maintain_Present_Err
     <19>    Ena_Lock_Addr_Err
     <18>    Ena_Wait_TO_Err
     <17>    Ena_Idle_TO_Err
     <16>    Ena_Rej_Abort_Err
     <15>    Ena_Rej_TO_Err
     <14>    Ena_Check_Err
     <13>    Ena_Reverse_Err
     <12>    Ena_Remote_Err
     <11>    Ena_Quick_Drop
     <10>    Ena_Priority_Promotion
      <9>    Ena_Interleaver
      <8>    Ena_Reject_Abort
      <7>    Ena_Reject_Timer
      <6>    Ena_Conn_Timer
      <5>    Ena_Switch_Frame
      <4>    Ena_REQ_BIU
      <3>    Ena_REQ_STU
      <2>    Ena_REQ_SRU
      <1>    Ena_REQ_CNT
      <0>    SPARE
```

Figure A-41 Bit definition — Requestor_ConfigB.

Route_Address_Mask[9] Configures the randomization mask for the Bus Interface Unit's translation of the Logical Route Address to the Physical Route Address. (See: "Route Address Generation")

The Enable Error bits allow the indicated errors to be reported(= 1), or to be unreported (= 0). With some noted exceptions they DO NOT prevent the errors from occurring. The error functions that these bits enable/disable are described in the "Error Handling" section. The bits are as follows:

Error Bit
= = = = = = = = = =
Ena_Stolen_Verify_Err — must ALWAYS be negated (= 0)
Ena_Maintain_Absent_Err — enable/disable detection

Ena_Maintain_Present_Err — enable/disable detection
Ena_Lock_Addr_Err — enable/disable detection
Ena_Wait_TO_Err — enable/disable detection
Ena_Idle_TO_Err — enable/disable detection
Ena_Rej_Abort_Err — enable/disable detection and mechanism
Ena_Rej_TO_Err
Ena_Check_Err
Ena_Reverse_Err
Ena_Remote_Err

Ena_Stolen_Verify_Err Must ALWAYS be negated (=0). (See: Anticipation Support)

Ena_Maintain_Absent_Err Enables (=1) or disables (=0) the detection of a Maintain_Absent_Error. Disabling this bit allows the Requestor to initiate a Switch transaction and pass through the incorrect T_LOCKOP<1..0> field. (See: "Error Detection and Reporting/FQ Errors")

Ena_Maintain_Present_Err Enables (=1) or disables (=0) the detection of a Maintain_Present_Error. Disabling this bit allows the Requestor to initiate a Switch transaction and pass through the incorrect T_LOCKOP<1..0> field. (See: "Error Detection and Reporting/FQ Errors")

Ena_Lock_Address_Err Enables (=1) or disables (=0) the detection of a Lock_Address_Error. Disabling this bit allows the Requestor to initiate a Switch transaction with a potentially incorrect Physical Route Address. (See: "Error Detection and Reporting/FQ Errors")

Ena_Wait_TO_Err Enables (=1) or disables (=0) the detection of a Wait_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Connection Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Idle_TO_Err Enables (=1) or disables (=0) the detection of an Idle_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Connection Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Rej_Abort_Err Enables (=1) or disables (=0) the generation AND detection of an Idle_TO_Error. Disabling this bit will prevent the Requestor from recognizing the state of the pin: M_REJ_ABORT. (See: "Error Detection and Reporting/Switch Errors")

Ena_Rej_TO_Err Enables (=1) or disables (=0) the detection of a Rej_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Reject Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Check_Err	Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")
Ena_Reverse_Err	Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")
Ena_Remote_Err	Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")
Ena_Quick_Drop	Enables (=1) or disables (=0) the Requestor Switch Transmitter to negate Frame as early as possible on an Unlocked operation. (See: "Quick Drop")
Ena_Priority_Promotion	Enables (=1) or disables (=0) the Priority Promotion mechanism. (See: "Priority Promotion")
Ena_Interleaver	Enables (=1) or disables (=0) the Requestor's detection of the INTERLEAVED pin. (See: "Physical Route Address Generation")
Ena_Reject_Abort	Enables (=1) or disables (=0) the Requestor's responding to the REJ_ABORT pin. (See: "Reject Timer")
Ena_Reject_Timer	Enables (=1) or disables (=0) the operation of the Reject Timer. This bit will override the Ena_Reject_Abort bit.
Ena_Conn_Timer	Enables (=1) or disables (=0) the operation of the Connection Timer.
Ena_Switch_Frame	Enables (=1) or disables (=0) the assertion of the REQ_SW_FRAME pin. This function overrides any other function which effects the assertion of the REQ_SW_FRAME pin.
Ena_REQ_BIU	Enables (=1) or resets (=0) the Requestor Bus Interface Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_STU, Ena_REQ_SRU, Ena_REQ_CNT. (See: "Disabled Operation")
Ena_REQ_STU	Enables (=1) or resets (=0) Requestor Switch Transmit Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_SRU, Ena_REQ_CNT. (See: "Disabled Operation")
Ena_REQ_SRU	Enables (=1) or resets (=0) Requestor Switch Receive Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_CNT. (See: "Disabled Operation")
Ena_REQ_CNT	Enables (=1) or resets (=0) Requestor Counter (Timer) Module. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_SRU. (See: "Disabled Operation")

Columns_1 Configures the Requestor to expect a 1-column Switch (=1). In this case, the Requestor still uses Columns_2 to determine the Bid construction. When negated (=0), the Requestor uses Columns_2 for both number of bids to be sent AND bid construction. (See: "Downstream Message Components")

A.5.1.1.15 Test Registers

The Requestor also contains a test register, Requestor_TestA. This register contains bits that are related to production testing of the SIGA, and unlike all other configuration registers, a read of Requestor_TestA does not yield the data last written. The write bits are initialized in their negated state and are related to production testing of the SIGA. The read bits are used to observe the internal state of the Requestor. They will yield no useful information during normal operation.

WARNING

Write bits of Req_TestA SHOULD NEVER BE ASSERTED DURING NORMAL OPERATION.

The write structure of Requestor_TestA is shown in Figure A–42.

Figure A–42 Register definition — Requestor_TestA (write).

```
Register: Requestor_TestA<31..0>

BIT/FIELD    FUNCTION (write)
=========    ================
     <31>    SPARE
     <30>    TST_CNT_RTP_FORCE_OMSP
     <29>    SPARE
     <28>    TST_CNT_RJT_LOAD_COUNTER

     <27>    TST_CNT_RJT_DECREMENT_COUNTER
     <26>    TST_CNT_COT_LOAD_ENABLE
     <25>    TST_CNT_RSR_BOC_COUNT_DISABLE
     <24>    TST_CNT_RSR_MSK_REG_COUNT_ENABLE

     <23>    TST_CNT_RSR_ADDER_LOAD_ENABLE/RANDOM_CLEAR
     <22>    TST_CNT_RSR_IDLE_DISABLE
     <21>    TST_TIO_RND
   <20..0>   SPARE[21]
```

The function of the bits in Figure A–42 are listed below:

TST_CNT_FORCE_OMSP This bit affects only the real time prescaler. When asserted, this bit will cause the One MicroSecond Pulse signal to be forced high continuously. This will cause other portions of the TM to count unusually quickly. When not asserted, the Real Time Prescaler will generate OMSP normally every n clock cycles.

TST_CNT_RJT_LOAD_COUNTER

When asserted, this bit will force the continuous loading of the four bit counter within the reject_timeout counter. The counter will be loaded with the initial value specified in the configuration register. When not asserted, the counter is loaded normally, at the beginning of every Requestor operation.

TST_CNT_RJT_DECREMENT_COUNTER

The reject_timeout counter will be decremented on every cycle when this bit is asserted. When both the decrement and load test bits are asserted, the load_counter takes precedence. A reject_timeout is issued from the reject_timer the cycle after the count reaches zero. Under normal operation, with this bit negated, the counter is decremented every n microseconds.

TST_CNT_COT_LOAD_ENABLE

This test bit is used to force the eight bit counter within the connection_timeout unit to load continuously from the configuration registers. With this bit negated, the counter is loaded at the beginning of every Requestor bid transmission.

TST_CNT_RSR_BOC_COUNT_DISABLE

When this bit is asserted, the back_off counter within the random restart/retry logic is disabled, preventing random timeouts from occurring when the counter reaches its terminal count.

TST_CNT_RSR_MSK_REG_COUNT_ENABLE

This bit is used to force the six bit mask_register_out Johnson counter to increment on every cycle. This permits the counter to be incremented without starting a Requestor operation. Under normal operation, this counter is incremented only after n switch rejects have occurred during an attempted Requestor transmission.

TST_CNT_RSR_ADDER_LOAD_ENABLE/RANDOM_CLEAR

This bit controls two testing functions. When asserted, this bit will force the backoff added to load a new value. This permits the adder to be tested without the normal requirement that the Requestor be issued a switch reject for each adder load operation. The other function of this bit is to clear the 12 bit random number generator. This will prevent random numbers from being OR'ed into the backoff counter, simplifying the testing of the module.

TST_CNT_RSR_IDLE_DISABLE

When asserted, this bit will disable the idle state of the Requestor from clearing the backoff adder and Johnson counter. With this bit asserted, the functions of these counters can be tested without starting a Requestor operation. When de-asserted, with the Requestor in the idle state, the backoff adder is cleared, and the Johnson counter will load the initial state specified in the Requestor configuration registers.

TST_TIO_RND The nine bit random number generator, used in the RQ_BI_TI module to fill the logical route address bits in the Requestor bids, is cleared whenever this bit is asserted. This simplifies the predictions of Requestor bids in testing since random numbers are no longer inserted.

The read structure of Requestor_TestA is shown in Figure A-43.

Figure A-43 Register definition – Requestor_TestA (read).

```
Register: Requestor_TestA<31..0>

BIT/FIELD    FUNCTION (read)
=========    ================
     <31>    TST_TM_SSR_SLOT_VALID
     <30>    TST_TM_RSR_PTS_ACTIVE
     <29>    TST_TM_RSR_RANDOM_TO
     <28>    TST_TM_RSR_RANDOM_GENERATOR

     <27>    TST_TM_RSR_CARRY_OUT
     <26>    TST_TM_COT_CONNECTION_TIMEOUT
     <25>    TST_TM_RJT_REJECT_TIMEOUT
     <24>    TST_TM_RJT_DECREMENT

     <23>    TST_TM_RTP_Q5
     <22>    TST_SR_REJ_DET
  <21..15>   TST_SR_FSM
     <14>    TST_ST_LOCKED
   <13..1>   TST_ST_FSM
      <0>    TST_ST_RAND_ROUTE
```

The function of the bits in Figure A-43 are listed below:

TST_TM_SSR_SLOT_VALID This bit indicates the state of the slotted start/retry module output SLOT_VALID. This signal may be used by the Requestor to retransmit a bid after a switch reject has put it into the backoff state.

TST_TM_RSR_PTS_ACTIVE The state of the Priority Time Slot signal is readable from this bit. The PTS signal will boost the priority of a bid after a switch reject has been issued.

TST_TM_RSR_RANDOM_TO The Random Timeout from the random start/retry module may be read, which has the same function as the slot_valid signal, depending on the message class of the Requestor operation.

TST_TM_RSR_RANDOM_GENERATOR
This bit reads the MSB of the 12-bit pseudo random number generator inside the backoff counter of the random start/retry module.

TST_TM_RSR_CARRY_OUT This bit is asserted when the backoff adder of the random start/retry module has an overflow, indicating that the Johnson counter will be incremented on the next backoff occurrence of the current Requestor operation.

TST_TM_COT_CONNECTION_TIMEOUT
The connection timeout indicates that the allotted Requestor connection time has expired. If the Requestor is in the WAIT state, the operation will be aborted.

TST_TM_RJT_REJECT_TIMEOUT
The reject timeout indicates that the time permitted for the Requestor to make a connection has expired. When the Requestor is in the backoff state, waiting to retransmit, the operation will be aborted.

TST_TM_RJT_DECREMENT This bit indicates when the reject_timeout module 4-bit counter receives a decrement pulse from the OMSP prescaler.

TST_TM_RTP_Q5 The RTP Q5 register may be read with this bit. This is the MSB of the Real Time Prescaler counter, which generates the OMSP signal. OMSP occurs on the falling edge of this signal.

TST_SR_REJ_DET This is the internal signal generated by the Requestor receiver indicating that a switch reject has been received during a Requestor operation.

TST_SR_FSM These seven bits show the current state of the Requestor receiver. The states, from MSB to LSB are: END, CHECK, BYTE2, BYTE1, W1_B1, ARMED, and IDLE.

TST_ST_LOCKED This bit is asserted whenever the Requestor transmitter is in a locked operation.

TST_ST_FSM These thirteen bits indicate the current state of the Requestor transmitter state machine. The state, from MSB to LSB are:. DONE, TO_IDLE, LOCKED_IDLE, LOCKED_DONE, BACKOFF2, BACKOFF1, WAIT, CHECK, DATA, CMD, BID1, HOLD, and IDLE.

TST_ST_RAND_ROUTE This bit reads the MSB of the 9-bit psuedo random number generator used in the logical route address of the Requestor bids.

A.5.1.2 Switch Message Protocol

The Requestor fully generates and supports the Butterfly Switch protocol. That support is described below.

A.5.1.2.1 Physical Route Address Generation

The Switch route address from the T-Bus field, T_AD<33..25>, is actually a logical address. This Logical Route Address, which has two possible sources, undergoes a transformation to derive the Physical Route Address. It is the Physical Route Address which is assembled into the bid symbols of the downstream Switch message. The Logical Route address is used in the calculation of the Header Partial Sum (see the Requestor/Checksum Calculation section). During a given function request, the two possible sources of Logical Route Address for the Requestor are the T-Bus (T_AD<33..25>) and the interleaver port (I_MOD<8..0>). The interleaver port is chosen if: (1) the I_INTERLEAVED pin is asserted on the SIGA during the T-Bus request cycle AND (2) the Enable_Interleave bit in the Requestor_ConfigB register is asserted.

NOTE

It is assumed that both the T-Bus Master making the request and the Interleaver will force any unused bits in Logical Route Address to "0" as it is presented to the pins of the SIGA.

Whichever routing address is actually chosen, that 9-bit quantity undergoes a transformation. It is modified to allow the randomization of a selectable number of the routing bits. The random bits that potentially replace routing bits are obtained from a 9-bit random number generator, the Random Route Generator, which runs at the T-Bus clock rate. A bit in the route address can be specified as random by setting a corresponding bit in the Route Address Mask register to a "1". The transformation for the Physical Route Address generation can be expressed by an equation as shown in Figure A-44.

Figure A–44 Equation — Physical Route Address generation.

```
temp<8..0> = MOD & INT & INT_EN
             # [T_SNN & (!INT # !INT_EN)]

 PRA<8..0> = (RAND & RAM) # (temp & !RAM)

     ...where,

     T_SNN     = T_AD<33..25>
     MOD       = I_MOD<8..0>
     INT       = T_INTERLEAVED
     INT_EN    = Req_ConfigB.Ena_Interleaver
     RAND      = RAND<8..0>, random # generator
     RAM       = Route_Address_Mask<8..0>
     PRA       = Physical Route Address
```

The first equation in Figure A–44 represents the selection of either the Interleaver port or the T–Bus port for the Logical Route Address. The second equation randomizes selected bits in the Logical Route Address. The Route Address Mask is located in the Req_ConfigB configuration register.

A.5.1.2.2 Downstream Message Components

Some of the relevant aspects of the downstream Switch message components are now discussed. For a more detailed explanation of Switch message definition and protocol, see the reference documents.

A.5.1.2.2.1 Header

The construction of the message header, which contains the bid symbols, varies depending on the modulus of the Switch, which can be either 8 or 16. The SIGA design will support both options, although the modulo–8 Switch is the most likely to be encountered. In addition, the Requestor can support a one, two or three column Switch. Figure A–45 shows the format of the bid symbols in both modulus configurations. As seen from Figure A–45, certain bid symbols may never be sent if the Switch is small enough. Note that a modulo–8 switch is always expected to have at least two switch columns and a modulo–16 can have as few as one. The random bits mentioned in Figure A–45 are obtained from a separate random number generator known as the Random Route Generator.

Figure A–45 Bit definition — downstream message header.

```
7                                   0
|                                   |
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 1)  (first sent)
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 2)       V
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 3)  (last sent)

...where,

P1..P0 = priority from T-Bus: PRIORITY<1..0>
Ra..Rd = Physical Route Address (see below...)
```

COL1	COL2	MOD8	BID1 Rd Rc Rb Ra	BID2 Rd Rc Rb Ra	BID3 Rd Rc Rb Ra
0	0	0	n2 n1 n0 R8	R7 R6 R5 R4	R3 R2 R1 R0
0	0	1	0 R8 R7 R6	0 R5 R4 R3	0 R2 R1 R0
0	1	0	R7 R6 R5 R4	R3 R2 R1 R0	----------
0	1	1	0 R5 R4 R3	0 R2 R1 R0	----------
1	0	0	n2 n1 n0 R8	----------	----------
1	0	1	0 R8 R7 R6	----------	----------
1	1	0	R7 R6 R5 R4	----------	----------
1	1	1	0 R5 R4 R3	----------	----------

```
...where,

COL2        = Requestor_ConfigA.Columns_2
COL1        = Requestor_ConfigA.Columns_1
MOD8        = Requestor_ConfigA.Modulo_8
n1,n2,n3    = random bits
---------   = Bid is NOT transmitted
```

A.5.1.2.2.2 Body

The message body, which contains the command, address, data and checksum bytes, varies based on the type of message being sent downstream. The general format is shown in Figure A–46. Figure A–46, of course, shows a single word write message. For multi–word write transfers there would be correspondingly more data bytes. For a read message, the difference would be that all data fields would be missing and bit S would be forced to a zero.

Figure A–46 Bit definition — downstream message body (write).

```
7                                                   0
|                                                   |
L1   L0   R1   R0   S2   S1   S0   A24  (first sent)
A23  A22  A21  A20  A19  A18  A17  A16        |
A15  A14  A13  A12  A11  A10  A9   A8         |
A7   A6   A5   A4   A3   A2   A1   A0         |
D31  D30  D29  D28  D27  D26  D25  D24        |
D23  D22  D21  D20  D19  D18  D17  D16        |
D15  D14  D13  D12  D11  D10  D9   D8         |
D7   D6   D5   D4   D3   D2   D1   D0         |
                    .                         |
   <possible additional write words>          |
                    .                         |
                                              V
F    0    0    S    CS3  CS2  CS1  CS0  (last sent)

...where,

L1..L0   = lock operation from T-Bus: T_LOCKOP<1..0>
R1..R0   = portion of field from T-Bus: T_RR<1..0>
           R1  R0
           ==  ==
           0   0    write
           0   1    read
           1   0    <unused>
           1   1    <unused>
S2..S0   = size information from T-Bus: T_SIZE<2..0>
A24..A0  = address information from T-Bus: T_AD<24..0>
D31..D0  = data information from T-Bus: T_AD<31..0>
F        = enable forward drivers
           F = 0   disable forward drivers next clock
           F = 1   enable forward drivers next clock
S        = Stolen Bit
CS3..CS0 = message checksum
```

NOTE

The current SIGA design ALWAYS forces the "F" bit to be a "0".

A.5.1.2.3 Checksum Support

The Requestor and Server each have two separate units of checksum logic. The first, known as the Transmit Checksum Unit, calculates the message checksum during its transmission. The second, known as the Receive Checksum Unit, calculates and verifies the checksum for the incoming message.

The elements included in the calculation of the checksum of a downstream message vary depending on the type of message being transmitted. For any initial message (locked or unlocked), the Requestor always initializes its Transmit Checksum Unit with the "flash" sum of the Logical Route Address. The Logical Route Address can, of course, come from either the MOD pins (interleaved access) or from the T-Bus (non-interleaved). For any locked messages, the Requestor always initializes its Transmit Checksum Unit to zero.

In the same way, the downstream Server must initialize its Receive Checksum Unit to ITS node checksum whenever it expects an initial message. This initialization value will, of course, match that calculated by a Requestor about to transmit to that Server's node. For locked messages, the Server will initialize its Receive Checksum Unit to zero, just as the Requestor does with its Transmit Checksum Unit.

In an upstream message, there are NEVER any routing bits to contend with. Therefore, the downstream Server always initializes its Transmit Checksum Unit to zero, as does the Requestor's Receive Checksum Unit.

A.5.1.2.4 Checksum Calculation

The checksum for a downstream message is actually calculated in two parts. If the message is an initial (locked or unlocked) one, a partial sum of the message header is calculated (by separate logic) and stored in the Transmit Checksum Unit. Then, the Transmit Checksum Unit adds (exor's) the initial value, if any, to the bytes of the body of the message as it is transmitted.

A.5.1.2.4.1 Header Partial Sum

The header partial sum is derived by considering only the Logical Route Address bits. This means that the priority and random bits are not included in the calculation. This approach eases the design of the checksum logic and makes it independent of the Switch modulus. The equation for this calculation is shown in Figure A-47.

Figure A-47 Equation — Requestor header partial sum calculation.

