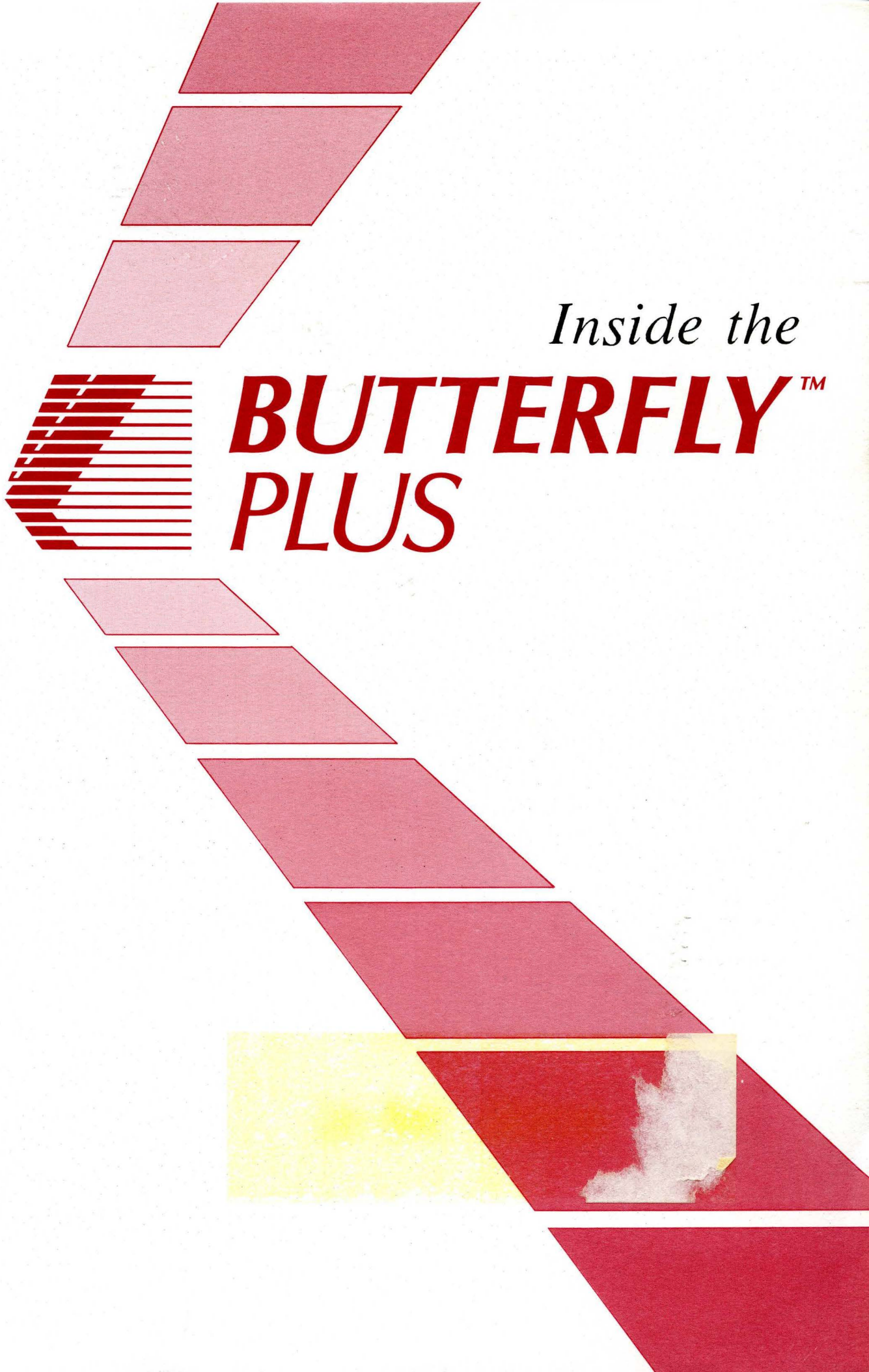


Inside the

BUTTERFLY™ PLUS

INSIDE THE

BUTTERFLY PLUS

October 28, 1987

RELEASE LEVEL

This manual conforms to the Beta Test version of the Butterfly Plus Parallel Processor released October, 1987, and Version 4.0 of the Chrysalis operating system.

NOTICE

Preface

This manual describes the theory of operation of the Butterfly Plus system. It provides detailed descriptions of the important components of the system, including the Processor Node, Butterfly Switch, Multibus Adapter, and VMEbus Adapter circuit cards, as well as an introduction to the overall design of the system. The manual also contains an introductory Chrysalis programming chapter to allow the reader a glimpse of the Butterfly Plus programming environment. This manual assumes that the reader is an engineer or programmer, sufficiently knowledgeable in the area of parallel processors.

Contents

Chapter 1 Butterfly Plus Overview

Chapter 2 The Processor Node

Chapter 3 The Butterfly Plus Switch

Chapter 4 The Multibus Adapter

Chapter 5 The VME Interface

Chapter 6 Programming the Butterfly Plus

Appendix A PNC Microcode Functions

Appendix B Physical Memory Map

Figures

Tables

Chapter 1

Butterfly Plus Overview

The Butterfly Plus parallel processor is a powerful, modular computer system composed of multiple processors and memory modules connected by a high performance network interconnect. Each processor, along with an associated memory module, occupies one circuit card called a processor node. All processor nodes in a Butterfly machine are identical; all connect to the Butterfly Plus switch in the same way and can work together interchangeably to run an application program. The Butterfly Plus is a multiple instruction, multiple data stream (MIMD) machine in which each processor can execute an independent program on local or shared memory data.

Collectively, the memory modules of all the processor nodes form the shared memory of the machine. Although each memory module is local to one particular processor node, any processor can access the local memory of any other processor by using the Butterfly Plus switch to make remote memory references. Special circuitry in each processor node interprets ordinary memory references as either local or remote memory references, as appropriate. The switch can complete a remote memory reference within a maximum of a few microseconds, regardless of where the data resides.

The distributed, shared memory architecture of the Butterfly Plus, together with the firmware and software of the Chrysalis operating system, provides a program execution environment in which tasks can be distributed among processors regardless of where the task data is located. The Butterfly Plus can be programmed in several different ways. Processors can be dedicated to individual tasks, as in a realtime system, or used as a pool of interchangeable

computing resources that are allocated to tasks dynamically. Interprocessor communication can occur through shared memory, with one processor writing data for another processor to read, or through message passing. The Chrysalis operating system and its associated set of programming languages, tools, and utilities support various different styles of parallel processing. The system software includes complete facilities for developing and debugging parallel programs.

ARCHITECTURAL FEATURES

The Butterfly Plus parallel processor has several important architectural features. Most importantly, the Butterfly Plus is an MIMD computer, able to execute multiple instruction streams on multiple data elements. The Butterfly Plus also has tightly coupled processors that allow for extensive interprocessor communications. In addition to these features, the Butterfly Plus architecture is flexible and expandable, and demonstrates a high degree of homogeneity and reliability.

Program instructions normally reside in the local memory of the processor that executes the instructions. As a result, each processor independently executes its own sequence of instructions on separate data in MIMD fashion, allowing programmers to structure programs and data in ways that are natural and efficient for their applications. The architecture supports a variety of software structures and allows Butterfly Plus systems to be used in a wide range of applications.

The processors in the Butterfly Plus are tightly coupled by the interconnect network, the Butterfly Plus switch. Tight coupling permits efficient interprocessor communication and gives each processor equal access to the global memory. As a result, programming is simplified, without sacrificing performance, by allowing programmers to organize data without undue concern for which memory module stores which part of the data. Program references to remote memory take only slightly longer to complete than references to local memory. Many applications achieve best performance when their data is scattered uniformly throughout the machine.

The Butterfly Plus is expandable over a wide range of configurations. Each processor node added to a Butterfly configuration contributes an increment of processing power, memory, and switch bandwidth. A Butterfly Plus system

can be configured with up to 256 processors to match the computational power needs of a particular application. Although the number of components increases with expansion, the complexity and cost of the interconnect network always remain at an acceptable level.

The Butterfly Plus is a homogeneous multiprocessor where each processor node is the same as every other. The uniformity of the Butterfly Plus architecture simplifies programming, since programmers need not allocate certain tasks to specific processors. Programmers can also write application software without concern for the number of processors that will be available to run the program. Such application software has several advantages. It can be developed and tested on small, inexpensive configurations, then run on larger, operational machines. Also, if the resource demands of an application exceed the capacity of its current Butterfly configuration, the application can be moved to a larger, more powerful Butterfly configuration and run without reprogramming. Application software written in this way improves overall system reliability and availability, since it can run on configurations reduced by missing or failed components.

The Butterfly Plus architecture is relatively insensitive to component failures, resulting in a reliable system. The machine can operate despite the absence of one or more of its processor nodes. A 128-processor machine can still run at approximately 98% of its capacity, for example, even after three of its processors are removed.

SYSTEM COMPONENTS

Butterfly Plus system components occupy from one to six freestanding 80-inch high racks in which circuit cards, card cages, and peripheral devices are mounted. Racks have hinged doors both at the front, for access to the circuit cards, and at the rear, for access to the power distribution and cooling systems. Three types of card cages can be installed inside the rack. Butterfly card cages hold processor nodes, VME nodes, switch cards, and clock cards. Multibus card cages hold standard Multibus cards, and VME card cages hold standard VME cards. Figure 1–1 illustrates a block diagram of the Butterfly Plus system. Every Butterfly Plus system has at least one Butterfly card cage and one Multibus card cage. Larger systems have several Butterfly card cages but rarely require a second Multibus card cage. The VME card cage is always optional. Table 1–1 lists the minimal number of circuit cards and card cages

required for various system configurations. Some Butterfly Plus configurations may require more cards and card cages than are listed.

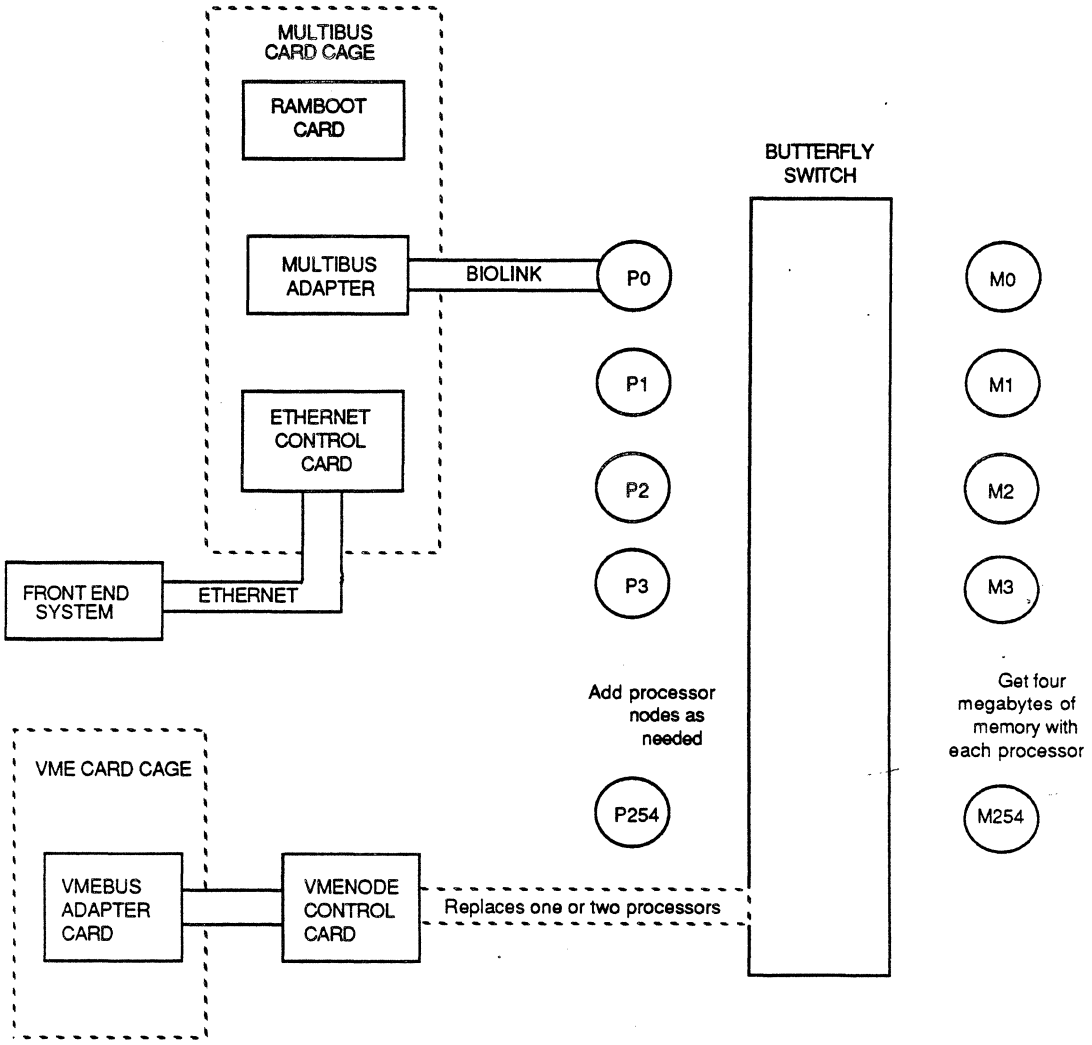


Figure 1-1
Butterfly Plus System Diagram

Table 1-1
Minimal Butterfly Configuration Guide

Racks	Processor Nodes	Butterfly Card Cages	Switch Cards	Clock Cards
1	1–16	1	1	0
	17–32	2	4	1
2	33–48	3	6	3
	49–64	4	8	3
3	65–80	5	10	4
	81–96	6	12	4
4	97–112	7	14	5
	113–128	8	16	5
4	129–144	11	22	8
	145–160	12	24	8
4	161–176	12	24	9
	177–192	13	26	9
5	193–208	14	28	10
	209–224	14	28	10
6	225–240	15	30	11
	241–256	16	32	11

A power distribution unit at the bottom of each Butterfly rack accepts 5-wire, 3-phase, 208-volt AC power from the wall outlet and distributes this AC power to the main Butterfly power supply, located at the top of the cage. This main power supply rectifies the 3-phase AC power and distributes the resulting unregulated DC power to the individual power converters located on every processor node and switch card. Aside from the main AC power distribution cable that carries 3-phase AC power to the main power supply, the power distribution unit has six 120-volt AC outlets from which peripherals mounted in the rack can draw AC power. Three neon lamps on the top of the power distribution unit indicate the state of the three phases of AC power. A central circuit breaker enables and disables AC power to the rack.

Butterfly Plus Card Cage

A Butterfly Plus card cage normally holds three types of circuit cards: processor nodes and clock cards, which mount vertically in the lower portion of the card cage, and switch cards, which mount horizontally at the top of the card cage. (An optional VME node replaces one or two of the processor nodes when a VME card is installed in the rack.) A rack normally holds up to two Butterfly card cages, each with 18 vertical slots and two horizontal slots. The horizontal slots accept only switch cards. The clock card, if present, must occupy the rightmost vertical slot. Up to 16 processor nodes normally occupy all but one of the remaining vertical slots.

The Butterfly card cage has an integral main power supply that accepts 3-phase AC power from the power distribution unit at the bottom of the rack and supplies unregulated DC power to the individual switching power supplies on each processor node, switch, and clock card. It also has an upper and lower fan bank for cooling, each with its own filter.

The Butterfly card cage has two AC circuit breakers and five DC circuit breakers. The two AC circuit breakers are located at the rear of the Butterfly card cage, above the lower fan assembly. The one on the left is a 4-handle breaker and the one on the right is a 3-handle breaker. The 3-handle breaker protects against current faults in either of the two cooling fan assemblies and trips the 4-handle circuit breaker when it detects a fault. If the 3-handle breaker is not in its on (up) position, the 4-handle breaker cannot be turned on. The 4-handle breaker detects current overloads on each of the three AC power phases and also detects ripple and phase faults.

The five DC circuit breakers are arranged in a row along the top of the front panel of the Butterfly card cage, between the processor nodes (installed vertically in card cage slots) and the lower Butterfly Plus switch card (installed horizontally in card cage slots). These breakers detect current or power cabling faults in the unregulated DC power that the DC power supply at the top of the cage provides to the Butterfly circuit cards. Each DC circuit breaker is a 20-ampere magnetic breaker that can power down a string of four processor, clock, or switch cards, allowing these cards to be removed without powering down the entire card cage.

A Butterfly Plus is shipped with its processor nodes already installed in the Butterfly card cage and secured by a shipping bracket to prevent accidental slipping. This shipping bracket may be left in place or removed upon arrival.

Multibus Card Cage

A Multibus card cage has nine horizontal slots. There are always at least three circuit cards in the first Multibus card cage: the Butterfly Plus Multibus adapter, which must occupy the topmost slot, a RAMboot card, and an Ethernet controller card. The lowest slot in the Multibus card cage cannot be occupied by a Multibus device requesting to be a master, and is generally kept empty.

Below the Multibus card cage is the Ethernet fantail, a narrow panel mounted horizontally, with cutouts for various types of cable connectors. The Multibus Ethernet controller card attaches to a DB-15 connector on the lefthand side of the fantail via a cable. Adjacent to the DB-15 Ethernet connector are two RS-232 host and console terminal ports. The UARTs of the Multibus adapter card connect to these two RS-232 connectors. The Ethernet fantail also provides a second Ethernet cable connector and a second pair of host and console terminal ports for use if a second Multibus card cage is needed. A bank of four additional RS-232 ports on the righthand side of the Ethernet fantail includes a pair of console and host ports on the left and right of the bank. These two ports terminate cables that attach to any processor node's diagnostic UART in the rack via that processor node's J4 connector. The middle two RS-232 connectors in this bank of four are unused, as are the two banks of RS-449 connectors. The Butterfly Plus needs only the ground, transmit data, and receive data signals on its RS-232 serial lines, all of which are standard DCE connections that accept male DB-25 connectors.

Processor Node

A Butterfly Plus can have up to 256 processor nodes. Each processor node is a single circuit card with the following components:

- MC68020 microprocessor with MC68881 floating point coprocessor and MC68851 paged memory management unit (PMMU)
- Four megabytes of main memory with memory controller

- 128-kilobytes of programmable read only memory (EPROM)
- Address decoding logic
- Processor node controller (PNC)
- Dual UART
- I/O bus adapter
- Interface to the Butterfly Plus switch
- Switching power supply.

These processor node components communicate with one another over a fast internal bus. Figure 1–2 shows the processor node.

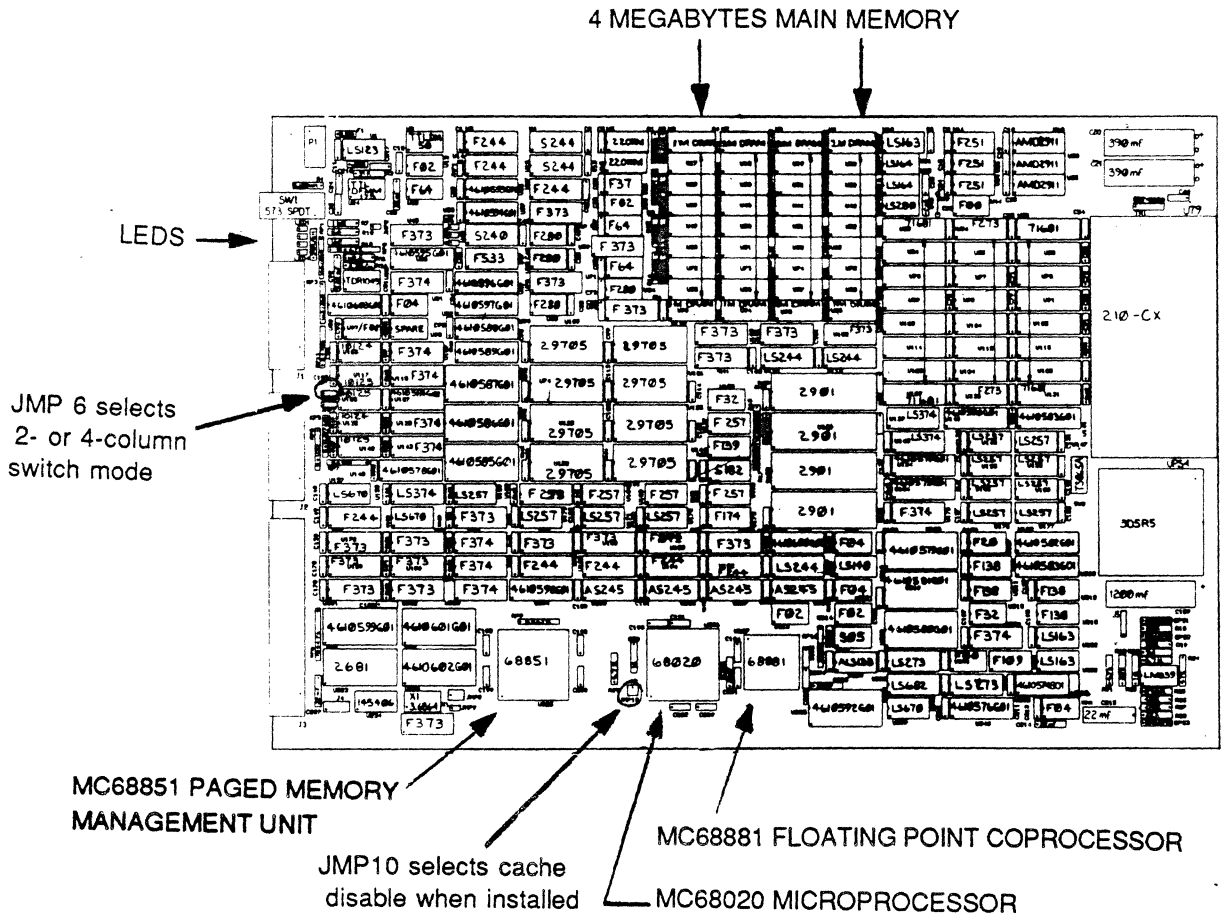


Figure 1-2
Processor Node

The MC68020 microprocessor is the main processing unit of a processor node. Associated with each MC68020 is an MC68881 floating point coprocessor. Paged memory management is provided by an MC68851. These three chips, combined with the address decoding logic, make up the primary computational elements of a processor node.

Main Memory

Each processor node has four megabytes of semiconductor dynamic random access memory (DRAM) for local and remote memory accesses. The four megabyte memory bank serves as the local memory of its processor node, which accesses that memory directly over a fast local bus. Any processor node in the system can remotely access the local memory of any other processor node by using the Butterfly Plus switch. Together, all the processor node memories make up the shared memory of the Butterfly Plus.

Circuitry in each processor node monitors the high-order physical address bits during a memory reference to determine whether the memory access is local or remote. If the high-order address bits match the processor number, the access is a local one and can be made directly, without involving the switch. If these address bits differ from the processor number, the access is a remote one and must be made through the switch. From the viewpoint of a program running on one processor, the only difference between references to local memory on its own processor node and references to remote memory on other processor nodes is that remote references take slightly longer to complete. Typical memory reference instructions that access local memory take about one microsecond. Those accessing remote memory take about five microseconds. Speeds of the processors, memories, and switch are balanced to let the system work efficiently in a wide range of configurations. The memory bandwidth is 102-Megabytes-per-second.

Every local memory access takes three cycles. The first cycle latches the memory address and type of operation (*e.g.*, read, word write, byte write). During a write operation, the new memory data is latched during the second cycle, thereby freeing the rest of the processor node to perform some other, unrelated operation during the third cycle. During a read operation, the memory returns the data to the processor node during the third cycle, leaving the rest of the processor node free to perform other work during the second cycle. The rest of the processor node is occupied during only two-thirds of the time it takes to access the memory.

Each four megabyte memory module has 36 one megabit DRAM chips organized into four banks of 8-bit bytes with one parity bit. A one megabit dynamic memory chip is a 1024×1024 array of binary cells. The memory controller selects particular cells by asserting first a 10-bit row address, derived from the low-order address bits, and then a 10-bit column address, derived from the high-order bits.

Memory cycles are pipelined to maximize performance. When a processor begins to execute a memory reference instruction, the memory controller starts a memory cycle by strobing in the row address even before the memory reference is fully decoded. If the address decoding logic finds that the address is not a memory location, the memory controller aborts its cycle without ever strobing in the column address.

Byte parity provides single bit error detection to protect main memory data. When data is written into memory, a parity bit is calculated and stored with each byte. After the data is read back from memory, its parity bits are again calculated and compared with the stored parity bits. If they differ, a processor level 6 interrupt is requested.

Switch Card

Switch cards combine the techniques of packet switching and sorting networks to establish communication paths between processor nodes. Besides routing data, switch cards also distribute clock and system reset signals to the processor nodes. The switch operates much like a packet switched network, formatting requests to read or write memory locations into data packets, then routing them directly from the processor node that initiated the request to the processor node where the data resides. Like most packet switched networks, the Butterfly Plus switch can establish and hold many different communication paths simultaneously. Every processor node can read or write a memory location through the switch—all at the same time—and provided that these simultaneous memory accesses do not conflict with one another, all can be performed in parallel within the same time it would take one processor to read only one memory location. Named after the fast Fourier transform butterfly, which it resembles, the Butterfly Plus switch can be expanded to accommodate up to 256 processors.

Conceptually, the switch consists of an array of switch nodes that connect to one another and to the processor nodes. Each input to a switch node is driven by the output from a processor node or another switch node. Similarly, each output from a switch node in turn drives a processor node or another switch node. The switch logic uses sophisticated mechanisms for routing, timing, flow control, and collision resolution to transport packets reliably from switch node to switch node.

Physically, the switch is made up of switch cards plus the cables that connect the switch cards to one another and to the processor nodes. One or two switch cards can mount directly into the Butterfly card cage horizontal slots just above the processor node vertical slots. A switch card contains eight complete switch nodes, each with four inputs and four outputs, arranged in two columns of four nodes. A metal panel mounted at the front of each switch card has 32 connectors, 16 for input to the four switch nodes in the first column and 16 for output from the four switch nodes in the second column. A Butterfly Plus system with up to 16 processors needs one switch card, which implements a 2-column switch. Butterfly Plus systems with from 17 to 32 processors need four switch cards connected in pairs to implement a 4-column switch. Larger systems extend the 4-column switch by adding two additional switch cards for every 16 processors. Systems can be typically configured with extra switch nodes to provide alternate paths between processor nodes.

Figure 1–3 shows the Butterfly Plus switch card.

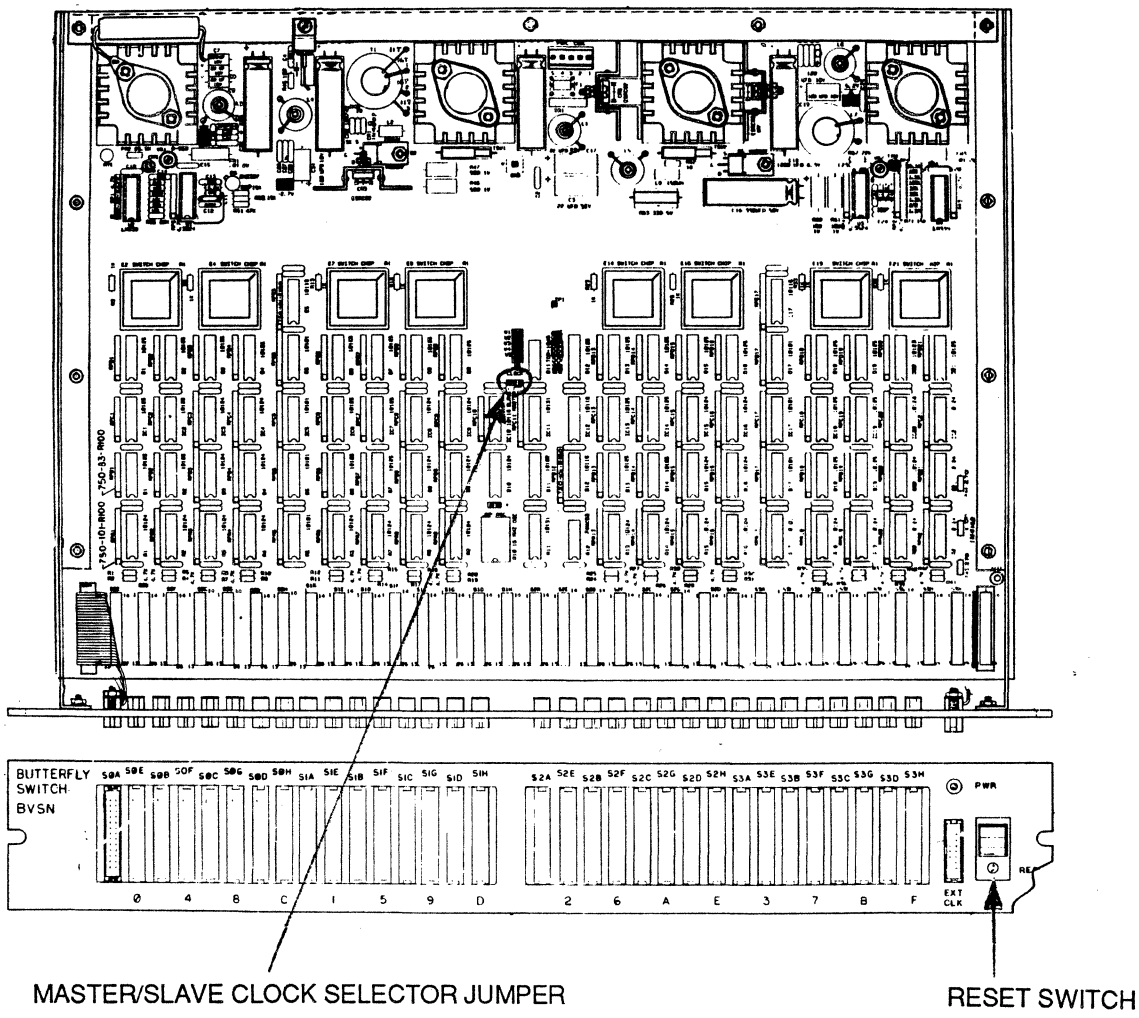


Figure 1-3
Switch Card

Clock Card

The Butterfly Plus is a fully synchronous machine in which all processor nodes use the same clock source. Systems with up to 16 processors use the single switch card clock as their system clock source. They do not need separate clock cards. Systems with more than 16 processors need a separate clock card and use the switch to distribute the system clock signal from this central source. The clock card occupies the rightmost slot of a Butterfly card cage. The Butterfly Plus clock card can operate as either a master or a slave.