```
HPS<3> = R8 $ R7 $ R3
HPS<2> = R6 $ R2
HPS<1> = R5 $ R1
HPS<0> = R4 $ R0

...where,

HPS<3..0>  = Header Partial Sum
R8..R0     = Logical Route Address
```

A.5.1.2.4.2 Message Checksum

As previously mentioned, the header partial sum is added (exor'ed) to the body of a downstream message if and only if that message is an initial message. The message checksum calculation is shown in Figure A–48. Figure A–48 shows the calculation for a single word write message. For write messages with more words, those bytes would be included in the same manner as the data bytes in the figure. For read messages, the data field would be missing entirely from the calculation.

Figure A–48 Equation — message checksum (see text).

```
CS<3> = HPS<3> $ exor(L1,S2,A23,A19,A15,A11,A7,A3,
                      D31,D27,D23,D19,D15,D11,D7,D3,F)

CS<2> = HPS<2> $ exor(L0,S1,A22,A18,A14,A10,A6,A2,
                      D30,D26,D22,D18,D14,D10,D6,D2,0)

CS<1> = HPS<1> $ exor(R1,S0,A21,A17,A13,A9,A5,A1,
                      D29,D25,D21,D17,D13,D9,D5,D1,0)

CS<0> = HPS<0> $ exor(R0,A24,A20,A16,A12,A8,A4,A0
                      D28,D24,D20,D16,D12,D8,D4,D0,S)

...where,

exor'ed components from: "Bit Definition - Message Body"
CS<3..0>  = message checksum
HPS<3..0> = Header Partial Sum
```

NOTE

The "F" field is always "0".

A.5.1.2.5 T–Bus Interface

The Requestor supports the standard T–Bus protocol with some small limitations. For one, the Requestor does NOT support unaligned transfers which fall across word (32–bit) boundaries. In addition, when it is locked to a T–Bus Master and in its "WAIT" state, the Requestor will always issue a REFUSED LOCKED to ANY T–Bus query while it is busy processing a split–cycle request. This means that it will even REFUSED LOCKED to its own T–Bus master! This is a hardware optimization which should cause no problems. The locking T–Bus master normally has no reason to query the Requestor until the Requestor finishes its current operation.

Figure A–49 shows the Requestor's state–dependent T–Bus responses while it is in some of its more interesting states.

Figure A–49 Requestor T–Bus responses (partial list).

```
NEXT RESPONSE                CONDITION
=============  ==================================================
State = IDLE  (satisfied a function request, waiting for new one):
 PROMISE          !LOCKED & !DROP_LOCK &  read
 PROMISE          !LOCKED & !DROP_LOCK &  write & !multi
 MORE             !LOCKED & !DROP_LOCK &  write &  multi
 REFUSED          !LOCKED &  DROP_LOCK
 REFUSED LOCKED    LOCKED & !DROP_LOCK & !my_master
 PROMISE           LOCKED & !DROP_LOCK &  my_master &  read
 PROMISE           LOCKED & !DROP_LOCK &  my_master &  write & !multi
 MORE              LOCKED & !DROP_LOCK &  my_master &  write &  multi
 REFUSED           LOCKED &  DROP_LOCK

State = WAIT  (waiting for function request to traverse Switch)
    -or-
State = BREQ  (making T-Bus request for T-Bus with split response):
 REFUSED          !LOCKED
 REFUSED LOCKED    LOCKED
```

A.5.1.2.6 LCON Interface

The LCON is a the physical and logical link between the SIGA Requestor and the "input" port of the Switch Gate Array (SGA). In other words, for the SIGA, the LCON interface is the logical Switch interface. The LCON provides the Requestor with: 1) level conversion to and from the ECL levels of the SGA and 2) reclocking of data, Frame, Reverse and the 65 ms pulse to and from the SGA.

Figure A–50 shows the Requestor's LCON (Switch) Interface Pins.

Figure A–50 Requestor LCON (Switch) interface pins.

PIN NAME	TYPE	FUNCTION
R_DATA<7..0>	bidirectional	Requestor-LCON data bus
R_FRAME	output	Frame output to Switch
R_REVERSE	input	Reverse input from Switch
R_NENA_BACK	output	LCON TTL driver enable
M_SIXTY_FIVE	input	65 ms timer input

A.5.1.2.6.1 Data Bus Enable Control

The Requestor controls the enables of both its own output drivers and the LCON's output drivers to the SIGA–LCON data interface — R_DATA<7..0>. To control its own output drivers, the Requestor generates an internal signal called, nena_out. When asserted (=0), nena_out enables the Requestor's R_DATA<7..0> drivers. To control the LCON, the Requestor provides the R_NENA_BACK signal to directly enable(=0)/disable(=1) the LCON's output drivers to R_DATA<7..0>. In addition, R_NENA_BACK, after a flip–flop delay, is used to enable/disable the LCON's Switch data ECL interface bus. When the Requestor is driving R_DATA<7..0>, it is in "Talk" Mode. When the LCON is driving that bus, the Requestor is in "Listen" Mode.

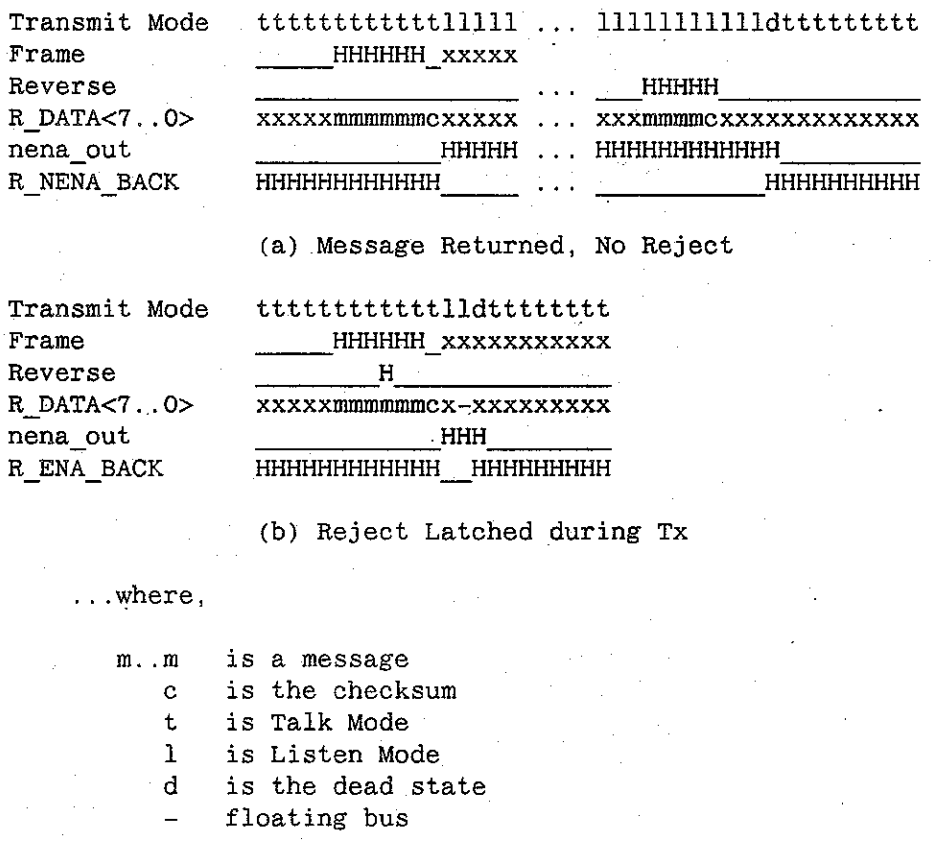
There are two major reasons why the Requestor separately provides the R_NENA_BACK signal. First, the Requestor already "knows" which direction the bus should be driving, and therefore this logic need not be repeated in the LCON. Second, this configuration gives the Requestor the ability to prevent bus contention.

Bus contention can occur when the direction of data changes on the LCON interface. If R_NENA_BACK changed on the same clock edge as nena_out, there would be contention on R_DATA<7..0> each time both of those signals changed. However, because of timing skew and minimum delays, contention is actually only a problem when the Requestor tries to enable its own drivers as it disables the LCON's backward drivers. This occurs during the transition from Listen to Talk Mode. But since the Requestor has separate control of its own output drivers and the LCON's, it can prevent this case of contention. It does this by inserting a "dead" state for one Switch Interval where neither the Requestor nor the LCON is driving R_DATA<7..0>.

The Requestor is considered "quiescent" when it is not transmitting messages and not waiting for any replies. When quiescent, the Requestor is in Talk Mode. The Requestor tries to stay in Talk Mode whenever possible, making the transition to Listen only for the absolute minimum time necessary. This situation is the mirror image to the Server. It is always in Listen Mode when quiescent and tries to stay in Talk Mode for as little time as possible.

When the Requestor finishes transmitting the checksum of an Initial or Locked message, it transitions directly into Listen Mode. Once there, it waits for either a Reject (which could have been detected and latched during the message transmission) or a return message. When either of those two events are complete, the Requestor transitions back to the Talk Mode, via the dead state. Figure A–51 shows this sequence for both a replied and a rejected Switch message. Note from Figure A–51 that there is a dead state only when making a transition from Listen to Talk Mode. Although not shown in the figure, subsequent Locked messages act in the exact same manner.

Figure A-51 **Timing — Requestor switch data bus enable.**



A.5.2 Server

The Server is described from the point of view of its overall operation and its two major interfaces: the T-Bus interface and the Switch Interface.

A.5.2.1 Operation

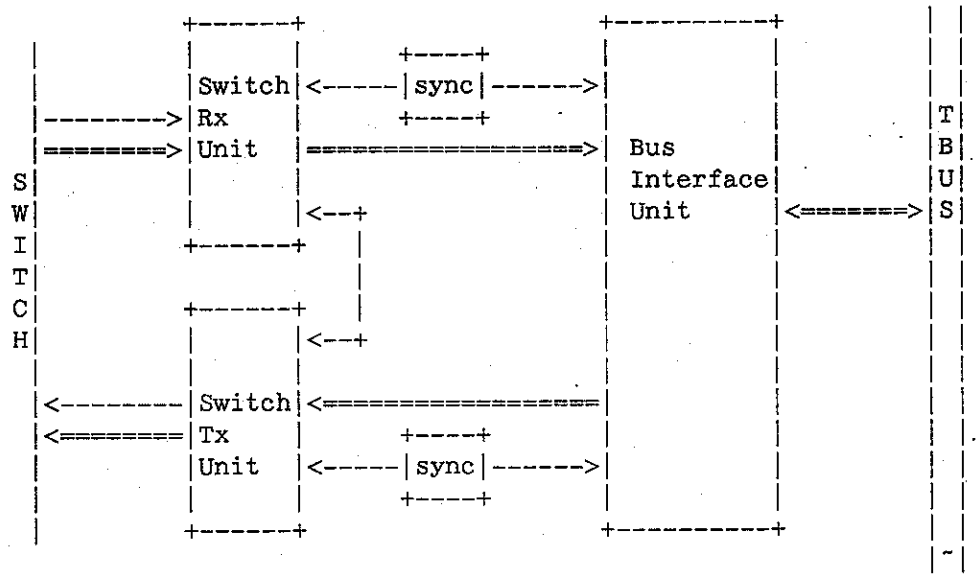
The operation of the Server is described by discussing its major functions.

A.5.2.1.1 Overview

The Server is a local T-Bus master which creates a logical coupling to a physically remote T-Bus slave via the Switch. The Server acts as the "responder" of this coupling on the Switch and thus can be thought of as a "master" on the T-Bus but a "slave" to the Switch. Referring to Figure A-52, the Server contains three major functional units: Bus Interface Unit (BIU), Switch Tx Unit (STU), and the Switch Rx Unit (SRU). The BIU is clocked by the T-Bus clock and both the STU and SRU are clocked by the Switch clock. Interfacing of control signals between these units is accomplished with handshake synchro-

nizers, as shown. The SRU receives function requests from the Switch and translates those requests into commands for the BIU. The BIU handles all of the T–Bus transactions of the Server to comply with a given function request. When a T–Bus slave device responds to a function request, the BIU picks up that response and passes it as a command to the STU. The STU then initiates an upstream Switch message to return the function response.

Figure A–52 Server block diagram.



The SRU detects the downstream message of a function request, verifies the checksum and alerts the BIU of the incoming message and the checksum status. The SRU also causes Switch rejects when either the BIU has explicitly commanded this action or when the SRU decides to on its own. The BIU will command a Switch reject when a function request is trying to access a T–Bus device which is locked to a T–Bus device other than the Server. The SRU will NOT initiate a reject without a command from the BIU and thus CANNOT correctly handle a non sequitur downstream message. A non sequitur would occur, for instance, when the SRU receives a function request (in the form of a downstream message) and knows that the STU has not even begun to send an upstream Switch message in response to the last function request.

The SRU has the additional responsibility of initiating a FREE–LOCKS command to the BIU when the Switch path is locked and the incoming Frame signal negates unexpectedly. This situation is known as "dropping a lock" and is the ONLY time when the Server does not create a Function Response as a result of an explicit function request.

The SRU/BIU interface is a streamlined request/response type interface where for each SRU request there is an BIU response. The SRU presents an encoded function request to the BIU and sets an "execute" flag. When the BIU

is done operating on that request, it sets a "done" flag and returns a status code and data to the SRU. The SRU also has the ability to "interrupt" the pending BIU operation. This is accomplished with a "terminate" handshake signal from the SRU. The "terminate" handshake receives a "terminate done" from the BIU when the BIU finishes. This "interrupt" path is used for situations where the BIU may be indefinitely "hung" because a failed T-Bus slave is continuously asserting Slave pause.

Both the SRU and BIU are responsible for handling their own functions independently and they have very little real-time knowledge of each other's state. This approach simplifies the Server design and carries the request/response philosophy throughout the system.

The BIU has three major responsibilities: (1) initiate T-Bus requests to comply with a command from the SRU; (2) receive responses from the T-Bus; (3) transfer those responses, along with any error indications, to the STU. To accomplish the T-Bus request/response transfer, the BIU supports most of the T-Bus protocol.

The STU is a fairly simple device. It acts on a function response from the BIU and initiates the upstream Switch message to carry out that response. The STU also is responsible for assembling and transmitting the data in an outgoing message.

A.5.2.1.2 Anticipation Support

The operation of the Server has two main goals: (1) to pass a downstream Switch function request to a T-Bus slave as quickly and efficiently as possible, and (2) to return the corresponding function response from that T-Bus slave as quickly and efficiently as possible. Certain techniques can be used to take advantage of the expected operation of the logic in the function request and response path. These techniques are known collectively as "anticipation". The use of anticipation in achieving the two main goals of the Server are now discussed.

A.5.2.1.2.1 Function Requests

Maximizing downstream function request efficiency in the Server involves balancing the desire for speed with the desire for eliminating unwanted side effects. The speed issue relates to the desire to transfer data from an incoming Switch message to the T-Bus as soon as it is available. Unwanted side effects involve taking any action on the T-Bus that would cause a change in stored data in a T-Bus slave device given that the downstream message was corrupted. Two extreme approaches could be taken in the design of the Server. First, the Server could wait until the entire downstream message had been received, including the checksum; verify the checksum; and then begin access

to the T-Bus. Second, the Server could begin access to the T-Bus immediately upon receiving a downstream message.

The first approach would cause the Server to waste valuable time in accessing the T-Bus, and the second could possibly cause unwanted side effects. Since one of the design goals of the Butterfly II is that data integrity should take precedence over speed, a compromise between the first and second approaches is implemented in the Server.

The Server "anticipates" the verification of the downstream checksum and begins it's request for T-Bus drivership. The timing is set up such that the Server BIU is commanded by the SRU to make a bus request at a specific moment in time. In fact, the SRU commands the BIU (input to the BIU synchronizer) to begin the T-Bus request EXACTLY five Switch intervals before the "Checksum_is_OK" signal is valid. This is true for both reads and writes. Therefore, the synchronizer setting, Server_ConfigA.BIU_Xfer_Sync<3..0> should be set accordingly. See "Synchronizer Settings" for more details.

A.5.2.1.2.2 Function Responses

The Server uses a similar technique as the Requestor for anticipating T-Bus transactions. Of course, in the case of the Server, the anticipation is for Function Responses rather than Function Requests. The Server_ConfigA.Multi_Head_Start<1..0> register is used to set the anticipation for multi-word writes. Figure A-53 illustrates its settings.

Figure A-53 Register definition — Server_ConfigA.Multi_Head_Start<1..0>.

```
Register: Server_ConfigA.Multi_Head_Start<1..0>

10    Wait until...
==    ============
00    all words are transferred
01    three words have been transferred
10    two words have been transferred
11    one word has been transferred
```

In addition, the Server_ConfigA.Ena_Byte_Head_Start bit, when asserted (=1), begins anticipation whenever the T-Bus Slave responds with EARLY-ACK.

Normally, the Server will anticipate for reads only. However, in some hardware configurations it is possible to anticipate on writes. When Server_ConfigB.Ena_Wr_Head_Start is asserted (=1), the Server treats writes exactly the same way as reads for all purposes.

WARNING

Using anticipation in multi-word writes can cause unusual side effects if the multi-word write does not complete in time. This is because the Server SRU may mistakenly believe that the write data buffers are actually stable until the upstream Requestor has seen the Function Response and taken some action. As seen by the Server, this response takes quite long, at least 4–6 Switch Intervals. Thus, if the multi-word write takes only this long to complete, there is no problem.

WARNING

Using read anticipation requires that the T-Bus Slave issue an ERROR before transferring any data.

NOTE

The EARLY-ACK response has no meaning for multi-word reads or writes, and this response is ignored by the Server. Also, the Server must examine the T_RR field even though T_SPAUSE may be asserted.

A.5.2.1.3 Locked Sequences

The Server's handling of locked sequences parallels that of the Requestor and is described in the "Requestor/Operation/Locked Sequences" section. Like the Requestor, the Server's locked sequence has three distinct events: opening, maintaining and dropping.

The Server becomes locked if and only if it receives an Initial Locked message (OPEN, by definition is the command). It remains locked as long as it returns any function response except Reject. When a lock is dropped at the upstream Requestor, Frame is negated. As mentioned in the "Requestor/Operation/Locked Sequences" section, a Requestor drop-lock function request can occur as the result of a T-Bus master issuing a FREE-LOCK or possibly a Requestor Switch Class error. The Server NEVER knows the reason for the drop-lock request, it simply issues the perfunctory FREE-LOCK to a T-Bus slave.

A.5.2.1.4 Stolen Bit Support

Because of the structure of the Switch message format, only one bit of Stolen information can be transferred between upstream and downstream nodes during a given message. Therefore, during byte reads, the Stolen bit from the Server's T-Bus is transported to the upstream Requestor exactly as it is read from T_AD<32> during the data transfer cycle of the T-Bus. For multi-word reads, the Server continues the T-Bus transaction, reading and storing all of

the intended words even when it encounters a Stolen bit BEFORE the last word of the transfer.

However, when the Server finally transmits that data to the upstream Requestor, it acts differently depending on whether or not the data contains a Stolen bit. If it does not, all of the multi-word data is included in the upstream message and the Stolen bit in the Checksum byte is sent negated. If it does, the Server ends transmission of the data AFTER it sends the Stolen word, and it asserts the Stolen bit in the Checksum byte. The upstream Requestor always assumes that the words of a multi-word transfer are NOT Stolen until it encounters an asserted Stolen bit in the Checksum byte. When this occurs, the LAST word and only the last word received by the Requestor is assumed to be Stolen.

For byte write transfers, the Server presents the state of the Stolen bit in the downstream Checksum byte to the downstream T-Bus bit, T_AD<32>. For multi-word writes however, the state of ALL Stolen bits transported downstream is assumed by the Server to be "0". In this case, the Server will ignore the state of the Stolen bit in the downstream Checksum byte.

A.5.2.1.5 Error Reporting

Errors delivered by the Server (Requestor "Remote Error" Class) are transported by the Server to the upstream Requestor via the function response Switch message. Those errors may have one of two sources: they could originate from the Server itself, or they could be errors passed to the Server from a downstream Slave. The error codes due to the Server are shown in Figure A-54. Their definitions are shown in Figure A-55. Other remote slave errors are described in other system documents.

Figure A–54 Server remote error codes and definitions.

```
Server Error Codes:

7       0
|       |
PPPPPPba

b a   Server Error
= =   ==================
0 0   Downstream_Refused
0 1   Downstream_Write
1 0   Downstream_Late
1 1   Downstream_OTL

...where,

   P..P = Server_ConfigA.Error_Prefix<5..0>
```

Figure A–55 Server remote error definitions.

Downstream_Write A downstream write error was detected from a T–Bus Slave while the downstream Server was sourcing data. Because of the direction of the data bus, the Server cannot return the actual error code.

Downstream_OTL A downstream T–Bus Slave did not respond to the Server's request. Specifically, the Slave did not assert T_DRIVEN in the T–Bus cycle following the Server's T–Bus request.

Downstream_Late A downstream T–Bus slave responded with a LATE ERROR.

Downstream_Refused A downstream T–Bus slave responded with REFUSED–LOCKED when the Server thought itself to be locked.

A.5.2.1.6 Disabled Operation

The Server can be disabled via a number of bits in the Server_ConfigB register. These include: Ena_BIU and Ena_SRU. These bits reset the two major blocks of the Server.

WARNING

In normal operation, these bits SHOULD ALWAYS BE ASSERTED/NEGATED AT THE SAME TIME. Otherwise, erratic Server operation may result.

A.5.2.1.7 Configuration Registers

The Server has two general Configuration Registers, known as Server_ConfigA and Server_ConfigB, which are used to set miscellaneous parameters and enable/disable certain functions. The structure of Server_ConfigA is shown in Figure A–56. The bit definition of Server_ConfigA is shown in Figure A–57. This register contains mostly configuration bits that affect the run–time parameters of the Server. All bits are "high true" and are reset (low) upon system reset. The structure of Server_ConfigB is shown in Figure A–58. The bit definition of Server_ConfigB is shown in Figure A–59. This register contains mostly configuration bits that affect the run–time parameters of the Server. All bits are "high true" and are reset (low) upon system reset. Only the low eight bits of Server_ConfigB are devoted to read/write configuration. The upper 24 bits are the read–only Server_TestA register.

Figure A–56 Register definition — Server_ConfigA.

Register: Server_ConfigA<31..0>

BIT/FIELD	FUNCTION (read/write)
<31>	Ena_Wr_Head_Start
<30>	Ena_Byte_Head_Start
<29..28>	Multi_Head_Start[2]
<27..24>	RX_Init_CS[4]
<23..18>	Error_Prefix[6]
<17>	Ena_BIU
<16>	Ena_SRU
<15..12>	STU_Freed_Sync[4]
<11..8>	STU_Done_Sync[4]
<7..4>	BIU_Free_Sync[4]
<3..0>	BIU_Xfer_Sync[4]

Figure A–57 Bit definition — Server_ConfigA.

Ena_Wr_Head_Start Enables the Server to anticipate during write–type Function Responses (=1). Otherwise, anticipation will only occur for read–type Function Responses. (See: "Anticipation Support")

Ena_Byte_Head_Start Enables the Server to anticipate during byte–type Function Responses (=1). Otherwise, anticipation will not occur for byte–type Function Responses (=0). (See: "Anticipation Support")

Multi_Head_Start[2] Configures the Server for the desired Function Response Anticipation for all multi–word operations. (See: "Anticipation Support")

RX_Init_CS[4]	Configures the initial checksum for Initial Messages. NOTE: This register must contain the logical INVERSE of the initial checksum. (See: "Checksum Calculation")
Error_Prefix[6]	Configures the Prefix (T-Bus bits: D7-D2) of the Error code response for Server error. (See: "Error Handling")
Ena_BIU	Enables the by releasing its reset signal (= 1). Otherwise, the BIU will be held in reset (=0). (See: "Disabled Operation")
Ena_SRU	Enables the SRU by releasing its reset signal (=1). Otherwise, the SRU will be held in reset (=0). (See: "Disabled Operation")
STU_Freed_Sync[4]	Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives a "freed" signal from the Bus Interface Unit (BIU). This signal indicates that the BIU has acted on a previous "free" command from the SRU. (See: "Synchronization")
STU_Done_Sync[4]	Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives a "done" signal from the Bus Interface Unit (BIU). This is used to indicate completion of a Function Request. (See: "Synchronization")
BIU_Free_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "free" signal from the Switch Receive Unit (SRU). This is used to issue a FREE-LOCK. (See: "Synchronization")
BIU_Xfer_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "xfer" from the Switch Receive Unit (SRU). This is used to initiate a Function Request. (See: "Synchronization")

Figure A-58 Register definition — Server_ConfigB/Server_TestA.

```
Register: Server_ConfigB<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    ====================================
  <31..8>    Server_TestA (read only, see below)
   <7..6>    spare
     <5>     Dis_Frame
     <4>     Ena_SOC
     <3>     Dis_Check_Err
   <2..0>    SER_Slave_Num[3]
```

Figure A–59 Bit definition — Server_ConfigB.

Dis_Frame	Disables the SRU by forcing it to see the incoming Frame negated, regardless of its actual state (= 1). Otherwise, the SRU will see the actual incoming Frame (= 0). (See: "Disabled Operation")
Ena_SOC	Enables the SRU to recognize the start of a new connection (= 1). Otherwise, the SRU will ignore this event (= 0). (See: "Disabled Operation")
Dis_Check_Err	Disables the detection of checksum errors (= 1). Otherwise, the detection is enabled (= 0). (See: "Checksum Calculation)
SER_Slave_Num[3]	Configures the Slave number that the Server will place on the T_SOURCE<2..0> pins when it is making a T-Bus Function Request.

A.5.2.1.8 Test Registers

The Server contains a read-only test register whose read value should NEVER be interpreted during normal operation. A write to this register is acceptable but has no meaning. Figure A–60 shows the structure of that register which is used mostly for observing internal states. Figure A–61 shows the bit definition of SOME of the bits in the Server_TestA register.

Figure A-60 Register definition — Server_TestA.

```
Register: Server_TestA<31..0>

BIT/FIELD   FUNCTION (read-only)
=========   ====================
     <31>   <unused>
     <30>   SRU believes it is locked
     <29>   SRU refusing new connections
     <28>   Synchronized "Enable New SOC's"
     <27>   SRU "Should be Checksum"
     <26>   SRU Checksum OK signal
     <25>   SRU Anticipation Signal
     <24>   Checksum errors occurred
  <23..20>  <unused>
  <19..16>  Running Version of Rx Checksum

   <15..8>  Internal State of SRU FSM
        <15>    SRU has seen Reverse come and go
                   and has seen Frame go away.
                   Transition to 9, 10, or 13 will occur
        <14>    SRU has seen first Reverse and is waiting
                   for the end of the Reverse transmission
        <13>    SRU is waiting for lock to be FREE-LOCKed
        <12>    SRU is waiting for first Reverse
        <11>    SRU receiving Checksum byte
        <10>    SRU receiving a command
         <9>    SRU is idle
         <8>    Bad SOC seen (low true)
    <7..0>      Server_ConfigB (see previous figure)
```

Figure A-61 Bit definition — Server_TestA.

SRU believes it is locked The BIU will issue a FREE-LOCKS request if Frame is negated for more than one Switch Interval.

SRU refusing new connections Indicates that there is no active connection and that new connections will be refused (with Reject). The SRU IS currently and WILL be idle until re-enabled. (See: "Disabled Operation")

Synchronized Enable New SOC's The synchronized version of Server_ConfigB.4. The programmer should check this bit before assuming that the SRU will Reject or accept new connections. (See: "Disabled Operation")

SRU "Should be Checksum" Indicates that the Checksum should have arrived. This is used in conjunction with the "SRU Anticipation Signal" to determine if the SRU is properly anticipating the reception of the Checksum byte.

SRU "Checksum OK" Indicates to the BIU that the T–Bus operation should, in fact, take place.

SRU Anticipation Signal Indicates to the BIU that it should begin the T–Bus request. See SRU "Should be Checksum" above.

Checksum errors occurred Indicates that a checksum error did occur sometime in the past. This bit is negated whenever Server_ConfigB.4 is negated.

A.5.2.2 Switch Message Protocol

The Server fully generates and supports the Butterfly Switch protocol. That support is described below.

A.5.2.2.1 Upstream Message Components

Unlike the Requestor, the Server never has to create a message header with routing information because the return path to the upstream Requestor has already been established. The Server need only return a checksum with data and/or error code information. Figure A–62 shows a typical upstream Server message as a response to a word–read function request. The significance of the "E" and "S" bits are described in: "Stolen and Error Messages". The upstream message body for a write is always of the same format whether the function request was multi–word or non–multiple word. Figure A–63 shows a typical upstream Server message as a response to a word write Function Request. The significance of the "E" and "S" bits are described in: "Stolen and Error Messages".

Figure A-62 **Bit definition — upstream message body (read).**

```
7                                                 0
|                                                 |
D31  D30  D29  D28  D27  D26  D25  D24  (first sent)
D23  D22  D21  D20  D19  D18  D17  D16        |
D15  D14  D13  D12  D11  D10  D9   D8         |
D7   D6   D5   D4   D3   D2   D1   D0         |
                                              |
   <possible additional read words>           |
                                              V
0    0    E    S    CS3  CS2  CS1  CS0  (last sent)

...where,

D31..D8  = data information from T-Bus: T_AD<31..8>
D7..D0   = error code (E=1), T_AD<7..0> (E=0)
E        = Error bit
S        = Stolen bit
CS3..CS0 = message checksum
```

Figure A-63 **Bit definition — upstream message body (write).**

```
7                                                 0
|                                                 |
D7   D6   D5   D4   D3   D2   D1   D0   (first sent)
0    0    E    S    CS3  CS2  CS1  CS0  (last sent)

...where,

D7..D0   = error code (E=1), unknown (E=0)
E        = Error bit
CS3..CS0 = message checksum
```

A.5.2.2.2 Stolen and Error Messages

When the Upstream Read message has Stolen and/or Error bits asserted in the checksum, their presence modify the meaning of the message byte (or bytes) PRECEDING the checksum byte. In the case of an asserted (= 1) Stolen bit, the Server is indicating that ONLY the previous four bytes are stolen. This is consistent with what can happen on the T-Bus side of the Server. There, a T-Bus Slave may happen to return a Stolen data word which is not necessarily the last word of the read operation. The Server's BIU will continue to read any data "past" the Stolen word, but its STU will always END transmission of the Upstream Switch Message on the Stolen word — ignoring the rest. The consequence for the Upstream Requestor is that the "S" bit always modifies

the LAST word received. The "S" bit has no meaning for Upstream write messages and is ignored.

When the Error bit is asserted (= 1) during an Upstream Read message, the Server is indicating that the byte immediately PRECEDING the Checksum contains the Error Code and that any other bytes in the message are "garbage" data. The T–Bus protocol demands that all Slaves respond with "ERROR" during the FIRST word transfer and that an "ERROR" response ends the T–Bus transfer. Therefore, an Upstream Read Message with E = 1 will only contain one word of data. Assertion of the "E" bit has higher priority than assertion of the "S" bit, so they will never be asserted simultaneously in a given Upstream message.

Figure A–64 shows a summary of the effect of the "E" and "S" bits on an Upstream Message.

Figure A–64 Interpretation of checksum E and S bits.

```
E S   previous byte is...
= =   ====================
0 0   Data byte, previous word is NOT stolen (reads only)
0 1   Data byte, previous word is stolen (reads only)
1 0   Error Code (reads or writes)

Note: the value ES = 11 will never occur
```

A.5.2.2.3 Upstream Message Types

The previous discussions about message formats can be brought together to produce an enumeration of the possible Upstream Message types. This summary is shown in Figure A–65.

Figure A-65 Upstream message types.

TYPE	#WORDS	STOLEN or ERRORS	RETURN MSG FORMAT
write	any	none	XC
	"	any error	ZC
read	non-multiple	none	DDDDC
	"	either on word1	DDDEC
	two words	none	DDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	three words	none	DDDDDDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	"	stolen on word3	DDDDDDDDDDDDC
	four words	none	DDDDDDDDDDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	"	stolen on word3	DDDDDDDDDDDDC
	"	stolen on word4	DDDDDDDDDDDDDDDDC

NOTE: Frame is high for entire return message.

X = don't care
Z = always an Error Code
E = Error Code (Checksum bit 5 = 1)
  = Data Byte (Checksum bit 5 = 0)
C = Checksum Byte

A.5.2.2.4 Checksum Calculation

Checksum support for the Server is described in the "Requestor/Operation/ Checksum Calculation" section. The actual calculation performed by the Server is shown in Figure A-66. Figure A-66 shown the calculation for a single word read message. For read messages with more words, those bytes would be included in the same manner as the data bytes in the figure. For write messages, the data field would be missing entirely from the calculation and only the error byte would be included.

Figure A-66 Equation — message checksum (single-word read, see text).

```
CS<3> =  exor(D31,D27,D23,D19,D15,D11,D7,D3,0)

CS<2> =  exor(D30,D26,D22,D18,D14,D10,D6,D2,0)

CS<1> =  exor(D29,D25,D21,D17,D13,D9,D5,D1,E)

CS<0> =  exor(D28,D24,D20,D16,D12,D8,D4,D0,S)

...where,

      CS<3..0>  = message checksum
```

A.5.2.2.5 Rejects

A Reject is the assertion of Reverse for exactly one Switch Interval. Rejects are not, strictly speaking, messages; because the Switch data pins do not carry any known data. The Server produces a Reject (assertion of Reverse for only one Switch Interval) in either of three conditions: 1) An addressed downstream T-Bus slave is found to be locked during an Initial Switch Message, 2) The Server has been configured to reject all Downstream messages, or 3) The Server's SRU state machine is busy while trying to return to its "idle" state.

During the Initial Switch message, the targeted Downstream device may, in fact, be locked to a device other than the Server. The Server issues a Reject to indicate this fact to the Upstream Requestor. Once the Server has successfully locked some device, it is still possible for a Locked Message to attempt an access to device other than one to which the Server is currently locked. In this situation however, the Server does NOT issue a Reject. Instead, it sends an error response to the upstream Requestor. (See: "Error Reporting")

The Server can also be configured — via the Requestor_ConfigA.Ena_SOC bit — to issue a reject on any new incoming message. This is a synchronized enable such that it can be asserted/negated at any time. The Server will continue to process any pending transactions but will prevent any new ones. Thus, the Server can be "gracefully" removed from the Switch interface.

Whenever the Server is in any state other than its "idle" state (locked or unlocked), it will refuse new attempts at a connection (Frame high preceded by Frame low for for at least two Switch Intervals) by issuing a Reject. There are many instances when a new connection attempt would indicate an Switch protocol violation, and thus a Reject issued by the Server would make little difference. However, there are some situations where the Server would correctly issue a Reject while it is off processing some event. For instance, a drop-lock would cause the Server to begin issuing a FREE-LOCK on the T-Bus. If new

downstream Switch message attempted to access the Server before it finished the transaction, the Server would issue a Reject.

A.5.2.3 T-Bus Interface

The Server supports the standard T-Bus protocol with some small limitations. For one, the Server does NOT support unaligned transfers which fall across word (32-bit) boundaries. The Server also expects to see an ERROR response as the FIRST response from a T-Bus Slave if that slave is going to issue any ERROR's. If the Slave cannot issue an ERROR in the cycle immediately following the T-Bus request (i.e., the first response cycle), it must assert T_NSPAUSE_xxx until it decides if the request is an error or not.

A.5.2.4 LCON Interface

The LCON is a the physical and logical link between the SIGA Server and the "input" port of the Switch Gate Array (SGA). In other words, for the SIGA, the LCON interface is the logical Switch interface. The LCON provides the Server with: 1) level conversion to and from the ECL levels of the SGA and 2) reclocking of data, Frame, Reverse to and from the SGA.

Figure A-67 shows the Server's LCON (Switch) Interface Pins.

Figure A-67 Server LCON (Switch) interface pins.

PIN NAME	TYPE	FUNCTION
S_DATA<7..0>	bidirectional	Server-LCON data bus
S_FRAME	input	Frame input from Switch
S_REVERSE	output	Reverse output to Switch
S_NENA_BACK	input	LCON TTL driver enable

A.5.2.4.1 Data Bus Enable Control

The Server controls the enables of both its own output drivers and the LCON's output drivers to the SIGA-LCON data interface -- S_DATA<7..0>. It does so in a manner complementary to the Requestor's method (see "Requestor/Operation/LCON Interface/Data Bus Enable Control"). The Server uses the same concept of "Talk" and "Listen" mode as the Requestor.

The Server is considered "quiescent" when it is not transmitting messages and not waiting for any replies. When quiescent, the Server is in Listen Mode. The Server tries to stay in Listen Mode whenever possible, making the transition to Talk only for the absolute minimum time necessary. This situation is the

mirror image to the Requestor. It is always in Talk Mode when quiescent and tries to stay in Listen mode for as little time as possible.

When the Server receives the checksum of a downstream message, it transitions to Talk mode — via the "dead" state. It remains in Talk mode until the T-Bus transaction is complete and the upstream return message has been sent. Once the upstream checksum has been sent, the Server transitions immediately into Listen mode (no contention is possible — as with the Requestor).

A.5.3 TCS Control Unit (TCU)

The basic purpose of the TCS Unit (TCU) is to allow the Test and Control System (TCS) Slave Processor access to the T-Bus interface — in essence, to act as a protocol converter. Normally, this involves the TCU acting like a T-Bus Master — performing reads and writes. However, the TCU is flexible enough so that it can also generate or "spoof" responses for any T-Bus Master or Slave. A "spoofed" response essentially involves issuing a response on the T-Bus in the absence of a request. This can used, for instance, to free up an observing T-Bus Master who's locked Slave has failed. In this case, the TCU can "make believe" that IT is the "failed" slave.

A secondary function of the TCU is to allow the TCS Slave Processor DIRECT access to the CSU Map, rather than forcing it to make an access via the T-Bus interface. This is useful for fault tolerance and bootstrapping.

A.5.3.1 I/O Description

The TCU interface is composed of four pins on the SIGA. The pins and their basic functions are shown in Figure A-68.

Figure A-68 TCU I/O signal description.

C_CLK	The data shift clock. Data is shifted into the SIGA on each rising edge of C_CLK. Data is shifted out of the SIGA on each falling edge of C_CLK.
C_IN	TCS data into the SIGA.
C_OUT	TCS data out of the SIGA. This is a tri-state signal which is driven when C_NEXECUTE is asserted (=0).

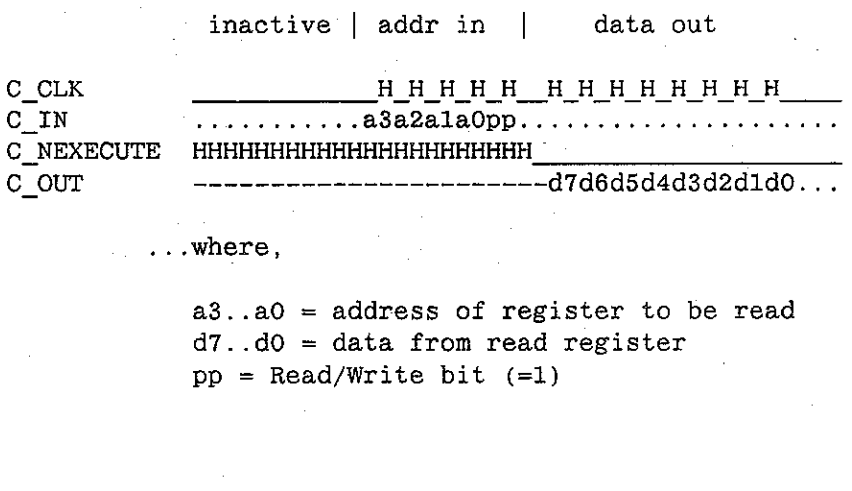
C_NEXECUTE — Asynchronously initiates execution of a command (=0) and enables C_OUT. In addition, negating C_NEXECUTE (=1) resets the TCU interface.

A.5.3.2 Read/Write Operation

The TCU contains 16 addressable registers — each 8 bits wide. The TCS Slave can read any register by clocking-in the required address (4 bits), a Read/Write bit (=1), and assert C_NEXECUTE (=0). A read operation is illustrated in Figure A-69. Some additional details for Read operations — not apparent from Figure A-69 — are now discussed.

1. C_IN data is clocked-in on the positive edge of C_CLK and C_OUT data is clocked-out on the negative edge of C_CLK.
2. Data can be clocked in or out at any desired rate, provided that the AC specifications of the C_CLK pin are not violated. The duty cycle of C_CLK is variable within the AC specifications. There is no MAXIMUM high (=1) or low (=0) time for C_CLK.
3. Reads are non-destructive and can be aborted at any time.
4. C_NEXECUTE is not synchronized with C_CLK and can be asserted at any time after the address and Read/Write bit has been clocked-in.
5. The C_OUT pin may be used to monitor, in real time, the value of a particular bit. This is done by reading the appropriate register, shifting out the desired bit using C_CLK, and then holding C_CLK steady. C_CLK can be held in either state (1 or 0) as long as it does not make another positive transition.
6. Extra data bits preceding the negative transition of C_NEXECUTE, are ignored.

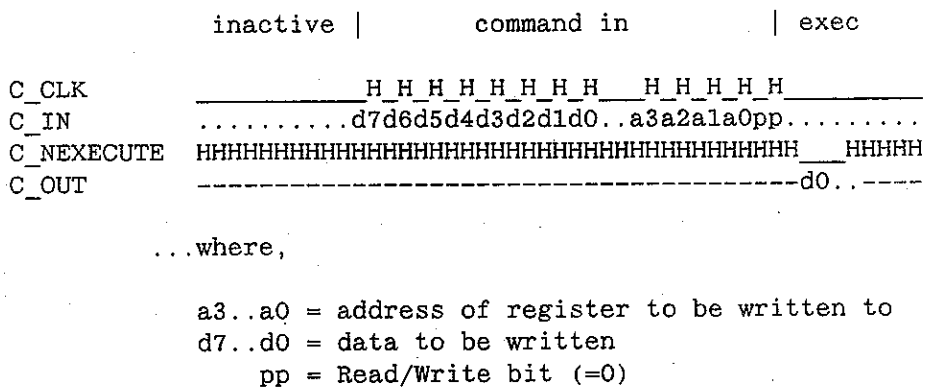
Figure A-69 Timing — TCU read operation.



A write operation is performed by clocking-in four bits of data, 4 bits of address, a Read/Write bit (=0), and then asserting C_NEXECUTE (=0). This is illustrated in Figure A-70. Some additional details for Write operations — not apparent from Figure A-70 — are now discussed.

1. C_IN data is clocked-in on the positive edge of C_CLK and C_OUT data is clocked-out on the negative edge of C_CLK.
2. Data can be clocked in or out at any desired rate, provided that the AC specifications of the C_CLK pin are not violated. The duty cycle of C_CLK is variable within the AC specifications. There is no MAXIMUM high (=1) or low (=0) time for C_CLK.
3. Reads are non-destructive and can be aborted at any time. Reads of the TBUS_Response register can be aborted as well. However, if this is done AFTER C_NEXECUTE has been asserted, the T-Bus operation may be aborted.
4. C_NEXECUTE is not synchronized with C_CLK and can be asserted at any time after the address and Read/Write bit has been clocked-in.
5. C_NEXECUTE need only be asserted for a short moment to begin execution of the command. The minimum low time is described in "AC Specifications".
6. Extra data bits preceding the negative transition of C_NEXECUTE, are ignored.

Figure A-70 Timing — TCU write operation.



A.5.3.3 Register Map

The register map for the 16 TCU registers is shown in Figure A-71.

Figure A–71 TCU register map.

a3..a0	DESCRIPTION
0	T_AD<7..0> (data)
1	T_AD<15..8> (data)
2	T_AD<23..16> (data)
3	T_AD<31..24> (data)
4	T_AD<7..0> (addr)
5	T_AD<15..8> (addr)
6	T_AD<23..16> (addr)
7	T_AD<31..24> (addr)
8	TBUS_Response
9	TBUS_Command
A	TBUS_Command_Modifier_0
B	TBUS_Command_Modifier_1
C	CSU Map<7..0>
D	CSU Map<8>
E	unused
F	unused

Referring to Figure A–71, registers 0 through 3 are special registers. For write operations, their contents are loaded, via the TCU interface, with the data to be written TO some T–Bus slave. For read operations, their contents are replaced with the data read FROM some T–Bus slave. Registers 4 through 7 are loaded ONLY by the TCU interface. The contents of these registers are placed on the T–Bus during the address phase of a T–Bus request.

The registers at address "C" and "D" are used to initialize CSU_Map<8..0>. Register "D" — bit "0", corresponds to CSU_Map<8>. Bits 7 through 1 of register "D" are unused. Figure A–72 shows the definition of the TBUS Response and Command Registers. Referring to Figure A–72, the TBUS_Response register is a read–only register which is valid after a T–Bus operation has been executed. The "Done" bit is monitored after a T–Bus command is initiated by the TCU. When asserted (=1), it indicates that the operation is complete. See the "T–Bus Operations" section for more detail. The "Drive_AD" bit indicates that the T_AD Bus was driven during a T–Bus access (=1). The remaining bits in the TBUS_Response register are the "responses" received from the T–Bus operation.

Figure A-72 **Register definitions — TBUS Response and Command registers.**