As master, it transmits a differential, emitter coupled logic signal, derived from an onboard oscillator, through the four connectors near the top of the card. The system reset signal, activated by the toggle switch near the top of the card, is distributed along with the clock signal. The four connectors mate with mass terminated cables that carry the clock and reset signals throughout the Butterfly Plus. One clock card can supply clock and reset signals to up to four switch cards (*i.e.*, two Butterfly card cages), which then redistribute these signals to their attached processor nodes.

When operating as a slave, the clock card receives system clock and reset signals from a master clock card through a connector on the front edge of the card, regenerates them, and passes them through its four output connectors. Systems with more than 32 processors have more than four switch cards and therefore require more than one clock source. Master and slave clock and reset signals are distributed through a hierarchical tree of clock cards in these larger configurations. The root clock card operates as master and sources the clock signals used by the next level of clock cards. Remaining clock cards operate as slaves, accepting signals from the previous level in the tree of clock cards and passing them on to the next level. Systems with 33–64 processors use a 2-level tree, and those with more than 64 processors use a 3-level tree.

There are two LEDs and a master reset toggle switch at the top of every clock card. The master reset switch asserts a system reset signal that resets every processor node. The red LED is lit to indicate that a system reset is in progress. The green LED is lit to indicate that power is applied.

INPUT/OUTPUT CAPABILITIES

The Butterfly Plus parallel processor supports two types of I/O devices: Multibus peripherals and VMEbus peripherals. The Multibus adapter card occupies the topmost slot in a Multibus card cage housed inside a Butterfly rack. It communicates with the chosen processor node (designated as the king node) via a ribbon cable that attaches to the I/O connector at the bottom of the processor node. The VME interface consists of two cards, one that fits into a VME card cage inside a Butterfly rack and one that replaces a processor node in one of the Butterfly card cage slots. Two ribbon cables connect these two VME interface cards, and another set of cables attaches the VME interface card in the Butterfly card cage to the Butterfly Plus switch.

Butterfly Plus I/O Link

Multibus I/O devices communicate with a processor node via a synchronous bus, called the Butterfly Plus I/O link (BIOLINK), that attaches the Multibus adapter to the processor node. Using the BIOLINK, a Butterfly Plus processor can access memory on the Multibus. The Butterfly Plus processor can also read or write the control registers and data registers of Multibus peripheral devices. On the other hand, a Multibus device can read or write the four megabytes of memory on the processor node to which the Multibus adapter is attached; it can issue an MC68020 interrupt request and supply an interrupt acknowledge response word to the Butterfly Plus processor; and it can also execute special multiprocessing functions on the processor node to which it is attached.

Multibus Adapter Card

The Multibus adapter contains the circuitry for connecting a Butterfly Plus processor node to I/O cards that conform to the IEEE 796 (Multibus) standard. The Multibus adapter is a printed circuit card that occupies the top slot in the Multibus card cage and attaches to the processor node I/O connector via a ribbon cable. The Multibus adapter supports local data transfers, event posting, and interrupt processing. Each Multibus interrupt request level can be mapped into one of two processor interrupts. Multibus interrupts can be individually enabled, and there is a general interrupt disable feature.

Two Multibus cards are included in every Butterfly Plus parallel processor: The RAMboot card (a Micro Memory MM–9000D Dynamic RAM Memory Module) and the Ethernet Controller Card (an Excelan EXOS 301 Ethernet Front-End Processor). Both of these Multibus cards are off-the-shelf IEEE-796 compatible. Both can occupy any slot in the Multibus card cage except for the top slot, which holds the Multibus adapter card, and the bottom slot, which cannot be occupied by a Multibus master. Figure 1–4 shows the Multibus adapter card.

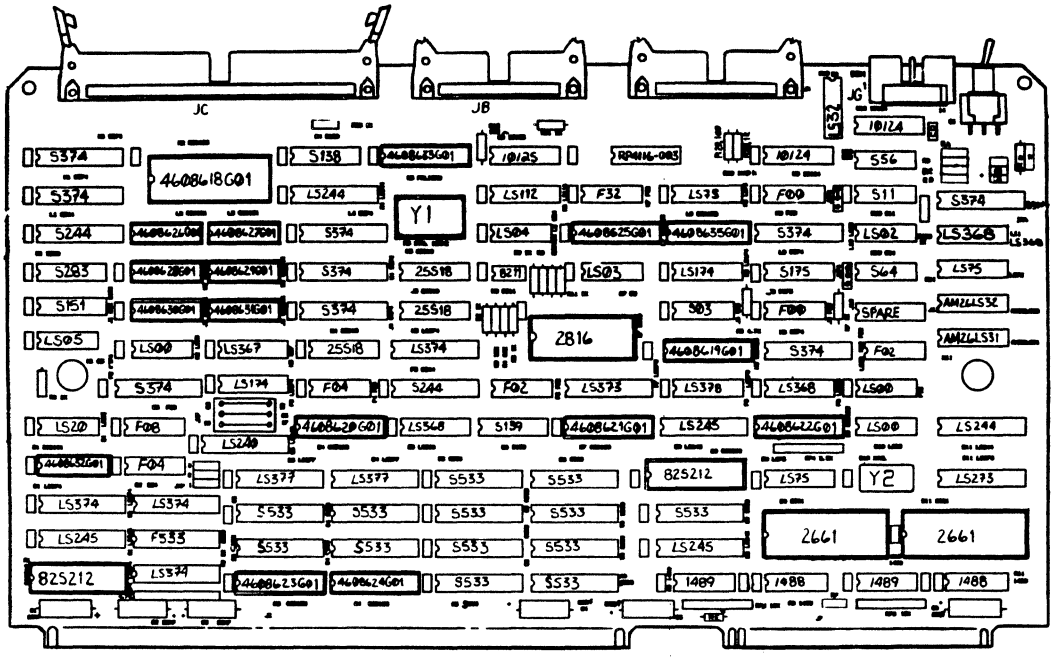


Figure 1-4
Multibus Adapter Card

Multibus RAMboot Card

The Multibus RAMboot Card is a DRAM memory that stores the operating system and other program code needed to perform a Butterfly Plus system reset as quickly as possible when necessary. Provided with one megabyte of memory, the RAMboot card can be expanded to four megabytes by filling empty chip sites on newer boards.

Multibus Ethernet Controller Card

The Multibus Ethernet controller card is an IEEE 802.3 local area network controller that interfaces the Butterfly Plus parallel processor to an Ethernet (version 1 or version 2) or other IEEE 802.3 local area network. The Ethernet controller card attaches to the local area network via a DB-15 connector on the Ethernet fantail, which allows the connection of a customer-supplied Ethernet transceiver. The processor node communicates with the Ethernet card via supplied control and data registers. It supports the industry standard TCP/IP protocol using the Ethernet controller card.

VME Interface

The VME interface supports high speed I/O devices whose bandwidth requirements exceed the I/O capacity of a processor node. With this interface, many industry standard, high bandwidth devices, such as frame grabbers, can interface to the Butterfly Plus efficiently. The VMEbus is a 32-bit bus with a peak data transfer rate of 40-megabytes-per-second and a typical data transfer rate approaching 20-megabytes-per-second.

The VME interface transfers data directly to and from the Butterfly Plus switch without passing the data through a processor node, as does the Multibus adapter. Direct attachment of the VME interface to the switch has several advantages. With the VME interface attached directly to the switch, I/O transfers can be spread across memory on many processor nodes, thereby reducing the load on any particular memory module. Also, the 20-megabyte-per-second nominal bandwidth of the VMEbus far exceeds the I/O bandwidth. Attaching the VME interface directly to either one or two switch ports allows the bandwidth of the connection between the Butterfly Plus switch and the VMEbus to be scaled up to about 20-megabytes. Physically, the VMEbus interface consists of two circuit cards connected by a cable up to 20 feet long. One of these, the VME Bus Adapter, is a double height VME card that plugs directly into the VME card cage. The other, the VME Node Controller, generally replaces two processor nodes in the Butterfly card cage and has four connectors that attach via ribbon cables to two separate ports on a Butterfly Plus switch card. Logically, the VME interface contains two complete interfaces to the Butterfly Plus switch, an interface to the VMEbus, and a microengine that translates between VMEbus transactions and Butterfly Plus switch messages.

Figure 1–5 shows the VME Node Controller and Figure 1–6 shows the VME Bus Adapter card.

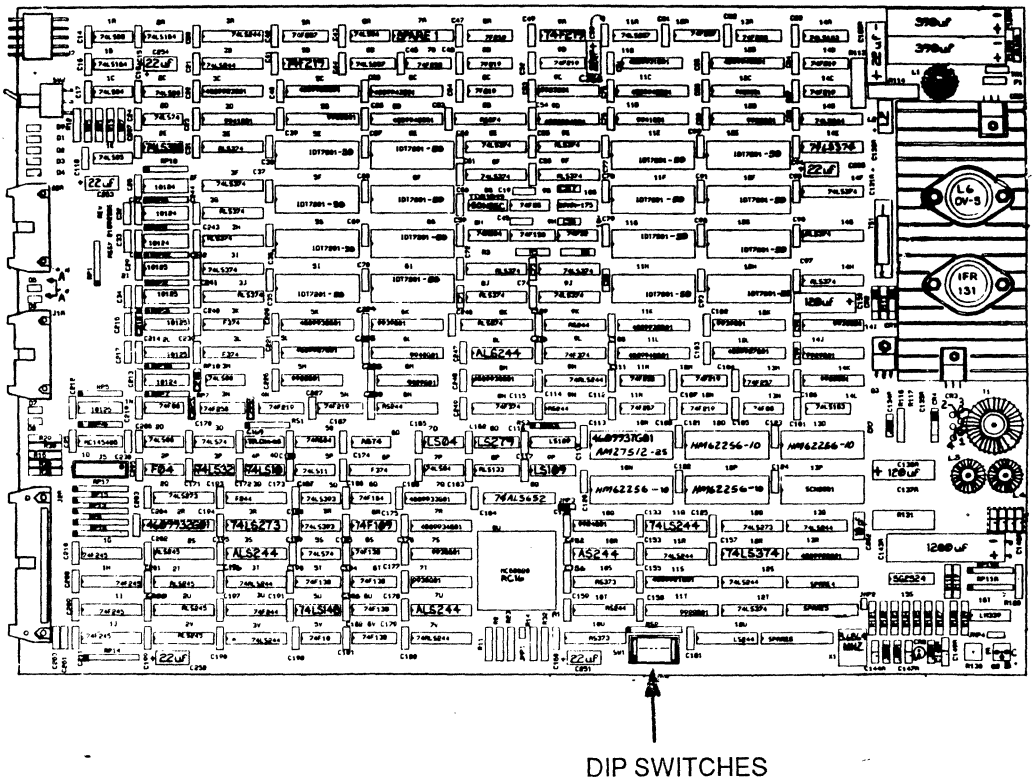


Figure 1-5
VME Node Controller

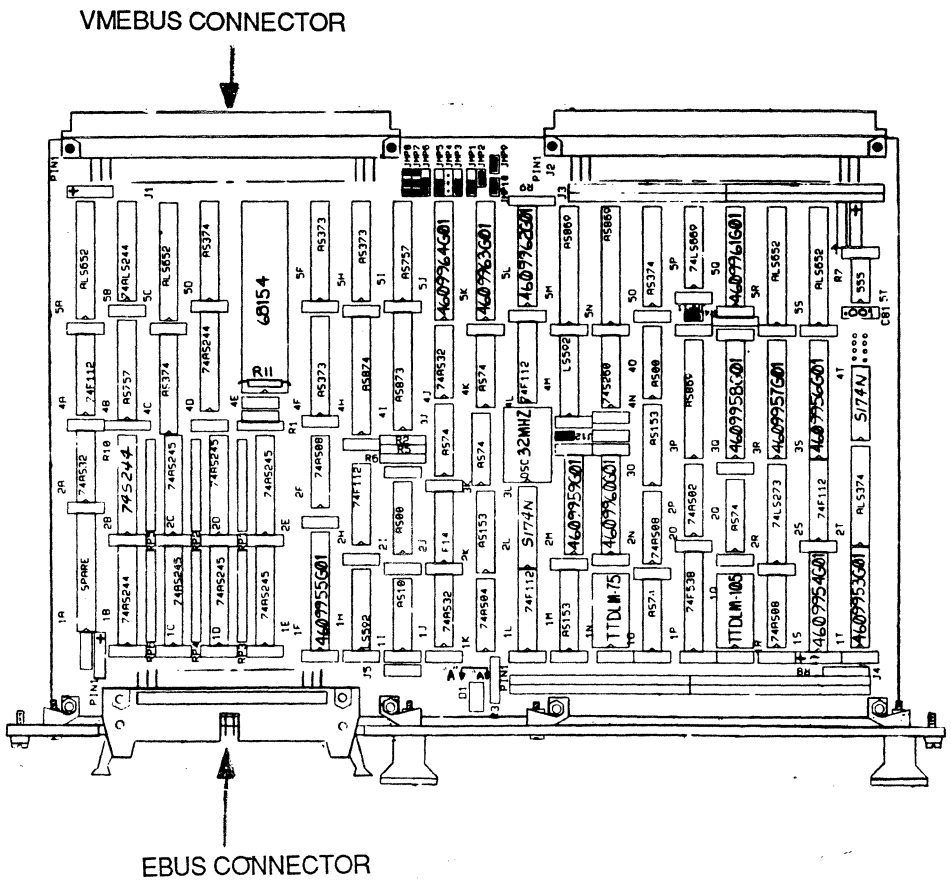


Figure 1-6
VME Bus Adapter Card

SYSTEM RESET

A Butterfly Plus reset can occur when the system reset switch is activated, when a restart message is received from another processor, when one of the power supplies detects that the power bus input voltage is too low, or when circuitry on the processor node detects an abnormal condition that can impair proper functioning of the board. In systems with up to 16 processors, the system reset switch is located on the Butterfly Plus switch card. In systems with 17–32 processors, the reset switch is located on the clock card. The system reset switch for machines with more than 32 processors is the reset switch on the master clock card, which generally occupies the rightmost slot in the

lower Butterfly card cage of the second to last rack (*i.e.*, the rack to the right of the leftmost rack). Operating the system reset switch initializes all processor nodes and switch nodes without clearing main memory. Except for the fact that memory data is preserved, a system reset has the same effect on the processor nodes and the switch as a momentary power outage. Toggling the system reset switch also resets the Multibus card cage. However, it does not reset the VMEbus.

PACKAGING

A 128-processor Butterfly Plus system, including the Multibus and VMEbus cages, occupies four Butterfly racks, each 36 inches deep by 24 inches wide by 80 inches high. Allow 40 inches by 100 inches of floor space for each Butterfly rack. This provides sufficient clearance to open the front and back doors, which are 19 inches wide. An additional 24 inches is required for the machine to be serviced.

The Butterfly Plus switch can be configured to provide any level of bandwidth needed for the machine. Alternate switch paths can be provided to reduce vulnerability to single point failures and potential contention. A 16-processor system uses a 1-card switch, for example, and a 64-processor system uses a switch distributed over eight cards.

Table 1–2 lists the specifications for the Butterfly Plus.

Table 1-2
Butterfly Plus Product Specification

Power Requirements	AC Plug	NEMA L21-30
	AC	5-wire, 3-phase power at 208 volts
	Per Processor Node Dissipation	65 watts
	Power Dissipation for 16-Node System	1450 watts
	Power Dissipation for 32-Node System	3275 watts
Operating Temperature	20 - 40°C	
Humidity	40% to 70% relative humidity, non-condensing	

Chapter 2

The Processor Node

A Butterfly Plus parallel processor can have up to 256 processor nodes connected by the Butterfly Plus switch. Each processor node contains an MC68020 microprocessor, MC68881 floating point coprocessor, MC68851 paged memory management unit, four megabytes of main memory, an AM2901 bitslice processor used as the processor node controller (PNC), an I/O bus adapter, a UART that drives the serial lines to the host and console diagnostic terminals, a bootstrap EPROM, and an interface to the switch. A local bus connects all of these various processor node elements, providing flexible control and minimal complexity. Figure 2–1 is a block diagram of the processor node.

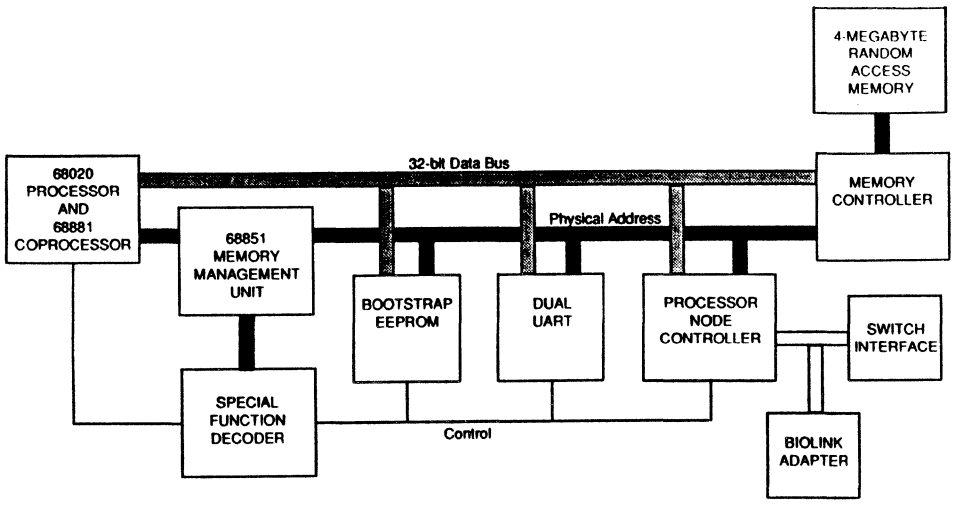


Figure 2-1
Processor Node Block Diagram

Each processor node consists of one 12 by 18 inch printed circuit card with an onboard switching power supply. Four cable connectors on the card edge provide unregulated DC power (upper, 5-wire connector), transmit and receive channels to the switch (the transmit connector is nearest to the power connector), and a BIOLINK channel that attaches to the Multibus adapter (the largest connector). A fifth connector, the J4 connector located near the card edge at the corner of the card, attaches to the dual UART for use by diagnostics.

Each processor node has four LEDs and one toggle power switch. The top LED is red and is controlled by software. The remaining three LEDs are green and are controlled by hardware. The red LED comes on at power up, or whenever the processor node is reset, to indicate that the processor node is running its power-up diagnostic tests. When the processor node completes its built-in diagnostic test, the red LED turns off to indicate that the processor node is now functioning properly. If the red indicator flashes or stays lit continuously, the card is probably faulty and must be checked.

The second LED is green and lights up when the processor node on-board power supply or regulator generates +5 volts of DC power. There can be live DC power on the processor node even if this green indicator light is off. Unregulated DC power might still be supplied from the power connector and distributed across the circuit card.

The third and forth LEDs are green and light up to indicate that a message is either being sent to the switch or being received from the switch, respectively.

PROCESSING ELEMENTS

The primary functional subsystem of a processor node is its processing elements group consisting of the main processor and floating point coprocessor, the paged memory management unit, the address decoding logic, and an arbiter for the processor node's internal bus. The MC68020 main processor and the MC68881 floating point coprocessor both run at their maximum rated speed of 16 MHz.

Processor

All Butterfly Plus application programs run on the MC68020 processor with MC68881 floating point coprocessor. These processing elements combine state-of-the-art technology and advanced circuit design techniques to achieve an architecturally advanced processing unit. The primary features of the MC68020 include:

- 32-bit data and address registers
- 16-gigabyte direct addressing range
- A wide variety of instruction types
- Five data types
- Memory mapped I/O for accessing peripheral devices through the main memory address space
- 14 memory addressing modes.

Refer to the *Motorola MC68020 32-Bit Microprocessor User's Manual, Second Edition* for further information.

Floating Point Coprocessor

The MC68881 floating point coprocessor implements the full IEEE 754 specification for floating point arithmetic. It uses the short (32-bit), long (64-bit), and extended (80-bit) number formats and implements microcoded square root, trigonometric, logarithmic, and transcendental functions. Since the MC68881 uses the MC68020 coprocessor facility, hardware floating point operations can be invoked by executing ordinary MC68020 instructions. The major features of the MC68881 floating processor include:

- Eight general purpose floating point data registers, each supporting a full 80-bit extended precision real number format
- A 67-bit arithmetic unit for fast calculation with intermediate precision greater than the extended precision format
- A 67-bit barrel shifter for fast shifting operations
- 46 instructions, including 35 arithmetic operations

- Support for trigonometric and transcendental functions
- Seven data types and 22 constants
- Virtual memory and virtual machine operations
- Instruction execution is fully concurrent with the main processor.

Refer to the Motorola *MC68881 Floating-Point coprocessor User's Manual, First Edition* for further information.

MEMORY MANAGEMENT UNIT

The MC68851 paged memory management unit supports the paged virtual memory of the MC68020. It translates the virtual addresses used by the processors into the physical addresses used by the PNC, memory, switch, and bus adapters. At the start of a memory read or write, the memory management unit receives a 32-bit virtual address from the MC68020 processor. It uses page descriptors stored in its address translation cache to translate this virtual address into a 32-bit physical address. The MC68851 also handles all access privilege checking and allows reprogramming of access privileges during operation. Refer to Appendix B, "Physical Memory Map", for a physical memory address table.

All Butterfly Plus components except the MC68020 processor and MC68881 floating point coprocessor use physical memory addresses. A 32-bit Butterfly Plus physical address is formatted as an 8-bit processor node number followed by the 24-bit local address of a location in the four megabyte memory module of that processor node. The processor and floating point coprocessor use virtual addresses, and since these are the two processing elements that execute most program instructions, virtual addresses are the only addresses that normally concern a programmer. The main distinction between physical and virtual addresses is that the physical address of a location is the same on all processor nodes, whereas virtual addresses can differ from one process to the next.

The operating system establishes the layout of the virtual address space. Since pages are an integral power of two in size, the low-order bits of a virtual address are identical to the low-order bits of the corresponding physical address. By convention, the text segment of a Chrysalis program usually starts at 40000 and the data segment usually starts at 20000. This is not a

requirement of the system, but merely a convention. The upper 16-megabytes of the virtual memory address space (addresses 0xFF000000 to 0xFFFFFFFF) are reserved as a special function subspace used to access certain memory mapped devices such as EPROM and UART, PNC special functions and other microcoded control functions, and Multibus I/O address space and memory. References to the special function subspace are translated into physical addresses within the memory management unit, which generates a 32-bit physical address if the access is allowed.

ADDRESS DECODING

Not all memory reference instructions access memory; some access memory mapped circuits that respond to memory reference instructions but do not necessarily store data. In general, the processors and other programmable components on a processor node can make four types of memory references:

- Dialog between the MC68020 processor and either the MC68851 paged memory management unit or the MC68881 floating point coprocessor
- Auxiliary system functions, such as executing code from EPROM, configuring main memory, downloading microcode to the writable control store of the processor node controller, or using the diagnostic UART serial channels to communicate with the host and console terminals
- Local memory access
- Any activity mediated by the processor node controller, including remote memory access; reading or writing Multibus memory or I/O device registers over the BIOLINK; and special functions like block transfer from one processor node to another, atomic queuing operations, and event posting.

PROCESSOR NODE FUNCTIONS

A processor node can perform many operations at the same time. As an example, one processor node can execute all of the following operations simultaneously:

- Add two integers in the MC68020 processor
- Divide two real numbers in the MC68881 floating point coprocessor

- Receive a message from a different processor node
- Send one word of a block of data to a different processor node
- Arbitrate a Multibus device request to access Butterfly Plus memory.

In high-throughput environments these multiple concurrent operations are both normal and desirable. The processor node provides adequate communication bandwidth between the various autonomous hardware resources while controlling their interactions. In addition to these conventional computer functions, a parallel processing architecture requires many special functions that the MC68020 processor does not provide. These special processor node functions include:

- A 32-bit time-of-day clock and timer, capable of interrupting the processor, for scheduling timed or delayed events
- DRAM refresh logic
- Conversion of remote memory references to switch messages
- Block memory transfer between processor nodes
- Remote restart of processor nodes from other processor nodes
- Interprocess communication among multiple processor nodes
- Message timeout handling for transmissions and solicited replies
- Processor interrupt acknowledgement and interrupt vector handling
- Error reporting of timeouts, checksum errors, and parity errors
- Event posting for the operating system
- Power fail interrupt and power-up sequence generation
- Interaction with I/O device controllers when performing direct memory access and special functions
- Process scheduling.

However, most of these special functions are implemented by microcode in the processor node controller; the remainder are implemented by the operating system.

INTERRUPT SYSTEM

The MC68020 responds to seven levels of prioritized, vectored interrupt requests. Level 7, the interrupt request level with highest priority, is the only level that is not maskable. When an interrupt is requested, the interrupt system software uses an interrupt vector supplied by the requestor to find the address of the appropriate interrupt service routine. Multiple interrupt requests can be pending simultaneously at all priority levels. Interrupt requests below a certain priority level can be masked off and ignored. The highest priority interrupt request triggers an interrupt if interrupts are enabled at that priority level. The highest priority element requesting an interrupt at that level then supplies an 8-bit interrupt vector to the processor.

The Butterfly Plus allocates the seven interrupt request levels, from highest to lowest priority, as follows:

Level 7: PNC and switch errors and power failure
Level 6: Memory parity error and diagnostic UART
Level 5: EPROM based debugger (USD) interrupt request
Level 4: High priority Multibus device interrupt request
Level 3: Low priority Multibus device interrupt request
Level 2: Interval timer interrupt request
Level 1: Scheduler interrupt (process scheduler request).

PROCESSOR NODE CONTROLLER

The processor node controller (PNC) consists of four basic subsystems. These include a microcode sequencer, a static RAM control store, an interrupt service routine address generator, and a bitslice processor. The main processing unit of the PNC is the AM2901 bitslice processor, which controls the various processor node resources and performs those functions that, because of throughput or indivisibility requirements, cannot be performed by the MC68020 processor. Microcode for the AM2901 is stored in an EPROM, from which it is downloaded into a 4096-word array of 64-bit static RAM control store. The AM2911 microsequencer controls the flow of the microprogram guided by the microcode. Figure 2–2 is a block diagram of the PNC showing its main components and how they interact.

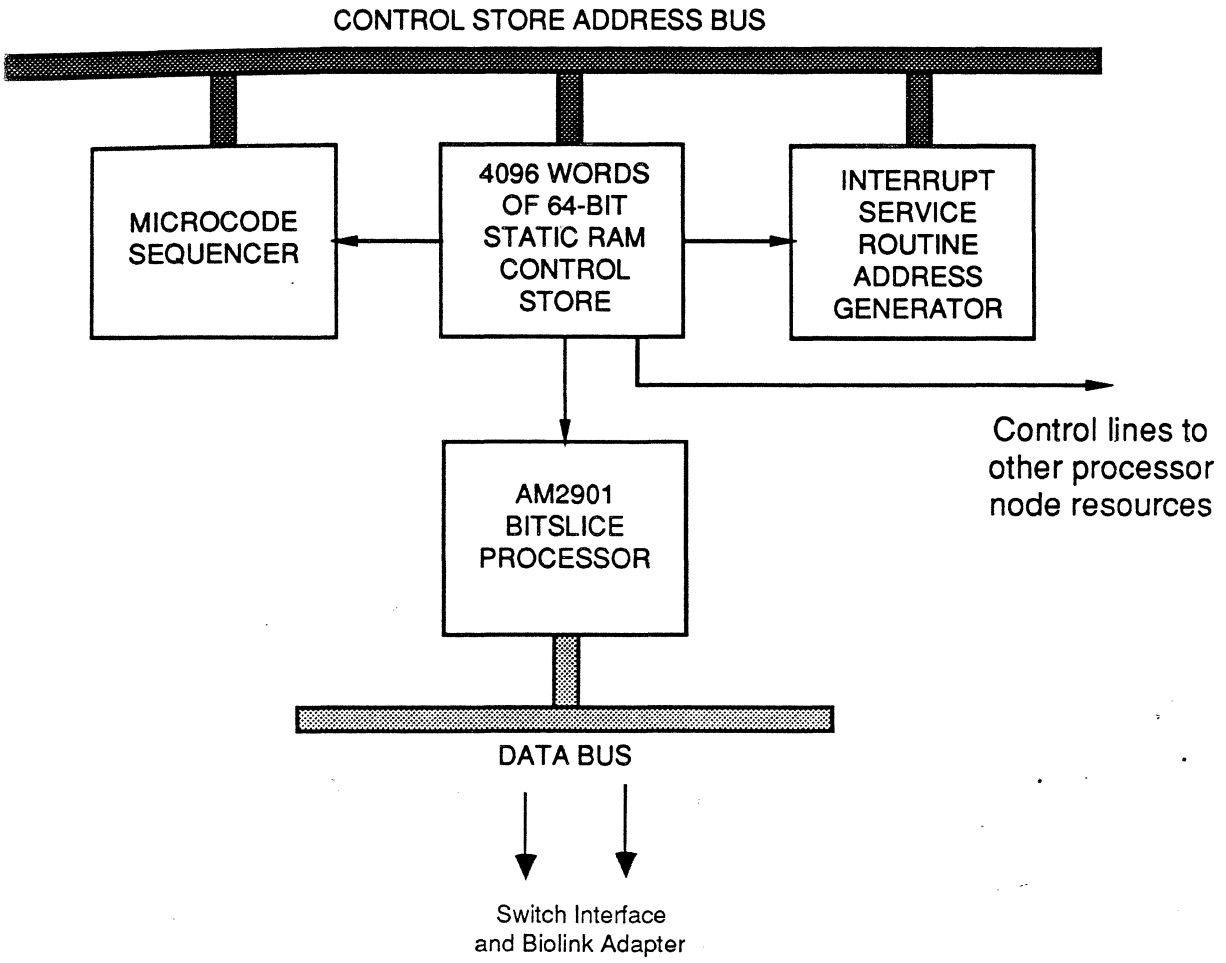


Figure 2-2
PNC Block Diagram

The PNC performs certain operations that greatly extend the parallel processing capability of the MC68020 processor. These include a suite of atomic arithmetic and logical operations, queuing primitives, operations that implement an event handling mechanism, and a process scheduler that works with the queuing and event handling mechanisms to provide efficient communication and synchronization between application software modules. Many of these PNC special functions manipulate data, and it is important to perform them in a way that prevents other processors from accessing the data while it is being used. Because the PNC controls all memory references made by either its own MC68020 processor or any remote processor node via the switch, it ensures that these special functions are indivisible or *atomic*.