```
Register: TBUS_Response<7..0> (read only)

BIT/FIELD  FUNCTION (read only)
=========  ======================
      <7>  Done
      <6>  Drive_AD
      <5>  T_DRIVEN
      <4>  M_PARITY
      <3>  T_AD<32>
   <2..0>  T_RR<2..0>

Register: TBUS_Command<7..0>

BIT/FIELD  FUNCTION
=========  ======================
   <7..6>  output T_AD<33..32> (addr)
   <5..3>  output T_SIZE<2..0>
   <2..0>  output T_RR<2..0>

Register: TBUS_Command_Modifier_0<7..0>

BIT/FIELD  FUNCTION
=========  ======================
   <7..4>  unused
      <3>  Response
      <2>  output T_AD<32> (data)
   <1..0>  output T_PATH<1..0>

Register: TBUS_Command_Modifier_1<7..0>

BIT/FIELD  FUNCTION
=========  ======================
      <7>  output T_SYNC
   <6..5>  output T_PRIORITY<1..0>
   <4..3>  output T_LOCKOP<1..0>
   <2..0>  output T_SOURCE<2..0>
```

The TBUS_Command and BUS_Command_Modifier_1 registers contains the indicated fields to be placed on the T-Bus during the address phase of any operation. The TBUS_Command_Modifier_0 register outputs the "T_PATH" field during the address phase of any operation and the T_AD<32> bit during the data phase of a write operation.

The "Response" field of the TBUS_Command_Modifier_0 register, has a special function. When asserted (=1), the TCU will place a "0" on the T_REQUEST and drive the T-Bus FOR A SINGLE CYCLE with the register settings intended for the address phase of a T-Bus cycle. This is used for

"spoofing" a T-Bus response. When the "Response" field is a "0", the TCU makes a normal T-Bus Request with T_REQUEST asserted (=1).

A.5.3.4 Normal T-Bus Operations

The TCU can be used to read and write, one to four bytes. Multi-word transfers are not allowed. The TCU can also OPEN and FREE locks although this is not recommended because the TCS Slave interface is relatively slow.

A read or write operation is set up by loading the desired data into the registers. The operation is actually initiated by a read of the TBUS_Response register. Since the MSB of this register is the "Done" bit, C_CLK should be disabled just after C_NEXECUTE is asserted (=0). This allows asynchronous monitoring of the "Done" bit. Terminating the read by negating (=1) C_NEXECUTE will abort the T-Bus request.

The TCU will retry after becoming REFUSED but will ignore a REFUSED LOCKED. In other words, the TCU will not become an "observing master".

A.5.3.5 Special T-Bus Operations

The TCU can FREE-LOCKS for any T-Bus master by specifying the correct T_SOURCE field value and performing a write operation. The TCU can also spoof any one-cycle response of a Slave by asserting the "Response" bit in the TBUS_Command_Modifier_0 register. For instance, it can issue a COMPLETED or ERROR for some Slave that is known to be faulty.

A.5.3.6 CSU Map Initialization

The CSU_Map is a 9-bit quantity which maps the SIGA CSU into a desired 8K page. This quantity is initialized by the TCU and is one of the first things that must be done to the SIGA upon power-up. If the CSU_Map is not initialized, it defaults to the setting of all 1's.

A.5.4 Configuration/Status Unit

The Configuration Status Unit (CSU) is the T-Bus Slave interface which allows any T-Bus master read and write access to the SIGA's configuration and status registers.

A.5.4.1 Normal Register Accesses

The CSU is limited in its support of the T-Bus protocol and is NOT optimized for minimum wait states (Slave pause cycles). The CSU will respond to a T-Bus query ONLY when T-Bus bits T_AD<24..16> match

CSU_Map<8..0>. The CSU_Map is initialized by the TCU (See: TCS Control Unit/CSU Map Initialization).

In the cycle following a request to the CSU, the CSU will either respond with an ERROR or go on to complete the requested function. Figure A–73 shows the TCU responding with an ERROR.

Figure A–73 Timing — CSU ERROR access.

```
T-Bus cycle #   |  0   |   1  |   2  |
T-Bus cycle     | req  | resp |  end |

T_NSPAUSE_SIGA HHHHHHHHHH_______HHHHH
T_RR<3..0>               xxxxxxxxeeeee

        ...where,

        x..x = invalid response
        e..e = ERROR response
```

Note from Figure A–73, that T_NSPAUSE_SIGA is asserted for only one cycle. The ERROR response is triggered by exactly two conditions: 1) T_SIZE<2> = 1, or 2) T_LOCKOP<1> = 1. This means that the CSU will not support multi-word writes or locking. A normal read and write operation are shown in Figure A–74. Note from Figure A–74 that T_AD<32> is always a "0" on a read and a "don't care" on a write. In addition, during write operations, data is set up to the configuration latches during cycle #1, written to them during cycle #2, and held at the configuration latches during cycle #3.

Figure A-74 Timing — normal CSU read/write.

```
T-Bus cycle #          |  0   |   1   |   2   |   3   |
T-Bus cycle            | req  | resp  | resp  |  end  |

T_NSPAUSE_SIGA         HHHHHHHHHH_____________HHHH
T_RR<3..0>             ??????????xxxxxxxxxxxxxcccc

T_AD<32>     (read)    ???????????XXXXXXXXXXXXX____
T_AD<31..0>  (read)    ???????????XXXXXXXXXXXXXRRRR

T_AD<32>     (write)   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
T_AD<31..0>  (write)   ?????????WWWWWWWWWWWWWWWWWWW

                ...where,

                x..x = invalid response
                c..c = COMPLETED response
                X..X = invalid data
                W..W = valid write data
```

A.5.4.2 Synchronized Accesses

Certain accesses to the CSU must be synchronized to the One Microsecond Pulse (OMSP). These include: 1) read/writes of the Real Time Clock, and 2) writes to the TONI_A or TONI_B registers. This mechanism is described in: "Requestor/Operation/RTC and Related Functions". Essentially, all this means to the CSU timing diagram in Figure A-74, is that cycle #2 is repeated until the synchronization pulse is received from the RTC or TONI_A/B controller.

A.5.4.3 Interleaver Loader

The CSU provides support for loading and reading the Interleaver Modulus Ram through the use of two special registers: Interleave_Address and Interleave_Data; and an external pin to the SIGA: I_NACCESS. Reads and writes to both the Interleave_Address and Interleave_Data registers are different than accesses to other configuration/status registers in the SIGA. The structure of the Interleaver_Address register is shown in Figure A-75. The structure of the Interleaver_Data register is shown in Figure A-76. As seen in Figure A-76, read/write access to the I_D register does not involve any data transfer within the SIGA.

Figure A–75 Register definition – Interleave_Address.

```
Register: Interleave_Address

31.............................0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

...where,

A..A = interleaver address
```

Figure A–76 Register definition – Interleave_Data.

```
Register: Interleave_Data

31.............................0
-------------------------------  (read)
-------------------------------  (write)
```

A.5.4.3.1 Address Register Access

When a T–Bus master reads the Address_Register, the CSU immediately responds with a Slave Pause cycle by asserting (=0) the T_NSPAUSE_SIGA pin, as it does with all other accesses. However, in the following cycle, the CSU also asserts the I_NACCESS pin and places the contents of the Interleave_Address register on the T–Bus. The CSU then waits for exactly seven (7) T–Bus cycles in this state. The mapping of the I_A register to the T–Bus during this "wait" state is shown in Figure A–77, part (a). In the cycle following the wait period, the CSU then negates (=1) both T_NSPAUSE_SIGA and I_NACCESS, and maps the I_A to the T–Bus as shown in Figure A–77, part (b).

Figure A–77 Interleave_Address register to T–Bus mapping.

```
T_AD<33>    = Interleave_Address<1>
T_AD<32>    = Interleave_Address<0>
T_AD<31..0> = Interleaver_Address<31..0>

          (a) wait (cycle 3 - 9)

T_AD<33>    = 0
T_AD<32>    = 0
T_AD<31..0> = Interleaver_Address<31..0>

          (b) end (cycle 10)
```

The timing for writes to the I_A register is exactly the same as for reads. The actual timing for Interleave_Address register read/write access is shown in Figure A–78.

Figure A–78 Timing – CSU Interleave_Address register read/write access.

```
T-Bus cycle # |  0   |   1  |  2   |...|  10  |  11  |
T-Bus cycle   | req  | resp | wait |...|  end |  ?   |

T_NSPAUSE_SIGA HHHHHHHHH_____________...____HHHHHHHHHHH
T_RR<3..0>                            ...  ccccccc

I_NACCESS   (read) HHHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (read)    ------???aaaaa...aaabbbb

I_NACCESS  (write) HHHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (write)     ddddddddddddd...dddd???

        ...where,

        c..c = COMPLETED response
        a..a = "wait" type read of I_A (bit swapping)
        b..b = "end" type read of I_A (bit masking)
        d..d = data written TO the I_A register
```

A.5.4.3.2 Data Register Access

The Interleave_Data access is EXACTLY the same as the Interleave_Address access EXCEPT for two key features: (1) during writes, no data is actually stored in the SIGA, and (2) during reads, the SIGA does NOT drive the T_AD<33..0> field. During this time, logic external to the SIGA will manipulate the Modulus Ram, and the SIGA is basically being used as an address decoder and T-Bus control signal driver. The actual timing for Interleave_Data register read/write access is shown in Figure A–79. Note from

Figure A–79 that the CSU temporarily drives the T–Bus during cycle #1. The data is unknown.

Figure A–79 Timing — CSU Interleave_Data read/write access.

```
T-Bus cycle # |  0   |   1   |  2   |...|  10   |  11   |
T-Bus cycle   | req  | resp  | wait |...|  end  |  ?    |

T_NSPAUSE_SIGA HHHHHHHHH_____________...____HHHHHHHHHHH
T_RR<3..0>                           ...   ccccc

I_NACCESS (read) HHHHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (read) xxxxxaaaaaaa---bb...bbbbbbb

I_NACCESS  (write) HHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (write)   ??????????xxx...xxxxxxxxx

        ...where,

        c..c = COMPLETED response
        a..a = unknown data driven by CSU (only for one cycle)
        b..b = data from/to Interleaver (not driven by SIGA)
```

A.5.4.4 Debug Support

The CSU supports "freezing" a CSU read or write for debugging purposes. This is accomplished by initiating a normal T–Bus access (see Figure A–74, "Timing — normal CSU read/write") and asserting (=0) and holding the pin, M_NDEBUG, during cycle #1 and #2. This will cause the CSU to repeat cycle #2 indefinitely until M_NDEBUG is negated (=1). When this occurs, the CSU will continue with cycle #3 as normal.

For read cycles this means that T_AD<31..0> will have the real–time state of any register being read. By reading a test register, for example, the state machine of the STU can be observed while it sends a message.

For write cycles, the use is somewhat limited. It simply means that T_AD<31..0> can be manipulated in real time from the master (or logic analyzer). Since during cycle #2 the configuration latches are transparent, so that any external manipulation will be seen internally in real time.

A.5.4.5 Restriction Summary

The following restrictions apply to CSU operation:

1. The CSU will flag as an ERROR any multi-word access or an OPEN or MAINTAIN. Therefore, the CSU does not support these operations. However, byte masking on writes IS supported.
2. The CSU will NOT check for unaligned transfers. It is illegal to request an operation with an unaligned address.
3. Synchronized Accesses rely on the presence of R_CLK to complete. If R_CLK is non-existent, the CSU will pause the T-Bus Master indefinitely. The only way to release the pause would be to assert the M_NRESET pin.
4. The Stolen bit (T_AD<32>) is not supported on either reads or writes.

A.6 Programming Model

This section provides a memory map of the previously defined SIGA registers, as well as a compilation of all SIGA Error Codes.

A.6.1 Memory Map

Figure A-80 shows the memory map of the various registers. Note from Figure A-80 that the "M" field is programmable via the CNU_Config.CSU_Map bits.

Figure A-80 SIGA memory map.

```
T_AD<24..0>                REGISTER
=====================      ======================================
  15  12         2 10
  |   |         | ||
M 000 XXXXXXXX000 bb  TONIA_Config
                001 bb  Time_Of_Next_InterruptA

M 001 XXXXXXXX000 bb  TONIB_Config
                001 bb  Time_Of_Next_InterruptB

M 100 0XXXXXXX000 bb  Protocol_Timer_Config | Message_Class
                001 bb  Transmit_Time_Config
                010 bb  Priority_Time_Config
                011 bb  Requestor_ConfigA
                100 bb  Requestor_ConfigB
                101 bb  Requestor_TestA
                110 bb  Real_Time_Clock (hi/lo)
                111 bb  <reserved>

      1XXXXXXX000 bb  Server_ConfigA
                001 bb  Server_ConfigB/Server_TestA

M 10I 0XXXXXXXXXX xx  Interleave_Address_Reg
      1XXXXXXXXXX xx  Interleave_Data_Reg

        ...where,

           M  = (T_AD<24..16> = CNU_Config.CSU_Map<8..0>)
           bb =  00  byte 0 <31..24>
                 01  byte 1 <23..16>
                 10  byte 2 <15..8>
                 11  byte 3 <7..0>
           xx =  no byte addressing capability
```

A.6.2 Error Code Summary

Figure A-81 presents an Error Code summary for the SIGA. Figure A-82 summarizes the Error Code definitions.

Figure A-81 Error code summary.

```
     Requestor/CSU Error Codes:

     7      0
     |      |
     PPPPdcba

     d c b a  Requestor/ CSU Error
     = = = =  ====================
     0 0 0 0  Maintain_Absent (2a)
     0 0 0 1  Maintain_Present (2b)
     0 0 1 0  Stolen_Verify (1)
     0 0 1 1  Lock_Address (2)
     0 1 0 0  Wait_TO (3a)
     0 1 0 1  Idle_TO (3b)
     0 1 1 0  Rej_Abort (4)
     0 1 1 1  Rej_TO (5)
     1 0 0 0  Reverse (6)
     1 0 0 1  Check (7)
     1 0 1 0  Misc. CSU Error

...where,

P..P = Requestor_ConfigA.Error_Prefix<3..0>
Priority is from highest (1) to lowest (8).
Within a given priority, errors are mutually
exclusive (i.e., 4a,b...).