The PNC also has two independent finite state machines, which work together with the AM2901 bitslice processor. The two finite state machines, called the switch receiver and the switch transmitter, manage the switch interface to handle incoming and outgoing messages.

The PNC regulates all Butterfly Plus switch transactions. It initiates all messages sent over the Butterfly Plus switch and processes all messages received from the switch.

The Butterfly Plus switch interface transfers message requests and replies between the processor node and the switch. Its two finite state machines have their own independent connectors on the processor node board. The switch interface communicates with the rest of the processor node through a pair of dual port memories. When a message is sent out across the switch, the message data is loaded into appropriate locations in the transmitter dual port memory, and the output finite state machine is notified. When a message comes in from the switch, the input finite state machine deposits it in the receiver dual port memory and notifies the processor node controller.

PNC Bitslice Processor

The AM2901 bitslice processor allows the PNC to perform arithmetic and logical operations, including conditional microcode branches. The 4096-word array of 64-bit microcode resides in a static RAM control store during execution. Every 125 nanoseconds a microcode sequencer feeds selected control store bits into a pipeline register. The next microinstruction is then fetched from the control store in parallel with execution of the current microinstruction. The microcode is stored in EPROM and downloaded to the static RAM microstore, where it can be overwritten by the operating system if required. Distinctive characteristics of the bipolar AM2901 bitslice processor are:

- 17 registers, each 16 bits wide, used for temporary calculation and data storage, and as permanent, fast access variables for timers, status flags, and pointers and counters for direct memory access
- 8-function arithmetic and logic unit (ALU)
- Dual-address architecture allowing simultaneous access to two working registers

- Flexible data source selection, which allows ALU data to be selected from five source ports, giving a total of 203 source operand pairs for every ALU function
- Left and right 16-bit rotates
- Carry, zero, and negative status flags for controlling conditional branches.

Refer to the *Advanced Micro Devices Bipolar Microprocessor Logic and Interface Data Book* for further information.

Control Store and Microcode Sequencer

The 12-bit control store address is sourced by a conventional microprogram sequencer when its code is in the process of performing a microinterrupt service routine. If the code is just about to begin a microinterrupt service routine, its address is sourced by the microinterrupt service routine address generator. The microprogram sequencer is implemented using the AM2911 microsequencer, augmented by two multiplexers that allow 2-condition, 4-way branching. Microsequencer features include a pushdown stack for saving up to four return addresses, and an internal address register for storing the addresses of commonly used routines.

The microsequencer allows the microcode to branch unconditionally, call and return from subroutines, or branch conditionally on the state of various flags and signals. Microprograms run at one of two execution levels. Dispatch level is reserved for microcode functions that respond to requests from the local processor, such as reading the realtime clock, initiating a block transfer, or referencing a Multibus address. Interrupt level is used for microcode functions that handle requests from the switch receiver and transmitter, direct memory access requests, and the interval timer.

Microcoded Special Functions

The PNC also operates the Butterfly Plus switch, and is responsible for all interactions with the switch interface. These interactions take many forms. The simplest occurs when the processor accesses a word of memory on another processor node. The PNC notes that the memory reference is a remote one, places the remote processor node number and memory address in the switch transmitter's dual port RAM, and tells the transmitter to begin the

transaction. While the message is en route and the processor is held in a wait state, the PNC can service direct memory access requests or other microinterrupts. When the message reaches the destination processor node, the remote PNC completes the memory reference and, if the access was a read, sends back a reply message. The reply returns to the originating node, and the value of the referenced memory location is returned to the processor just as though it had performed a local memory reference. Because the hardware is heavily overlapped, this entire remote memory reference occurs in five microseconds under normal loads.

In addition to single word transfers, a PNC can transfer blocks of memory between any two processor nodes in the machine. Block transfers occur at high speed, limited only by the bandwidth of the switch port (about 32-megabits-per-second) and the bandwidth of the memory module (about 102-megabits-per-second). Note that this is *not* the aggregate bandwidth of the Butterfly Plus; it is only the bandwidth of one processor node.

The PNC also performs a variety of indivisible primitive operations. For example, the PNC can post events without using any system locks. The processor tells the PNC to post an event by writing the address of a parameter block into a special memory location that traps to the microcode. Writing the address of a parameter block into a memory location is a general mechanism used by such functions as block transfer, enqueue, and the like. It causes the PNC to send a special message to the destination node that specifies the location of the data structure associated with an event. The PNC at the destination node stops all other memory references and updates the event data structure. If the process that owns the posted event needs to run immediately, the destination PNC also invokes the process scheduler on that node. Other special functions implemented by the PNC include a realtime clock, an interval timer, indivisible mask-and-add-to-memory instructions, and dual queue functions. The operating system uses PNC primitives like these to build higher level synchronization primitives such as locks, semaphores, barriers, and monitors. Appendix B, "PNC Microcode Functions", describes the PNC special functions in detail.

Microinterrupt Requirements

Most processor node functions involve a sequence of transfers from one internal bus element to another, and all activate bus element control lines. Each

processor node function occurs when the PNC executes the appropriate sequence of microinstructions from its control store. Certain data transactions required during processor node functions must be performed as quickly as possible. The delay when a processor reads or writes its local memory, for example, has a dramatic impact on system performance. Other transactions can take somewhat longer to complete. Direct memory access operations that transfer data across the BIOLINK always require low latency. Memory refresh, in contrast, can tolerate a relatively long delay. Switch transactions have various timing requirements, depending on the priority of the message that initiated the transaction.

The microinterrupt driven PNC architecture is ideal for implementing functions with diverse timing requirements because low-priority functions can be kept pending while the PNC attends to higher priority tasks. Microinterrupt requests are issued to initiate processor node functions. Special circuitry locates the address of the microinterrupt service routine that performs the highest priority function among those with pending requests. Once PNC microinterrupts are enabled, that service routine begins executing almost immediately. Other microinterrupt requests remain active until the higher priority function has been performed.

Realtime Clock and Timer

The PNC maintains a 32-bit realtime clock and a 16-bit interval timer, each with 62.5-microsecond resolution. The maximum timer interval is about four seconds and the realtime clock period is somewhat greater than 74 hours. The interval timer issues a level-2 interrupt request when it reaches zero. It is then reset to request another interrupt about four seconds later. This is normally subsequently changed by the system level 2 interrupt handler.

SWITCH INTERFACE

The Butterfly Plus switch interface connects the processor node to the switch and is controlled by the PNC. All message transactions between a processor and a memory module on a different processor node—or between one processor and another—travel through the switch interface. Two finite state machines make up the switch interface: the switch receiver and the switch transmitter.

Switch Receiver Micromachine

The PNC uses one of its independent micromachines, the switch receiver, to accept incoming messages. The switch receiver communicates with the PNC via various control signals and a 16-word dual port memory, called the RxRAM, which is divided into two independent request and acknowledgement buffers. These two input buffers are provided primarily to prevent deadlock. One of these, the request buffer, accepts messages that require the processor node to send a reply; the other, the acknowledgement buffer, holds answers or reply messages that can be processed without using the switch. The receiver examines each incoming message and either places the message in one of the two input buffers or rejects it. Only one message at a time can enter the receiver, but a message of one type can be held in the RxRAM while a message of the other type is received. A special restart message is accepted even if both switch receiver buffers are full, but all other messages are rejected if the appropriate buffer has not yet been emptied. For example, if a message is a read request, and the input buffer for messages requiring a reply is full, the message is rejected.

Request messages include all query messages and some control messages. Acknowledgement messages include the remaining control messages plus all informational messages. The restart control message needs no processor node resources and is always processed immediately by the receiver itself, without intervention by the PNC, and before the restart takes effect. Request messages do not generate a PNC microinterrupt until the transmitter acknowledgement buffer is available, and therefore are not processed by the PNC until then. The request buffer becomes available as soon as the PNC reads the incoming message, but the acknowledgement buffer is only released when the response to the previous request message is completely processed. Acknowledgement messages are less complex, since the PNC can process them immediately.

While processing a request message, the PNC may need to transmit one or more acknowledgement messages, which are of two types. Reply messages are sent in direct response to query messages. Informational messages, in contrast, are unsolicited by their recipient. At the receiver both message types go into the acknowledgement buffer, which has two sections, the header section and the first in, first out (FIFO) section. Messages held in these buffers can be short, using only the header area; long, using both the header and the FIFO area; or of variable length, using the FIFO dynamically. Variable length messages are used only during block transfer transactions.

Two input buffers are provided primarily to prevent deadlock. Assembled in one buffer are those message types (request messages) that require the processor node to send out a reply message. Assembled in the other are answer or reply messages that can be processed without using the switch. The receiver examines each incoming message and either places the message in one of the two input buffers or rejects it. Message rejection occurs when some resource needed to assemble the message is currently in use. If the message is a read request, for example, and the input buffer for messages requiring a reply is full, the message is rejected. A restart message cannot be rejected.

Receiver Circuit Operation

Figure 2–3 is a block diagram of the switch receiver, which has four parts:

- A finite state machine controlled by ROM and programmable logic array firmware
- A set of line drivers and receivers that convert from transistor-to-transistor logic (TTL) to emitter coupled logic (ECL)
- A 16-word dual port memory with 16 bits per word
- A checksum generator.

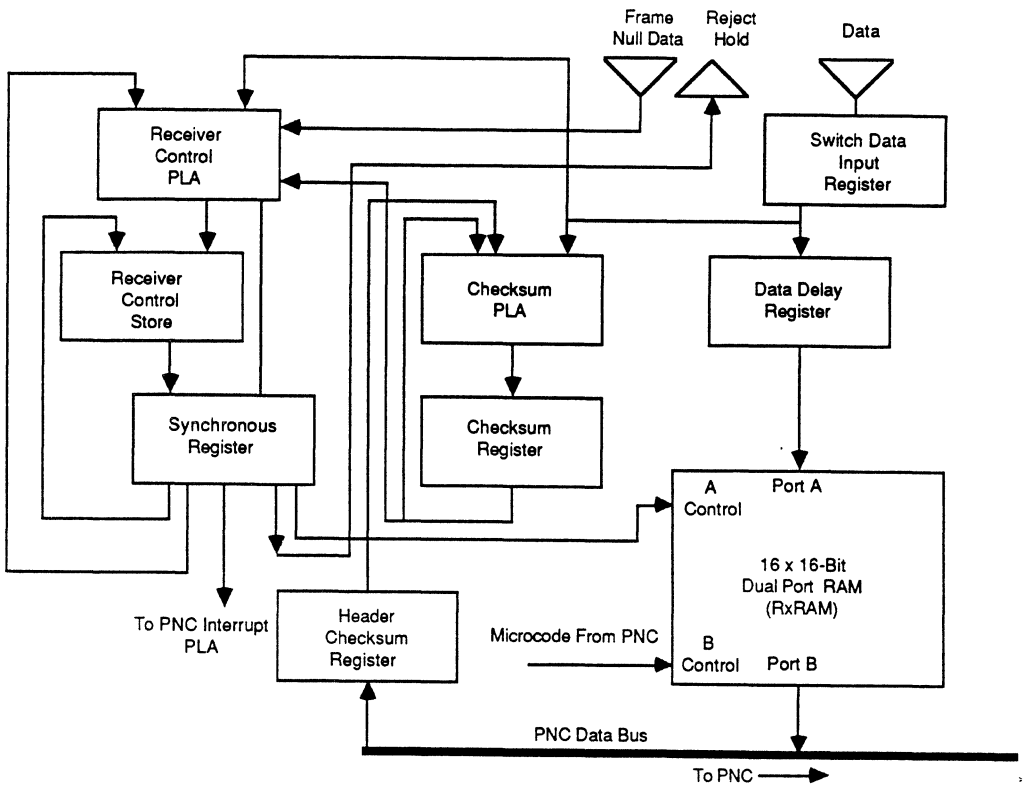


Figure 2-3
Switch Receiver Block Diagram

The switch receiver is implemented as a microcoded finite state machine, an approach that allows a variety of message formats and makes it easy to synchronize the receiver and the PNC. More importantly, the state machine approach frees the PNC to perform other functions while a received message is being assembled. It also allows recognition of a restart message without PNC intervention. The state machine drives the register control and state update logic. It consists mainly of a programmable logic array that sources the address lines of a ROM control store and runs at the same clock frequency as the PNC. The balanced differential ECL line drivers and receivers have a common mode noise rejection of at least one volt to facilitate grounding and reduce intercable crosstalk. All communications between Butterfly Plus switch nodes—and between processor nodes and switch nodes—use balanced, ECL-compatible signal protocols.

The RxRAM dual port memory allows the receiver to assemble a message in one of its input buffers while the PNC accesses data from the other buffer during the same microstep. The receiver collects incoming messages from the Butterfly Plus switch and assembles them into an RxRAM input buffer.

Sometime during message assembly, the receiver requests microinterrupt service from the PNC. In the resulting interrupt service routine, the PNC uses the message type from the input buffer to decide what to do with the input buffer data.

Message checksums are calculated by a programmable logic array that can either load the 4-bit contents of the header checksum register (which is loaded by the processor) into the checksum register, or update the checksum register with the data from the switch output port.

If a restart message arrives, the receiver checks a 16-bit password and, if it is correct, issues a restart to the rest of the processor node. Receipt of a restart message generally indicates that a program on the node sending the restart has determined the processor node receiving the restart to be in a state where all other means of communication with it have failed. Since there are few software failures that prevent one processor node from communicating with another, the Chrysalis operating system determines whether a hardware failure has occurred via the start-up code or a user-invoked program (such as *restart*). The advantage of having the receiver interpret the restart message is that only a small section of the circuitry is involved in initializing all of the components of a processor node, including the switch receiver.

Switch Transmitter Micromachine

The PNC uses another independent micromachine, the switch transmitter, to send messages. Like the switch receiver, the transmitter communicates with its PNC via various control signals and a 16-word dual port memory, called the TxRAM, which is divided into two independent buffers. One of these, the request buffer, initiates new transactions; the other, the acknowledgement buffer, sends secondary messages in response to messages entering from the switch. Only one type of message (*e.g.*, request or acknowledgement) can be sent at any one time. While one message is being sent, however, a message of the other type can be assembled and stored in the TxRAM.

From the transmitting processor node point of view, there are two kinds of request messages: query messages, which always wait for the destination processor node to return a reply message, and control messages, which do not wait. The PNC sets a return address register in its microinterrupt system and enables a microinterrupt at the end of transmission. For control messages, it

also releases the processor. In either case, the PNC then sets a timer and returns to its idle loop.

Transmitter microcode is responsible for getting a message to its destination, if possible. Before it can start sending a message, the transmitter might have to finish sending an earlier acknowledgement message. The transmitter will randomly select the request message buffer or the acknowledge message buffer to send next, if both contain unsent messages. When it becomes available, the transmitter then tries to send the current message. If the message is rejected, the transmitter automatically tries one of its alternate output paths. When sending a block transfer data message, the receiver can hold off the transmitter on a byte-by-byte basis once the message header is accepted. The transmitter appends a checksum to the end of the message.

Once a message is completely sent, the transmitter stops, requests a PNC microinterrupt at the address specified in the return address register, and then marks the request buffer as being available. The PNC resets the timer and microinterrupt request, and, if a reply message is expected, sets the timer and return address register to wait for the reply message. If the timer runs out, the PNC timer routine causes the transmitter to abort, notes the error, and simulates a transmitter completion. If the processor is waiting for a reply, this timeout causes a bus error.

Transmitter Circuit Operation

Figure 2–4 is a block diagram of the switch transmitter, which has four parts:

- A set of ECL to TTL line drivers and receivers
- A 16-word dual-port memory with 16 bits per word
- A checksum generator and data multiplexer
- A finite state machine controlled by ROM and a programmable logic array.

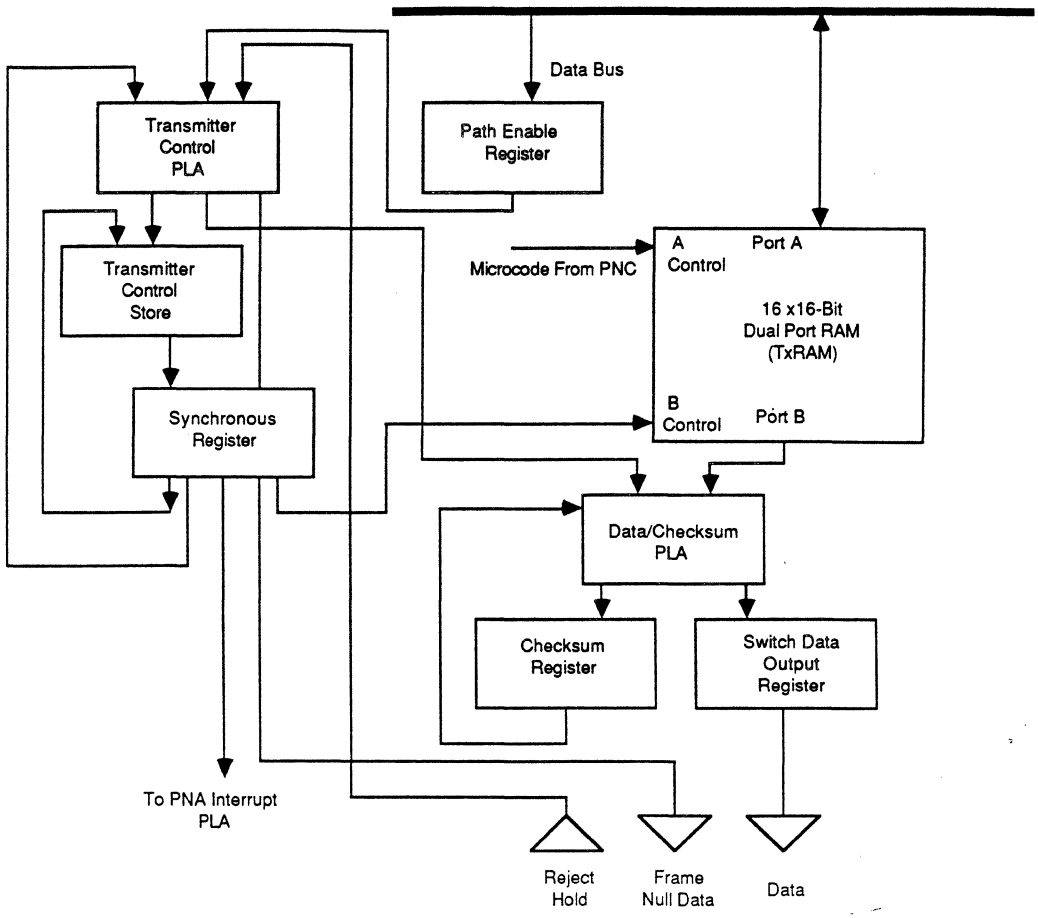


Figure 2-4
Switch Transmitter Block Diagram

The transmitter is structurally similar to the receiver. The PNC can gate one of the 16-bit words onto the data bus and load the low and/or high bytes of any word. Only 12 of the 16 RAM words are used for the two output buffers, so four words of temporary storage are added to the PNC resources. The switch data originates in a programmable logic array that provides the various sources (*e.g.*, zero, RAM data, shifted TxRAM data, alternate path routing, checksum, and ones). Part of the programmable logic array also handles alternate path selection.

All switch transactions originate in the MC68020 processor and are initiated by microcode in the PNC. In general, switch transactions between two processor nodes occur as follows. Initially, one PNC sends a message to another PNC. If the PNC receiving the message has a full request buffer, it cannot acknowledge the receipt of this message until after the request buffer is empty. While waiting for its request buffer to empty, the PNC that received the

message can service microinterrupts from other sources. (However, the MC68020 processor of this PNC remains idle.) Once the request buffer is empty, the PNC that received the message builds an acknowledgement message in its TxRAM and signals the transmitter to send this message to the PNC that originally sent the message. The PNC is uninterruptable while it builds its acknowledgement message. Therefore, it can begin to send this message before it is fully assembled.

Each time the PNC assembles a message in the transmitter request or acknowledgement output buffers, it initializes one of two timeout counters to detect deadlock. The memory refresh service routine decrements both counters every 62.5 microseconds. If the request timeout counter reaches zero while the processor is waiting for a reply, the transmitter request buffer is released, an error flag in the PNC status register is set, and the processor is given a bus error. Once a query request message is completely sent, a PNC microinterrupt routine resets the request timeout counter while waiting for the reply. If the processor is waiting for a reply that fails to arrive in the allotted interval, an error flag is set in the PNC status register, and the processor is given a bus error. If the acknowledgement timeout counter reaches zero, a flag is set in the PNC microinterrupt control register and the transmitter acknowledgement buffer is released with no further direct action. Thus, if a transaction involves sending several acknowledgement messages (*e.g.*, in a block transfer), an attempt is made to send all of them even if some of them time out.

Because long messages improve switch bandwidth-utilization, the transmitter uses an output FIFO. The output FIFO allows the transmitter state machine to begin sending a message even before the first data byte is loaded. This technique greatly reduces the delay normally encountered in store-and-forward techniques. The two flow control lines required for the receiver input FIFO now also provide flow control on the transmit side. Hence, when the PNC becomes so busy (*e.g.*, responding to I/O controller access requests to local memory and/or servicing the receiver) that the transmitter output FIFO becomes empty, the transmitter simply asserts the flow control line that indicates that the current nibble should be ignored. Eventually the PNC becomes available and completes transmission of the message. The transmit FIFO is four words long. It is only used for block transfer messages.

The transmitter has two output buffers to improve PNC bandwidth utilization and assure fairness, which becomes an issue when a processor node must transmit both a request and a reply message at the same time. Providing two output buffers also streamlines interactions with the PNC. When the PNC has data to be sent stored in its output buffer, it sets a flipflop in the transmitter, telling the transmitter that the corresponding output buffer is not empty. When the transmitter is in idle state, which occurs after transmission of a message or when a rejection is sensed, it tests the flipflop and begins sending the message. At the end of transmission, the transmitter clears the flipflop, enabling the PNC to assemble another message. The transmitter has full responsibility for retransmission on rejection, checksum generation, and alternate path selection. If neither output buffer is empty, the transmitter alternates their transmission-upon-rejection responses randomly, guaranteeing proper arbitration.

BOOTSTRAP EPROM

The EPROM consists of up to 128-kilobytes of erasable, programmable, read-only memory that stores microcode for immediate bootstrap upon power up. In addition to the bootstrap, other programs such as the bootstrap debugger and some diagnostics also reside in the EPROM. The microcode is downloaded to the PNC static RAM microstore for the immediate execution of instructions when power is applied. The 128-kilobytes of EPROM are located in two 27512-type devices. When the code is downloaded from the EPROM to the PNC microstore, it can be used as is or reprogrammed.

The EPROM resides in two different areas of the memory map. Normally, it lies between FFC00000 and FFFDFFFF of the virtual memory special function subspace. The first 64-kilobytes of EPROM is also aliased between physical addresses xx000000 and xx00FFFF of the supervisor program space. This allows the MC68020 to fetch a reset vector from the first eight bytes of the EPROM. Excluding the supervisor program space, accesses to this physical address range reference the lowest segment 64-kilobytes of main memory.

DIAGNOSTIC UART

The UART is a dual channel RS-232 serial port transceiver chip used for diagnostic I/O without PNC intervention. A special cable interfaces the UART by attaching the J4 connector near the corner of the processor node to the RS-232

host and console ports on the Ethernet fantail. The UART consists of an SCN2681 (with a counter/timer module), an MC140456 serial line driver and receiver, and a 3.6864-MHz crystal to provide the time base. It supports serial channels operating at up to 19,200 baud. The SCN2681 interface consists of several 1-byte registers addressed on consecutive word boundaries. The auxiliary function control logic acknowledges 1-word transfers to the MC68020 processor for UART accesses. The UART interrupts the MC68020 at level 6. Refer to the Signetics SCN2681 data sheet for programming information. This data sheet may be found in the *Signetics MOS Microprocessor Data Manual 1983*.

USD BOOTSTRAP DEBUGGER

Along with the power up bootstrap, the EPROM also stores a simple debugger program, called USD, for bootstrap diagnostics. Access to USD is available through either the diagnostic UART host and console serial ports or the Multibus Adapter host and console serial ports on the Ethernet fantail. The host serial port cable connects to the host computer and the console serial port cable connects to a terminal. Although only one processor node at a time can be cabled to the fantail host and console ports for debugging, the USD running on that processor node can be used to examine the state of any other processor node in the system. USD also allows a simple downloading protocol followed by the Chrysalis *bld* utility.

USD is stored in EPROM. It provides power up diagnostics and simple debugging options. At power up, each processor node's low-level USD starts immediately and runs from EPROM. It lights the red LED on the processor node and performs some basic memory and processor diagnostics. If the diagnostics fail, the red LED remains on; otherwise, it goes out after the power up diagnostics complete. Because it is interrupt driven, USD can also be used to examine the state of a running program. Refer to the *Chrysalis Programmer's Manual* for further details about USD.

Chapter 3

The Butterfly Plus Switch

The Butterfly Plus switch implements fast, reliable, and economical communication among processors. It is a collection of switching nodes, configured as a serial decision network, which interconnects processor nodes and gives each processor equal access to the shared global memory. It supports reading and writing memory on remote processor nodes, block transfer of memory data between processor node memories, special atomic functions of the operating system, and processor node reset.

Each Butterfly Plus switching node is a 4-input, 4-output switching element implemented by a custom VLSI chip. Eight VLSI switch chips are packaged on a single printed circuit card to produce a 16-input, 16-output switch module that occupies a horizontal slot at the top of the Butterfly card cage. The eight switching nodes on a switch card are logically arranged in two columns of four nodes each, as shown in Figure 3–1. A 3 by 19 inch panel mounted at the front of each switch card provides input and output connections for up to 16 processor nodes. A switch card can be used by itself in a 16-processor Butterfly Plus, or it can be cabled together with other switch cards to form the switch for a larger machine.

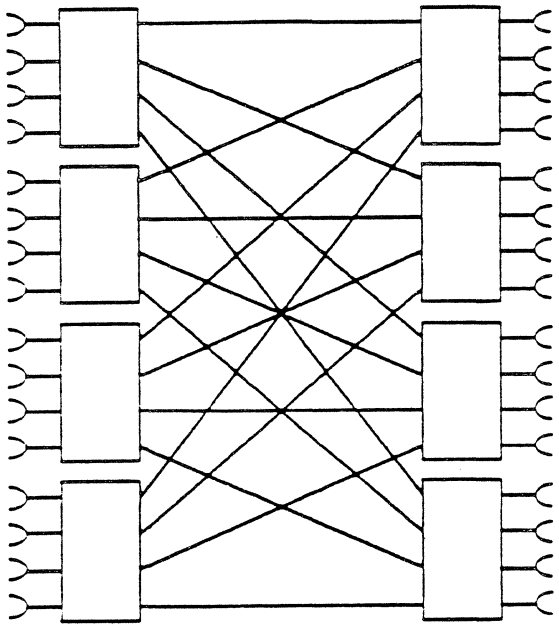


Figure 3-1
16-Port, 8-Node Switch

The 32 connectors on a 16-input, 16-output switch card attach to cables that carry signals to and from processor nodes, other switch nodes, and the optional VMEbus adapter. Onboard clock and reset logic are included for systems that require only one switch card. A rocker switch on the righthand side of the switch card panel activates the reset logic. A red LED beneath this rocker switch comes on briefly when the rocker switch is toggled. A green LED to the left of the reset switch comes on to indicate that regulated DC power is applied to the switch. In systems with more than one switch card, the switch card accepts external system clock and reset signals through a 16-pin connector on its front panel.

A path through the switch connects each processor node to every other processor node. Each path through the Butterfly Plus switch is four bits wide. These 4-bit nibbles move from one switching node to the next on each clock tick. Switch operation is similar to that of a packet switching network. The switching nodes use packet address bits to route a packet through the switch from a source processor node to a destination processor node. Each switching node uses two bits of the packet address to select one of its four output ports. Figure 3–2 shows a packet in transit through a 16-input, 16-output switch like the one shown in Figure 3–1.

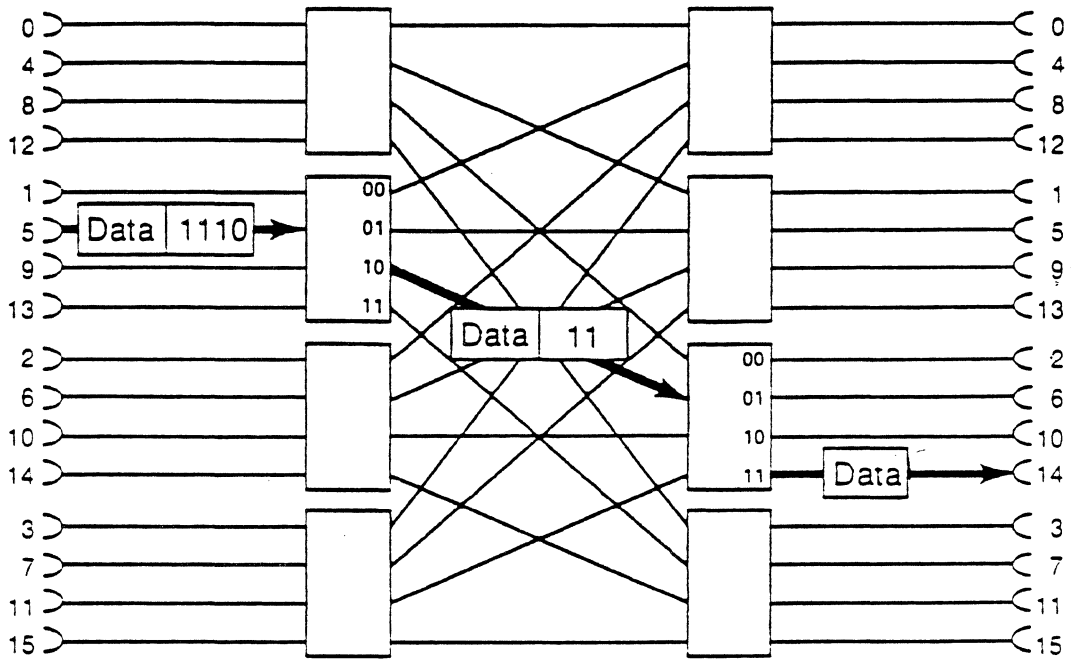


Figure 3-2
A Packet Moves through an 8-Node Switch

To send a message to node 14, node 5 builds a packet containing the 4-bit address of node 14 (binary 1110), followed by the message data, then sends the packet into the switch. The first switching node strips the two least significant address bits (binary 10) off of the packet and uses these two bits to route the remainder of the packet out of its port 2 (binary 10). The next switching node strips off the next two address bits (binary 11) to switch the packet out of its port 3 (binary 11) to node 14. The structure of the switch network ensures that packets with binary address 1110 will always be routed, with the same number of steps, to node 14, regardless of which processor node sent them.

The Butterfly Plus switch expands easily as processors are added to the system. By adding additional rows and columns of switch nodes, the 16-processor switch can be expanded to handle more processors. For example, the number of connected processors can be quadrupled to 64, as shown in Figure 3–3, by connecting two additional switch node columns and enough rows to accommodate 48 more processors. The complexity of the switch (as measured by the number of switch elements or the number of signal paths) for an N-processor system grows as $N\log_4 N$. For large configurations this is significantly less than other switch designs (*e.g.*, the N^2 elements that a full crossbar switch would require). At the usual 8-MHz clock frequency, each path through the switch supports interprocessor data transfers at a bandwidth of 32-megabits-per-second. Thus, a switch for a 16-processor system has a maximum bandwidth of 512-megabits-per-second, and a switch for a 64-processor system, such as the one shown in Figure 3–3, has a maximum bandwidth of 2,048-megabits-per-second.

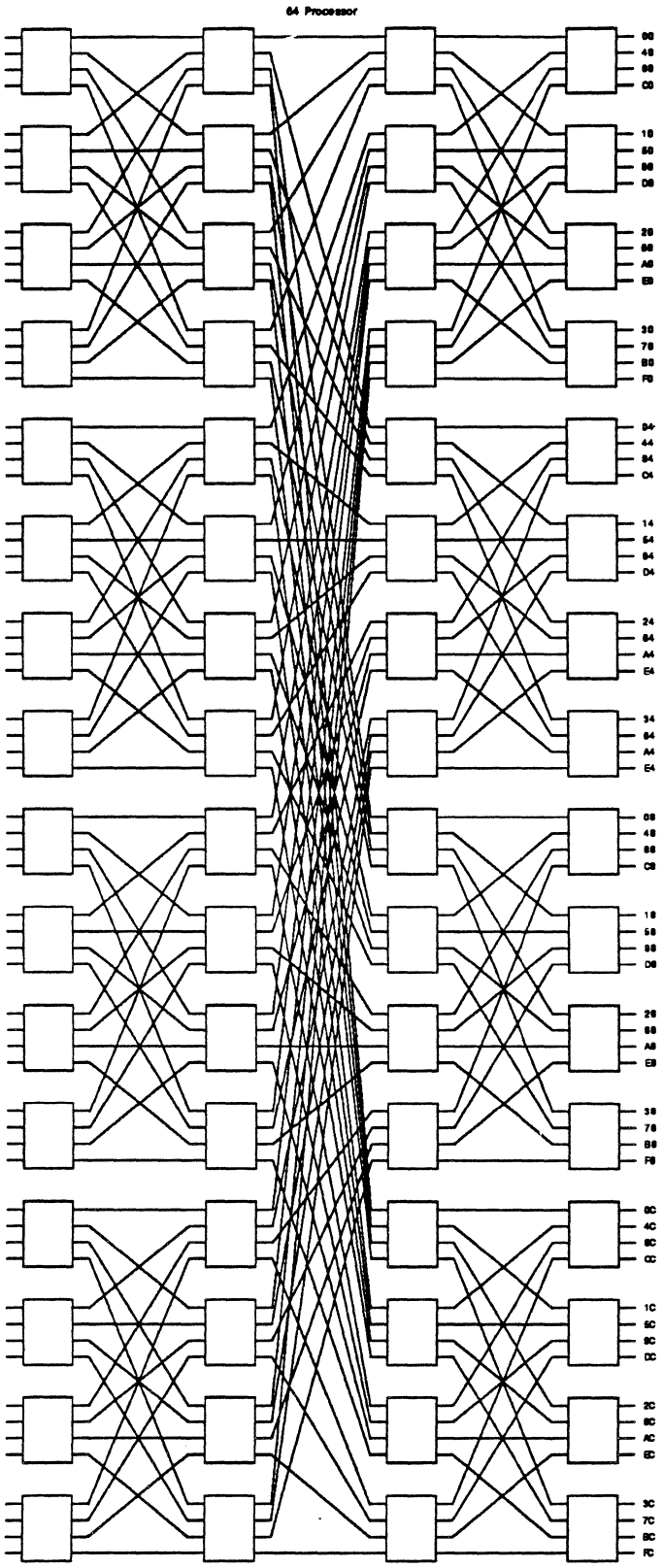


Figure 3-3
Switch for a 64-Processor Butterfly Plus

ALTERNATIVE SWITCH STRUCTURES

There are many forms of multiprocessors and many types of switches in the parallel processing industry. In general, there are three ways to connect the processors and memory modules of a parallel processor system:

- Common bus
- Crossbar switch
- Butterfly switch.

Of the three communication switches, the simplest is the common bus, or singly connected switch. This is the standard data bus concept used in numerous minicomputer architecture systems. It connects every processor and every memory module along one bus (see Figure 3–4). Other structures that provide the same interconnect concept include the ring, the Ethernet, and broadcast channels.

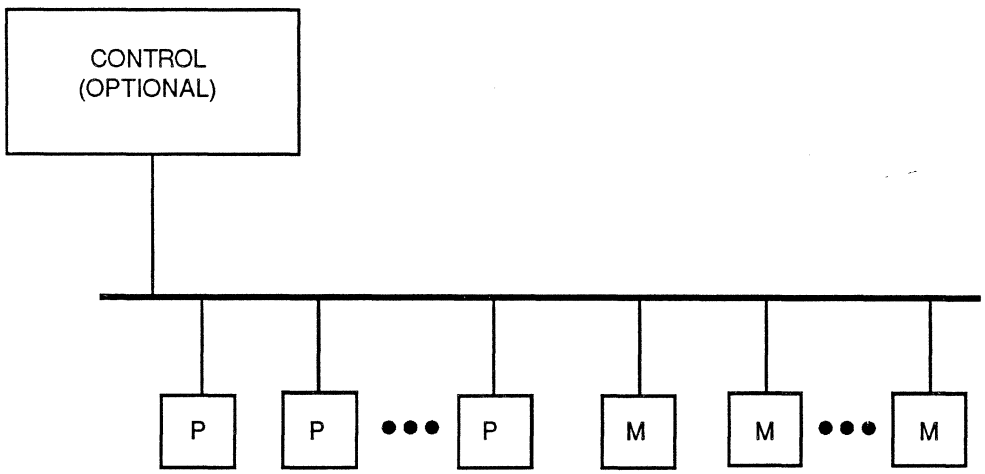


Figure 3-4
Common Bus

The common bus architecture presents a simple, inexpensive communication structure where all the processor and memory modules provide the bus driver, receiver, transmitter, and arbiter circuitry. All communication flows through this one path. The processors and memory modules share the path according to some protocol (*e.g.* time division multiplexing or arbitration). In some systems, there can be more than one path for reliability or bandwidth reasons, but

in general this number is small. The addition of more buses impacts each of the connected elements, which then must have an arbitrary fanout. The common bus is efficient and cost effective for a simple computer system with a relatively small number of processors (20 to 30 processors). This structure expands smoothly, since its cost goes up linearly with the number of elements it connects. However, as the needs for large numbers of parallel processors increase, the complexity of the bus system also increases. Faster buses will require more hardware on each processor module, resulting in increasingly complex processor circuitry. When the number of elements exceeds the necessarily finite bandwidth of the pathway, the common bus structure is no longer usable.

Particular implementations of singly connected switches have a number of disadvantages. Because the pathway bandwidth is severely limited, buses are hard to expand beyond certain limits, unless they are very carefully designed for this goal. However, a bus type switch designed to be easily expandable frequently loses in efficiency. Buses are also hard to debug because of the difficulty in identifying the problem module. Although rings solve efficiency and debugging problems by effectively splitting up the bus into short pieces, ring buses consequently have longer delays.

The simplest solution to the problem of the common bus architecture for parallel processor systems is the crossbar switch, a completely connected switch. This architecture is basically an extension of the common bus structure. The crossbar switch attempts to reduce the contention found in a common bus architecture by using a separate data bus for every processor and a separate data bus for every memory module (see Figure 3–5).

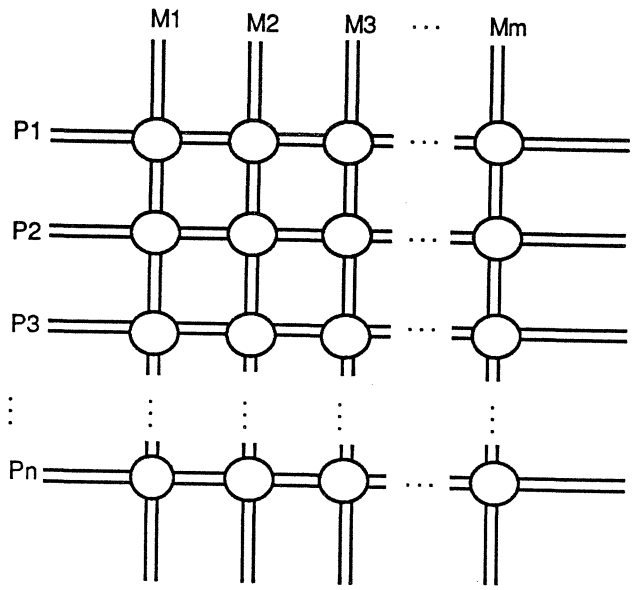


Figure 3-5
Crossbar Switch

The buses for the P processor and M memory modules crisscross to form grids with intersecting processor and memory buses, resulting in a $P \times M$ crossbar switch. The advantage of the crossbar switch is that it allows all processors to access memory simultaneously, as long as no memory module is being accessed by more than one processor. It also provides high potential bandwidth, since each source has its own private path independent of and in parallel with any other source. Bus arbitration is also provided in the event of bus contention. The fundamental disadvantage of the crossbar lies in the $P \times M$ number of nodes required by an P-processor system. This disadvantage results in a system that grows as P^2 (for an equal number of processors and memories) in complexity, configuration size, and cost. As systems get larger, the cost of the switch tends to dominate the overall system cost.

The serially connected Butterfly Plus switch is a more efficent alternative than the crossbar switch, and attempts to reduce crossbar switch complexities by linking multiple crossbar switches as nodes in a treelike or multistage structure. For example, if a large (say 100×100) crossbar switch were divided into two smaller switches, one 100 by 10, and the other 10 by 100, as shown in Figure 3–6, this change would reduce the number of crosspoint elements

required from 10,000 to 2,000, an 80% savings. Dividing the crossbar, however, introduces sharing of switch paths between sources at a point that is within the switch rather than only at the destination. As a result, the maximum potential bandwidth of the switch is reduced by a factor of ten. Interconnections within the Butterfly Plus switch eliminate the loss of maximum bandwidth, however, because a vertical cross section anywhere through the switch always has at least as many paths as there are processors. Also, although there are slight delays that can be encountered from the cumulative transit time through the multiple Butterfly switch stages, these delays are negligible compared to the advantages reaped from the reduction in switch complexity and size.

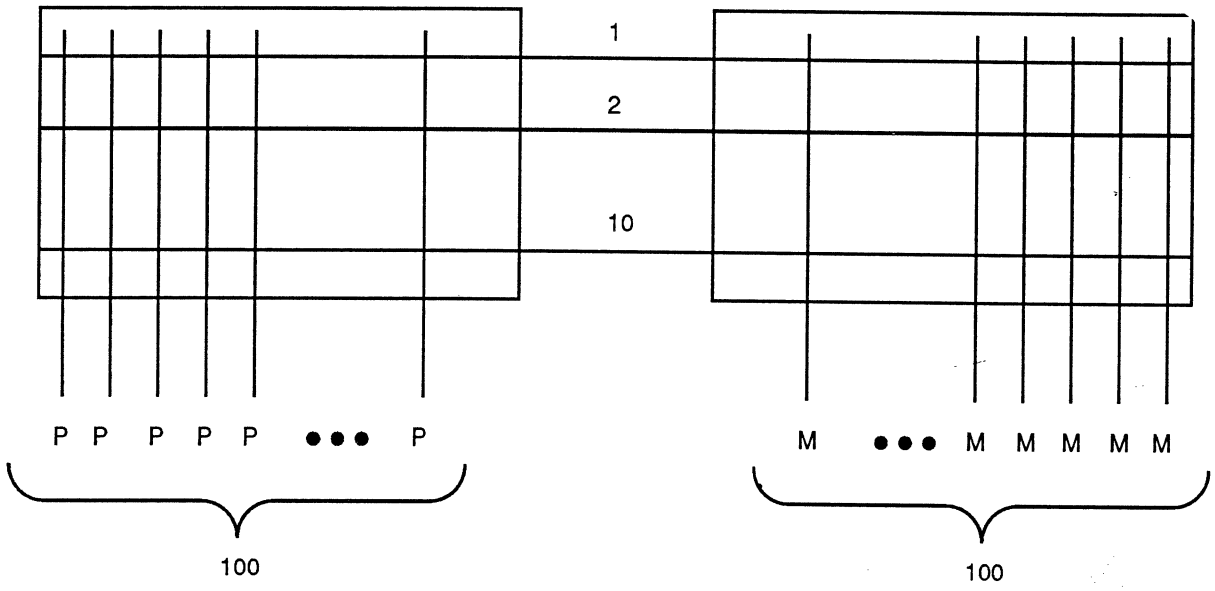


Figure 3-6
Divided Crossbar Switch

Unlike the crossbar switch, which grows as $P \times M$ (or P^2 for an equal number of processors and memories), treelike or multistage networks such as the Butterfly switch connect P processors to P memories at a growth rate of PlogP. The factor logP, which corresponds to the number of levels in the basic tree, is the number of stages. The logarithmic base, which corresponds to tree fan out, is the number of inputs and outputs to each switch node. The factor of P, corresponding to the number of nodes at the lowest level in the basic tree, is the number of processors (and memories) in the tree.

CHARACTERISTICS OF A BUTTERFLY PLUS SWITCH

The Butterfly Plus switch offers many unique features that distinguish it from common bus and crossbar interconnects:

- The Butterfly Plus switch is purely a communication medium. There are no processing elements in the switch. This contrasts with other interconnects that use networks of processors for both computation and communication.
- Messages passing through the Butterfly Plus switch are independent. No effort is made to coordinate the messages in the switch; rather, messages are sent whenever a processor happens to make a remote memory reference. Furthermore, actions of the various processors are not coordinated at the memory reference level. Thus, no effort need be made to match the configuration of the switch to the algorithm being executed on the machine.
- The Butterfly Plus switch connects complete processors to the globally shared memory, not pieces of an arithmetic and logic unit.
- The Butterfly Plus switch is self-routing. Each processor node has one switch port through which it sends all transactions, regardless of destination, and another port through which it receives transactions for that node.
- The Butterfly Plus switch is basically serial. Transaction routing decisions are serial, proceeding from one node to another. Although there is 4-bit parallelism in the data paths of the switch, this parallelism is explicitly for the purpose of improving switch performance. Since the data paths are inherently serial, data streams are supported easily and efficiently.
- The Butterfly Plus switch provides alternate paths so that processors trying to take the same path will not have to contend for that path. Instead, one of the processors can be rerouted along an alternate path.

The Butterfly Plus switch offers the following benefits.

Expandability New nodes are easy to add, and there are no significant cost jumps as particular sizes are reached and exceeded.

Routing Routing is simple and fixed, without the need for a complicated dynamic algorithm.

Reconfiguration	All nodes in the network understand all they need to know about their connectivity and topology without any external configuration parameters, such as switches or jumpers.
Homogeneity	All switch nodes are identical.
Balance	The system is well balanced with no pronounced bottlenecks, and can be tuned to fit its environment.
Reliability	The network operates reliably, even when some of the switching nodes and/or pathways are inoperable; a single defective processor does not impair switch functioning substantially.
Interface	Hardware and software interact easily with the switch.

Like any general purpose data communication structure, the Butterfly Plus switch provides services such as data connection, addressing, flow control (*i.e.*, speed matching), and arbitration. It also has important characteristics such as speed, geographic extent, and protocols. Aside from these services and attributes, the switch is transparent to the rest of the machine.

BUTTERFLY PLUS SWITCH OPERATION

The Butterfly Plus switch performs all communication between processor nodes. All switch transactions are initiated by one of the processors. When a processor makes a service request, microinterrupt code in the processor node controller (PNC) transmits an appropriate message through the switch transmitter circuit of the requesting processor node. This message is routed from node to node through the switch until it appears at the switch receiver circuit in the destination processor node. There the receiver requests a microinterrupt, and the PNC performs the action specified in the message, which can include sending one or more additional messages to various other processors. In some cases the requesting processor waits for one of these responses to complete the transaction, while in other cases it simply initiates the transaction and proceeds to its next task immediately. Timers are used to recover from certain error conditions.

A brief description of a read access to remote memory shows how processor nodes use the switch. When a processor makes a read reference, the memory management unit transforms the supplied virtual address into a physical

address. If the referenced location is not in memory on its processor node, the local PNC is addressed, and it sends a packet through the switch, addressed to the remote processor node, and requests the contents of that physical memory address. The remote PNC receives the packet, reads the referenced memory location, and sends a reply packet containing the value back through the switch to the source processor node. When the local PNC receives the reply, it satisfies the processor read request with the value obtained from the reply. The round trip time for a normal remote memory reference is about five microseconds. In addition to single word transfers, the switch can also transfer blocks of data efficiently between any pair of processor nodes at its full 32-megabit-per-second bandwidth.

The Butterfly Plus provides for a variety of message transactions. Each message includes at least the address of its destination processor, its message type, some data bytes, and a checksum. There are several different classes of messages: fixed versus variable length messages, messages initiated by the processor versus those initiated by other messages, and messages that can initiate other messages versus those that do not. The processor node implements several types of switch transactions: single byte, word, or long word reads and writes; block transfers; interrupt requests; processor node resets; and a class of special transactions that includes event synchronization, queuing and dequeuing, and similar multiprocessing functions.

Simple memory transactions go into the switch via the paged memory management unit, which maps ordinary processor memory references into physical addresses that refer to specific memory locations on specific processor nodes accessed via the switch. To the MC68020, remote accesses appear no different from ordinary local reads. The other types of transactions begin when a processor stores the address of a parameter block in one of several special locations whose addresses are decoded by the PNC. Microcode checks the parameters and sends out the appropriate message. Some transactions send several messages.

Handling Contention

Because the Butterfly Plus switch eliminates the need for a dedicated path between each pair of processor nodes, it is possible for two messages simultaneously passing through the switch to reach the same switch node and require the same switch node output port. When contention of this sort

occurs, one of the messages is allowed to proceed to its destination and the other is automatically retransmitted after a short random delay. However, messages arriving at the same switching element, but not requiring the same output port, do not collide; they are handled concurrently. A single switching node can handle as many as four messages simultaneously, provided that each of the four accesses a different output port.

A switch containing an alternate path between every pair of processor nodes can be built by adding extra switching nodes. This is typically implemented in larger configurations to make the switch resilient to switching node failures. Currently, all systems with 17–128 processors are configured to have redundant paths. If a switching node on the path between two processors fails, packets are automatically routed along the alternate path. Reduced switch contention is another benefit of a switch configured with alternate paths. When contention within a switch having alternate paths requires message retransmission, an alternate path is used immediately.

Butterfly Plus switches are configured to minimize the likelihood of message collision. As a result, the transit time through the switch is dominated by the time required for a message to pass through the switch in a bit-serial fashion, rather than by contention. The amount of contention is, of course, application dependent. Contention overhead usually amounts to less than a few percent (typically 1% to 5%) of the total application run time in tuned programs. Most collisions occur at the destination node (memory contention) rather than in the interior of the switch (switch contention).

Error Detection and Handling

Errors are detected and handled in several ways:

- Each message includes a 4-bit checksum, which is generated and checked automatically.
- Variable length block transfer data messages include an additional checksum early in the message, just after the address and length information.
- Rejected messages are retried automatically, using the alternate paths cyclically if alternate paths are available.
- Timers detect dead states for all messages and for processors waiting for a reply.

- The application program or operating system can make additional checks as appropriate.

These facilities detect errors in the receiver, the transmitter, the PNC, the processor, and the switch itself.

The 4-bit checksum in each message detects most errors, but there remains a probability of about 6% that any particular error might not be detected. If errors are frequent, the hardware should be taken out of service and repaired. Rather than try to retransmit a message with a bad checksum, the checksum handler declares that the hardware has failed, so that undetected errors are not introduced into the system. The checksum error handler aborts the transaction in progress and reports the error to the operating system at the destination where the error is detected.

A switch with extra columns has alternate addressing paths, which the processor can enable and disable independently. If a switch path fails, the switch quickly and automatically retries alternate paths for as long as the message continues to be rejected.

Block Transfers

Although shorter messages reduce problems of switch contention with other types of messages, thus alleviating certain performance bottlenecks (*e.g.*, spin locks), long messages have the more important advantages of reduced set-up time and lower switch overhead. During a block transfer, the originating node can request transmission of a block of memory data from any node to any other node. There are no restrictions on the locations of the source, destination, or originating node. The maximum size of a block transfer is 65,536 bytes. The operating system normally breaks long transfers into a series of smaller transfers.

The originating processor node becomes free as soon as the block transfer request has been accepted by the destination processor node; however, accesses to either the source or destination processor nodes during the transfer are likely to be rejected. Direct memory access transactions are not affected by block transfers, since they have higher priority, but computational performance is reduced in the source and destination processors. In the absence of

other activity, a block transfer uses 75% of the total memory bandwidth, leaving 25% of normal memory bandwidth for the processor.

Although the pipelined structure of a Butterfly Plus switch favors long messages over short messages, the structure of a tightly coupled multiprocessor also requires the ability to read and write single words across the switch. As a result, the hardware supports both single- and multiple-word transfers.

Routing Decisions

In any type of switch, the control circuitry (as opposed to the data) consists of one or more "switching systems" that route a transaction from a source to a destination. In a bus system, for example, the arbitration mechanism, along with various address recognizers and gateable bus drivers, forms the switching system. In a crossbar switch, there are different switching systems for each destination, all working in parallel.

In a common bus or crossbar system, the actual switching decision for each individual transaction takes place at only one instant of time. This implies that each decision must be an N-way decision, and that the switching system implementing the decision must have a fanout of N. There are two problems in such systems: N is large and N is not well defined. It is easy to build a switching node that decides between a small fixed number of alternatives. When this number gets to be very large and/or variable, the design of the switching node and the resulting switching system becomes awkward, costly, and/or slow.

In the Butterfly Plus switch, the switching decisions are not made all at once, but rather are spread out in time. As the transaction moves towards its destination, it encounters several switching nodes; at each node a decision is made about its route. Each of the nodes can thus have a very small fanout (*e.g.*, two), while still allowing a large and expandable overall system fanout. This kind of switching system is known as a serial decision network (see Figure 3–7). A message originating from a processor (P) eventually makes its way to a memory module (M) after passing through at least three switching nodes. At each node a decision is made as to which of the two outgoing paths the message will take.

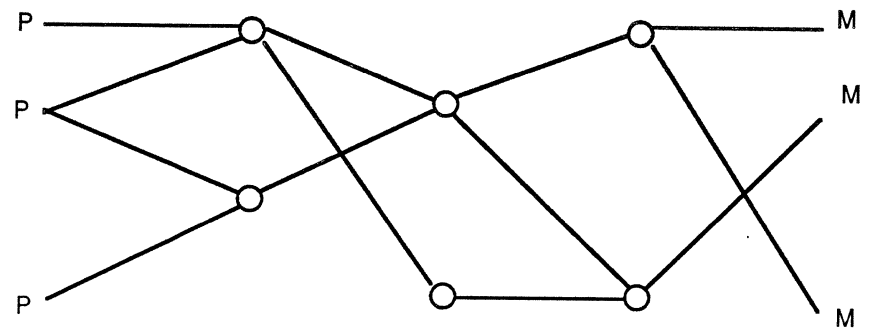


Figure 3-7
Serial Decision Network

A good analogy can be made to a computer network like the ARPANET. Here the switching nodes are actually computers, and data paths are communications lines that span large distances. Transactions entering the network contain the address of their destination within them. Each switching node routes each transaction along the best output path to its destination. Fanout from each individual switch node is small, but the entire network interconnects hundreds of sources and destinations.

The environment of a single computer, on the other hand, clearly involves a different set of concerns than does the computer network. For a single machine, a serial decision network with a simple, regular structure like the Butterfly Plus switch is more easily implemented. There is no need for elaborate protocols or complicated schemes for routing and retransmission in this relatively simple environment. The switching nodes thus become simple devices that need no programmed processor because the necessary algorithms are easily implemented in dedicated hardware. There is also no need to buffer an entire message; instead, only a few bits at a time will suffice.

Contents of the transactions is another key difference between a geographically distributed network and a single-machine network. Messages in a geographically distributed network pass between essentially independent computers and therefore each message must contain overhead data to extablish its context. This can also occur in a single machine if it is structured as a

collection of loosely coupled processors sending messages to one another. In a more tightly coupled machine like the Butterfly Plus, messages are the actual requests by processors to fetch and store the contents of memory locations. A transaction consists simply of an address and some data to write into that location, or a request to read a particular location, or the contents of a location being returned to the processor that requested a read. No large overheads are incurred.

The Butterfly Plus switch design differs from that used in many machines where a processor establishes a path to memory through available switching, then holds that path until the memory returns the requested data. In the Butterfly Plus, each stage of the switch is only busy long enough to pass a request on to the next stage. Data returning from memory to processor is contained in an independent transaction that finds its own way through the network. Between the time a data request is made and a reply message returns the requested data, many otherwise unrelated transactions can travel along the same path.

Bidirectional Communication

Transactions must flow not only from processors to memories, but also from memories to processors, as when the results of a memory read are returned to a processor. One way to implement two-way communication is to superimpose a second network on the original network to handle the return flows. The Butterfly Plus switch provides two-way communication by bringing the stubs on one side of the network around to meet those of the other side. Conceptually, the switch appears to be a cylinder, as shown in Figure 3–8. The processors and memories are located along a line down the side of the cylinder, sending their outputs to the right and receiving their inputs from the left. All transactions flow in a counterclockwise direction as viewed from the top of the cylinder.

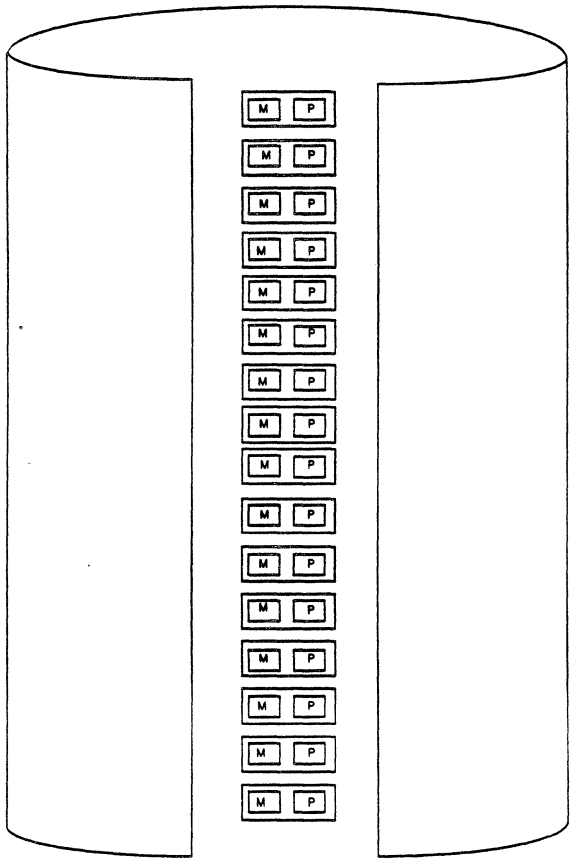


Figure 3-8
Butterfly Plus Switch as a Cylinder

The processors and memories, which up until this point have been pictured on opposite ends of the switch, are now next to each other. The next step combines a processor and some memory into a processor node. These nodes interface to switch entry ports. Now the picture looks like Figure 3–9 and the switch cylinder has become a high-bandwidth communication path between nodes.