     Server Error Codes:

     7      0
     |      |
     PPPPPPba

     b a  Server Error
     = =  ==================
     0 0  Downstream_Refused
     0 1  Downstream_Write
     1 0  Downstream_Late
     1 1  Downstream_OTL

...where,

P..P = Server_ConfigA.Error_Prefix<5..0>
```

Figure A-82 Error code definition summary.

Maintain_Absent	An NORMAL was issued to the Requestor during its idle state and it was locked.
Maintain_Present	A MAINTAIN was issued to the Requestor during its idle state and it was NOT locked.
Lock_Address	A Function Request was made to a locked Requestor during its idle state with a node address was different than that which opened the locked sequence.
Wait_TO	The Switch Transmit Connection Timer overflowed while the Requestor was waiting for a Function Response.
Idle_TO	The Switch Transmit Connection Timer overflowed while the Requestor was in its idle state.
Rej_Abort	The Switch Transmit Reject Timer was forced into overflow by the REJ_ABORT input pin.
Rej_TO	The Switch Transmit Reject Timer overflowed while the Requestor was attempting to open a connection.
Reverse	The Requestor detected an incorrect polarity of the Reverse signal during a Function Response.
Check	The Requestor detected an incorrect Checksum during a Function Response.
CSU Error	An error was made accessing the CSU. It could be one or both of the of the following: 1) An OPEN lock was requested or 2) A Multi-word transfer was requested.
Downstream_Write	A downstream write error was detected while the downstream Server was sourcing data.
Downstream_OTL	A downstream T-Bus slave did not respond to the Server's request.
Downstream_Late	A downstream T-Bus slave responded with a LATE ERROR.

Downstream_Refused A downstream T-Bus slave responded with REFUSED-LOCKED when the Server thought itself locked.

A.7 Special Topics

This section describes some of the special topics relating to SIGA operation.

A.7.1 Initialization States

The external Reset signal is resynchronized by the SIGA for use by all synchronous logic clocked by all three major clocks (R_CLK, S_CLK and T_CLK). When Reset is applied and then released, all internal storage logic that needs to be initialized, will be so initialized. The SIGA will now be in its first initialization state, known as the Quiescent State.

In this state, the SIGA Switch and T-Bus interfaces are partially disabled. The Server's Switch interface responds to any assertions of downstream Frame with Rejects. The Requestor's Switch interface ignores any assertions of the upstream Reverse. The Server's T-Bus interface makes no T-Bus requests and the Requestor's T-Bus interface responds to any remote function requests with a REFUSED. The Configuration/Status Unit and the TCU, however, are operational. Normally, in the Quiescent state, the TCU will initialize the CSU's mapping logic via the CNU_Config.CSU_Map<8..0> register. Once the Control Net initializes the CSU_Map, any T-Bus master can then initialize the SIGA registers via the CSU.

Once this is accomplished, the SIGA is in the Operational State. The Operational State is the normal operational mode of the SIGA.

A.7.2 Synchronization

Because of the use of multiple clocks, the SIGA design inherently requires the use of synchronizers to implement handshaking across clock boundaries. Some of these synchronizers are in non-critical paths and are thus implemented in the most cost effective manner. In particular, these synchronizers are of the "large uncertainty, fixed delay" variety. This means that there delay is not programmable and that "input-to-output" delay is not constant over changes in input. These are used in areas such as: 1) Between the external reset pin, M_NRESET, and the internal reset destinations, 2) Between the TCU negation of C_NEXECUTE and the T_Bus access. These synchronizers are designed to provide a MINIMUM of 100 ns settling time (T_CLK < = 22 MHz, R_CLK,S_CLK < = 45 MHz).

The other variety of synchronizers — used in critical path applications — are the "variable delay, zero uncertainty" synchronizers. These are used between

the T-Bus and Switch interfaces along the Function request/response paths. These are the synchronizers which have four bits of configuration to control the settling time. Figure A-83 shows the various settings for ALL variable-delay synchronizers. Figure A-83 should be used in combination with the clock period of the logic RECEIVING the synchronizer data to determine the actual settling time. For instance, if a 100 ns settling time on the positive edge is desired for the STU Synchronizer, the register: Requestor_ConfigA.STU_Sync<3..0>, should be set to a "0110". This is because assuming R_CLK = 40 MHz (25 ns period), the synchronizer will require four clock periods — at 25 ns apiece — to obtain the total of 100 ns.

On the other hand, the BIU Synchronizer control, set by Requestor_ConfigA.BIU_Sync<3..0>, would need a setting of "0010" to obtain the same settling time. Here, of course, the clock period is twice as long as the STU Synchronizer so the number of synchronizer clock delays is half.

Figure A-83 Variable-delay synchronizer settings.

3210	# CLOCK DELAYS	TRANSFER EDGE
0000	1	Positive
0001	1	Negative
0010	2	Positive
0011	2	Negative
0100	3	Positive
0101	3	Negative
0110	4	Positive
0111	4	Negative
1000	5	Positive
1001	5	Negative
1010	ILLEGAL	-
1011	ILLEGAL	-
1100	ILLEGAL	-
1101	ILLEGAL	-
1110	ILLEGAL	-
1111	ILLEGAL	-

NOTE

Currently, it is recommended that only the POSITIVE transfer edge be used for any setting.

NOTE

It has been determined that a settling time of 100 ns is a reasonable goal for the variable-delay synchronizers.

A.8 Pin Description and Pinout

The next page begins a pin description of the SIGA.

PIN NAME	TYPE	DESCRIPTION
C_CLK	IN	TCU input clock
C_IN	IN	TCU data input
C_NEXECUTE	IN	TCU execute handshake input
C_OUT	OUT	TCU data output
F_AD<24..16>	IN	T-Bus input for T_AD<24..16>
F_PATH<1..0>	IN	T-Bus input for T_PATH<1..0>
F_REQUEST	IN	T-Bus input for T_REQUEST
F_RR<2..0>	IN	T-Bus input for T_RR<2..0>
F_SIZE_2	IN	T-Bus input for T_SIZE_2
F_SOURCE<2..0>	IN	T-Bus input for T_SOURCE<2..0>
I_INTERLEAVED	IN	=0: do NOT use I_MOD<8..0> for route address =1: use I_MOD<8..0> for route address
I_MOD<8..0>	IN	Interleaver data input
I_NACCESS	OUT	=0: CSU Interleaver loader is active =1: CSU Interleaver loader is NOT active
M_NDEBUG	IN	=0: Debug mode during CSU access (TEST ONLY) =1: Do NOT enter debug mode (NORMAL MODE)
M_NFLOAT	IN	=0: Tri-state all outputs (TEST ONLY) =1: Normal output operation (NORMAL MODE)
M_NRESET	IN	=0: Hardware reset to SIGA =1: Normal operational mode
M_NSELECT	IN	=0: Select CSU, attach to T_PATH<1/0> =1: Do NOT select CSU
M_PARA	OUT	Parametric nand tree output (TEST ONLY)
M_PARITY	IN	=0: No parity error during T-Bus response =1: Parity error during T-Bus response
M_REJ_ABORT	IN	=0: Do NOT abort Switch retries =1: Abort Switch retries
M_SIXTY_FIVE	IN	=0: 65 ms pulse NOT active =1: 65 ms pulse active (one R_CLK period)
M_TONIA_INT	OUT	=0: TONIA interrupt is active =1: TONIA interrupt is NOT active
M_TONIB_INT	OUT	=0: TONIB interrupt is active =1: TONIB interrupt is NOT active
R_CLK	IN	Requestor clock input
R_DATA<7..0>	BID	Requestor Switch data interface
R_FRAME	OUT	Requestor Switch Frame output
R_NENA_BACK	OUT	=0: Enable LCON to drive R_DATA<7..0> =1: Disable LCON from driving R_DATA<7..0>
R_REVERSE	IN	Requestor Switch Reverse Input
S_CLK	IN	Server clock input
S_DATA<7..0>	BID	Server Switch data interface
S_FRAME	IN	Server Switch Frame input
S_NENA_BACK	OUT	=0: Disable LCON from driving S_DATA<7..0> =1: Enable LCON to drive S_DATA<7..0>
S_REVERSE	OUT	Server Switch Reverse Input
T_AD<33..25>	BID	T-Bus input/output for T_AD<33..25>
T_AD<24..16>	OUT	T-Bus output for T_AD<24..16>
T_AD<15..0>	BID	T-Bus input/output for T_AD<15..0>
T_CLK	IN	T-Bus input clock
T_DRIVEN	OUT	T-Bus output for T_DRIVEN

T_ENA_HOLD	IN	=0: Disable T-Bus input latches
		=1: Enable T-Bus input latches
T_ENA_TDAT.2	OUT	=0: Enable T_AD<33..0> drivers
		=1: Disable T_AD<33..0> drivers
T_ENA_TDAT<1..0>	OUT	=0: Disable T_AD<33..0> drivers
		=1: Enable T_AD<33..0> drivers
T_ENA_TRANS.1	OUT	=0: Enable transaction T-Bus field
		=1: Disable transaction T-Bus field
T_ENA_TRANS.0	OUT	=0: Disable transaction T-Bus field
		=1: Enable transaction T-Bus field
T_LOCKOP<1..0>	BID	T-Bus input/output for T_LOCKOP<1..0>
T_MPAUSE	OUT	T-Bus output for T_MPAUSE
T_NBGRANT_SIGM	IN	=0: SIGA Master granted next T-Bus
		=1: SIGA Master NOT granted next T-Bus
T_NBGRANT_SIGS	IN	=0: SIGA Slave granted next T-Bus
		=1: SIGA Slave NOT granted next T-Bus
T_NBREQ_SIGM	OUT	=0: SIGA Master is requesting T-Bus
		=1: SIGA Master is NOT requesting T-Bus
T_NBREQ_SIGS	OUT	=0: SIGA Slave is requesting T-Bus
		=1: SIGA Slave is NOT requesting T-Bus
T_NDRIVEN_SIGA	OUT	=0: SIGA is driving T-Bus next cycle
		=1: SIGA is NOT driving T-Bus next cycle
T_NSPAUSE_SIGA	OUT	=0: SIGA is pausing T-Bus next cycle
		=1: SIGA is NOT pausing T-Bus next cycle
T_PATH<1..0>	OUT	T-Bus output for T_PATH<1..0>
T_PRIORITY<1..0>	BID	T-Bus input/output for T_PRIORITY<1..0>
T_REQUEST	OUT	T-Bus output for T_REQUEST
T_RR<2..0>	OUT	T-Bus output for T_RR<2..0>
T_SIZE.2	OUT	T-Bus output for T_SIZE.2
T_SIZE<1..0>	BID	T-Bus input/output for T_SIZE<1..0>
T_SOURCE<2..0>	OUT	T-Bus output for T_SOURCE<2..0>
T_SPAUSE	OUT	T-Bus output for T_SPAUSE
T_SYNC	BID	T-Bus input/output for T_SYNC

The following page shows the SIGA pinout sorted by pin function.

SIGA PINOUT SORTED BY PIN FUNCTION

R15	C_CLK	R06	R_DATA.6	B12	T_DRIVEN
T14	C_IN	P06	R_DATA.7	C12	T_ENA_HOLD
R14	C_NEXECUTE	R05	R_FRAME	C03	T_ENA_TDAT.0
P13	C_OUT	T05	R_NENA_BACK	B03	T_ENA_TDAT.1
B09	F_AD.16	T04	R_REVERSE	A03	T_ENA_TDAT.2
C09	F_AD.17	T13	S_CLK	C14	T_ENA_TRANS.0
A10	F_AD.18	T12	S_DATA.0	C15	T_ENA_TRANS.1
B10	F_AD.19	P11	S_DATA.1	D01	T_LOCKOP.0
C10	F_AD.20	R11	S_DATA.2	D02	T_LOCKOP.1
A11	F_AD.21	T11	S_DATA.3	E01	T_MPAUSE
B11	F_AD.22	P10	S_DATA.4	A06	T_NBGRANT_SIGM
C11	F_AD.23	R10	S_DATA.5	C07	T_NBGRANT_SIGS
A12	F_AD.24	T10	S_DATA.6	C05	T_NBREQ_SIGM
A05	F_CLK	P09	S_DATA.7	B05	T_NBREQ_SIGS
A07	F_PATH.0	R13	S_FRAME	C06	T_NDRIVEN_SIGA
C08	F_PATH.1	R12	S_NENA_BACK	B06	T_NSPAUSE_SIGA
B14	F_REQUEST	P12	S_REVERSE	C13	T_PATH.0
C04	F_RR.0	P02	T_AD.0	A14	T_PATH.1
B04	F_RR.1	N03	T_AD.1	E02	T_PRIORITY.0
A04	F_RR.2	F14	T_AD.10	E03	T_PRIORITY.1
F03	F_SIZE_2	F15	T_AD.11	A13	T_REQUEST
G03	F_SOURCE.0	F16	T_AD.12	D14	T_RR.0
F01	F_SOURCE.1	G14	T_AD.13	D15	T_RR.1
F02	F_SOURCE.2	G15	T_AD.14	D16	T_RR.2
B02	I_INTERLEAVED	G16	T_AD.15	E14	T_SIZE.0
M02	I_MOD.0	H14	T_AD.16	E15	T_SIZE.1
M01	I_MOD.1	H15	T_AD.17	E16	T_SIZE.2
L03	I_MOD.2	J15	T_AD.18	D03	T_SOURCE.0
L02	I_MOD.3	J14	T_AD.19	C01	T_SOURCE.1
L01	I_MOD.4	P01	T_AD.2	C02	T_SOURCE.2
K03	I_MOD.5	K16	T_AD.20	B15	T_SPAUSE
K02	I_MOD.6	K15	T_AD.21	B13	T_SYNC
K01	I_MOD.7	K14	T_AD.22	A09	VDD
J03	I_MOD.8	L16	T_AD.23	A15	VDD
P03	I_NACCESS	L15	T_AD.24	B01	VDD
R02	M_NDEBUG	L14	T_AD.25	B16	VDD
P14	M_NFLOAT	M16	T_AD.26	J01	VDD
T15	M_NRESET	M15	T_AD.27	J16	VDD
B07	M_NSELECT	M14	T_AD.28	T01	VDD
R03	M_PARA	N16	T_AD.29	T08	VDD
C16	M_PARITY	N02	T_AD.3	T16	VDD
R04	M_REJ_ABORT	N15	T_AD.30	A02	VSS
J02	M_SIXTY_FIVE	N14	T_AD.31	A08	VSS
P04	M_TONIA_INT	P16	T_AD.32	A16	VSS
T03	M_TONIB_INT	P15	T_AD.33	H01	VSS
P05	R_CLK	N01	T_AD.4	H16	VSS
R09	R_DATA.0	M03	T_AD.5	R01	VSS
R08	R_DATA.1	H02	T_AD.6	R16	VSS
P08	R_DATA.2	H03	T_AD.7	T02	VSS
R07	R_DATA.3	G01	T_AD.8	T07	VSS

P07	R_DATA.4	G02	T_AD.9	T09	VSS
T06	R_DATA.5	B08	T_CLK		

The following page shows the SIGA pinout sorted by pin number.

SIGA PINOUT SORTED BY PIN NUMBER

A02	VSS	D16	T_RR.2	N14	T_AD.31
A03	T_ENA_TDAT.2	E01	T_MPAUSE	N15	T_AD.30
A04	F_RR.2	E02	T_PRIORITY.0	N16	T_AD.29
A05	F_CLK	E03	T_PRIORITY.1	P01	T_AD.2
A06	T_NBGRANT_SIGM	E14	T_SIZE.0	P02	T_AD.0
A07	F_PATH.0	E15	T_SIZE.1	P03	I_NACCESS
A08	VSS	E16	T_SIZE.2	P04	M_TONIA_INT
A09	VDD	F01	F_SOURCE.1	P05	R_CLK
A10	F_AD.18	F02	F_SOURCE.2	P06	R_DATA.7
A11	F_AD.21	F03	F_SIZE_2	P07	R_DATA.4
A12	F_AD.24	F14	T_AD.10	P08	R_DATA.2
A13	T_REQUEST	F15	T_AD.11	P09	S_DATA.7
A14	T_PATH.1	F16	T_AD.12	P10	S_DATA.4
A15	VDD	G01	T_AD.8	P11	S_DATA.1
A16	VSS	G02	T_AD.9	P12	S_REVERSE
B01	VDD	G03	F_SOURCE.0	P13	C_OUT
B02	I_INTERLEAVED	G14	T_AD.13	P14	M_NFLOAT
B03	T_ENA_TDAT.1	G15	T_AD.14	P15	T_AD.33
B04	F_RR.1	G16	T_AD.15	P16	T_AD.32
B05	T_NBREQ_SIGS	H01	VSS	R01	VSS
B06	T_NSPAUSE_SIGA	H02	T_AD.6	R02	M_NDEBUG
B07	M_NSELECT	H03	T_AD.7	R03	M_PARA
B08	T_CLK	H14	T_AD.16	R04	M_REJ_ABORT
B09	F_AD.16	H15	T_AD.17	R05	R_FRAME
B10	F_AD.19	H16	VSS	R06	R_DATA.6
B11	F_AD.22	J01	VDD	R07	R_DATA.3
B12	T_DRIVEN	J02	M_SIXTY_FIVE	R08	R_DATA.1
B13	T_SYNC	J03	I_MOD.8	R09	R_DATA.0
B14	F_REQUEST	J14	T_AD.19	R10	S_DATA.5
B15	T_SPAUSE	J15	T_AD.18	R11	S_DATA.2
B16	VDD	J16	VDD	R12	S_NENA_BACK
C01	T_SOURCE.1	K01	I_MOD.7	R13	S_FRAME
C02	T_SOURCE.2	K02	I_MOD.6	R14	C_NEXECUTE
C03	T_ENA_TDAT.0	K03	I_MOD.5	R15	C_CLK
C04	F_RR.0	K14	T_AD.22	R16	VSS
C05	T_NBREQ_SIGM	K15	T_AD.21	T01	VDD
C06	T_NDRIVEN_SIGA	K16	T_AD.20	T02	VSS
C07	T_NBGRANT_SIGS	L01	I_MOD.4	T03	M_TONIB_INT
C08	F_PATH.1	L02	I_MOD.3	T04	R_REVERSE
C09	F_AD.17	L03	I_MOD.2	T05	R_NENA_BACK
C10	F_AD.20	L14	T_AD.25	T06	R_DATA.5
C11	F_AD.23	L15	T_AD.24	T07	VSS
C12	T_ENA_HOLD	L16	T_AD.23	T08	VDD
C13	T_PATH.0	M01	I_MOD.1	T09	VSS
C14	T_ENA_TRANS.0	M02	I_MOD.0	T10	S_DATA.6
C15	T_ENA_TRANS.1	M03	T_AD.5	T11	S_DATA.3
C16	M_PARITY	M14	T_AD.28	T12	S_DATA.0
D01	T_LOCKOP.0	M15	T_AD.27	T13	S_CLK
D02	T_LOCKOP.1	M16	T_AD.26	T14	C_IN
D03	T_SOURCE.0	N01	T_AD.4	T15	M_NRESET

```
D14  T_RR.0          |  N02  T_AD.3        |  T16  VDD
D15  T_RR.1          |  N03  T_AD.1        |
```

A.9 A.C./D.C. Parameters

All SIGA input and bidirectional pins have a light pullup resistor, a diode protection network (max = 2000V) and latch-up (max = 200 ma). All inputs and output have standard TTL VIL/VIH and VOL/VOH characteristics. All outputs and bidirectional pins have 4 ma drive capability — except T_ENA_TDAT<2..0> and T_ENA_TRANS<1..0>, which have 8 ma drive capability. The SIGA will dissipate less than 3 watts.

The following page shows the A.C. timing parameters.

NOTE

For the B2VME, the following A.C. parameters override the normal ones:

PIN/CLASS	Tsu	Thld	Tpd (min/max)	LOAD
T_NDRIVEN_SIGA	-	-	2.0/11.0	20.0
F_SOURCE<2..0>	21.0	0.0	-	-

SIGA A.C. CHARACTERISTICS

PIN/CLASS	Tsu	Thld	Tpd (min/max)	LOAD
TBUS:				
T_DRIVEN	25.0	0.0	-	-
T_MPAUSE	25.0	0.0	-	-
T_SPAUSE	25.0	0.0	-	-
T_NBGRANT_SIGM	25.0	0.0	-	-
T_NBGRANT_SIGS	25.0	0.0	-	-
T_REQUEST	(a)	(a)	2.0/18.0	30.0
T_RR<2..0>	(a)	(a)	2.0/18.0	30.0
T_PATH<1..0>	(a)	(a)	2.0/18.0	30.0
T_SOURCE<2..0>	(a)	(a)	2.0/18.0	30.0
T_SIZE.2	(a)	(a)	2.0/18.0	30.0
T_SIZE<1..0>	20.0	0.0	2.0/18.0	30.0
T_SYNC	20.0	0.0	2.0/18.0	30.0
T_LOCKOP<1..0>	20.0	0.0	2.0/18.0	30.0
T_PRIORITY<1..0>	20.0	0.0	2.0/18.0	30.0
T_AD<33..0>	20.0	0.0	2.0/18.0	30.0
T_NBREQ_SIGM	-	-	2.0/13.0	20.0
T_NBREQ_SIGS	-	-	2.0/13.0	20.0
T_NDRIVEN_SIGA	-	-	2.0/13.0	20.0
T_NSPAUSE_SIGA	-	-	2.0/13.0	20.0
T_ENA_TDAT<2..0>	-	-	2.0/15.0	30.0
T_ENA_TRANS<1..0>	-	-	2.0/15.0	30.0
T_ENA_HOLD	(d)			
FAST:				
F_REQUEST	25.0	0.0	-	-
F_RR<2..0>	24.0	0.0	-	-
F_SOURCE<2..0>	25.0	0.0	-	-
F_PATH<1..0>	25.0	0.0	-	-
F_SIZE_2	25.0	0.0	-	-
F_AD<24..16>	25.0	0.0	-	-
SWITCH - REQ:				
R_DATA<7..0>	2.0	6.0	2.0/13.0	20.0
R_REVERSE	2.0	6.0	-	-
R_FRAME	-	-	2.0/13.0	20.0
R_NENA_BACK	-	-	2.0/13.0	20.0
R_CLK	(c)			
SWITCH - SER:				
S_DATA<7..0>	2.0	2.0	2.0/13.0	20.0
S_FRAME	2.0	6.0	-	-

S_REVERSE	-	-	2.0/13.0	20.0
S_NENA_BACK	-	-	2.0/13.0	20.0
R_CLK		(c)		
TCS:				

C_IN	50.0	50.0	-	-
C_OUT	-	-	2.0/50.0	20.0
C_NEXECUTE	50.0	50.0	-	-
INTERLEAVER:				

I_MOD<8..0>	17.0	0.0	-	-
I_INTERLEAVED	24.0	0.0	-	-
I_NACCESS	-	-	2.0/30.0	20.0
MISCELLANEOUS:				

M_TONIA_INT	-	-	2.0/30.0	20.0
M_TONIB_INT	-	-	2.0/30.0	20.0
M_PARITY	21.0	0.0	-	-
M_NSELECT	25.0	0.0	-	-
M_NDEBUG	25.0	24.0	-	-
M_SIXTY_FIVE	7.9	14.9	-	-
M_NRESET	(b)	(b)	-	-
M_REJ_ABORT	(b)	(b)	-	-

NOTES:
======

specific:

(a) No internal connection to SIGA - timing is unimportant
(b) Synchronized within SIGA - timing is unimportant
(c) MINIMUM HIGH time for [RS]_CLK = 5.3 ns
(d) [TF]_CLK rising to T_ENA_HOLD rising = 15 ns minimum
T_ENA_HOLD minimum HIGH time = 6 ns
T_ENA_HOLD falling to [TF]_CLK rising = 4.5 ns minimum

general:

1. All times in nanoseconds
2. All loads in picofarads
3. TBUS, FAST and INTERLEAVER timing are relative to rising T_CLK
4. SWITCH - REQ timing is relative to rising R_CLK
5. SWITCH - SER timing is relative to rising S_CLK
6. TCS timing is relative to falling C_CLK

B

T–bus Specification

This appendix is derived from the T–Bus Specification dated September 30, 1988.

B.1 Introduction

The T–Bus, or Transaction Bus, describes the transaction protocol and physical layer of the microprocessor bus architecture of the B2VME board. The T–Bus was designed to address the three main necessities of a Switch–based, multi–processor architecture: split cycles, Locking, and arbitration. In achieving these goals, the T–Bus has aquired some features which make it significantly different from most conventional microprocessor bus architectures.

This document describes both the physical and protocol layers of the T–Bus. Details about the operation of any specific devices residing ON the T–Bus are outside the scope of this documet. In addition, this document assumes that the reader have a basic understanding of bus architectures.

B.2 Logical Operators

Figure B–1 shows the standard operators used in this document.

Figure B-1 **Example — logical operators.**