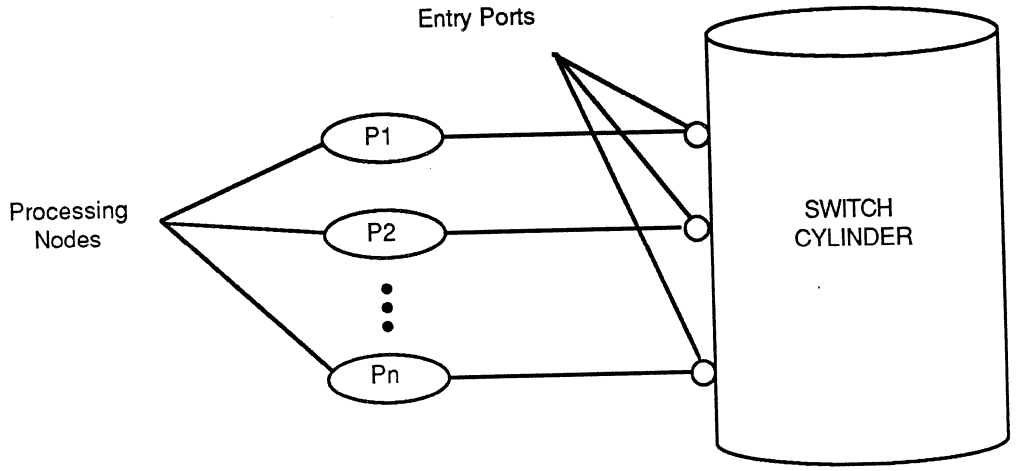


Figure 3-9
Switch as a Channel between Processor Nodes

Even though the memory is now segmented into a piece in each node, it is all still globally accessible, and we can ignore the association between the processor and the memory in the same node. On the other hand, now that a processor has some memory within the same node, there is a potential direct path to that memory without going through the switch. The memory can be used both as a local and as a common memory, and in fact can be dynamically allocated between these uses.

Conflict Resolution Strategies

Conflicts arise when two transactions at a particular switch node want to use the same exit path. Clearly both of them cannot, so an appropriate strategy must be chosen to resolve the conflict. One strategy is to let one transaction wait until the other has completely passed. This simple algorithm is not suitable for a bidirectional switch, however, since it leads to deadlock.

The Butterfly Plus uses an alternative strategy, which sends the losing message back to its source and has it retry some time later. If the switch interface is designed with care, this strategy does not necessarily lead to deadlock. This strategy also seems to have better performance than the wait strategy because it reduces the profile of a message. It reduces the number of secondary conflicts that result from the original conflict.

To clarify the retreat strategy further, see the example in Figure 3–10, and assume that the simpler wait strategy is being used. Transaction 1 is trying to get to exit port A via the dotted line. Transaction 2 is trying to use the same path, however, so transaction 1 is blocked at node B. Transaction 1 must therefore wait until transaction 2 has passed through. At the same time, transaction 3 is trying to get to C via the dashed line. It runs into a conflict at D with transaction 1. So now transaction 3 must also wait, first until transaction 2 passes, so that transaction 1 can start, and then until transaction 1 goes by.

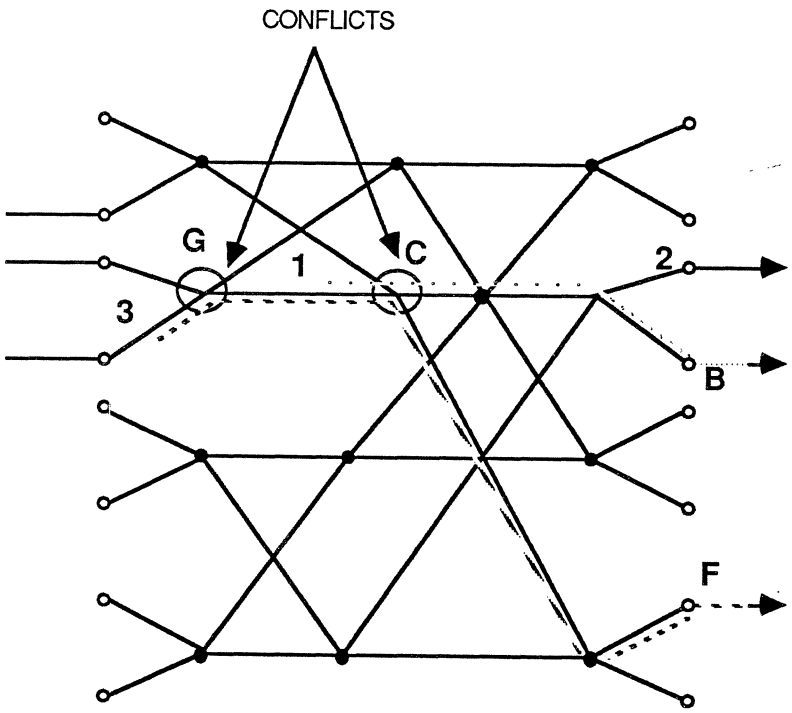


Figure 3-10
Secondary Blockage in the Switch

Since the Butterfly Plus uses a retreating strategy, this type of secondary contention is avoided. The blocked message presents a much smaller average profile and a reduced tendency to obstruct other messages, because it spends less time actually sitting on the network. The result is an increase in the overall network bandwidth. Higher bandwidth decreases the average delay of a transaction significantly, especially when the network is heavily loaded.

In a retreat strategy, a transaction encountering a conflict retreats to its entry port (is cancelled) and then retries. A retreat is only possible if the tail of the transaction has not left the entry port; otherwise the retreat path is not known. Once the head of the transaction reaches the exit port, it can no longer conflict with another transmission. Therefore, the messages moving through the Butterfly Plus switch have a minimum time length for the head of a message to reach the other end of the switch, plus the latency during which the remote switch interface could reject the message, plus any delay in passing the retreat signal back to the source.

The retreat strategy offers one other valuable benefit. Because the retreat signal is asserted by the switch interface, a processor node can examine a message without making a commitment to actually accept it. This feature of the retreat strategy ultimately leads to a switch interface without deadlock.

An important system concern is the maximum delay required to send a message through the switch, which determines the ability to establish timeouts for lost or incorrect messages. With the wait strategy, the worst case is that all sources simultaneously address the same destination. With the retreat strategy, on the other hand, there is theoretically no upper limit to the maximum delay; it is possible, though wildly unlikely, to have a transaction take five minutes or a week to traverse the interconnect. In reality, problems that might arise from delayed transactions are circumvented by the Butterfly Plus operating system. If a transaction does not complete within a reasonable time, timeout logic signals the process that sent the stale transaction.

Parallel Data Paths

Parallel data paths improve the bandwidth of a Butterfly Plus switch to maximize system performance. Parallelism also has the advantage of amortizing the overhead required for the control portion of the switch. The term *thickness* is a synonym for data parallelism when referring to the dimensions of a Butterfly Plus switch.

Although it is possible to design switches that permit a transaction to complete in only one clock tick, such an approach would be a design overkill. Large amounts of parallelism become unattractive when the marginal system advantage of an extra data path becomes less than the marginal increase in the cost of the switch. A small amount of parallelism, however, can produce a switch in which one path has a bandwidth of many tens of megabits. The Butterfly Plus switch implements a 4-bit thick data path, thus achieving an increase in bandwidth while incurring only a small marginal increase in cost.

Alternate Paths and Extra Columns

To minimize the effect of switch node failures, the Butterfly Plus switch uses additional switch columns. The failure of an individual switch node can affect a wedge of switch nodes as well as entry points emanating in both directions from the failed node (Figure 3–11). If the sources and destinations connected via the entry points can somehow be isolated in either the left or right wedge, the rest of the system is still fully connected. The severity of a failure increases the closer the failed node is to the center of the switching network. A failure at the periphery of the switch might only take out one or two processors, not a very serious occurrence in a large system.

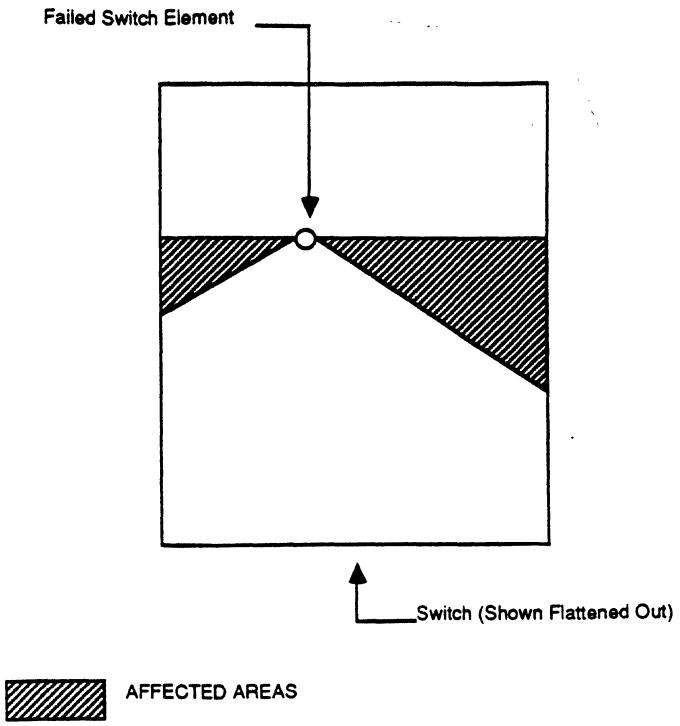


Figure 3-11
Effects of Switch Failure

Introducing redundancy makes the Butterfly Plus switch network more resilient to failures. The addition of some extra columns of switching nodes accomplishes this. The switch thus has multiple paths between each source and destination pair. Each destination has several addresses, any of which can be used to reach that destination. This is shown in Figure 3–12. Failure of a node in the interior of the switch (that is, excluding nodes in the leftmost and rightmost columns) does not affect the connectivity of the switch. This increase in reliability has been achieved at the relatively low cost of adding only one column to the switch.

The extra column is added to the right side of the switch. The additional column thus has the same connectivity as the column at the far left side of the switch. A switch node permits a transaction to move from one row of switch nodes to another. In the Butterfly Plus switch, each column selects an orthogonal set of such moves. The extra column that has the same connectivity as another column furnishes an alternate method of providing such motion. Thus, the best way to protect against the largest number of switch node failures is to place these two columns as far apart as possible.

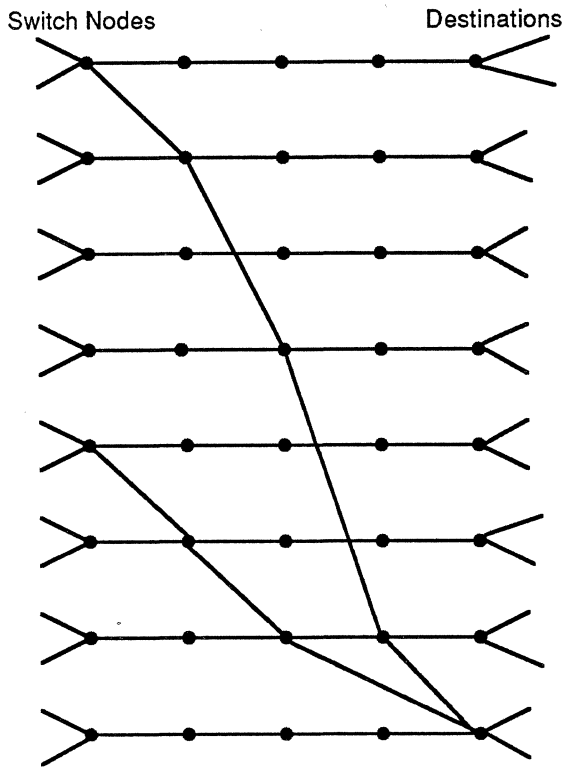


Figure 3-12
Alternate Paths through a Switch

Alternate message paths increase switch throughput and provide resiliency if individual switch nodes should fail. The Butterfly Plus uses a combined hardware and software approach to dealing with these alternate paths through the switch. At the software level, the alternate paths manifest themselves as several addresses for any given destination.

At the hardware level, the switch transmitter automatically chooses a new alternate path for each message transmission. If a transaction fails to reach its destination, an appropriate address bit is changed and the transaction is sent again down a different path. Alternate paths are also useful in decreasing the delay due to conflicts. If a conflict is encountered along one path, the hardware tries the alternate paths until a clear one is found. To prevent unworkable paths from being used, a 4-bit path enable register allows the processor to enable or inhibit one or more of the alternate paths. This register also ensures that only valid paths are enabled. Alternate paths to a given destination are handled by hardware.

SPEED ISSUES IN SWITCH DESIGN

The bandwidth has a tremendous impact on the speed of a Butterfly Plus switch. It is a function of several factors:

Switch thickness	The bandwidth is linearly related to how many bits move across the switch during one clock period
Switch clock frequency	The bandwidth is linearly related to the switch clock frequency
Control overhead	Each message consists of data bits and control bits, thus the bandwidth is related to the ratio of data bits to message bits (*i.e.*, data bits plus control bits)
Memory bandwidth	If the switch speed is too high, data cannot be written into or read from the local memory fast enough to keep the switch path busy and still permit the local processor an occasional access to its memory.

The switch clock frequency is limited either by the time it takes to establish a connection between a switch node input and output port or by the time it takes to propagate the data to the next switch node. In very large switch configurations, the propagation time may be larger than the connection time.

The potential bandwidth of 32 MHz is further reduced because not every switch clock period transfers data. Assuming a switch base of four, with 128 processors and a thickness of four, Table 3-1 shows the reduction in bandwidth due to the inclusion of control bits in each message for several data block sizes for the unidirectional switch.

Table 3-1
Switch Bandwidth versus Message Size

Data bits	Message size	Effective Bandwidth
16	96	5.3 MHz
32	112	9.1 MHz
64	144	14.2 MHz
256	336	24.4 MHz

As can be seen, data sent or retrieved from remote memory in multiple word blocks uses the Butterfly Plus switch more efficiently.

DEAD STATES AND FLOW CONTROL

Most communications systems need flow control to prevent data from being lost, either initially, while a communications channel is being established, or later, if some processing element is unable to keep up with the system as a whole. In the Butterfly Plus switch, two types of flow control are required, since either the sending or the receiving PNC might not be able to keep up with the switch. Introducing flow control mechanisms typically causes the secondary problem of dead states: system states that persist indefinitely. Dead states can occur because of either hardware or software malfunctions, or as the result of design problems. Dead states that result from system design are termed deadlocks. The switch avoids deadlocks and times out all dead states caused by user error or hardware problems external to a working processor node. Most dead states caused by local hardware problems will also time out. The occurrence of these dead states is reported to the local operating system.

In the Butterfly Plus, dead states can occur for several reasons. Mainly, they occur because messages can be rejected due to switch contention (a type of flow control), and this rejection can lead to dead states when any of the following conditions occur:

- The addressed hardware is missing, inoperative, or flooded by some external malfunction
- The local receiver is malfunctioning or flooded
- The local transmitter or part of the switch is malfunctioning.

Once a message has been accepted, an empty transmitter buffer or a full receiver buffer can cause nulls to be sent. Hardware malfunctions can simulate these conditions, causing a dead state. A dead state can occur when a message requiring a reply is not processed because of a bad checksum or hardware error.

Two independent timers are used to time out dead states. When a request is submitted to the transmitter, one timer ensures that the request goes out within a reasonable period. If a reply is expected, the same timer is used to ensure

that it arrives within a reasonable period. A second timer does the same for the transmission of acknowledgement messages. The timeout intervals are long enough to make it unlikely that normal switch contention will exceed them, and yet short enough for the operating system to recover smoothly if timeouts occur.

Deadlocks are possible whenever a resource is needed to complete a task, but that resource is already in use and cannot be freed until the task completes. If a switch interface has only a single input message buffer and a single output message buffer, a deadlock such as that shown in Figure 3–13 can result from the following scenario:

1. Both node A and node B send memory request messages to each other simultaneously (memory request messages expect a reply).
2. After these requests have been processed, but before the replies have been sent, requests from two other nodes, X and Y, arrive at the inputs.

At this point, the replies in node A and node B cannot be sent because the receive buffers are occupied by the later requests. Yet, no further processing of requests can be done to free the receive buffers until the replies have been sent. The system is locked up. By ensuring that a reply message can always be sent and accepted, however, deadlock can be avoided. This requires a mechanism that accepts replies even if the receiver buffer is occupied. Providing two input buffers and two output buffers in each processor node, as in the Butterfly Plus, and adopting rules for how these buffers are used avoids the problem of deadlock.

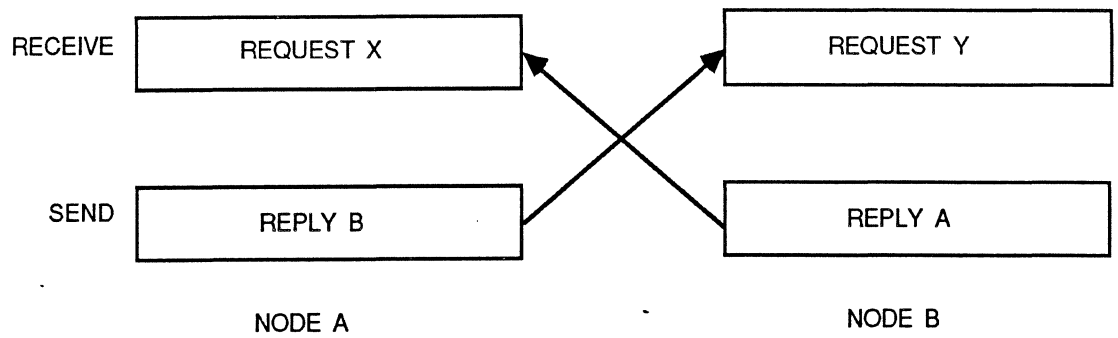


Figure 3-13
Deadlock in a Switch

Using two transmit buffers and allocating one exclusively for replies ensures that a reply can be sent in the first place. After a request is rejected and before retransmitting the request, the transmitter checks the reply buffer and, if a reply is present, may (randomly) send it before the request. The level of randomness in the switch and a reject retransmission delay randomizer in the switch interface transmitter ensure that replies do not repeatedly encounter the same or other request messages in the switch, thus also avoiding deadlock.

SWITCH DESIGN SUMMARY

In summary, the Butterfly Plus switch has the following characteristics:

Conflict Resolution	Uses the retreat strategy.
Parallel Data Paths	Data parallelism of four bits.
Switch Base	Uses a base of four.
Partial Switches	If the number of processors does not match a standard switch size, uses the next larger size of switch and depopulate its input ports for better performance.
Long and Short Messages	Supports both long messages (block transfers) and short messages (single word transfers), but breaks long messages into blocks of 256 bytes or less to keep the latency of short messages low.
Speed	Clocks the switch at 8-megabits-per-second for an effective bandwidth of 32 MHz per path; in a switch with 256 ports, this implies a maximum aggregate data rate of 8-gigabits-per-second.
Deadlock	Avoids by using the retreat strategy in the switch and by ensuring that the switch interface can always accept a reply.

Error Control

The switch itself does not perform error control, but the switch interfaces use check bits to detect both data errors and routing errors.

SWITCH NODE IMPLEMENTATION

Each channel through the switch comprises eight pathways, as shown in Figure 3–14. Four data paths, bits 0, 1, 2, and 3, together with one control path, FRAME, carry signals downstream; one control path, REJECT, carries a signal upstream. Two other signals, ND and HOLD, provide flow control during block transfers. Data path signals encode the routing information first, then the body of data, followed by a checksum, and then padding bits if required. (Padding might be needed if the message is so short that it would completely exit the transmitter before a possible REJECT signal could be returned by the switch.)

The FRAME signal in each message defines its head and tail. FRAME is generated by the interface unit supplying input to each node. Headers are used by the switch node in routing, and are passed downstream from the switch node in the form of a new FRAME signal to delimit the transmission. A header consists of the routing information, plus the message type, and, if the message is a request, the source address. As the head enters a node, a crosspoint link is formed; as the tail leaves the node, the link is broken.

The REJECT signal is generated by any node that is not free to establish the required link when the message head appears. The signal travels upstream, effecting a *retreat* by unlinking all segments of the channel as it goes, then directing the interface unit to take down the connection and try again later. A REJECT signal can also be generated by the destination switch interface to abort a message in progress.

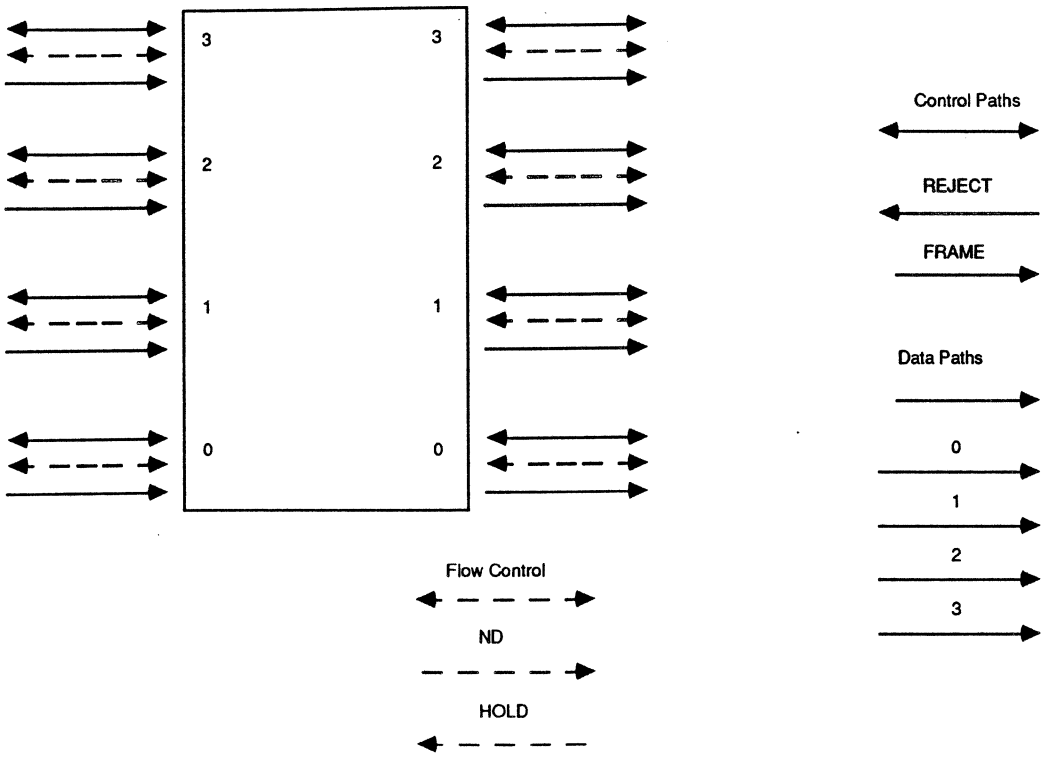


Figure 3-14
Base 4 Switch Node I/O Diagram

Circuitry along the buffered FRAME pathway detects the head and tail of the message. The head signal and the routing bits (encoded on data paths 0 and 1 during the leading digit of the message) are sent to the bid priority and crossbar assignment logic block. Taken together, head and routing bits constitute a bid to establish a crossbar link to the designated downstream port.

If the port is available and if no other message with higher priority is simultaneously bidding for use of the same port, the link is established; the assigned connection code is latched for the duration of the message, and fed from there to the crossbar switch to establish the connection. The data and FRAME signals are then sent through to the downstream port. The FRAME signal is delayed by one clock tick with respect to the data, thus discarding the old leading digit of the header (*i.e.*, the routing bits). This signal announces to the bid logic that the port is unavailable to other bidders because there is a message in transit.

If the port is unavailable at bid time, or if a higher priority bid is simultaneously being made, a local REJECT signal is returned to the losing bidder's channel logic, where the signal resets the frame flipflops and continues

upstream in retreat, becoming a *downstream REJECT* signal at the next node upstream. Here the signal again clears the frame flipflops and continues upstream, traversing one node per clock tick.

Chapter 4

The Multibus Adapter

The Butterfly Plus Multibus adapter is a printed circuit card that plugs into a 9-slot Multibus card cage and attaches to a Butterfly Plus processor node via its BIOLINK cable. On the Multibus adapter card edge are a toggle switch and four connectors. The toggle switch activates a watchdog timer and is normally toggled off (to the right) to prevent the Watchdog Timer from affecting the Butterfly Plus. The 14-pin connector supplies a system clock for small Butterfly configurations. The two 26-pin connectors form a null switch interface that attaches to the processor node and emulates a 2-column switch for small Butterfly configurations. The 60-pin connector attaches the Multibus adapter to the BIOLINK connector of a processor node.

The Multibus card cage is a 9-slot horizontal card cage that installs directly into the Butterfly rack. It has a reset switch and a toggle on/off switch on the right front panel. The reset switch initializes only the Multibus card cage. There are always a minimum of three cards installed in the first Multibus card cage: the Multibus adapter card, a RAMboot card, and an Ethernet controller card. The Multibus Adapter card is always in the topmost slot of the card cage to accomodate the serial lines. The bottom slot is reserved for slave boards only. This slot has no connections for interrupt signals.

By communicating with a processor node over its BIOLINK, the Multibus adapter makes Multibus memory and I/O devices accessible to the Butterfly Plus. The Multibus adapter conforms electrically, mechanically, and functionally to the IEEE 796 (Multibus) specification with two exceptions: it uses some of the undefined signals on the Multibus P2 connector, and, when acting

as a Multibus slave, it does not always return a data acknowledge signal to the Multibus master within the IEEE 796 specification of eight microseconds. Because of BIOLINK latency, the Multibus adapter returns data acknowledgement signals within 25 microseconds.

The Multibus adapter consists of five independent subsystems:

- A data channel connecting the Multibus to a processor node
- A null switch interface
- A host and console UARTs
- A clock and reset connector
- An EPROM.

The data channel between the BIOLINK and the Multibus allows the Butterfly Plus processor node to access memory and I/O device data on the Multibus, and allows devices on the Multibus to access local memory on the processor node to which the Multibus adapter is attached. Besides transferring data between the Multibus and a processor node, the adapter also facilitates Multibus interrupt processing by mapping each Multibus interrupt request level into one of two MC68020 interrupt requests, in either vectored or non-vectored format. Each interrupt request level can be enabled individually, and there is a general interrupt disable function. The data channel also allows Multibus devices to post events on the Butterfly Plus.

Three of the five functional subsystems in the Multibus adapter, the null switch interface, the host and console UARTs, and the clock and reset connector, are used to reduce hardware needs in small Butterfly systems. Each of these subsections duplicates key functions of certain Butterfly Plus circuit cards. Butterfly Plus clock cards, switch cards, and the host and console connections on the Ethernet fantail all have functional counterparts on the Multibus adapter. In addition, the adapter contains a 2-kilobyte EPROM to provide configuration information.

Butterfly Plus software controls each subsystem within the Multibus adapter through a set of memory mapped control registers. Many functions of the Multibus adapter are implemented by setting and clearing bits in its Misc Register, which is mapped at FFF7D026 in the Butterfly Plus virtual address space. The Multibus adapter Misc Register is instrumental in controlling

Multibus interrupt requests and memory management functions, as well as other tasks. A layout of the Misc Register appears later in this chapter in Table 4–8.

NULL SWITCH INTERFACE

The null switch interface is an independent subsystem within the Multibus adapter that provides some of the functions of a 2-column Butterfly Plus switch. This allows development systems with only one processor to operate without a Butterfly Plus switch card. The null switch interface also generates its own master clock and reset signals, eliminating the need for a separate clock card in the Butterfly Plus. The null switch interface connects to a processor node through its pair of 26-pin switch receiver and transmitter cables. The transmitter cable connects the leftmost 26-pin connector on the front of the Multibus adapter card to the switch transmitter port on the processor node. The receiver cable connects the rightmost 26-pin connector on the Multibus adapter to the switch receiver port on the processor node.

Even in a 1-processor Butterfly Plus system, the processor still needs to send messages across the switch under certain circumstances. The null switch interface allows this by emulating a 2-column switch and passing messages from the switch transmitter to the switch receiver selectively, based upon their addresses. The null switch does this by stripping off the address bits before passing the messages to the processor node. It passes only messages addressed to processor 0. It handles all the communication details of a real switch, such as deleting the address from the beginning of a message. Messages that are not addressed to processor 0 are ignored. A Butterfly Plus processor node obtains its master clock and reset signals from a switch card via its switch receiver and transmitter connectors. The switch card in turn obtains clock and reset signals from a clock card. In a 1-processor system that has no switch card, the null switch interface supplies master clock and reset signals to the processor node. The reset signal has two possible sources within the null switch interface: a pair of pins on the Multibus P2 connector and the watchdog timer circuit. The reset signal is fed to the transmitter connector of the null switch interface, where it initiates a power up reset of the processor node connected to the interface.

Two pins on the Multibus P2 connector, pins 11 and 13, control a flipflop that drives the reset signal. A TTL low signal applied to pin 13 sets the flipflop and asserts the reset line. A TTL low signal applied to pin 11 resets the flipflop and inactivates the reset line. Because the flipflop effectively debounces pins 11 and 13, a mechanical switch can be connected to these pins directly.

The standard Multibus reset signal, called INIT, does not reset the Multibus adapter, nor does it cause the adapter to reset the processor node. The Butterfly Plus I/O system reset signal, called BIORES, does reset the Multibus adapter, however, and causes the adapter to assert the Multibus INIT signal. Multibus resets do not reset the Butterfly Plus.

A connector on the Multibus adapter card can supply system clock and reset signals to one or two switch cards. This 14-pin clock connector is identical to the connector on a standard Butterfly Plus clock card. In Butterfly Plus systems with up to 16 processors that require only one switch card, the Multibus adapter can replace the clock card.

WATCHDOG TIMER

The Multibus adapter contains a watchdog timer, controlled by a bit in the Misc Register, that can reset the processor connected to the clock connector. Once enabled, the watchdog timer activates the reset signal for 500 nanoseconds if register bit 4, the watchdog timer control bit, is not toggled at least once in about 12 seconds. The watchdog timer is disabled when the Multibus adapter is reset or power is cycled. It is enabled when the watchdog timer control bit in the Misc Register is set for the first time. To enable the watchdog timer, the word 0010H is ORed into location FFF7D026. Thereafter, the word 0010H must be XORed into the Misc Register at least about once every five seconds to prevent the watchdog timer from issuing a reset. If for some reason the watchdog timer does issue a reset, the toggle mechanism automatically begins to reXOR the word 0010H into the Misc Register every five seconds. A toggle switch on the front of the Multibus adapter card disables the watchdog timer when toggled to the right.

HOST AND CONSOLE UARTS

The Multibus adapter contains two UARTs, brought out to connectors on the Ethernet fantail, that allow console terminals, load devices, or a host computer

to be attached. The host and console UARTs use independent programmable serial communication channels. Each can interrupt the processor at level 4 on either transmitter empty or receiver full conditions. These UARTs are normally set to operate at 9600 baud with eight data bits, one stop bit, and no parity.

Each of the two UARTs used in the Multibus adapter is an SC2661 enhanced programmable communication interface (EPCI) that performs full duplex serial communication in either synchronous or asynchronous format at data rates up to 19,200 baud. Both UARTs are programmed through a set of six control registers. The registers have the following structure:

```
struct UART  {
  short   data ;      /* EPCI data register          */
  short   status ;    /* EPCI status register        */
  short   mode ;      /* EPCI mode control register  */
  short   command ;   /* EPCI command register       */
  short   vector ;    /* interrupt vector register   */
  short   control ;   /* interrupt control register  */
} ;
```

Each register is 16-bits wide. The high-order byte is undefined on reads and ignored on writes. The first four registers, the data, status, mode, and command registers, are inside the EPCI. These registers control the baud rate, parity, and other aspects of the actual communication. Refer to the EPCI data sheet to initialize and use these registers. The interrupt vector and interrupt control registers are implemented in the Multibus adapter.

The host and console UARTs use their associated interrupt control register and interrupt vector register to enable transmitter and receiver interrupts. Receiver full and transmitter empty interrupts are individually enabled for each UART by configuring the appropriate interrupt control register. In each interrupt control register, bits 15–2 are unused and must be zero, bit 1 is set to enable transmitter empty interrupts, and bit 0 is set to enable receiver full interrupts. Figure 4–1 illustrates the bit map for the interrupt control registers.

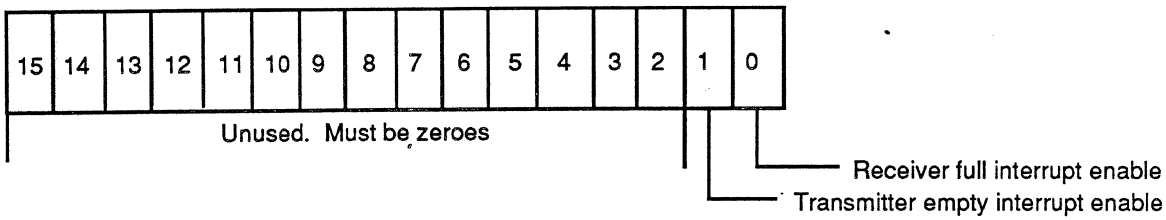


Figure 4-1
UART Interrupt Control Register

The interrupt vector registers contain the values supplied to the processor when it responds to a UART interrupt. In each interrupt vector register, bits 15–8 are unused and must be zero, bits 7–1 are the interrupt vector, and bit 0 is a read-only bit set when there is a receiver interrupt pending. This distinguishes between receiver and transmitter interrupts during processor vector fetches. Figure 4–2 illustrates the bit map for the interrupt vector registers.

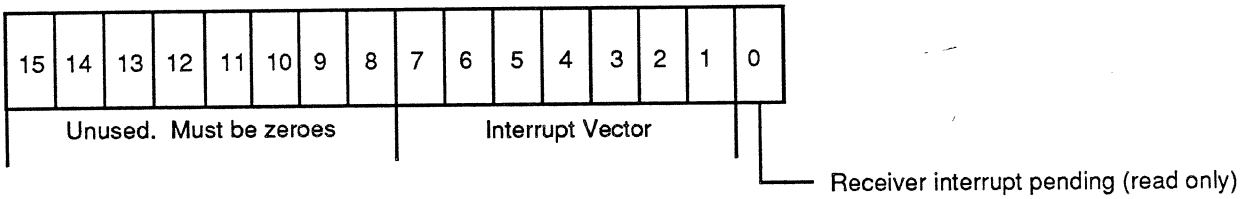


Figure 4-2
UART Interrupt Vector Register

The control registers for the host and console UARTs are mapped into the Butterfly Plus address space beginning at FFF7D000, as shown in Table 4–1.

Table 4-1
UART Control Register Addresses FFF7D000

Address	Register
FFF7D000	Console Data Register
FFF7D002	Console Status/Synch Register
FFF7D004	Console Mode Register
FFF7D006	Console Command Register
FFF7D008	Console Interrupt Vector Register
FFF7D00A	Console Interrupt Control Register
FFF7D010	Host Data Register
FFF7D012	Host Status/Synch Register
FFF7D014	Host Mode Register
FFF7D016	Host Command Register
FFF7D018	Host Interrupt Vector Register
FFF7D01A	Host Interrupt Control Register

Host and console UART interrupt requests enter the Butterfly Plus processor node at level 4 and always have higher priority than any Multibus interrupt request. Receiver interrupts have higher priority than transmitter interrupts. When the PNC responds to a UART interrupt from the Multibus adapter, it reads the appropriate interrupt vector register and passes the resulting value to the processor after setting the low-order bit (to distinguish between receiver and transmitter interrupts).

The processor (and consequently, user software) is effectively insulated from the interrupt vector fetching process. The PNC determines where an interrupt originated and then acquires the correct interrupt vector. For example, the Multibus adapter can have five separate interrupt requests pending at level 4: receive and transmit interrupts from both the host and the console, and a Multibus interrupt mapped into level 4. Microcode in the PNC determines the order in which these interrupts are serviced, while the processor merely fetches a vector from the PNC. The PNC services I/O interrupts in the following order:

1. Host receiver (highest priority)
2. Console receiver
3. Host transmitter
4. Console transmitter
5. Multibus interrupts mapped into MC68020 interrupt request level 4
6. Level 4 interrupt requests from other BIOLINK cards
7. Multibus interrupts mapped into MC68020 interrupt request level 3
8. Level 3 requests from other BIOLINK cards (lowest priority).

EPROM

For applications that require non-volatile storage, the Multibus adapter contains a 2-kilobyte EPROM. This EPROM can be used for configuration information that may be useful after a power outage. User programs can read and write this EPROM through its address register and its data and control register, which are mapped into Butterfly Plus address space at FFF7D020 and FFF7D022, respectively. The reads/write cycle of the EPROMs is about 1,000 read/writes, so diagnostics should be performed carefully to save the EPROM. The EPROM Address Register is used to drive the address lines of the EPROM. Although this register is 16 bits wide, only the low-order eleven bits (bits 10–0) are used for the EPROM address. The EPROM Data and Control Register is used both to supply data to the EPROM and to manipulate its control signals. Table 4–2 shows the EPROM Data and Control Register format. Figure 4–3 illustrates the bit settings of this register.

Table 4-2
EPROM Data and Control Register (FFF7D022) Layout

Bits	Active Level	Function
15–12		Unused.
11	high	Write data to data holding register.
10	high	Strobe data into EPROM.
9	low	Read data from EPROM.
8	low	Enable EEPROM.
7–0		Data bits 7–0.

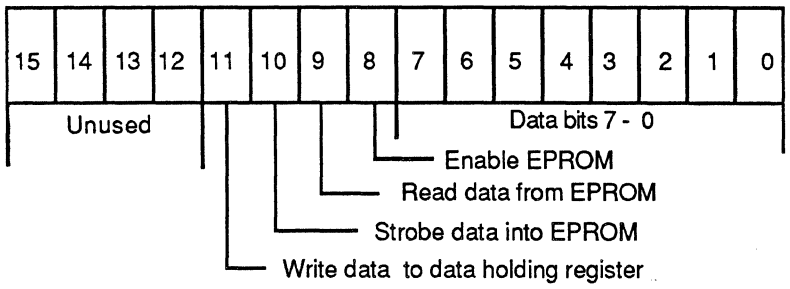


Figure 4-3
EPROM Data and Control Register Bit Layout

A program can read and write the EPROM by following the procedures described in Figures 4–4 (reading an EPROM) and 4–5 (writing an EPROM). Note that "xx" represents a byte of data.

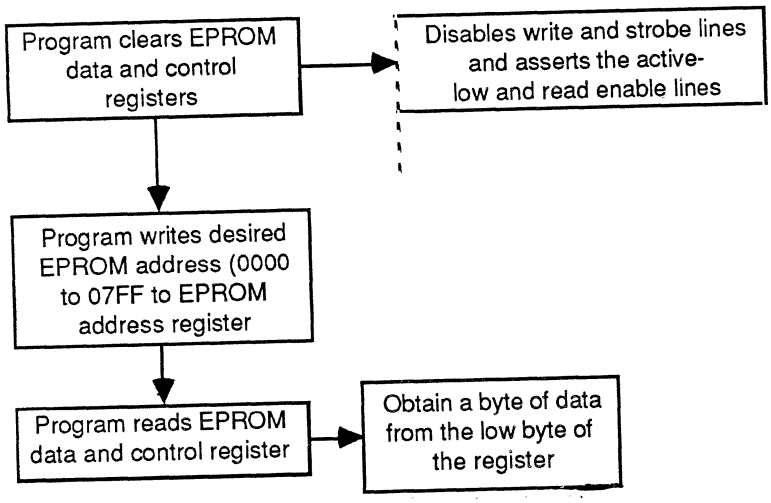


Figure 4-4
Reading the EPROM

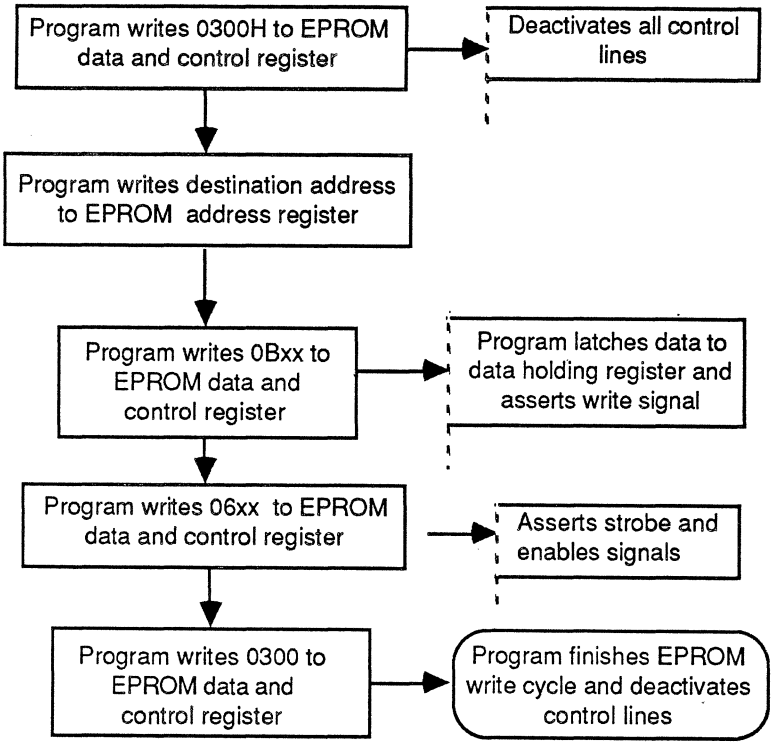


Figure 4-5
Writing the EPROM

MULTIBUS DATA TRANSFERS

The Multibus adapter is primarily a communication channel between the Multibus and the BIOLINK. It has a 16-bit data path that supports both word and byte operations and a typical data transfer rate of two million bytes per second. A Butterfly Plus longword access to Multibus memory or I/O data takes two Multibus cycles, but is otherwise identical to a 16-bit transfer. The detailed mechanics of data transfer are transparent to both the Multibus device and the Butterfly Plus MC68020 processor. That is, the local memory of the Butterfly Plus processor node to which the Multibus adapter is attached appears to be "on" the Multibus, while Multibus memory or I/O device registers appear to be "on" the BIOLINK of the processor node.

The adapter maps Multibus addresses to Butterfly Plus physical addresses, and the processor node maps Butterfly Plus physical addresses to Multibus addresses. A Butterfly Plus program accesses the Multibus using Butterfly Plus virtual addresses. The MC68851 paged memory management unit translates these virtual addresses into physical addresses. The low-order bits of a virtual address are generally identical to the low-order bits of the corresponding physical address. Since Multibus adapter circuitry decodes only these low-order bits, the distinction between virtual and physical addresses is usually unimportant if programmers realize that the high-order bits of a virtual address are established by system software and subject to change. All addresses cited in this chapter are virtual addresses. To protect against address mapping changes, programs should reference these addresses symbolically through use of an operating system header file.

There is a basic byte order incompatibility between the MC68020 processor and the Multibus. In word-oriented data, the Multibus addressing convention assumes that the least significant byte is stored at the lower (even numbered) address and the most significant byte is stored at the higher (odd numbered) address. The MC68020 addressing conventions use the opposite byte order. To remedy this problem, the Multibus adapter has a programmable byte swapping facility. During word transfers, when swapping is enabled, the byte order is reversed on the data bus. During byte transfers the least significant address bit is inverted. Thus, when swapping is enabled, all even addresses access odd byte locations and all odd addresses access even byte locations.

Multibus Access to Butterfly Plus Memory

A Multibus device can only access the Butterfly Plus memory of the processor node to which the Multibus adapter is attached. Multibus devices cannot access memory on remote processor nodes over the Butterfly Plus switch. Four types of Multibus access to Butterfly Plus local memory are supported: word reads, word writes, byte reads, and byte writes. A pipeline register can be used to reduce BIOLINK bus latency for block transfers that access consecutive Butterfly Plus memory locations. Multibus devices can also post events on the local Butterfly Plus node.

The Multibus has 24 address lines, giving it a 16-megabyte address space. By convention, half of the 16-megabyte Multibus address space is used to map four megabytes of Butterfly Plus memory in both normal and byte swapped

form, leaving eight megabytes of address space for Multibus memory and memory mapped I/O devices. The Multibus adapter maps Butterfly Plus local memory (on the processor node to which is attached) directly into the Multibus memory address space at the same physical address. It also maps a byte swapped version of this same Butterfly Plus memory into the Multibus memory address space beginning at location 400000H. Therefore, a Multibus device reading or writing any location between 000000 and 3FFFFF accesses the word or byte of memory at that physical address on the Butterfly Plus processor node, and by reading or writing 400000H higher than that location, the Multibus device will access the same Butterfly Plus memory data in byte swapped form.

Multibus memory mapped I/O devices are mapped into the Multibus memory address space between location 800000H and location BF0000. The Multibus adapter maps certain PNC special functions into a 64-kilobyte block of Multibus address space beginning at location BF0000 for use in posting events on the Butterfly Plus. The remaining Multibus address space is available for Multibus memory devices, such as the Multibus RAMboot and Ethernet cards. Application programs can alter the standard Multibus address mappings by changing entries in the adapter's mapping RAM; however, standard system software relies on the conventional mapping.

Butterfly Plus Access to Multibus Data

The Butterfly Plus processor node to which a Multibus adapter is attached can access both memory and I/O device addresses on the Multibus. Adapter hardware supports four types of Butterfly Plus access to Multibus memory directly: word reads, word writes, byte reads, and byte writes. A Butterfly Plus program can use longword reads and writes to access Multibus memory, however the Multibus adapter performs these 32-bit accesses in two separate 16-bit Multibus cycles.

Chrysalis maps the 64-kilobyte Multibus I/O address space into Butterfly Plus virtual address space beginning at address FFE40000; therefore, the MC68020 can access a Multibus I/O device register by reading or writing the Butterfly Plus memory location at FFE40000 more than the Multibus I/O register address. Chrysalis also maps a byte swapped version of this same Multibus address space beginning at virtual address FFF40000.

Usable Multibus memory addresses begin at 800000H, since two copies of Butterfly Plus memory are mapped into the lower half of the Multibus memory address space. This 8-megabyte region of Multibus memory address space is mapped into Butterfly Plus virtual address space beginning at address FF00000. A portion of the byte swapped version of this same Multibus memory data can be mapped into Butterfly Plus virtual address space, beginning at address FF000000, by means of the **Map_MB_Swapped** call to the Chrysalis operating system. Therefore, the MC68020 can access a Multibus memory location by reading or writing the Butterfly Plus memory location at FE800000 more than the Multibus memory address, and it can also obtain the byte swapped version of this same data after calling **Map_MB_Swapped**. Rather than using the absolute virtual addresses cited here, a program would invoke the Chrysalis function **Map_MB** (or **Map_MB_Swapped**) and obtain a pointer to the Multibus memory data. Figure 4–6 illustrates the Butterfly Plus and Multibus virtual address maps.

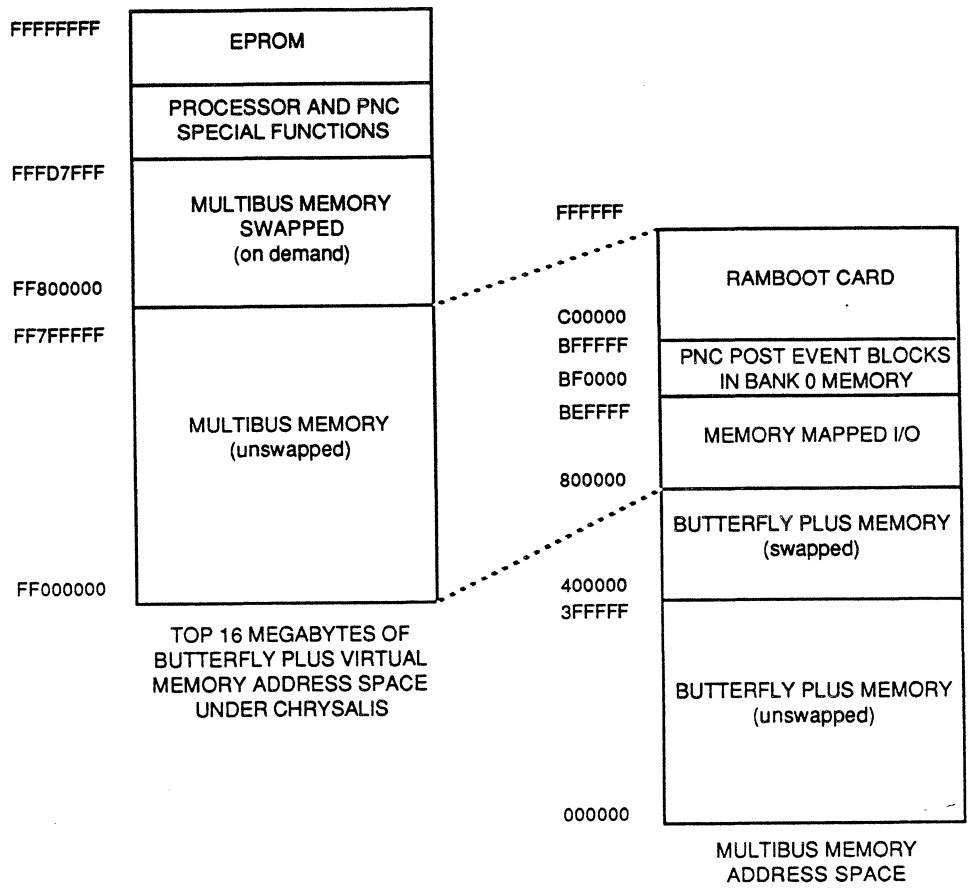


Figure 4-6
Butterfly Plus and Multibus Address Maps under Chrysalis

THE MULTIBUS ADAPTER PIPELINE

When a Multibus device reads or writes to the BIOLINK, the PNC can take as long as 25 microseconds to respond. This possible lag in response occurs because the PNC disables microinterrupts while performing certain functions, some of which—like posting events or scheduling tasks—take many microseconds to complete. To reduce the effects of this latency, the Multibus adapter implements a simple pipeline between Multibus devices and Butterfly Plus memory. The pipeline works for both byte and word operations. It is always active when the Multibus writes to the Butterfly Plus and it can be selectively enabled or disabled when the Multibus reads from the Butterfly

Plus. Misc Register bit 3 (mask value 0x08) is the pipeline enable bit, which is set to one to enable the pipeline and cleared to zero to disable it. To enable pipelined reads of Butterfly Plus memory, for example, OR the word value 0008H into location FFF7D026. The pipeline is not used for transfers that originate at the Butterfly Plus, when the Butterfly Plus reads or writes to the Multibus.

Pipelined Writes to the Butterfly Plus

When a Multibus device writes a word or byte to Butterfly Plus memory, the adapter buffers the address and data in its own registers. It then acknowledges the transaction to the Multibus device, freeing the Multibus for another transaction. Meanwhile, the Multibus adapter waits, if necessary, for the Butterfly Plus to respond, and then completes the transaction.

Pipelined Reads from the Butterfly Plus

When a Multibus device reads a word or byte from Butterfly Plus memory, the Multibus adapter and the PNC collaborate on a prefetching scheme, which assumes that the next address read by a Multibus device will be two greater than the last address that was read. When the pipeline is enabled during a Multibus read request to the Butterfly Plus, the Multibus adapter and the PNC execute one of two sequences, depending on whether the pipeline is full or empty. (The pipeline is empty after a Multibus reset and after any Multibus operation except a pipelined read. It is full only after a pipelined read.) If the pipeline is empty, the Multibus adapter and the PNC execute a preset sequence whenever a Multibus device reads Butterfly Plus memory. Figure 4–7 illustrates this sequence in a flowchart.

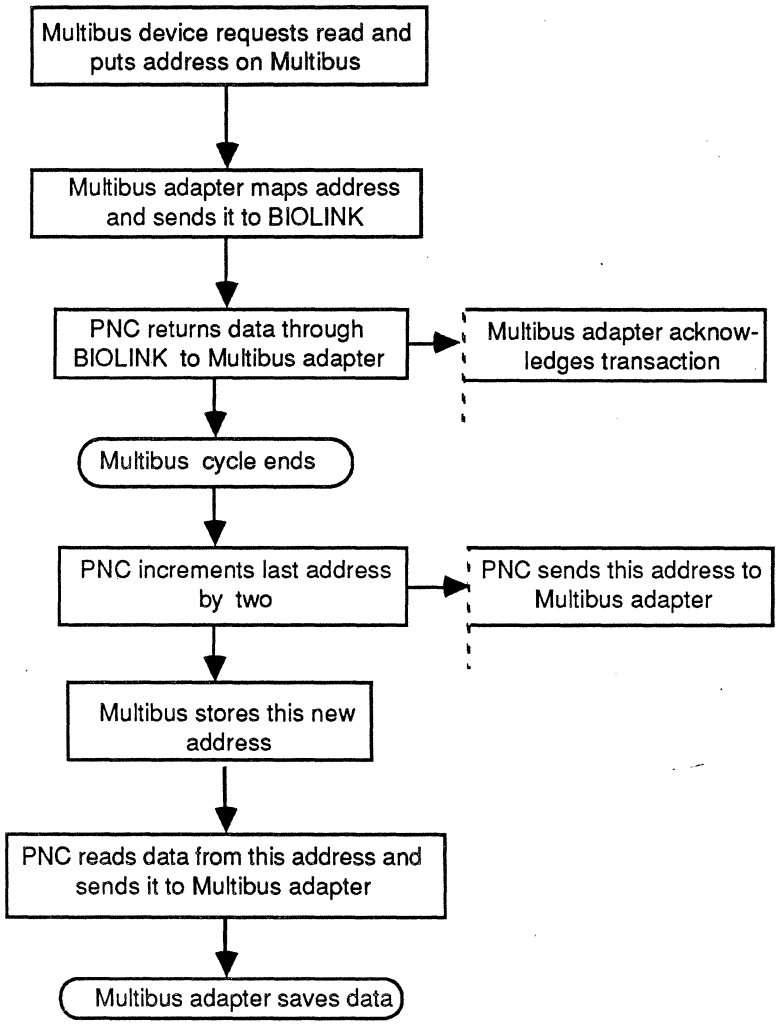


Figure 4-7
Pipeline Empty Multibus Read

If the pipeline is full when it is enabled, a preset sequence is performed whenever a Multibus device reads Butterfly Plus memory. Figure 4–8 illustrates this sequence in a flowchart.

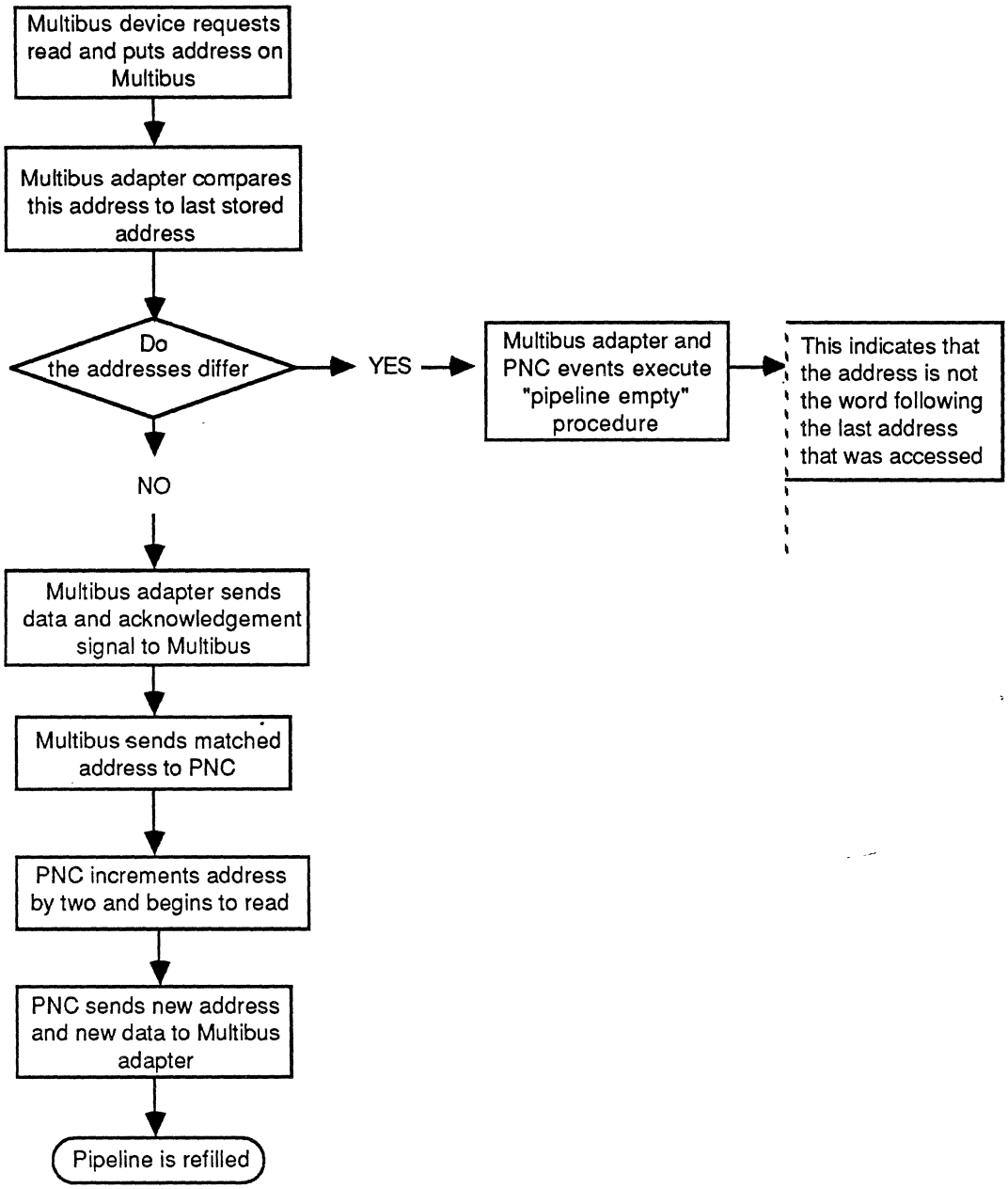


Figure 4-8
Pipeline Full Multibus Read

Although pipelining can be enabled or disabled at any time during read operations, the only case where pipelined reads result in a significant savings of time is on large block transfer operations, where the ratio of address matches to total accesses is high. If this ratio is not high enough, the pipeline can actually slow down the transfer.

Multibus Data Transfer Timing

The amount of time required to perform a Multibus data transfer can vary widely. Table 4–3 shows the best, worst, and typical total cycle time for each type of access, measured from the time one access begins until the earliest time that a subsequent access can begin. The table shows timing for full word accesses only. Note that byte access takes the same amount of time as word access, and longword access takes twice as long. The table also shows the elapsed time from the beginning of an access until data acknowledge is returned. When a Butterfly Plus node accesses the Multibus, the access cycle begins after the processor asserts the address strobe. When a Multibus device accesses the Butterfly Plus node, the cycle begins after the Multibus device asserts its bus request line.