```
OPERATOR        FUNCTION
========        ===========
&               logical "and"
#               logical "or"
$               logical "exor"
!               logical "not"
!$              logical "exnor"
|               concatenate
=               is defined as
:=              is defined after next
                positive edge of T_CLK as
...             continue equation from
                the previous line
```

B.3 Major T-Bus Concepts

The following desribes the Major T-Bus concepts with an emphasis on differences between the T-Bus and conventional bus architectures.

B.3.1 Master, Slaves and Drivership

The conventional notion of bus Masters and bus Slaves is not applicable to the T-Bus. Normally, a bus Master is the ONLY device that may request and get "ownership" of the bus. A conventional bus Slave responds immediately to the Master's requests and never initiates ownership of the bus for itself. In contrast, the T-Bus design liberates the Slave's typical role to allow the Slave limited ownership of the bus. This is done to support split-cycles (see: "Deferred Transactions"). In this document, the terms Master and Slave are used only for historical consistenecy and the reader is cautioned NOT to apply the conventional meaning to these terms.

Along with their new roles in the T-Bus domain, Masters and Slaves force the application of a new term to replace the conventional term of bus ownership. Since there is now some parity between Masters and Slaves, the term "ownership" is no longer applicable. A T-Bus Master or Slave is considered to have bus "drivership" (not ownership) when that device is "requesting" (Master) or "responding" (Slave) on the T-Bus. By definition, the device that is causing the assertion of the T_DRIVEN (see: "Signal Field Description Summary" for details) signal is the current T-Bus driver.

Arbitration is interesting on the T-Bus because Slaves as well as Masters must now be allowed to arbitrate for T-Bus drivership. This necessitates an arbitration scheme that is impervious to Master/Slave lockout situations.

B.3.2 Split Cycles

A Slave responding to a Master's Function Request (request to do some task) may not be able to immediately carry-out that request. In fact, the Slave may have to perform internal operations which span many dozens of T-Bus cycles before it can respond. If the Slave where to tie-up the T-Bus by not responding immediately, no other devices could make forward progress and significant system degradation would result. The concept of split-cycles solves this problem. When a Master makes a Function Request, the Slave can say, "I have received your request and I will respond to it later." This is known as a "defer response".

Following the defer response, the Master and Slave then release the T-Bus while the Slave obtains its Function Response (answer to the Function Request). In the meanwhile, transactions from other T-Bus devices can occur as normal. When the Slave has finished "building" its Function Response, it then accesses the T-Bus and hands its "split response" to the Master; which has been anticipating the Slaves response.

While the Master is waiting for the Slave, it is called a "split Master". The Slave that is off processing the Master's Function request is called a "split Slave". Split sequences can be used between any Master/Slave pair that support this function. Split sequences can be freely intermingled with non-split sequences and locked sequences (see: Locked Operations).

B.3.3 Locks, Stolen

The T-Bus supports the concepts of Stolen and Locks. Stealing is supported by the addition of an extra bit field in the address/data portion of the T-Bus. When a "stolen" data word is transfered across the T-Bus interface to a Slave, that bit is asserted and recorded in the Slave as a "tag" on that word. This technique helps to identify a word (or words) as having a special status. For instance, the stolen bit may be used by one device to prevent other devices from evaluating a word of memory.

Locking is the technique where a T-Bus Master obtains exclusive use of a T-Bus Slave for one or many sequences. That Slave is responsible for preventing other Masters from gaining access to it (with one exception). A locked transaction has three distinct parts: opening, maintaining, and closing.

Opening involves a Master getting some Slave to "lock" itself to that Master. This commits that Slave to servicing only the locking Master, and is accomplished with a special Function Request. The Slave records the Master's unique "Master Number" for future reference. A Master may lock multiple Slaves to it but a Slave can only have one Master.

Once the Master has opened the lock sucessfully, it can continue to perform most sequences. The Slave will continue to comply with Function Requests

from its locked Master but will "rebuff" those from other Masters (again, with one exception – see: "Denied Responses"). The slave recognizes a request from its locked Master by comparing its stored Master Number with that of the requesting device. When the Master is done with the lock, it "closes" it by issuing a specific Function Request (see: "Closing a Lock"). After this event, all Slaves that recognize the requesting Master as their locked Master become unlocked and can once again respond to ANY Master.

B.3.4 Observing Master State

When a locked Slave rebuffs a Master to which it is not locked, that Master enters the "observing Master" state. This is a state of "suspended animation" for the Master while "observes" the T-Bus for a "free request" from the Master "owning" the lock. When the Master recognizes the this event, it arbitrates for T-Bus drivership and re-requests the original Function Request.

B.3.5 High-Level Protocol Definitions

The T-Bus operates at the lowest level of protocol with the concept of requests and responses. A request cycle is defined as the T-Bus cycle where a Master – following a bus request/grant sequence – drives the T-Bus for exactly one cycle and "asks" some Slave to perform a task. A response immediately follows the request and is defined as a "requested" Slave "answering" the "requesting" Master and possibly transfers data. A response may occupy a varied number of T-Bus cycles, due to two factors: 1) the number of words to be transfered, and 2) the possibility of a Slave Pause or Master Pause condition (described later). A transaction is defined as the combination of request and related response which occur between explicit arbitration cycles. Explicit arbitration is defined in more detail in: "Arbitration". The length of a transaction will vary as the length of a response.

When a Master requests some action of a Slave, that Slave may respond by starting a split-cycle. These events, of course, comprise a transaction. As previously described, the Master then releases the bus and other T-Bus activity takes over. Later on, the Slave returns the answer to the Master. This entire process, which may take place over many T-Bus cycles, is called a sequence. The original "asking" by the Master is defined as the Function Request and the eventual (after the split-cycle) "answer" by the Slave is defined as the Function Response. Another way of looking at this is that the Function Request is the "original request" and the Function Response is the "ULTIMATE answer". The combination of a Function Request and Function Response are defined as a sequence.

Note that a sequence and a transaction will coincide for "immediate" Slave responses. Here, the response is the same as the Function Response. It is only because of split-cycles that the concept of Function Request, Function Re-

sponse and sequence are necessary. Note also that the request and Function Request ALWAYS coincide (split-cycle or not).

At the highest level of abstraction is the operation. An operation is defined as a series of sequences required to accomplish some goal. For instance, to perform some locked read/writes on some remote node, the Master must open a lock (with an open-lock transaction), perform some read/writes (using normal transactions) and then close the lock (using a free-lock transaction).

B.4 Signal Field Description

The entire T-Bus consists of 61 signals which can be broken down into two major fields: the "Transaction" (TRANS) and the "T-Bus Address/Data" (TDAT). The TRANS field is the control field and contains the 27 (minimum) signals necessary to implement the T-Bus Transaction protocol. The TDAT field contains the 34 signals necessary to transfer the data and address information. The width of the TRANS field will increase as more Masters and/or Slaves are included in a particular instantiation of the T-Bus. The width of 27 represents the minimum configuration (one Master, one Slave). The TDAT and TRANS fields are further broken-down into subfields seen in Figure B-2. Each subfield is then broken-down into signal fields. Often, a signal field will consist of only one signal.

Figure B-2 T-Bus TDAT and TRANS subfields.

TRANS Subfields	TDAT Subfields
STATUS	LRA (T_REQUEST = 1)
ARBITRATION	OFFSET (T_REQUEST = 1)
TRANSACTION (normal)	STOLEN (T_REQUEST = 0)
TRANSACTION (extended)	DATA (T_REQUEST = 0)
CLOCKS	

These function of these signal fields are described in greater detail in the following sections. The description is on a symbolic level - actual signal polarities and encodings are described in: "Logical Signal Assignments". In the description, the terms "sink" and "source" are used to describe signal direction. Sink indicates that some T-Bus device must RECEIVE that signal from the T-Bus. Source indicates that some T-Bus device must TRANSMIT that signal to the T-Bus. As indicated in some cases, a signal can be sink/sourced from/to the T-Bus. In addition the terms "valid request" and "valid response" are introduced in the description of the T_REQUEST signal.

Refer to "Logical Signal Assignments" for the following discussion.

B.4.1 TRANS Field

The following describes the subfields and signal fields of the TRANS field.

Figure B-3 T-Bus STATUS subfield.

Signal Name	Signal Direction
T_NDRIVEN_*	(source)
T_NMPAUSE_*	(source)
T_NSPAUSE_*	(source)
T_DRIVEN	(sink)
T_MPAUSE	(sink)
T_SPAUSE	(sink)

The STATUS field is used to indicate condititions which may modify the interpretation of most of the other T-Bus fields. The STATUS field is also used for arbitration. T_MPAUSE and T_SPAUSE are the "highest order" T-Bus signals because their state indicates IF any other T-Bus fields should be considered valid.

T_MPAUSE is the "Master Pause" indicator for T-Bus Masters. It indicates that one or more T-Bus Masters is requesting a "pause". T_MPAUSE is derived from all of the T_NMPAUSE_* signals, where "*" is a wildcard representing all of the Master devices which want to have pause capability. Note that a Master is not REQUIRED to have pause capability. When it wants to pause, one (AND ONLY ONE) Master asserts its own T_NMPAUSE_* signal. When ANY of the T_NMPAUSE_* signals from any of the Masters is asserted, T_MPAUSE is asserted. This defines a "Master Pause" condition. Master Pause is activated by a Master's desire to "throttle" a read/write data transfer from/to responding Slave during a transaction. The Slave detects this request by observing T_MPAUSE.

T_SPAUSE is the "Slave Pause" indicator for T-Bus Slaves. In an analogous way to T_MPAUSE, T_SPAUSE is derived from all of the T_NSPAUSE_* signals wishing to have pause capability. Note that like a Master, a Slave is not REQUIRED to have pause capability. When it wants to pause, one (AND ONLY ONE) Slave asserts its own T_NSPAUSE_* signal. When ANY of the T_NSPAUSE_* signals is asserted, T_SPAUSE is asserted. This defines a "Slave Pause" condition. Slave Pause is activated by a Slave's desire to "throttle" a read/write data transfer to/from a requesting Master during a transaction. The Master recognizes this desire by observing T_SPAUSE.

Asserting T_[MS]PAUSE "overrides" ALL other T-Bus signals EXCEPT those in the TDAT field. See the "Pausing" section for more details. This means that the indicated fields should be ignored during Master and Slave Pause. Conceptually, T_MPAUSE and T_SPAUSE are used to "stretch" a par-

ticular transaction by telling all Masters and Slaves to: "ignore the T-Bus for now, see you in the next cycle!"

T_DRIVEN, when asserted, indicates that either a single Master or single Slave device is driving the TRANS field. The T_REQUEST signal (described later) is then used to differentiate between a Master and a Slave. T_DRIVEN is also used to define the T-Bus arbitration period (see: Arbitration). Analagous to T_MPAUSE and T_SPAUSE, T_DRIVEN is derived from from all Masters and Slaves which drive their own signal, T_NDRIVEN_*. When any T_NDRIVEN_* is asserted, T_DRIVEN is assserted. Every Master and Slave MUST source its own T_NDRIVEN_* signal, but ONLY ONE can assert it during a given T-Bus period.

When T_DRIVEN is negated, the T-Bus is in its "dead" state (see "Arbitration"). While in the dead state, no other signals in the TRANSACTION or TDAT fields (except T_MPAUSE and T_SPAUSE) should be interpreted.

B.4.2 ARBITRATION Subfield

Figure B-4 T-Bus ARBITRATION subfield.

Signal Name	Signal Direction
T_NBREQ_*	(source)
T_NBGRANT_*	(sink)

The ARBITRATION field is used exclusively byany T-Bus Master or Slave to request and be granted T-Bus drivership. Normally, all Masters and some or all Slaves will want the ability to arbitrate for drivership. Slaves need this ability ONLY if they support split-cycle sequences.

T_NBREQ_* represents the separate T-Bus request signals from all the Masters and Slaves desiring T-Bus arbitration. They are inputs to the T-Bus arbitration logic. The outputs of this logic are the T_NBGRANT_* signals which, when asserted, indicate to a particular Master or Slave that they will have T-Bus drivership in the next T-Bus cycle. Arbitration is discussed in detail in the "Arbitration" section.

B.4.3 TRANSACTION (STANDARD) Subfield

Figure B-5 T-Bus TRANSACTION (STANDARD) subfield.

Signal/Field Name	Signal Direction
T_REQUEST	(source/sink)
T_SOURCE<2..0>	(source/sink)
T_RR<2..0>	(source/sink)
T_SIZE<2..0>	(source/sink)
T_LOCKOP<1..0>	(source/sink)
T_PATH<1..0>	(source/sink)

The TRANSACTION (STANDARD) field implements the "heart" of a T-Bus operation. It defines the characteristics of a particular transaction. The group of signals considered here are for "standard" sequences, meaning sequences which do not require a Switch access. The signals related to Switch accesses are in the "extended" portion of the TRANSACTION field and are described later.

T_REQUEST is the "request/response" modifier for the T-Bus. As with all TRANSACTION signals, T_REQUEST is only valid (can be interpreted) when T_DRIVEN is asserted and T_MPAUSE and T_SPAUSE are negated. A valid T_REQUEST signal indicates whether a Master or a Slave currently has T-Bus drivership. In doing so, the valid T_REQUEST also modifies the meaning of the other subfields of the TRANSACTION field. All descriptions of the remaining TRANSACTION fields include a reference to the T_REQUEST modifier. Essentially, when a valid T_REQUEST is asserted, the current cycle is a "request cycle" and the Master has drivership. When a valid T_REQUEST is negated, the current cycle is a "response cycle" and the Slave has drivership. A Master or Slave pause condition during what would NORMALLY be a response cycle, causes that cycle to become a "paused response cycle". Note that a paused response cycle is a subset of a response cycle.

T_SOURCE, is used to indicate the "source number" of the device with current drivership. During a request cycle, T_SOURCE indicates the requesting Master's source number. During a response cycle, it indicates the respondings Slave's source number. The Slave source number is used by Masters in split-cycle sequences. The Master source number is used by Slaves in locked sequences.

The set of all Masters in a particular implementation of the T-Bus MUST have unique source numbers. With one exception, the set of all Slaves must have unique source numbers as well. Any non-split-cycle Slave may use the reserved "anonymous" Slave number of "000". This optimization frees-up more of the possible Slave source numbers for use by other devices.

During a request cycle, the T_RR field indicates whether the request is a "READ", "WRITE", or "FREE-LOCKS". In addition, T_RR can indicate an "AUX_READ" or "AUX_WRITE". In the current instantiation of the B2VME, "AUX_READ" and "AUX_WRITE" are treated by Slaves in exactly the same manner as "READ" and "WRITE".

During a response cycle, the T_RR field indicates the Slave's response. An "ERROR" response indicates that the Slave detected an error and wishes to end the sequence. A "PROMISE" response indicates that the Slave will enter its split-cycle state and return the Function Response at a later time. A "REFUSED" response indicates to the requesting Master that the Slave is temporarily "busy". In this case, the Master would typically try again as soon as it regains drivership of the bus. A "COMPLETED" indicates the end of a sequence. A "MORE" indicates to the requesting Master that the Slave will source additional data in the next cycle (reads) or will sink additional data on the next cycle (writes). An "EARLY-ACK" response is issued by a Slave as an indicator that the transaction will end within two bus cycles. This response is an optimization for certain Masters and is NOT required for Slaves. A "REFUSED-LOCKED" response indicates to the requesting Master that the Slave is currently locked to another Master and is denying the requesting Master's function request.

During a request cycle, the T_SIZE field indicates that the number of words in the transfer and thus defines the minimum length of the transaction. Note that the T_SIZE field reserves the code for "1-word"; a Master must request a "4-byte" size instead. This is done to allow Slaves which do not support multi-word transfers, to easily detect those transfers. During a response cycle, T_SIZE has no meaning and need not be driven by the responding Slave.

During a request cycle, the T_LOCKOP field indicates what type of locking requests, if any, are being made of the addressed Slave. The "NORMAL" request indicates that the Master is requesting a non-locked sequence. The "BYPASS" request indicates that the requested Slave should ignore any other lock it may "own" (from other another Master) and respond to the requesting Master's Function Request. An "OPEN" request indicates to the requested Slave that the requesting Master wishes to begin a locked operation. A "MAINTAIN" request indicates to the requested Slave that the Master wishes to continue a locked sequence. Both the "BYPASS" and "NORMAL" requests will act like a "MAINTAIN" in that they preserve the current – if any – locked sequence. During a response cycle, T_LOCKOP has no meaning and need not be driven by the responding Slave.

During a request cycle, the T_PATH field indicates whether the Function Request is local or remote (over the Switch). If remote, the TRANSACTION (EXTENDED) field must be interpreted by the Slave.

B.4.4 TRANSACTION (EXTENDED) Subfield

Figure B-6 T-Bus TRANSACTION (EXTENDED) subfield.

Signal/Field Name	Signal Direction
T_SYNC	(source/sink)
T_PRIORITY<1..0>	(source/sink)

The TRANSACTION (EXTENDED) field adds Switch access capabilities to a Master's Function Request. This field is only valid during a request cycle and need not be driven by a Slave during the response cycle. In addition, only Masters which need to access the Switch need drive this field during a Function Request. Therfore, the TRANSACTION (EXTENDED) field is interpreted only when the T_PATH field is "SIGA_A" or "SIGA_B".

During a request cycle, the T_SYNC field is used to command the Switch Interface Gate Array's (SIGA) Requestor interface to force a "message class" of "00" for that particular Function Request. See the SIGA Functional Specification for more details. T_SYNC need not be driven by a Master which does not require Switch accesses.

During a request cycle, the T_PRIORITY field indicates the DESIRED Switch message priority. The ACTUAL Switch message priority is determined by many factors within the SIGA. See the SIGA Functional Specification for more details.

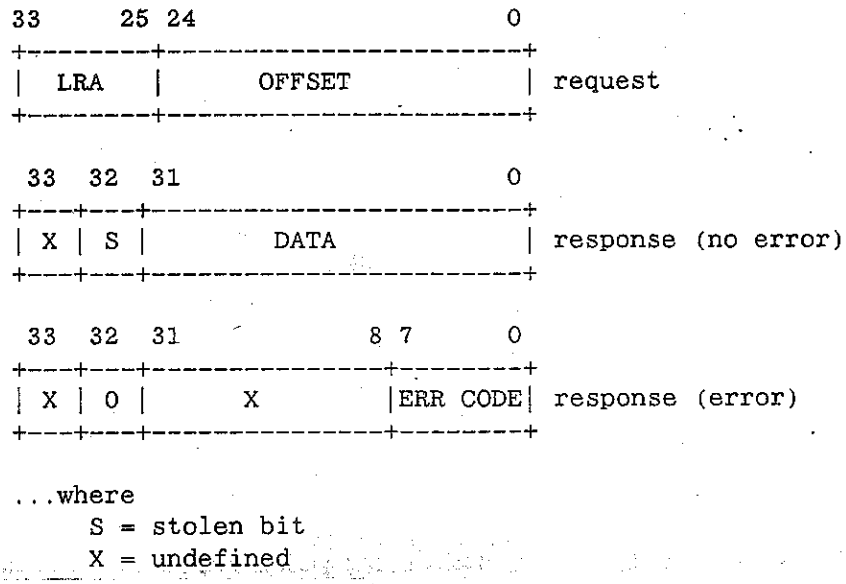
B.4.5 TDAT Field

The 34-bit TDAT field supports the multiplexed address/data architecture of the T-Bus. The subfields have different definitions depending on whether the T-Bus is currently in a request or response cycle. The following describes the subfields of the TDAT field.

During a request cycle, the Master sources the address onto the TDAT field. It can be accessing either a local or remote device. In either case, the OFFSET describes the address offset. The minimum width of this field is defined by the maximum memory configuration expected on a given instantiation of the T-Bus. For the B2VME, the maximum memory is 32 Mb and thus the OFFSET field is: T_AD<24..0>. Only the minimum width of the field need be driven by the requesting Master which performs only local accesses. During remote accesses (T_PATH = "SIGA_A" or "SIGA_B"), the LRA field indicates the Logical Route Address to the Slave (always the SIGA-Requestor). The Logical Route Address is transformed by the SIGA into the Physical Route Address. See the SIGA Specification for details.

During a response cycle, the DATA subfield holds the read/write data for the appropriate Function Request. If the response is an "ERROR", T_AD<7..0> will hold the error code. The STOLEN subfield, when asserted, indicates whether or not a particular word of data is "stolen". The UNDEFINED field of the TDAT field need not be driven during responses. Figure B–7 shows the various definitions of the TDAT field.

Figure B–7 T–Bus TDAT field definition.



B.5 Operation

The following describes the operation of the T–Bus in detail.

B.5.1 Arbitration

Arbitration is the process by which a T–Bus Master or Slave which did not previously have drivership, requests (possibly) and eventually gets drivership. This can happen in two ways: explicitly and implicitly. An explicit grant is when a Master or Slavé (returning a split–response) first requests, then is granted drivership of the T–Bus. Implicit drivership occurs only in a Slave; when it recognizes that a Master is making a request to it, and then drives the bus in the next cycle with its response. Both concepts are discussed below.

B.5.1.1 Explicit

Explicit arbitration is "look-ahead" in the sense that when a device is "granted" the T-Bus in say, cycle #1; that device is expected to drive the bus in cycle #2. In addition, an explicit T-Bus grant CAN (and does in the B2VME) be given to a device in the same cycle as the original request. The A.C. timing specification of the T-Bus allows for this possibility. Explicit arbitration begins when a Master or Slave asserts its T_NBREQ_* signal. The arbitration logic asserts exactly one of the T_NBGRANT_* signals based on all of its T_NBREQ_* inputs. The "granted" Master or Slave must then wait for either an explicit arbitration cycle or a "dead" cycle on the T-Bus (if it does not already exist) before it can reconize the assertion of its T_NBGRANT_* as valid.

An explicit arbitration cycle is defined as follows: Some device is responding with anything except a "MORE" or "EARLY-ACK", and there are no Master or Slave pauses. This implies a cycle where a Slave is at the "end" of its response cycle(s). A dead cycle is defined when no T-Bus device is driving, and both Slave and Master Pause are inactive. In both cases, the Requesting device MUST also check to see whether or not it actually requested the bus! This scheme places much of the burden of arbitration with the various T-Bus Masters and Slaves. A device which receives a grant during as explicit arbitration cycle is said to have received an "explicit grant". A formal definition of an explicit grant is defined in Figure B-8.

Figure B-8 Definition — T-Bus explicit grant.