Table 4-3
Multibus Data Transfer Timing

	Total Cycle Time (μsec)			Time to Data Acknowledge		
	Best	Worst	Typ.	Best	Worst	Typ.
Butterfly reads Multibus	2.1	20	2.4	2.1	20	2.4
Butterfly writes Multibus	2.4	20	2.7	2.4	20	2.7
Multibus reads Butterfly (without pipeline)	1.3	25	1.5	0.9	25	1.0
Multibus reads Butterfly (with pipeline)	1.3	25	1.5	0.3	0.3	0.3
Multibus writes Butterfly	0.8	25	1.0	0.3	0.3	0.3

POSTING EVENTS FROM THE MULTIBUS

Multibus devices can post events on the local Butterfly Plus processor node by writing to a pre-defined special address in the Multibus memory address space. Using its mapping RAM, the adapter converts this Multibus address into a Butterfly Plus physical address, passes it to the PNC, and notifies the PNC that it is requesting a post event operation (as opposed to a normal memory access). The PNC uses the Butterfly Plus physical address as a pointer to the parameter block for the event, and then proceeds to post the data from the Multibus to this event. When implementing Multibus post event operations, a Butterfly Plus program should use the following sequence of operations:

1. Allocate and initialize an event block through a call to the Chrysalis **Make_Event** function. **Make_Event** returns an event handle, which serves as a pointer to the event block.

2. Create a parameter block for the event. The parameter block has the following structure:

```
struct  Event_Parameter_Block {
  long    Event_Handle;
  long    Post_Data;
  short   Reply_Code;
  } ;
```

   By convention, the parameter block should occupy bank 0, the lowest 64 kilobytes of Butterfly Plus physical memory (*i.e.*, the header segment), which is the only area normally mapped to Multibus address space for this purpose. The Chrysalis function **Make_PNC_Block** can be used to obtain bank 0 storage space for a parameter block.

3. Pass the address of the parameter block to the Multibus device. The procedure for this depends on which particular device will do the posting.

4. If the parameter block is not in bank 0, the lowest 64 kilobytes of Butterfly Plus physical memory, configure the Multibus adapter mapping RAM location that corresponds to the physical address of the parameter block. For this location, the segment attribute field must be set to *special.* The Butterfly Plus segment field must correspond to bits 21–16 of the physical address of the parameter block. If necessary, enable the mapping RAM. Omit this step if the parameter block is in bank 0, which is already mapped to Multibus address BF0000, or if **Make_PNC_Block** was used to obtain header block storage space.

Upon receiving the post event request, the PNC checks the parameter block to ensure that the event handle is valid and that the request is appropriate. If there are no errors, the PNC writes the data word from the Multibus into the low-order word of the **Post_Data** field in the parameter block, posts the data with the event specified by the **Event_Handle**, and writes the **Reply_Code** into the parameter block. The following is a programming example for the posting of events from the Multibus.

```
char * MB_Event ( ) Creates an event for use by a Multibus device.
                    Returns the Multibus address to which the
                    device should write a 0 to cause the post.
{  short tempsr = entker;
   oid pblk;
```

```
    char * MB post addr;
    pncblock pbp;
    pnc_pb * pb;
      pblk = Make_PNC_Block ( );
      pbp = map_oab (pblk,0,0);
      pb = & (pbp->pnc_blk.post);
      pb->p_handle = Make_Event (0,0,0,0);
      pb->p_postdata = <any datum>;
      pb->p_reply = 0;
      unmap_oab (pbp);
      restoreker (tempsr);
      return (((int)pb & 0xffff)    0xbf0000);
}
```

SERVICING MULTIBUS INTERRUPT REQUESTS

The Multibus adapter can forward Multibus interrupt requests to the MC68020 in the processor node. Software can map each Multibus interrupt request level separately into either a level 3 or a level 4 Butterfly Plus processor interrupt, and assign the priority of Multibus interrupts. Each Multibus interrupt level can be either vectored or non-vectored. In the case of vectored interrupts, the Multibus adapter can perform a 2-cycle interrupt acknowledge operation on the Multibus and pass the resulting interrupt vector to the PNC. Each Multibus interrupt level can be individually enabled or disabled, and there is also a general interrupt disable function.

There are eight Multibus interrupt levels, numbered 0 to 7. The Multibus adapter contains a user-programmable interrupt vector RAM, which determines the hierarchy of these interrupts and specifies the Butterfly Plus processor interrupt level, if any, that corresponds to each Multibus interrupt level. The vector RAM uses the eight Multibus interrupt request lines as its address lines. Every possible combination of interrupt requests selects one location in the vector RAM. The content of each vector RAM location specifies the highest priority interrupt request, among those currently active, and determines whether that interrupt request is mapped to MC68020 interrupt level 3 or level 4.

The Multibus adapter requests a Butterfly Plus processor interrupt in two situations:

- When one or more Multibus interrupt lines become active and select an interrupt vector RAM location that indicates a level 3 or level 4 Multibus interrupt request

- When one of the UARTs on the Multibus adapter has an interrupt pending.

When the Butterfly Plus processor acknowledges a level 3 or level 4 interrupt request, the PNC first determines if it is attached to a Multibus adapter. If so, the PNC reads the adapter Interrupt Status Register to determine the contents of the currently selected interrupt vector RAM location and the status of the adapter host and console UART. The layout of the Interrupt Status Register is shown in Table 4–4 and Figure 4–9.

Table 4-4
Multibus Adapter Interrupt Status Register Layout FFF7D028

Bit	Description
15–11	Not used.
10	Set to one if Multibus interrupts are enabled.
9	Set to one if there is a host UART interrupt pending.
8	Set to one if there is a console UART interrupt pending.
7–0	Interrupt vector RAM contents; indicates Multibus interrupt status.

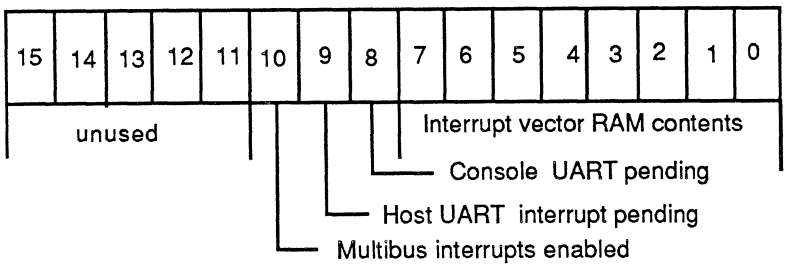


Figure 4-9
Multibus Adapter Interrupt Status Register Bit Map

If either of the two UART interrupt request bits (8 or 9) is set, the PNC services the UART interrupt first. If there is a pending Multibus interrupt request (bit 10), the PNC examines the interrupt vector RAM to determine which Multibus interrupt request level initiated the interrupt request. Bits 0 to 7 of the

Multibus Adapter Interrupt Status Register indicate to the MC68020 microprocessor whether the Multibus adapter wants to interrupt the Butterfly Plus at level 3 or level 4 (bits 3 or 7), and what Multibus interrupt level is being invoked at that Butterfly Plus processor level (bits 0 to 2 or bits 4 to 6). The PNC then takes the Multibus interrupt request level represented by bits 0 to 2 or 4 to 6 and multiplies it times two to obtain the index to read one of the eight 16-bit values from a vector table that begins at FF00005E (symbolic label BMA_vec) in Butterfly Plus address space. If this 16-bit value is nonzero, it is returned to the Butterfly Plus processor as the default interrupt vector. If the 16-bit value is zero, the PNC instructs the Multibus adapter to perform an interrupt acknowledge cycle on the Multibus and return the value obtained. It then passes this value to the Butterfly Plus processor as the interrupt vector. If no Multibus device responds to the interrupt acknowledge cycle, the PNC passes the spurious interrupt vector to the Butterfly Plus processor. The boot EPROM initializes BMA_VEC to 48H - 4FH.

Programming the Interrupt Vector RAM

The boot EPROM configures the Multibus Adapter so that Multibus interrupt requests are normally passed to the Butterfly Plus processor as MC68020 level 3 interrupt requests in standard priority sequence (*i.e.*, Multibus level 7 interrupt requests have highest priority). By changing the adapter's interrupt vector RAM, a program can alter the Butterfly Plus interrupt request level for Multibus interrupt requests, change the sequence in which Multibus interrupt requests are forwarded to the MC68020 at that level, and selectively disable Multibus interrupt requests. The adapter's interrupt vector RAM begins at FFF7F000 in the Butterfly Plus address space. The lower eight bits of an interrupt vector RAM address correspond exactly to combinations of Multibus interrupt request lines 7 to 0, where the level 7 request line is the most significant bit of the address. Each 1-byte interrupt vector RAM location has the layout shown in Table 4–5 and Figure 4–10.

Table 4-5
Multibus Adapter Interrupt Vector RAM Layout FFF7F000

Bit	Description
7	Set to indicate a level 4 interrupt request.
6–4	Specify which currently active Multibus interrupt level has the highest priority at level 4.
3	Set to indicate a level 3 interrupt.
2–0	Specify which currently active Multibus interrupt level has the highest priority at level 3.

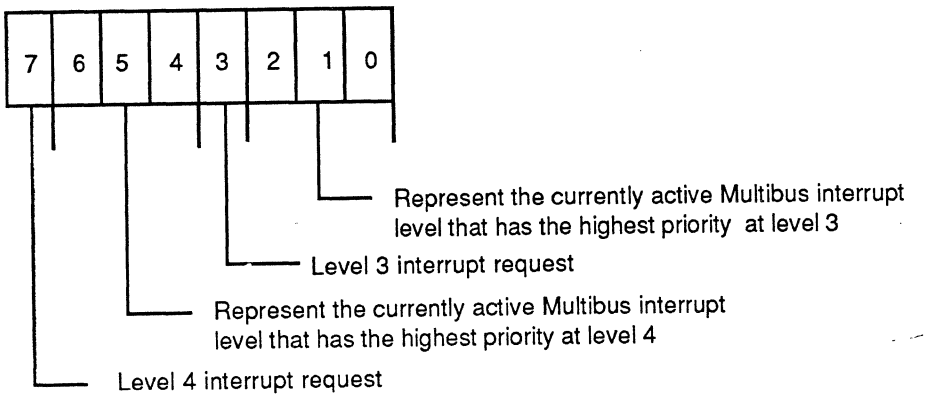


Figure 4-10
Multibus Adapter Interrupt Vector RAM Bit Map

As a simple example, assume that only Multibus interrupt request levels 1 and 2 are in use. These two interrupt request levels together have four possible states: neither is active, only level 1 is active, only level 2 is active, or both 1 and 2 are active. The four states correspond, respectively, to the four interrupt vector RAM locations FFF7F000, FFF7F002, FFF7F004, and FFF7F006, which could be programmed as shown in Table 4–6. In this example, both of the Multibus interrupt request levels were mapped into MC68020 interrupt request level 3. Vector RAM location FFF7F006, corresponding to both interrupts active, gives Multibus level 1 requests priority over Multibus level 2 requests.

Table 4-6
Vector RAM Programming Example

Active	Address	Level 4 (bit 7)	Highest at 4 (bits 6–4)	Level 3 (bit 3)	Highest at 3 (bits 2–0)
None	FFF7F000	0	x	0	x
1	FFF7F002	0	x	1	1
2	FFF7F004	0	x	1	2
1 and 2	FFF7F006	0	x	1	1

Enabling Multibus Interrupts

Before the Multibus can interrupt the Butterfly Plus processor, Multibus interrupts must be explicitly enabled by setting bit 0 in the Multibus Adapter Misc Register. Clearing Misc Register bit 0 disables Multibus interrupts. This interrupt enable bit affects Multibus interrupts only; host and console UART interrupts are enabled separately. The boot EPROM enables Multibus interrupts during initialization.

MULTIBUS MEMORY MANAGEMENT

The Multibus adapter divides the Multibus memory address space into 256 segments, each 64-kilobytes long. The eight most significant bits of a Multibus physical address specify which segment is being accessed. Each Multibus memory segment can be mapped to access Butterfly Plus physical memory or left unmapped to access Multibus memory. The Multibus adapter stores a 2-bit attribute tag for each segment that determines how Multibus devices access the segment. Table 4–7 shows the four possible values for this 2-bit attribute tag. The *bit value* column refers to the coding of bits 6 and 7 of the Multibus mapping RAM.

Table 4-7
Segment Attribute Tag Values

Bit Value	Code	Type	Description
00	0	Normal	Multibus devices access Butterfly Plus memory normally.
01	1	Special	Multibus devices access PNC special registers, potentially invoking a Butterfly special function.
10	2	Swapped	Multibus devices access Butterfly memory, but the byte order is reversed.
11	3	No Reply	The Multibus adapter will not respond to any addresses in this segment.

The Multibus adapter performs address mapping by using the eight high-order Multibus address bits as an index into a 256-byte lookup table called the mapping RAM. Each mapping RAM entry contains address bits 21–16 of a Butterfly Plus physical memory segment, along with two bits that indicate the attribute of the segment. Figure 4–11 diagrams this address translation process.

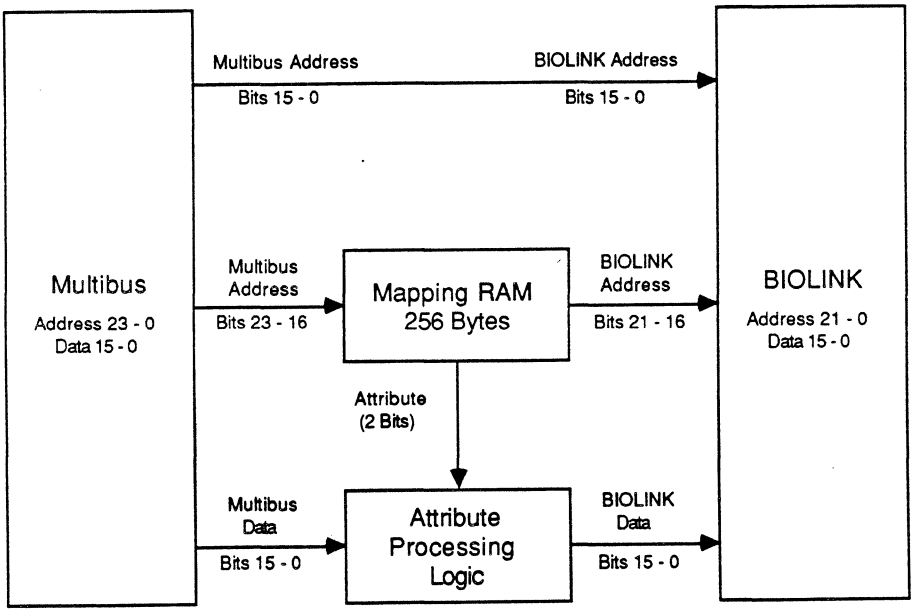


Figure 4-11
Mapping Multibus Addresses onto the BIOLINK

User programs running on the processor node to which the Multibus adapter is attached can access its mapping RAM directly beginning at FFF7E000 in Butterfly Plus address space. Each mapping RAM location consists of one byte with the segment attribute encoded in bits 7 and 6, and with Butterfly Plus physical address bits 21–16 in bits 5–0.

The eight address bits of a mapping RAM location correspond exactly to the eight high-order Multibus address bits that select that location. To map Multibus logical segment 0x70000 into Butterfly Plus physical segment 0x20000, for example, simply write the byte value 02 into location FFF7E007. To swap the byte order in this segment, write the byte value 82H to this same location.

Although the mapping RAM can be read or written at any time, the Multibus adapter will not respond to Multibus addresses until the mapping RAM is explicitly enabled by setting bit 1 in the adapter's Misc Register. For example, to enable the mapping RAM, simply OR the word value 0002H into location FFF7D026.

The boot EPROM configures and enables the mapping RAM during initialization. Multibus addresses 000000 - 3FFFFF are configured to access the bottom four megabytes of the Butterfly local memory, addresses 400000 - 7FFFFF to access the same memory byte swapped, and addresses BF00000 - BFFFFF to access the bottom 64-kilobytes of Butterfly memory in special function mode. All other Multibus addresses are initialized to "no reply".

LOCK SIGNAL AND JUMPER SETTINGS

Besides controlling the watchdog timer, mapping RAM, Multibus interrupts, and pipeline, there are two other functions of the Misc Register: enabling the LOCK signal and reading jumper settings. The LOCK signal is a Multibus control signal that implements indivisible read-modify-write cycles to dual port RAM. The LOCK signal disables one port of a RAM to aid semaphore signaling among Multibus masters. (Refer to Section 2.2.2.8 of the IEEE Standard 796-1983 specification for a detailed description of the LOCK signal.) Bit 2 in the Misc Register is used to assert the LOCK signal from the processor node. Setting this bit to one causes LOCK to be asserted whenever the Butterfly Plus accesses Multibus addresses. Clearing the bit to zero disables this function.

There are eight jumpers on the Multibus adapter, all reserved for use by application software, which can read the settings of these jumpers in bits 15–8 of the Misc Register. Each jumper has a position where the corresponding bit reads as a one and a position where it reads as a zero.

The remaining bits in the Misc Register are connected to pins on the Multibus P2 connector. Bits 7, 6, and 5 of the Misc Register are not used. Table 4–8 and Figure 4–12 show the Misc Register layout.

Table 4-8
Multibus Adapter Misc Register (FFF7D026) Layout

Bits	Function
15–8	Read jumper settings.
7	Spare (P2 connector pin 46).
6	Spare (P2 pin 48).
5	Spare (P2 pin 50).
4	Watchdog Timer Control.
3	Pipeline Read Enable.
2	Assert Multibus LOCK Signal.
1	Mapping RAM Enable.
0	Multibus Interrupt Enable.

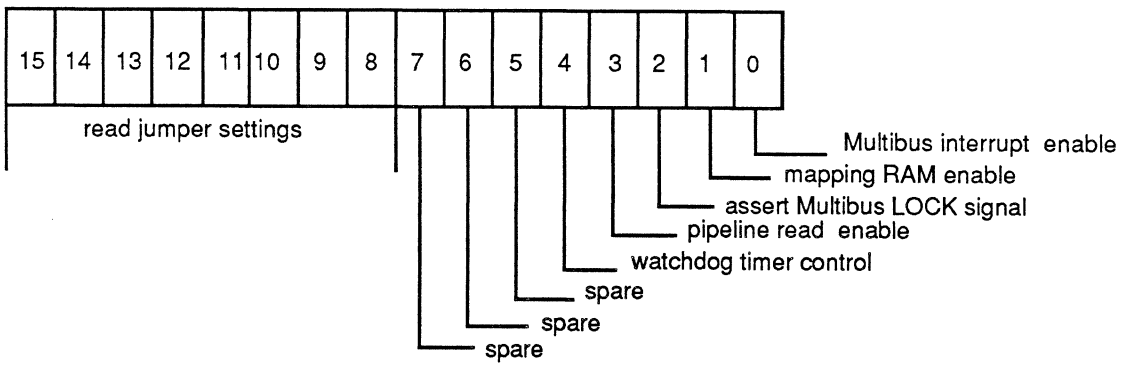


Figure 4-12
Multibus Adapter Misc Register Bit Map

INSTALLING THE MULTIBUS ADAPTER

Before the Multibus adapter card can be installed in the Multibus card cage, changes must be made to the topmost Multibus P2 connector, the smaller of the two Multibus backplane connectors. Since the Multibus adapter card must occupy the top slot in the Multibus card cage, only the P2 connecter in the top socket, labelled J1, is modified. To make the change, cut all signal traces from pins 1 to 50 of the P2 connector on the top (J1) socket. Pin 1 is the upper right hand pin when the socket is viewed from the front with the J1 connector on top. Odd numbered pins are at the top and even numbered pins are at the bottom. Thus, all traces to the rightmost 25 pins on the top and the bottom of the connector must be cut, leaving only traces to the leftmost five pins on the top and the bottom. Table 4–9 shows the P2 connector pinout.

Table 4-9
Multibus P2 Connector Pin Assignments

Pin	Description	Pin	Description
1	RS-232 host DSR	2	Ground
3	RS-232 host DCD	4	RS-232 host receive data
5	RS-232 host DTR	6	Ground
7	RS-232 host RTS	8	RS-232 host transmit data
9	RS-232 host CTS	10	Ground
11		12	
13		14	
15		16	
17	Ground	18	Ground
19	RS-232 console CTS	20	
21	RS-442 – sync clock	22	
23	RS-442 + host transmit	24	
25	RS-442 + sync clock	26	RS-442 – host transmit
27		28	Ground
29	RS-232 console DCD	30	RS-232 console receive data
31	RS-232 console DTR	32	Ground
33	RS-232 console RTS	34	RS-232 console transmit data
35	RS-442 – console transmit	36	RS-442 – console ext clock
37	RS-442 + console transmit	38	RS-442 – host ext clock
39	RS-442 + console ext clock	40	RS-442 + host ext clock
41		42	
43		44	
45		46	Misc Register bit 7
47		48	Front panel clock
49		50	Front panel data
51		52	
53		54	
55	Address bit 22	56	Address bit 23
57	Address bit 20	58	Address bit 21
59		60	

The standard Multibus adapter configuration requires two cables terminated with female DB-25 connectors: one for the host UART and one for the console UART. These connectors attach to the back of the Multibus card cage. Each cable should be at least three feet long and should have three conductors. Unterminated ends of the host cable must be wired to the J1 connector in the Multibus card cage with DB-25 connector pins 2, 3, and 7 attached to J1 connector pins 4, 8, and 10, respectively. Pins 2, 3, and 7 of the console cable are similarly wired to pins 30, 34, and 32 (respectively) of the Multibus J1 connector. The other ends of these cables plug into the leftmost RS-232 connector slots on the Multibus Ethernet Fantail.

MULTIBUS ADAPTER REGISTER SUMMARY

Layouts for the Multibus adapter control registers (Console UART Control Register, Host UART Control Register, and battery backed-up RAM) are shown in Table 4–10. An additional control register, the Misc Register, was shown in Table 4–8.

Table 4-10
Multibus Adapter Control Registers

Console UART Registers (FFF7D000 to FFF7D00A)	
Offset	**Register**
FFF7D000	Console Data Register.
FFF7D002	Console Status and Synchronization Register.
FFF7D004	Console Mode Register.
FFF7D006	Console Command Register.
FFF7D008	Console Interrupt Vector Register. Bits 15–8 must be zero. Bits 7–1 hold the interrupt vector and bit 0 is set to one if there is a receiver interrupt pending.
FFF7D00A	Console Interrupt Control Register. Bits 15–2 must be zero. Bit is set to enable transmitter-empty interrupts. Bit 0 is set to enabl receiver-full interrupts.
Host UART Registers (FFF7D010 to FFF7D01A)	
Offset	**Register**
FFF7D010	Host Data Register.
FFF7D012	Host Status and Synchronization Register.
FFF7D014	Host Mode Register.

FFF7D016	Host Command Register.
FFF7D018	Host Interrupt Vector Register. Bits 15–8 must be zero. Bits 7–1 hold the interrupt vector and bit 0 is set to one if there is a receiver interrupt pending.
FFF7D01A	Host Interrupt Control Register. Bits 15–2 must be zero. Bit 1 is set to enable transmitter-empty interrupts. Bit 0 is set to enable receiver-full interrupts.

EPROM (FFF7D020 to FFF7D022)

Offset	Register
FFF7D020	EPROM Address Register. Bits 15–11 are not used. Bits 10–0 hold the EEPROM address.
FFF7D022	EPROM Data and Control Register. Bits 15–12 are not used. Bit 11 is set to one to write data to the holding register. Bit 10 is set to one to strobe data into the EPROM. Bit 9 is cleared to zero to read data from the EPROM. Bit 8 is cleared to zero to enable the EPROM. Bits 7–0 hold the EPROM data.

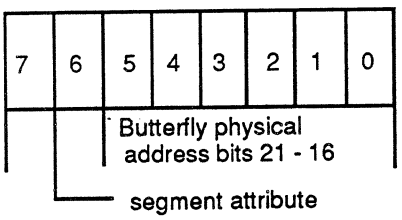
Interrupt Status Register (FFF7D028)

Bits	Function
15–11	Unused.
10	Multibus interrupts enabled.
9	Host interrupt pending.
8	Console interrupt pending.
7	Level 4 interrupt pending.
6–4	Interrupt with highest priority at level 4.
3	Level 3 interrupt pending.
2–0	Interrupt with highest priority at level 3.

Mapping RAM (0xFFF7E000 to 0xFFF7E0FF)

Bits	Function
7–6	Segment attribute (00 = normal, 01 = special, 10 = swapped, 11 = no response).
5–0	Butterfly physical address bits 21–16.

Mapping RAM Bit Map



Interrupt Vector RAM (0xFFF7F000 to 0xFFF7F0FF)

Bits	Function
7	Level 4 interrupt pending.
6–4	Interrupt with highest priority at level 4.
3	Level 3 interrupt pending.
2–0	Interrupt with highest priority at level 3.

Chapter 5

The VME Interface

The Butterfly Plus VME interface attaches a VME card cage to the Butterfly Plus parallel processor. It consists of two printed circuit cards: a VME Node Controller, which replaces one or two processor nodes in a Butterfly card cage; and a VME Bus Adapter, which installs in the VME card cage. Ribbon cables connect the VME Node Controller to the VME Bus Adapter. The VME Node Controller has two complete Butterfly Plus switch ports that connect it directly to the switch. Most of this VMEbus address space is mapped directly through to devices on the VMEbus.

The VME interface conforms to the IEEE P1014/D1.2 specifications, which cover electrical, mechanical, and signal protocol issues. It allows the Butterfly Plus to access data processing, data storage, and peripheral control devices on the VMEbus.

The VME interface provides access to a wide variety of VMEbus devices, and can be used in several different ways. Most especially, the design of the Butterfly Plus VME interface was directed towards two high bandwidth I/O paradigms. The first paradigm involves a device such as a graphics display with a video RAM, and requires the VME interface to move data from the switch interface into the RAM accessed as part of the VME address space. The second paradigm involves a disk, disk controller, and memory on the VMEbus. In this example, the VME memory is used as a buffer, and the disk controller transfers data between the disk and the VME buffer memory. Data transfers between the VME buffer memory and the Butterfly Plus switch are provided by the VME interface. The VME interface is used only as a VMEbus master.

OVERVIEW OF THE VME INTERFACE

The Butterfly Plus VME interface can be used as a bus master, a restricted bus arbiter, an interrupt request generator, or an interrupt handler on the VMEbus. It connects directly to the Butterfly Plus switch, through which all Butterfly Plus/VME communications are made. The VME interface can transfer data to or from sequential VME addresses, and it can also transfer successive data values to or from the same VME address (*e.g.*, to access VME device FIFOs). The Butterfly Plus VME interface uses full 32-bit data and address transactions.

As either an interrupt generator or an interrupt handler, the VME interface supports all seven VME interrupt levels, generating *post event* messages in response to VME interrupts. Sent over the Butterfly Plus switch, these messages appear to Butterfly Plus programs as events. The VME interface acts on messages from the Butterfly Plus requesting:

- Block and 32-bit word transfers between Butterfly Plus and VMEbus memory
- 16-bit atomic mask-and-add instructions performed on VMEbus memory
- VMEbus interrupt requests
- VME interface reset.

Performance

High I/O throughput is a major reason for using the Butterfly Plus VME interface. The VMEbus operates up to ten times faster than the Multibus, the other common bus to which a Butterfly Plus can be attached. The VME interface is also better able to handle the 4-megabytes per second peak bandwidth of each Butterfly Plus switch port. The VME interface contains two complete switch ports, each with average throughput of approximately 3-megabytes-per-second. It achieves about 5.5-megabytes-per-second of I/O bandwidth when both switch ports are used. Additional VME interfaces can be added for even higher throughput. The intent of the Butterfly Plus VME interface design was to achieve the highest possible throughput. A minimal amount of consideration was given to single-word reads across the VME interface (about 20 microseconds) and single-word writes (about 15 microseconds).

Architecture

The Butterfly Plus VME interface consists of two printed circuit cards—a VME Node Controller and a VME Bus Adapter card—that communicate via an external bus. The VME Node Controller is a 12 by 18 inch printed circuit card that fits into a Butterfly card cage, replacing one or two of the Butterfly Plus processor nodes. It attaches directly to the Butterfly Plus switch via two cables, if only one switch port is used, or via four cables, when two switch ports are used. The VME Node Controller contains two complete full duplex switch interfaces, each with an independent switch receiver and transmitter and the same two card edge connectors found on a processor node. Attaching the VME Node Controller to two switch ports nearly doubles the bandwidth of the data path between the Butterfly Plus and the VMEbus. If higher bandwidth is not required, the VME Node Controller can attach to only a single Butterfly Plus switch port. The total number of switch ports used by all VME Node Controllers and all processor nodes cannot exceed 256. The VME Node Controller has an onboard MC68020 processor; writable program memory, including 128-kilobytes of RAM; and 64-kilobytes of bootstrap EPROM.

The VME Bus Adapter is a 6 by 9 inch, double height VME circuit card that provides the actual interface between the VME Node Controller and the VMEbus. Installed in a VME card cage that fits inside the Butterfly rack, the VME Bus Adapter performs mainly as a high speed direct memory access controller on the VMEbus. The external bus that connects the VME Node Controller in the Butterfly card cage to the VME Bus Adapter in the VME card cage is a pair of 40-conductor ribbon cables, up to 25 feet long, used as a 40-bit, partially multiplexed bus. The length of the external bus contributes about four nanoseconds per foot to the time required to perform a data transfer. Figure 5–1 is a block diagram of the VME interface.