```
<explicit grant> = <bus request> & <explicit arbitration cycle>
                   ... & <bus grant>

<explicit arbitration cycle> = <arbitration cycle> # <dead cycle>

<arbitration cycle> = !(T_MPAUSE + T_SPAUSE)
                      ... & T_DRIVEN & T_RESPONSE
                      ... & !<more or early-ack>

<dead cycle> = !(T_MPAUSE # T_SPAUSE) & !T_DRIVEN

<more or early-ack> = T_RR is "MORE" or "EARLY-ACK"

<bus request> = T_NBREQ_* is asserted

<bus grant> = T_NBGRANT_*is asserted
```

A typical explicit arbitration sequence is shown in Figure B-9. Note that Master device "Foo" negates T_NBREQ_Foo when another arbitration cycle arrives. In fact, a granted device MUST negate its T_NBREQ_* during the next arbitration cycle if it no longer wants to drive the T-Bus. In addition, although

Figure B-9 does not show it, Slaves can arbitrate for the bus in the exact same manner to return a split-cycle Function Response. Slaves must also negate their T_BREQ_* during the last cycle of their response.

Figure B-9 State simulation — explicit arbitration example for Master.

```
...given CYC = <tbus cycle number>
         NBRQ = T_NBREQ_Foo
         NBGR = T_NBGRANT_Foo
```

CYC	NBRQ	NBGR	Cycle Type	Comments
1	1	X	<dont care>	no request yet
2	0	1	<dont care>	no grant yet
3	0	0	<request>	a different Master on now
4	0	0	<response>	its a 3-cycle transaction
5	0	0	<response>	almost done
6	0	0	<arbitration>	Done! Foo will get next drivership
7	X	X	<request>	Foo now drives
8	X	X	<response>	its another 3-cycle transaction
9	1	X	<arbitration>	Done! Foo loses next drivership
10	1	X	<dont care>	another device (if any) drives

The T-Bus protocol enforces only ONE restriction on the arbitration logic: All Slaves must have higher priority than all Masters. This is explained fully in the "Denied Responses" section.

B.5.1.2 Implicit

Implicit arbitration is only used by Slaves. It occurs when a Master makes a request on the T-Bus and the requested Slave, recognizing the request to it, acts by driving the T-Bus in the following cycle in response. The Slave than receives what is called an "implicit grant". The actual "arbitration" occurs when the Slave recognizes that a request for its address is on the T-Bus. The Slave detects its address based on some combination of address bits and the T_PATH<1.0> field. The specific combination depends on the system memory map. In general, however, a "local" slave will respond to some combination of T_AD<24..0> with the T_PATH = "LOCAL". The SIGA is a special case of implicitly responding Slaves because it will respond when the T_PATH field is "SIGA_A" or "SIGA_B" - regardless of the TDAT field. Figure B-10 defines the implicit arbitration cycle.

Figure B-10 Definition — T-Bus implicit arbitration cycle.

```
<implicit grant> = <implicit arbitration cycle>
                   ... & <my address>

<implicit arbitration cycle> = !(T_MPAUSE  +  T_SPAUSE)
                               ... &  T_DRIVEN  & !T_RESPONSE
                               ... & !<free lock>

<my address> = <my local address> # <my remote address>

<my local address> = T_PATH is "LOCAL"
                     ... & T_AD<24..0> "is in my address space"

<my remote address> = T_PATH is "SIGA_A" and I am Siga A
                      ... or T_PATH is "SIGA_B" and I am Siga B

<free lock> = T_RR is "FREE-LOCK"
```

Note that it is impossible for implicit and explicit arbitration to occur simultaneously because explicit arbitration cannot, by definition, occur during a request cycle. Implicit arbitration, on the other hand, can ONLY occur during a request cycle.

B.5.2 Driving Rules

As stated previously, T-Bus drivership is defined by the device which is, or will be, driving the TRANS field. The TDAT field however, will not always be driven by the same which has T-Bus drivership. In fact, the rules for driving (not necessarily drivership) can be stated in a precise format as seen in Figure B-11. Here, it is seen that TRANS and TDAT fields are driven by different devices during write sequences.

Figure B-11 Driving rules for T-Bus Masters and Slaves.

```
...given <explicit> = <explicit arbitration cycle> & !<free>
         <implicit> = <implicit arbitration>
         <noarb> = !<explicit arbitration>
                   ... & !<implicit arbitration>
         free = <free-locks request>
```

Cycle Type	Request Type	Next drive of TDAT to device...	Next drive of TRANS to device...
<explicit>	-	that was granted	that was granted
<implicit>	read	BEING requested	BEING requested
<implicit>	write	MAKING the request	BEING requested
<implicit>	free	none	none
<noarb>	-	currently driving	currently driving

Note in Figure B-11, that a free-lock request has no device driving during the response phase. This is done for a number of reasons which are described in a later section.

B.5.3 Cycle Definitions

A formal definition of a request and response cycle is necessary for the rest of the discussion. Figure B-12 gives those definitions.

Figure B-12 Definition — request and response cycles.

```
<request cycle> = !(T_MPAUSE + T_SPAUSE)
                  ... & T_DRIVEN & !T_RESPONSE

<response cycle> = !(T_MPAUSE + T_SPAUSE)
                   ... & T_DRIVEN & T_RESPONSE
```

Note that a response cycle can simultaneously be an explicit arbitration cycle. In addition, a request cycle is ALWAYS an explicit arbitration cycle. A response cycle that is interrupted by an asserted T_MPAUSE or T_SPAUSE signal is called a "paused response cycle".

B.5.4 Immediate Transactions

The simplest transaction is the "immediate Function Response", or "immediate" transaction. This is where a Master, after arbitrating for the T-Bus, makes a request that the requested Slave can service "immediately" - before the next arbitration cycle. By definition, this is a case where the transaction's response cycle and the Function Response coincide. Figure B-13 shows a simple example of a 1-word read and write.

Figure B-13 State simulation — immediate, byte read/write.

```
READ:

CYC Driver    T_RR        Comments
=== ========  =========   ===================================
 1  Slave n   COMPLETED   end previous transaction; arbitrate
 2  Master A  READ        request: read 1 word from Slave B
 3  Slave B   COMPLETED   response: here is data; arbitrate
 4    ?         ?         some other transaction

WRITE:

CYC Driver    T_RR        Comments
=== ========  =========   ===================================
 1  Slave n   COMPLETED   end previous transaction; arbitrate
 2  Master A  WRITE       request: write 1 word to Slave B
 3  Slave B   COMPLETED   response: got data; arbitrate
 4    ?         ?         some other transaction
```

Figure B-13 shows that the read and write look quite similar; the only difference being the direction of the TDAT field during cycle # 3, which is not shown. In general, the direction of the TDAT field in a State Simulation Diagram is obvious and thus will not be shown. The reader should refer to the "Driving Rules" section of this document for more details on TDAT driving rules. In addition, the explicit arbitration portion of the transaction is shown in Figure B-13 but will not be shown in subsequent figures.

When performing a transfer of more than one word, the Slave must respond with "placeholder" response rather than just respond with "COMPLETED" for each word transferred. This is because "COMPLETED" is used by all other Masters and Slaves to determine when arbitration occurs and must be reserved for the end of the response. That placeholder is the "MORE" response. Figure B-14 shows an immediate multi-word read and write.

Figure B-14 State simulation — immediate, 3-word read/write.

```
READ:

CYC Driver    T_RR        Comments
=== ========  ==========  ==================================
 1  Master A  READ        request: read 3-words from Slave B
 2  Slave B   MORE        response: sourcing the 1st word
 3  Slave B   MORE        response: sourcing the 2nd word
 4  Slave B   COMPLETED   response: sourcing the last word

WRITE:

CYC Driver    T_RR        Comments
=== ========  ==========  ==================================
 1  Master A  WRITE       request: write 3-words to Slave B
 2  Slave B   MORE        response: sinking the 1st word
 3  Slave B   MORE        response: sinking the 2nd word
 4  Slave B   COMPLETED   response: sinking the last word
```

Of course, there will normally be "n-1" valid "MORE" responses, where "n" is the number of words to be transfered. This number will change if there are any errors (described later). The two and four-word reads and writes are obvious extensions of the simulation in Figure B-14.

B.5.5 Pausing

Pausing is the method by which Masters or Slaves can "extend" a single transaction in order to have more time to process or perform their task for that transaction. As mentioned in the "Signal Field Description", a Master or Slave can request a pause by asserting its T_NMPAUSE_* or T_NSPAUSE_* signal, respectively. This will cause either T_MPAUSE or T_SPAUSE to be asserted. Five important rules apply to pauses and are shown in Figure B-15.

Figure B-15 Rules for pausing.

1. Slaves can assert T_NSPAUSE_* when and only when they are making a response. Masters can assert T_NMPAUSE_* only when they were granted drivership during the most recent arbitration cycle. Once this condition is met, there are no restrictions on WHEN the device can pause.
2. As a corallary to rule #1: Of the set of all Masters and Slaves, exactly ONE can be asserting its individual pause signal in a given T-Bus cycle.
3. Pausing overrides ALL other TRANS signals on the T-Bus. It basically indicates: "Ignore this cycle in all respects and continue as normal in the next cycle."

4. There is no protocol restriction on how many cycles a device may pause for. However, practical restrictions – like T-Bus throughput and bandwidth – do apply.

5. Any device sourcing data while being paused by the sinking device MUST HOLD that data stable and valid until the pause is released.

When particular Master is making a request, the responding Slave may pause by asserting T_NSPAUSE_* for a variety of reasons. The most likley reasons are: 1) The Slave is not ready to source the requested data on a read, 2) The Slave is not ready to sink the requested data on a write, or 3) The Slave needs more than one T-Bus cycle to calculate its response. Figure B-16 shows how a Slave Pause would work in a multi-word read/write situation (Reasons #1, #2).

Figure B-16 State simulation — immediate, 2-word read/write with Slave.

READ:

CYC	Driver	T_SPAUSE	T_RR	Comments
1	Master A	negated	READ	req: read 2-words from Slave B
2	Slave B	asserted	X	resp: pause, I'm not ready
3	Slave B	negated	MORE	resp: sourcing the 1st word
4	Slave B	asserted	X	resp: pause, I'm not ready
5	Slave B	negated	COMPLETED	resp: sourcing last word, arb

WRITE:

CYC	Driver	T_SPAUSE	T_RR	Comments
1	Master A	negated	WRITE	req: write 2-words to Slave C
2	Slave C	asserted	X	resp: pause, I'm not ready
3	Slave C	asserted	X	resp: pause, I'm not ready
4	Slave C	asserted	X	resp: pause, I'm not ready
5	Slave C	negated	MORE	resp: sinking the 1st word
6	Slave C	negated	COMPLETED	resp: sinking last word, arb

Note from Figure B-16 that the T_RR field is a "don't-care" during the paused cycles. Again, PAUSES ALWAYS OVERRIDE REQUEST OR RESPONSE CYCLES. Cycle #1 in both transactions is, of course, an implicit arbitration cycle. Cycles #2 and #4 in the read tranasaction and #2, #3, #4 in the write transaction are known as paused response cycles. Note also that Slave C appears to be slow in responding to the request. Although the T-Bus protocol supports a theoretically infinite pause period, there is obvously a practical threshold which impacts system performance. This threshold must be decided upon when the particular implementation of a T-Bus is designed. If a Slave could not meet that threshold for response time, it should then issue a PROMISE (see: "Refusing") and return the response later.

A requesting Master may pause by asserting T_NMPAUSE_* for a variety of reasons. The most likley reasons are: 1) The Master is not ready to sink the requested data on a read, 2) The Master is not ready to source the requested data on a write, or 3) The Master has just gained drivership of the T-Bus and needs more than one T-Bus cycle to calculate its request. Reason #3 is an unusual case and is not implemented on the B2VME. Figure B-17 shows how a Master Pause would work in a single-word write situation (Reason #1).

Figure B-17 State simulation — immediate, 1-word read with Master.

CYC	Driver	M_SPAUSE	T_RR	Comments
1	Master A	negated	READ	req: read 1-word from Slave B
2	Slave B	asserted	X	resp: I am waiting for you
3	Slave B	asserted	X	resp: I am waiting for you
4	Slave B	negated	COMPLETED	resp: sourcing word, arbitrate

Note that the Master can assert Master Pause even when it is receiving data from a split-response. In addition, it is possible for both Master and Slave pause to occur while a Slave is responding.

B.5.6 Early Acknowledge

A T-Bus Slave will issue a pause when it cannot complete a task and respond quickly enough. Often however, a slave MAY be able to predict WHEN it will finish. Transfering this knowledge to the requesting Master enables that Master to anticipate the eventual Function Response. For the SIGA-Server, for instance, the "early warning" from a requested Slave will allow the SIGA to begin transmitting its upstream Switch message header in anticipation of the data.

By their nature, some Slaves (like memory) that pause DO know when their response cycle will end. For this reason, Slaves may implement what is known as an early acknowledge. This is accomplished by responding with an "EARLY-ACK" before the end of a Function Response. In fact, the "EARLY-ACK" must occur no more than TWO T-Bus cycles before the end of the a Function Response. The "EARLY-ACK" token in a response is a "placeholder" in the same sense as a "MORE" token. This can be seen in the previous definition of an arbitration cycle. An "EARLY-ACK" of course, should NOT be accompanied by Slave pause within the same cycle. However, an "EARLY-ACK" does imply that the TDAT field should NOT be considered valid. A typcial early acknowledge is shown in Figure B-18.

Figure B-18 State simulation — early acknowledge.

```
CYC Driver    T_SPAUSE  T_RR       Comments
=== ========  ========  =========  ==============================
 1  Master A  negated   READ       req: read 2-words from Slave B
 2  Slave B   asserted     X       resp: pause, I'm not ready
 3  Slave B   negated   EARLY-ACK  resp: resp: early-ack
 4  Slave B   asserted     X       resp: pause, I'm not ready
 5  Slave B   negated   COMPLETED  resp: sourcing last word, arb
```

B.5.7 Deferred Transactions

A more complex transaction type, known as a "deferred transaction" contains two new concepts: "deferred responses" and "denied responses". These are described in the following discussion.

B.5.7.1 Deferred Responses

A deferred transaction is when a requested Slave knows that it cannot comply with a Function Request immediately, and must "defer" its response until later. The slave indicates this fact to the requesting Master by responding with a "PROMISE" (also known as deferred) response. In the cycle following the deferred response, both the transacting Master and Slave release drivership of the T-Bus in the same exact way as the end of an immediate response. Figure B-19 shows a deferred write.

Figure B-19 State simulation — deferred write.

```
DEFERRED 3-WORD WRITE, NO PAUSING

CYC Driver    T_RR       Comments
=== ========  =========  ======================================
 1  Master A  WRITE      req: write 3-words to Slave C
 2  Slave C   MORE       resp: sinking the 1st word
 3  Slave C   MORE       resp: sinking the 2nd word
 4  Slave C   PROMISE    resp: sinking last word, deferred, arb
 .     .        .              .
 .     .        .              .
 .     .        .              .
 n  Master X  READ       req: read 1 word from Slave Y
n+1 Slave Y   COMPLETED  resp: sinking last word, done
 .     .        .              .
 .     .        .              .
 .     .        .              .
 m  Slave C   <no drive> Slave C requests and is granted
m+1 Slave C   COMPLETED  resp: I'm done, no errors
```

A number of T-Bus concepts are introduced/refined in Figure B-19. Slave C decides that it must defer its response and issues a "PROMISE" to its requesting Master A. Note that it is up to the Slave and NOT the Master to decide whether or not the Slave should defer its response. In a typical implementation, a Slave will either be designed to ALWAYS split or NEVER split, although that NOT need be the case. The T-Bus WOULD support a Slave that SOMETIMES split, or split only for certain types of sequences. Of course, a Master must be able to handle a split-response if it wishes to access a split-cycle Slave.

Figure B-19 illustrates some of the previously defined terms. Cycle #1 is both a Function Request and request. Cycles #2-#4 are the response and Cycle #4 is the deferred response. Cycles #5 through n-1 (not shown) include any other type of valid T-Bus activity, including: 1) requests and responses from T-Bus Masters and Slaves and 2) dead T-Bus cycles. Cycles n and n + 1 show a specific example of these other transactions. Cycle m is where Slave B was finally ready to return its split response, and so requests the T-Bus. Figure B-19 shows that Slave B happens to receive its grant within the same cycle (Cycle m) as the request - although this will not necessarily occur. Note that Cycle m + 1 is really just an acknowledgement for the Function Request in Cycle #1. No data is transfered during Cycle m + 1 unless the split response was an "ERROR" response. In this case Slave C would present an error code to Master A.

At the end of Cycle #4, Master A, noticing that Slave C is about to defer its response, records the slave number of Slave C from the T_SOURCE field. Master A becomes a "split-master" in Cycle #5 (not shown) and Slave C becomes a split-slave. Master A must now observe ALL cycles of T-Bus activity and watch for a response from its split slave, Slave C. That response finally arrives during cycle m + 1. Master A interprets that response as if it actually occured during Cycle #4. Cycle m + 1 ends the sequence. After Cycle #4, Master A MUST desist from making T-Bus requests. Figure B-20 shows the definition of a split Master's split response.

Figure B-20 Definition — Master interprets a split response.

```
<split response> = <I am split> & <response>
                 ... & <my split Slave number>
                 ... & <split RR field>

<I am split> = "I got a deferred response
               to my Function Request"

<my split Slave number> = "T_SOURCE is the same as
                           the Slave that issued the
                           original deferred response"

<split RR field> = T_RR is not "REFUSED" or "REFUSED-LOCKED"

<response> = "defined previously"
```

B.5.7.2 Denied Responses

While Slave C is split, the Function Request is "unresolved". In terms of the T-Bus protocol, both Master A and Slave C are still in the middle of a sequence. During this time, Masters OTHER than the split Master (Master A) may legitimately make a request to split Slave C. When this occurs, Slave C must indicate to the requesting Master that its request is being "denied". It does so by issuing a "REFUSED" response which is also known as a denied response. Figure B-21 demonstrates this response in action during a 3-word read.

Figure B-21 State simulation — deferred read.

```
DEFERRED 3-WORD READ, NO PAUSING

CYC Driver    T_RR        Comments
=== ========  ==========  ================================
 1  Master A  READ        req: read 3-words from  Slave C
 3  Slave C   PROMISE     resp: got request, deferred
 .     .         .                    .
 .     .         .                    .
 .     .         .                    .
 n  Master X  READ        req: read 1 word from Slave C
n+1 Slave C   REFUSED     resp: denied!
 .     .         .                    .
 .     .         .                    .
 .     .         .                    .
 m  Slave B   <no drive>  Slave B requests and is granted
m+1 Slave B   MORE        resp: sourcing 1st word
m+2 Slave B   MORE        resp: sourcing 2nd word
m+3 Slave B   COMPLETED   resp: sourcing last word, done
```

Note from Figure B-21 that in Cycle n + 1, Slave C denied a Function Request from Master X. If Slave C responded with any other response token ("COMPLETED", "ERROR", etc.), Master A would interpret that as a response to its Function Request. However, the T-Bus protocol would allow Slave C to make a Function Response to Master A by "faking" its slave number with the reserved "anonymous Slave number": T_SOURCE<2..0> = "000". Although this is technically legal, it both violates the spirit of the split-cycles and complicates the design of the split Slave, and thus should be avoided. In addition, denied accesses have one major restriction: a responding split Slave CANNOT first respond with data and then issue the "REFUSED" response.

When a Master is issued a denied response, it may try the request again as soon as it can re-arbitrate for the T-Bus. The specific retry schedule for a denied Master is implementation dependent. However, if the Master is designed to retry immediately, Slaves MUST be given higher priority that Masters. This is so because otherwise, the Master will consume the entire T-Bus bandwidth with retries and thus never let the Slave issue its split response.

B.5.8 Locked Operations

Locked Operations describe a scenario where certain Slaves can become "locked" to certain Masters. While the Slave is locked to a Master, it becomes unable to service MOST requests from any other Master. Locking is used when a Master wishes to have eclusive use of some Slave for more than one sequence. Locking is somewhat an analogous to split cycles in that the Slave is acting on a Function Request while freeing the T-Bus for other activity.

The contrast beween locking and split cycles concerns the Slave's relationship to the Master after the initial Function Request. Split Slaves are acting on a particular Function Request and don't know (or care) anything about which Master requested it. In addition, split cycles only span one sequence. Locked Slaves, on the other hand, DO care about which Master is involved. And locked operations span at least two sequences.

Locking is divided into three distinct phases: 1) Opening, 2) maintaining, and 3) closing. These phases are now described in detail.

B.5.8.1 Opening a Lock

A Master can open a lock by making a Function Request and setting the T_LOCKOP field to an "OPEN" token. This informs the requested Slave that it is to become "locked" to that Master. The Slave records the Master's source number during the request cycle for possible later use. A typical open lock is shown in Figure B-22.

Figure B-22 State simulation — opening/bypassing and refused-locked.

CYC	Driver	T_LOCKOP	T_RR	Comments
1	Master A	OPEN	READ	req: read 1-word/lock Slave B
2	Slave B	-	COMPLETED	resp: source data, lock to A
m	Master D	OPEN	WRITE	req: write 4-words to Slave B
m+1	Slave B	-	RFSD-LCKD	resp: refused-locked
n	Master E	OPEN	WRITE	req: write 4-words to Slave B
n+1	Slave B	-	RFSD-LCKD	resp: refused-locked
o	Master E	BYPASS	WRITE	req: write 4-words to Slave B
o+1	Slave B	-	MORE	resp: sinking 1st word
o+2	Slave B	-	MORE	resp: sinking 2nd word
o+3	Slave B	-	MORE	resp: sinking 3rd word
o+4	Slave B	-	COMPLETED	resp: sinking last word

From Figure B-22 it can be seen that the read request was sucessful (no "ERROR" response) and that Slave B is now locked to Master A. In T-Bus terminology, Slave B is "locked" to Master A. Note also that two other Masters (D and E) attempted to request Slave B in cycles m and n. Slave B, knowing that it was locked to Master A, issued a "REFUSED-LOCKED" token to Masters D and E. This response is known as the "denied-locked" response and is analogous to the denied response for split-cycle Slaves.

Note that it is legitimate for a locked and split Slave to issue a denied-locked response to its own locked Master. This is just an optimization to ease design of a lockable Slave's response logic. By T-Bus protocol, the split (and locked) Master should ALWAYS wait for its Function Response before attempting another request.

Masters D and E subsequently become "observing" Masters. An observing Master releases the T-Bus immediately - in this case by not driving data during m+2 and n+2. This is a T-Bus requirement. The observing Masters MUST then wait until Master A has "finished" with its lock on Slave B before they make another T-Bus request. This "finishing" event is described later.

Note from Figure B-22 that in cycle o, Master E makes a "BYPASS" request. The bypass request is the "override" mechanism for Masters attempting to request Slaves locked to another Master. Slave B, recognizing the bypass attempt, allows the sequence to continue, effectively ignoring the T_SOURCE field. The bypass operation does NOT affect, in any way, the lock between Slave B and Master A.

B.5.8.2 Maintaining a Lock

Note that is is perfectly legal for Master A to now open a lock to a NEW Slave once its Function Response has arrived (in this case immediately). In fact, a given Master can lock as many Slaves as is practical. The open request will normally be given to Slaves that the requesting Master has not yet locked. However, it is NOT illegal for the Master to issue two consecutive open requests to a single Slave. The Slave will interpret that request as a "MAINTAIN", which is described next.

Master A can now make exclusive use of Slave B simply by making more Function Requests while T_LOCKOP field set to the "MAINTAIN" token. This indicates to Slave B that the current lock should continue. A maintain is shown in Figure B-23.

Figure B-23 State simulation — maintaining a lock.

CYC	Driver	T_LOCKOP	T_RR	Comments
1	Master A	OPEN	READ	req: read 1-word/lock Slave B
2	Slave B	-	COMPLETED	resp: source data, lock to A
m	Master A	MAINTAIN	WRITE	req: write 4-words to Slave B
m+1	Slave B	-	MORE	resp: sinking 1st word
m+2	Slave B	-	MORE	resp: sinking 2nd word
m+3	Slave B	-	MORE	resp: sinking 3rd word
m+4	Slave B	-	COMPLETED	resp: sinking last word

Figure B-24 summarizes the legal T_LOCKOP tokens that can be presented to a Slave under various conditions.

Figure B-24 Valid T_LOCKOPs between locking/locked Master and Slave.

CURRENT SLAVE CONDITION	VALID T_LOCKOP DURING REQUEST FROM LOCKING/LOCKED MASTER
UNLOCKED	OPEN, BYPASS, NORMAL
LOCKED	OPEN, BYPASS

B.5.8.3 Closing a Lock

A lock to a Slave is closed when a Master makes the "free lock" request with the "FREE-LOCK" token in the T_RR field. All lockable Slaves which are currently locked to that Master become "unlocked" in the cycle following the

request. The Slaves use the T_SOURCE field to recognize their locked Masters during the request; they do NOT need to see an implicit grant. Figure B-25 shows the definition of Locking/Unlocking for a Slave.

Figure B-25 Definition — Slave locking and unlocking.

```
<locked> := <newly locking> # (<locked> & !<my free lock>)

<newly locking> = <implicit grant> & <open lock> & !<locked>

<my free lock> = <request> & (T_SOURCE = <my locked master>)
                 ... & <free lock>

<my locked master> := <newly locking> & T_SOURCE<2..0>
                      ... # (!<newly locking> & <my locked master>)

<open lock> = T_LOCKOP is "OPEN"

<free lock> = T_LOCKOP is "FREE-LOCK"
```

A free lock Function Request is an anomoly in terms of T-Bus protocol because it is the ONLY transaction which does NOT have a corresponding response cycle. In fact, the cycle following a free lock request, by definition MUST be a dead T-Bus cycle (previously defined). Of course, arbitration can take place during that dead cycle.

ALL observing Masters "wake-up" when they observe ANY free lock request go by on the T-Bus, even though the free lock may NOT be for THEIR locked Slave. The newly "arisen" Masters then arbitrate for the T-Bus and once again attempt their Function Request. Figure B-26 shows a typical free locks.

Figure B-26 State simulation – freeing a lock.

CYC	Driver	T_LOCKOP	T_RR	Comments
1	Master A	OPEN	READ	req: read 1-word/lock Slave B
2	Slave B	-	COMPLETED	resp: source data, lock to A
m	Master A	MAINTAIN	WRITE	req: write 1-word to Slave B
m+1	Slave B	-	COMPLETED	resp: sinking last word
n	Master D	X	WRITE	req: write 2-words to Slave B
n+1	Slave B	-	RFSD-LCKD	resp: refused-locked
n+2	-	-	-	Master D becomes observing
o	Master A	FREE-LOCK	READ	req: read 1-word/lock Slave B
o+1	-	-	-	dead cycle, Master D arbitrates
o+2	Master D	NORMAL	WRITE	req: write 2-words to Slave B
o+3	Slave B	X	MORE	resp: sinking 1st word
o+4	Slave B	X	COMPLETED	resp: sinking last word, arb

Figure B-26 shows Master D re-arbitrating for the T-Bus when it observes a free lock request. Again, Master D does not care whether or not the free lock request will free-up the Slave that denied it. Master D only cares that it saw ANY free lock request. In Cycle o + 1, Master D is granted the T-Bus – but of course, some other Master or slave could have gotten it in that cycle.

Locked Sequences MUST ALWAYS end with a free lock request, otherwise observing Masters would be left "hanging". A "lock timer" should be implemented by the Master so that the lock will not be held for too long. This has major implications for Switch transactions when the SIGA is locked. The lock timer should start after the response to the "OPEN" request by the Master.

B.5.9 Locked/Split Sequences

It is certainly legal (and the norm for the SIGA) for a Master and Slave to be both locked AND in split cycles. This situation is shown in Figure B-27.

Figure B-27 State simulation – opening/bypassing and refused-locked.

CYC	Driver	T_LOCKOP	T_RR	Comments
1	Master A	OPEN	READ	req: read 2-words/lock Slave B
2	Slave B	-	PROMISE	resp: lock to A
m	Master D	NORMAL	WRITE	req: write 4-words to Slave B
m+1	Slave B	-	RFSD-LCKD	resp: refused-locked
n	Slave B	-	MORE	resp: sourcing 1st word
n+1	Slave B	-	COMPLETED	resp: sourcing last word, arb
o	Master A	MAINTAIN	WRITE	req: write 2-words to Slave B
o+1	Slave B	-	MORE	resp: sink 1st word
o+2	Slave B	-	PROMISE	resp: sink last word, promise
p	Slave B	-	COMPLETED	resp: O.K., no errors
q	Master A	X	FREE-LOCK	req: free any locks to me
q+1	-	-	-	dead cycle, Master D arbitrates
q+2	Master D	NORMAL	WRITE	req: write 2-words to Slave B
q+3	Slave B	X	MORE	resp: sinking 1st word
q+4	Slave B	X	COMPLETED	resp: sinking last word, arb

B.5.10 Stolen

The stolen bit, T_AD<32>, implies certain restrictions on T-Bus transactions. The stolen bit can never be asserted during a multi-word write – Slaves can make this assumption. For multi-word reads, an asserted stolen bit does not necessarily end the cycle. However, the Master can legitimately ignore any read words occuring after a stolen word. The Slave, on the other hand, can end the cycle on any stolen word.

B.5.11 Error Response

Sometimes during a response, a Slave recognizes that there was an error in the request or as a result of the request. If the Slave is not going to make a defer response, it MUST detect the error before it makes its first response to the Master. The Slave CANNOT transfer data (sink or source) for one or more cycles and THEN respond with "ERROR". Of course, the Slave can always pause if it can't decide immediately if there is an error or not.

The Slave can return an "ERROR" response in one of two places: 1) during a split response, in the first response cycle following the Slave grant or 2) in the first response cycle immediately following an implicit grant (the defer or

immediate response). The "ERROR" response ends the transaction, the sequence - and depending on the implementation - the operation.

For read sequences (split or non-split) the TDAT field holds the 8-bit error code that is implementation dependent. For non- split write sequences, the TDAT field is in the "wrong" direction and thus the error code cannot be presented to the requesting Master. Normally for split writes, the split response is just an acknowledgement - no data is transfered. Split write sequences, therfore give the written Slave a chance to return the error code with the split response. In fact, if returning an error code for both read and write sequences is important enough, a Slave can be design as split cycle solely for that purpose.

Figure B-28 shows the error response in three different situations.

Figure B-28 State simulation — error response.

ERROR WITHOUT PAUSE:

CYC	Driver	M_SPAUSE	T_RR	Comments
1	Master A	negated	READ	req: read 3-words from Slave B
2	Slave B	negated	ERROR	resp: sourcing error code

ERROR ON A WRITE:

CYC	Driver	M_SPAUSE	T_RR	Comments
1	Master A	negated	WRITE	req: write 2-words to Slave B
2	Slave B	negated	ERROR	resp: sinking data, can't ...source error code

PAUSED, THEN ERROR:

CYC	Driver	M_SPAUSE	T_RR	Comments
1	Master A	negated	READ	req: read 3-words from Slave B
2	Slave B	asserted	X	resp: I am waiting for you
3	Slave B	asserted	X	resp: I am waiting for you
4	Slave B	negated	ERROR	resp: sourcing error code

SPLIT RESPONSE WITH ERROR AND PAUSE:

CYC	Driver	M_SPAUSE	T_RR	Comments
1	Slave B	asserted	X	resp: I'm not ready yet
2	Slave B	negated	ERROR	resp: sourcing error code

An error resonse need not end an operation. This is up to the Master involved. Also, an error response does NOT imply a free lock request to a locked Slave that responded with the error.

B.6 Logical Signal Assignments

Figure B-29 Logical encoding — ARBITRATION subfield.

```
T_NBREQ_*       = 0, device requesting bus drivership
                = 1, device NOT requesting bus drivership

T_NBGRANT_*     = 0, device granted bus drivership
                = 1, device NOT granted bus drivership
```

Figure B-30 Logical encoding — STATUS subfield.

```
T_NDRIVEN_*     = 0, device is driving bus
                = 1, device is NOT driving bus

T_NMPAUSE_*     = 0, master is requesting pause
                = 1, master is NOT requesting pause

T_NSPAUSE_*     = 0, slave is requesting pause
                = 1, slave is NOT requesting pause

T_DRIVEN        = 0, the T-Bus is NOT currently driven
                = 1, the T-Bus is currently driven

T_MPAUSE        = 0, a T-Bus master is NOT currently pausing
                = 1, a T-Bus master is currently pausing

T_SPAUSE        = 0, a T-Bus slave is NOT currently pausing
                = 1, a T-Bus slave is currently pausing
```

Figure B-31 Logical encoding — TRANSACTION (STANDARD) subfield.

```
T_REQUEST       = 0, current cycle is a Response
                = 1, current cycle is a Request

T_SOURCE<2..0>
                T_REQUEST=1     T_REQUEST=0
                ===========     ===========
000             SIGA_AB/CNU     SIGA_AB/CSU
001             SIGA_A/SERV     SIGA_A/REQ
010             SIGA_B/SERV     SIGA_B/REQ
011             VMES            VMEM
100             CPUA            MEMORY
101             CPUB            <reserved>
110             <reserved>      <reserved>
111             <reserved>      <reserved>

T_RR<2..0>
                T_REQUEST=1     T_REQUEST=0
                ===========     ===========
000             WRITE           ERROR
001             READ            PROMISE
010             <reserved>      REFUSED
011             <reserved>      COMPLETED
100             FREE_LOCKS      MORE
101             <reserved>      EARLY-ACK
110             <reserved>      REFUSED-LOCKED
111             <illegal>       <illegal>
```

(continued)

Figure B-32 **Logical encoding — TRANSACTION (STANDARD) subfield (continued).**

```
T_SIZE<2..0>
                    T_REQUEST=1     T_REQUEST=0
                    ===========     ===========
000                 4 BYTES         -
001                 1 BYTE          -
010                 2 BYTES         -
011                 3 BYTES         -
100                 4 WORDS         -
101                 <illegal>       -
110                 2 WORDS         -
111                 3 WORDS         -

T_LOCKOP<1..0>
                    T_REQUEST=1     T_REQUEST=0
                    ===========     ===========
00                  NORMAL            -
01                  BYPASS            -
10                  OPEN              -
11                  MAINTAIN          -

T_PATH<1..0>
                    T_REQUEST=1     T_REQUEST=0
                    ===========     ===========
00                  ILLEGAL          -
01                  SIGA_B           -
10                  SIGA_A           -
11                  LOCAL            -
```

Figure B-33 **Logical encoding — TRANSACTION (EXTENDED) subfield.**

```
T_SYNC              = 0, normal message start
                    = 1, force Slot 0 message start

T_PRIORITY<1..0>
                    T_REQUEST=1     T_REQUEST=0
                    ===========     ===========
00                  EXPRESS          -
01                  FOREGROUND       -
10                  BACKGROUND       -
11                  <illegal>        -
```

Figure B-34 Logical encoding — TDAT subfield.

```
T_AD<33..0>
                    T_REQUEST=1      T_REQUEST=0
                    ===========      ===========
                    T_AD<33..25>     T_AD<33>
                    is SNN           is undefined

                    T_AD<24..0>      T_AD<32>
                    is OFFSET        is STOLEN bit

                                     T_AD<31..0>
                                     is DATA
```

Figure B-35 Logical encoding — CLOCK subfield.

```
T_CLK   rising edge is active
```

B.7 Transaction Syntax

The following shows the extended-BNF syntax of some legal T-Bus sequences. These definitions are meant to enumerate some of the lower-level T-Bus events on a cycle-by-cycle basis. They do not describe the entire set of T-Bus events. Identifiers which appear in all capitals represent subfields of T-Bus TRANS field and are terminal identifiers.

The extended BNF format include the following definitions:

Figure B-36 Definition — extended BNF operators.

```
Symbol  Definition
======  ============
::=     can be replaced by
{  }    zero or "n" occurences of enclosed
...     continue line from above
 |      or
```

The Extended BNF constructions follow:

```
<sequence> ::= <free transaction> <sequence>
             | <immediate sequence> <sequence>
             | <deferred sequence> <sequence>
             | <denied sequence> <sequence>

<free transaction> ::= <free-request> <no response>
```

```
<immediate sequence> ::= <function request> <function response>

<deferred sequence> ::= <function request> <deferred response>
                        ...<function response>

<denied sequence> ::= <function request> <denied response>
                      ...{<function request>  <denied  response>}
                      ...<function request> <function response>

<function request> ::= <request>

<function response> ::= <error>
                        | <1-word fr>
                        | <2-word fr>
                        | <3-word fr>
                        | <4-word fr>

<1-word fr> ::= <completed>
                | <EARLY-ACK> <COMPLETED>
                | <EARLY-ACK> <1 pause> <COMPLETED>

<2-word fr> ::= <more> <completed>
                | <early ack> <completed>
                | <early ack> <1 pause> <completed>

<3-word fr> ::= <more> <more> <completed>
                | <more> <early ack> <completed>
                | <more> <early ack> <1 pause> <completed>

<4-word fr> ::= <more> <more> <more> <completed>
                | <more> <more> <early ack> <completed>
                | <more> <more> <early ack> <1 pause> <completed>

<1 pause> ::= <>
              | <master_pause>
              | <slave_pause>
              | <master_and_slave_pause>

<pause> ::= <>
            | <master_pause>
            | <slave_pause>
            | <master_and_slave_pause>
            | <pause> <pause>

<more> ::= <pause> <MORE>

<early-ack> ::= <pause> <EARLY-ACK>

<completed> ::= <pause> <COMPLETED>

<error> ::= <pause> <ERROR>

<deferred response> ::= <pause> <PROMISE>

<denied response> ::= <pause> <REFUSED>

<request> ::= "as defined in this document"

<master pause> ::= T_MPAUSE

<slave pause> ::= T_SPAUSE
```

<master_and_slave_pause> ::= T_MPAUSE and T_SPAUSE

B.8 A.C. Timing Specifications

The following are the A.C. requirements for the T-Bus. It refers to a "double-standard" in that non-SIGA devices have tighter timing.

Figure B-37 A.C. timing.

TRANSACTION and TDAT signals:

Type	Parameter	NON-SIGA	SIGA
Source	TPD-max	19	23
Source	TPD-min	4	4
Sink	Setup-max	20	24
Sink	Hold-min	2	2

ARBITRATION signals:

Type	Parameter	NON-SIGA	SIGA
Source	TPD-max	8	13
Source	TPD-min	2	2
Sink	Setup-max	20	20
Sink	Hold-min	2	2

STATUS signals:

Type	Parameter	NON-SIGA	SIGA
Source	TPD-max	8	13
Source	TPD-min	2	2
Sink	Setup-max	20	24
Sink	Hold-min	2	2