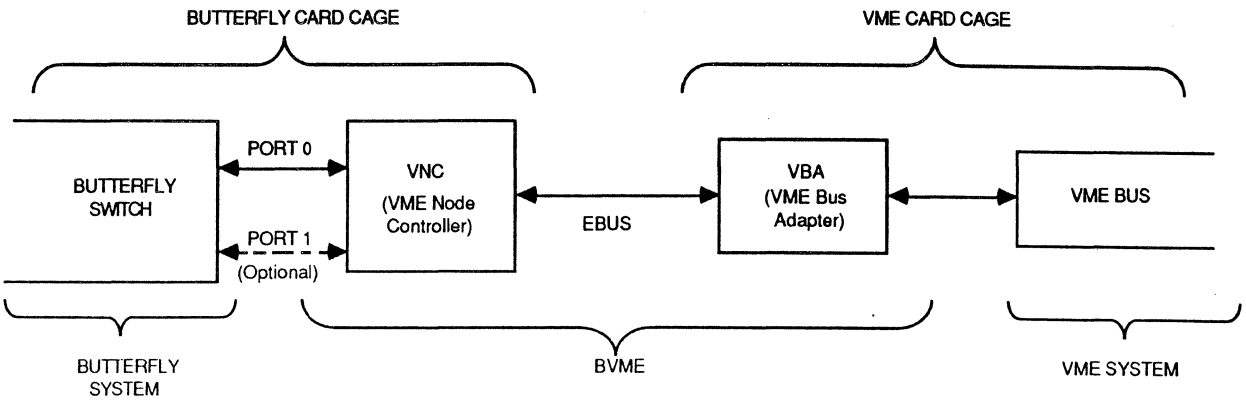


Figure 5-1
VME Interface Block Diagram

VME NODE CONTROLLER

The VME Node Controller has two ports (0 and 1) that connect to the Butterfly Plus switch. Each of these ports has two finite state machines, one for transmitting to the Butterfly Plus switch and one for receiving from the Butterfly Plus switch. The VME Node Controller derives clock and reset signals from its port 0 Butterfly Plus switch connection; therefore, if only one Butterfly Plus switch port is connected, it must be port 0. Each finite state machine contains a first-in, first-out (FIFO) memory for buffering data. Note that the terms *transmit* and *receive*, when applied to the finite state machines and their FIFOs, are defined relative to the VME Node Controller. For example, the channel 0 transmit FIFO holds data that the VME Node Controller is sending to port 0 of the Butterfly Plus switch. A FIFO bus attaches the four FIFOs to the external bus interface, where the ribbon cable to the VME Bus Adapter connects. Figure 5–2 is a block diagram of the VME Node Controller.

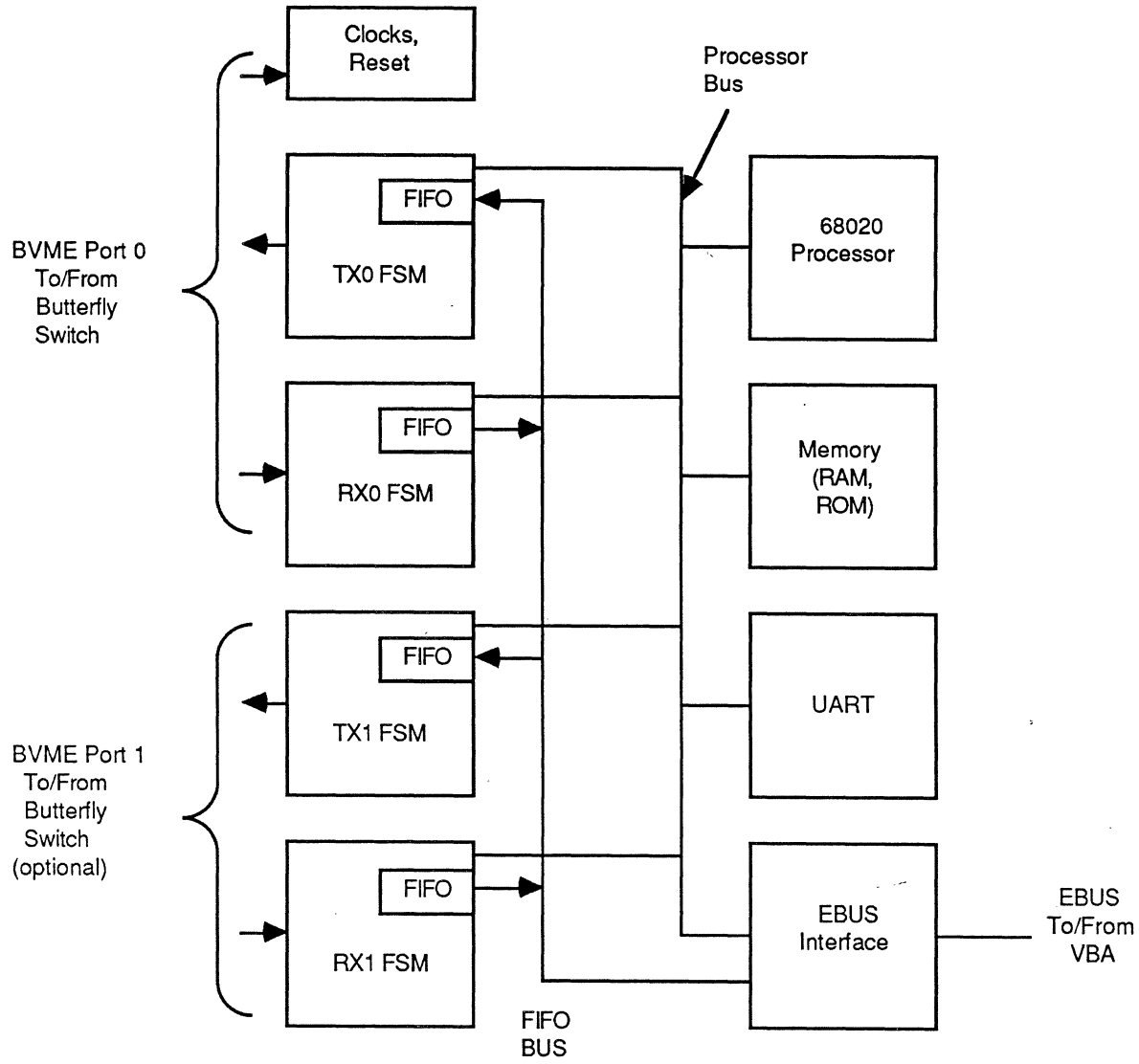


Figure 5-2
VME Node Controller Block Diagram

Operation of the entire VME interface is controlled and coordinated by firmware stored in read-only memory and executed by the MC68020 processor on the VME Node Controller. Random access memory is also available for use by the VME Node Controller processor.

Internally, the VME Node Controller main bus links the finite state machines, the processor, memory, and external bus interface. There is a diagnostic UART on the VME Node Controller main bus for testing and maintenance. Figure 5–3 diagrams the physical layout of the VME Node Controller circuit card, showing the locations of the jumpers and DIP switch, the onboard MC68020 microprocessor, and the diagnostic UART.

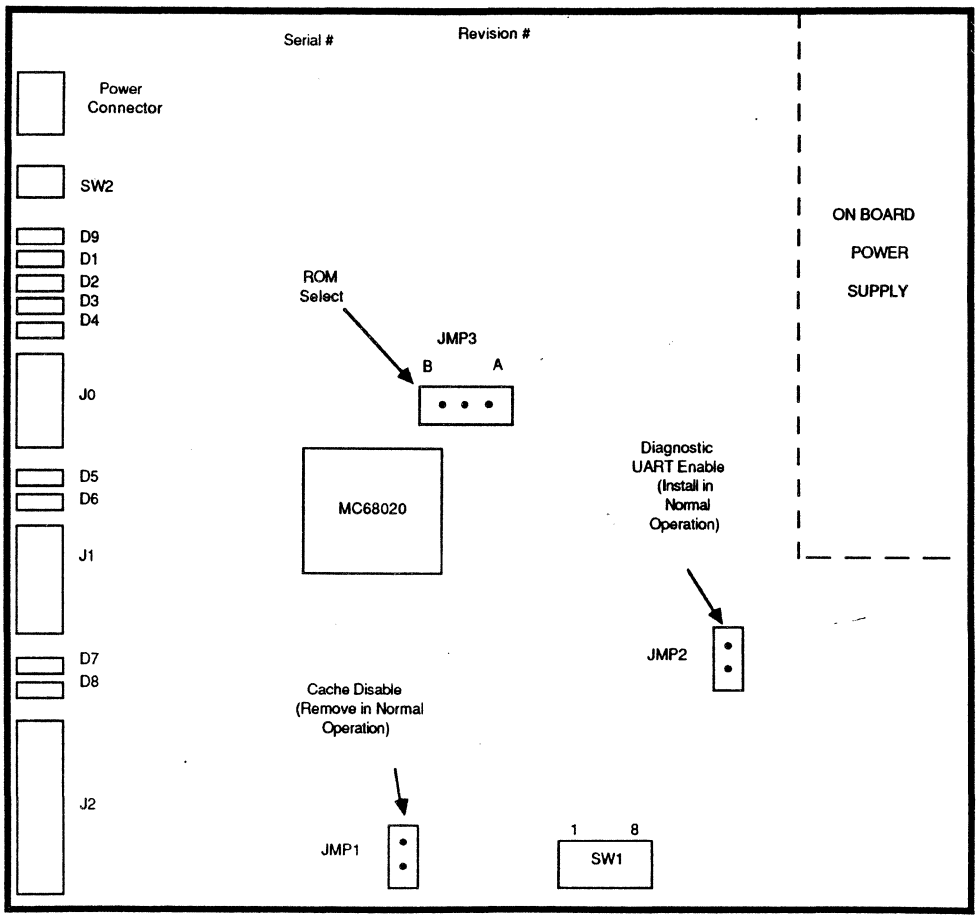


Figure 5-3
VME Node Controller Layout

LED Indicators

The front edge of the VME Node Controller has nine LED indicators (two red, two amber, and five green), one toggle switch, and seven connectors (see Figure 5–4).

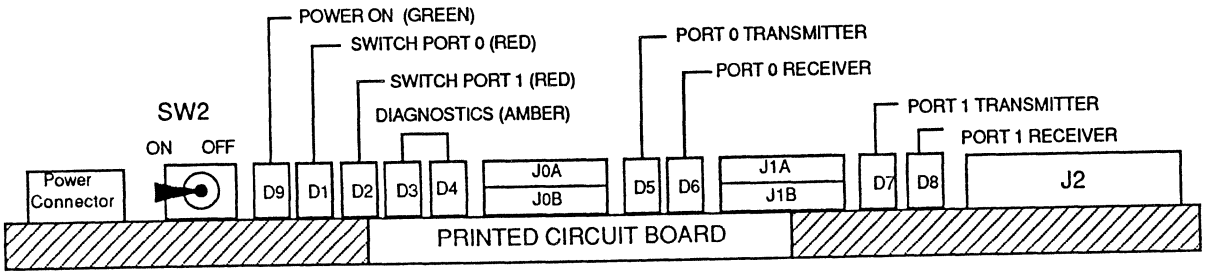


Figure 5-4
VME Node Controller Front Edge

D1 is a red LED indicator, driven by bit 2 in the VME Node Controller Misc register. It indicates whether switch port 0 was initialized properly. This indicator lights up briefly upon power up or reset and turns off when initialization is finished successfully and the port 0 address is discovered.

D2 is a red LED indicator, driven by bit 3 in the VME Node Controller Misc register. It indicates whether switch port 1 was initialized properly. Like D1, it lights up briefly upon power up or reset and turns off when initialization is finished successfully (*i.e.*, the port 1 address is discovered). It stays on if the VME Node Controller is attached only to switch port 0.

D3 and D4 are amber LED indicators, driven by bits 4 and 5, respectively, in the VME Node Controller Misc register. They are used by VME interface diagnostics.

D5, D6, D7, and D8 are green LED indicators that display the values of the switch transmitter and receiver FRAME signals for switch port 0 and switch port 1, as shown in Table 5–1. Each LED turns on when the associated signal is asserted to indicate that a message is flowing between the VME Node Controller and the Butterfly Plus switch.

D9 is a green LED indicator, which lights up to indicate that DC power is applied at the appropriate voltage level for proper operation.

Table 5-1
VME Node Controller LED Indicators

LED	Frame Signal	Indicates a Message
D5	Port 0 transmitter	From VME interface to Butterfly
D6	Port 0 receiver	From Butterfly to VME interface
D7	Port 1 transmitter	From VME interface to Butterfly
D8	Port 1 receiver	From Butterfly to VME interface

Power Switch and Connectors

A toggle switch, SW2, on the front edge of the VME Node Controller, powers up the card by toggling DC power from the Butterfly card cage main power supply. The on position is toward the power connector.

The VME Node Controller has seven connectors on the front edge of the card: two pairs of switch transmitter and receiver connectors, two external bus connectors, and a power connector. J0A is the switch transmitter connector for switch port 0. The VME Node Controller transmits data to the Butterfly Plus switch on this connector, and it also receives the system reset signal from the switch on this connector. J0B is the switch transmitter connector for switch port 1, which is functionally identical to switch port 0, except that transmitter port 1 ignores the system reset signal. J0A is always connected, whereas use of J0B is optional.

J1A is the switch receiver connector for switch port 0. The VME interface receives data from the Butterfly Plus switch on this connector, and it also receives the system clock signal from the switch on this connector. J1B is the switch receiver connector for switch port 1, which is functionally identical to switch port 0, except that receiver port 1 ignores the Butterfly Plus clock signal. J1A is always connected, whereas use of J1B is optional.

J2A and J2B are two 40-pin connectors for the external bus cable that attaches the VME Node Controller to the VME Bus Adapter.

The power connector supplies unregulated 30-volt DC power to the VME Node Controller. The VME Node Controller and its VME Bus Adapter can be powered up or down independently. The VME Node Controller responds to

Butterfly Plus switch messages even when the VME Bus Adapter is powered off; however, it does not respond to the VMEbus unless the adapter is turned on.

Jumpers

There are three jumpers on the VME Node Controller, positioned as shown in Figure 5–5. JMP1 is a 2-pin jumper that, when installed, disables the instruction cache in the MC68020 processor. It should be installed only during hardware debugging. JMP2 is a 2-pin jumper that, when removed, disables the diagnostic UART oscillator. It is always installed during normal operation. JMP3 is a 3-pin jumper that selects the EPROM size. When this jumper is in the A position, it selects a 32-kilobyte EPROM; when in the B position, it selects a 64-kilobyte EPROM, the usual size of EPROM.

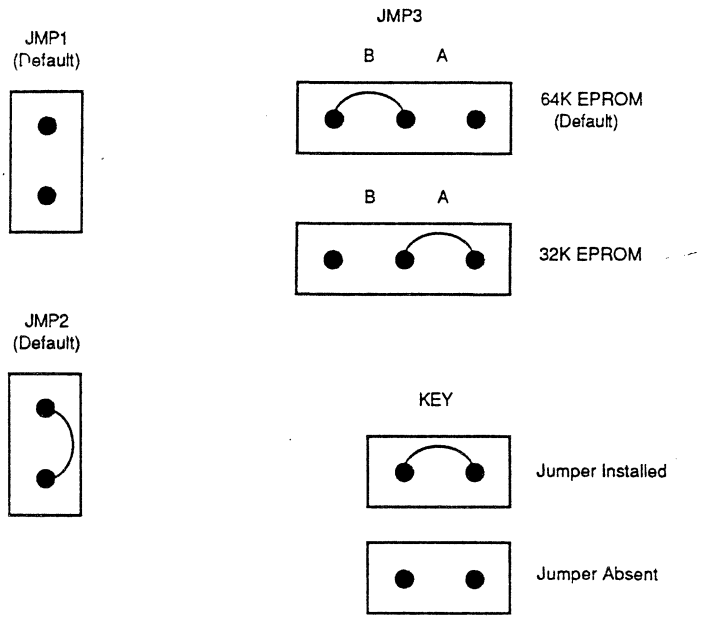


Figure 5-5
VME Node Controller Jumper Positions

DIP Switch Settings

There is one switch block, SW1, on the VME Node Controller. Its eight switches define a value that can be read from the VME Misc register. Switches SW1-1 and SW1-2 are used to indicate the size of the Butterfly Plus

switch. Switches SW1-3 through SW1-8 are available to the VNC programmer. See Table 5–2 for the correct setting of these switches.

Table 5-2
DIP Switch Settings

SW1-1	SW1-2	SW1-3 through SW1-8	Sets VME Node Controller for
On	On	Off	2-column switch
Off	On	Off	3-column switch
On	Off	Off	4-column switch
Off	Off	Off	5-column switch

VME BUS ADAPTER

The VME Bus Adapter provides a direct link between the VME Node Controller and the VMEbus. It has one external bus connector that attaches to the VME Node Controller. The following six main elements are included: interrupt request generator and handler, VMEbus arbiter, VMEbus requester, VMEbus control circuit, direct memory access byte count logic, and direct memory access address generator. Figure 5–6 is a block diagram of the VME Bus Adapter.

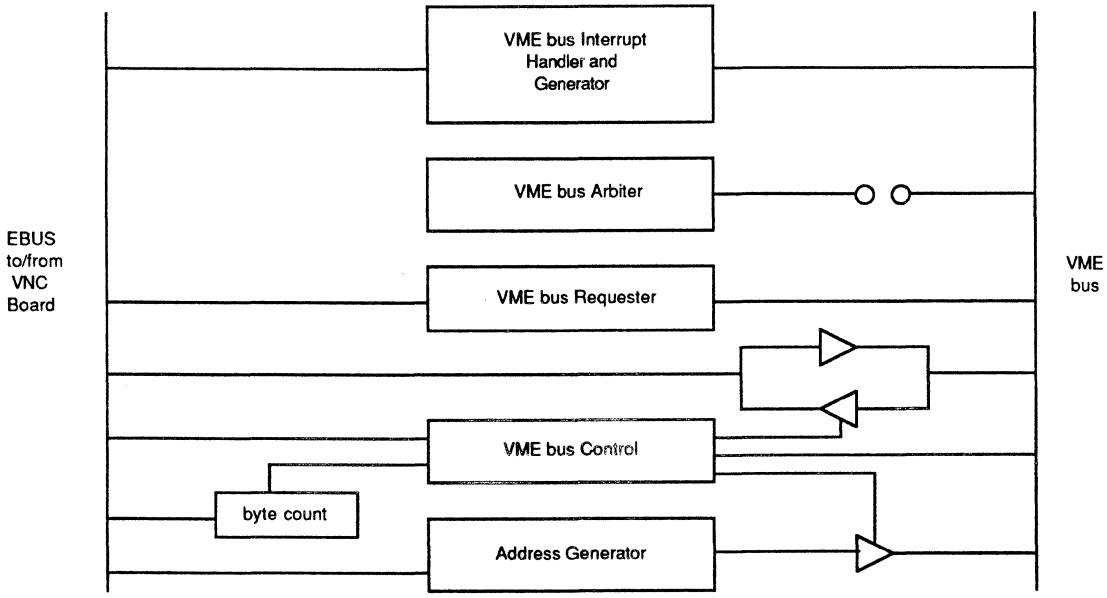


Figure 5-6
VME Bus Adapter Block Diagram

The interrupt request generator and handler manage interrupt requests directed to, or arriving from, the VMEbus respectively. The VME Bus Adapter can enable interrupt requests from the VMEbus and receive notification when a VMEbus interrupt request arrives. It can also generate requests for VMEbus interrupts to be handled by other devices on the VMEbus.

The use of the VMEbus arbiter is optional. It is used only if no other device on the VMEbus is acting as the bus arbiter. If the VME Bus Adapter is placed in slot 1, the VMEbus arbiter jumpers must be set for operation. If the VME Bus Adapter is placed elsewhere in the VMEbus card cage, the VMEbus jumpers must be set to disable the arbiter.

The VMEbus requester, under control of the VME Node Controller firmware, requests mastership of the VMEbus. It is a **Release When Done** REQUESTER as described by the *Motorola VMEbus Specification*.

The address generator and byte count logic are used to transfer data across the external bus, between VMEbus memory and VME Node Controller FIFO memory, under firmware control. Once the firmware sets up the parameters for a DMA transfer, these functional units perform all bus cycling required to complete the transfer. Figure 5–7 illustrates the VME Bus Adapter card.

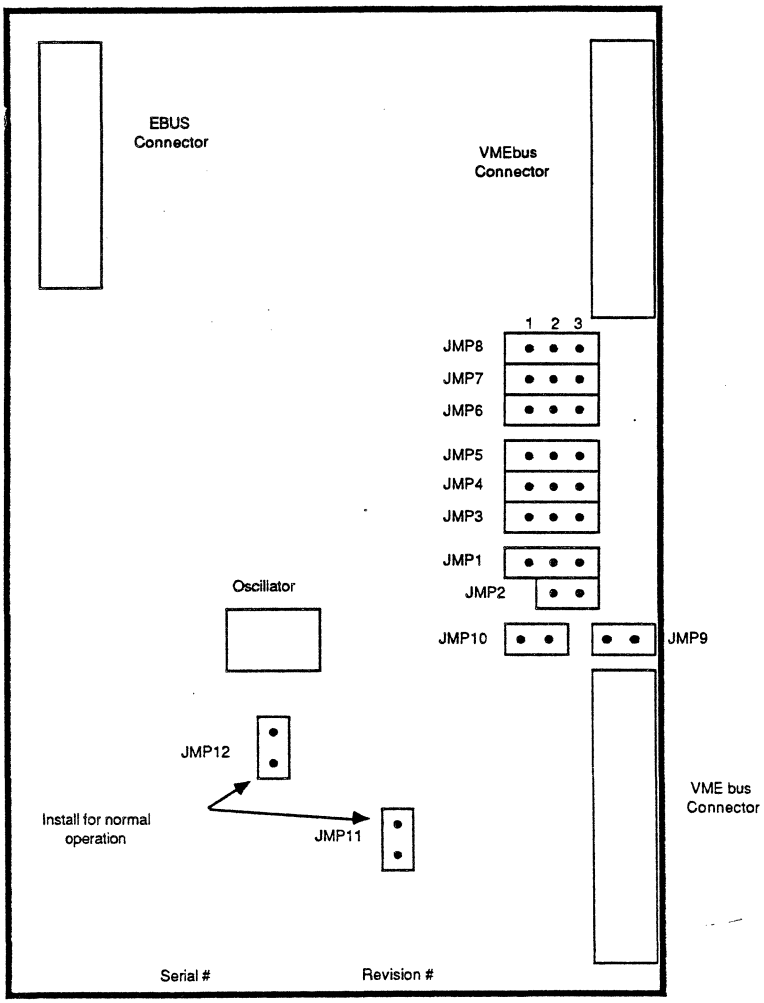


Figure 5-7
VME Bus Adapter Card Layout

There are 12 jumpers on the VME Bus Adapter. See Figure 5–8 for the correct placement of these jumpers. Jumpers JMP1 and JMP2 can be positioned to disable the onboard arbiter and allow the VME Bus Adapter to operate in VME card cage positions other than slot 1. Jumpers JMP3 through JMP8 determine the bus grant level the VME interface uses, and propagate daisy-chained signals. (See the *Motorola VMEbus Specification* for details of bus grant operation.) Jumpers JMP9 and JMP10 must be installed at all times. Jumpers JMP11 and JMP12 are removed to disable the onboard oscillators that feed the timeout counter clock and the finite state machine clock. These jumpers should be removed only for testing and must be installed for normal operation.

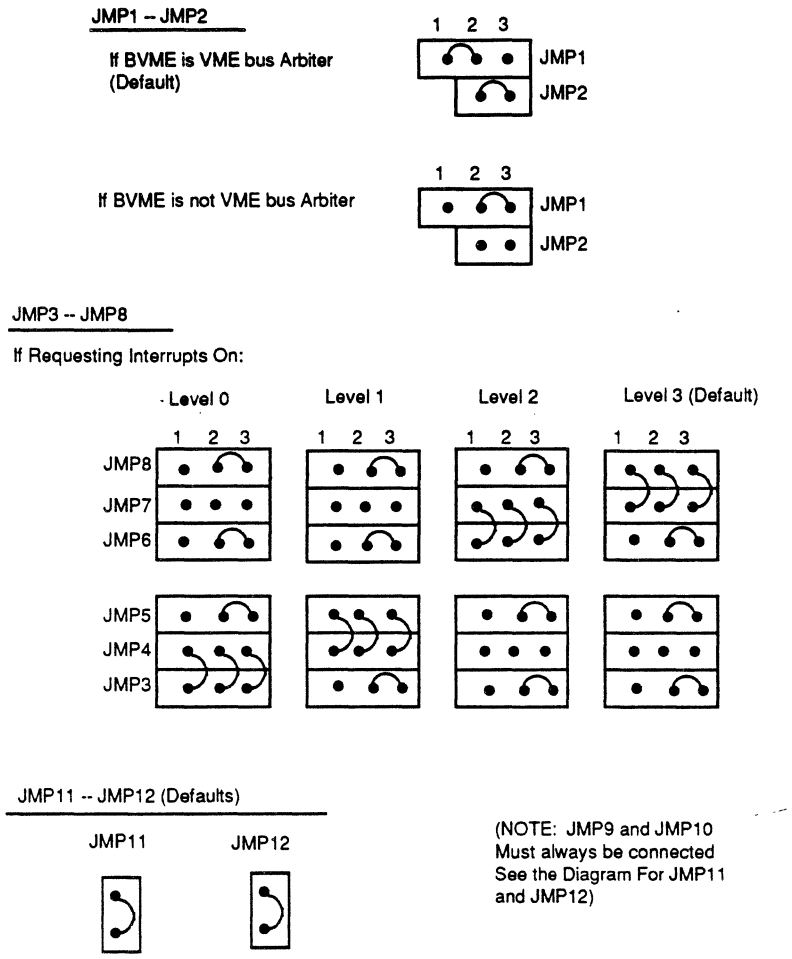


Figure 5-8
VME Bus Adapter Jumper Positions

VME INTERFACE AS BUS REQUESTER

The VME interface can act as a bus requester (master) on the VMEbus to control the transfer of data on the bus. An important parameter in using any bus master device is how long the device can retain bus mastership. Although some VME devices release the bus on request from another device, the VME interface does not. Once the VME interface acquires the bus, it holds the bus until it completes its operation(s). However, the number of bus cycles that a VME interface retains the bus is the number of bytes in the block, divided by four; the maximum time the VME interface can retain bus mastership is 256

divided by four, or 64 bus cycles. To attain the role of bus master, a device requests control; its request is then arbitrated against any other requests by a bus arbiter. The bus arbiter selects one requesting device and grants mastership. When the VME interface is used in direct access mode, bus mastership is released after one bus cycle (read, write, or read-modify-write) is completed.

CONFIGURATION ON THE VMEBUS

The Butterfly Plus VME Bus Adapter can reside in any slot of the VMEbus card cage. Note that the jumper settings change when the VME is the system arbiter.

VME Interface as Arbiter

In a VMEbus system, the device occupying slot 1 must contain an arbiter. For applications where the VME interface is the only bus master or where it must occupy slot 1 of the VME cage, a restricted VME arbiter is provided. This arbiter uses a single-level arbitration scheme and is activated by jumpers 1 and 2 on the VME Bus Adapter.

VME ADDRESS SPACE

Physical memory accessed via the VME interface is equally accessible through either switch port. The VME interface provides a four megabyte window between Butterfly Plus physical address space and VME address space.

VMEbus memory locations that are addressable by the Butterfly Plus do not replace memory on other processor nodes; rather, the VMEbus address space augments or extends the existing Butterfly Plus memory. A VME Node Controller can be viewed as an unusual processor node that runs no user code and whose memory resides in VMEbus devices.

Address Translation

When a process running on a Butterfly Plus processor node references a remote memory location, the virtual address presented by the remote processor is translated to a 32-bit physical address. The high-order eight bits of the physical address specify a switch port and the low-order 22 bits specify an offset into the memory which is accessible through that switch port.

When the target switch port is connected to a processor node, the 22-bit offset specifies a location in the memory of that processor node. When the target switch port is connected to a VME Node Controller, a simple mapping scheme converts the 22-bit offset into a 32-bit VMEbus address. With this scheme, the low-order 16 bits of the 22-bit offset pass directly through to the VMEbus. The high-order six bits index a translation table that supplies the high-order 16 bits of the VMEbus address and the six bits of address modifier required by the VMEbus specification. The same translation table is used for both switch ports.

The translation table, comprising 64 mapping registers, divides the four megabytes of accessible address space into 64-kilobyte pages that can be located anywhere in VME address space. In the simplest case, these pages are mapped to a contiguous four megabyte block of VME address space. The address map can be altered by changing the **addr_map** field of the structure **control_reg_struct** in bank 0 memory. Once the mapping is established, ordinary **move** instructions can be used to transfer data. Although the VME interface address mapping facility can be used to provide access protection, mapping is not intended primarily as a protection mechanism.

Bank 0 Memory

Most of the four megabytes of Butterfly Plus address space occupied by the VME interface is mapped directly through to devices on the VMEbus. The lower 64-kilobytes, however, is bank 0 memory reserved for use by the VME interface. Most of bank 0 memory is mapped to RAM on the VME interface itself. This RAM is used for the VME interface control program, interrupt vectors, the switch node discovery table, and address mapping tables. Part of the bank 0 address space maps control registers that a Butterfly Plus program can use to influence operation of the VME interface.

VME control register space starts at 0000FE00 in bank 0, filling the last 512 bytes of the bank. All register offsets are relative to this starting address. Registers must be accessed as long operands, using block transfer operations. To ensure proper alignment, block transfers that access registers in the VME interface must have a starting address and a transfer length that divide evenly by four. Any non-aligned access to VME interface registers is ignored.

Control Registers

The bank 0 control registers available to a Butterfly Plus program are laid out according to the following structure:

```
struct control_reg_struct{
long addr_map[64];          /* Butterfly to VMEbus address map   */
long rtc;                   /* realtime clock                    */
long switch_timeout;        /* timeout for switch retransmission */
long vba_timeout;           /* timeout for block transfer        */
long vba_restart_timeout;   /* timeout to restart external bus   */
short bxfers[2];            /* outstanding block transfers       */
long rx_rm_errors[2];       /* receive request message errors    */
long rx_am_errors[2];       /* receive answer message errors     */
long tx_rm_timeouts[2];     /* transmit request message timeouts */
long tx_am_timeouts[2];     /* transmit answer message timeouts  */
long vba_timeouts;          /* no response from adapter to node  */
long spare[2];              /* aligned to end at address 13F     */
long intr_event[16];        /* event handle for interrupt        */
long transfer_ctl;          /* controls VMEbus access mode       */
long vba_int_req;           /* controls VME interrupt requests   */
long jump_addr;             /* jam into PC of MC68020 on node    */
long alt_paths;             /* Butterfly switch alternate paths  */
long vba_bus_timeout;       /* VMEbus error, arbiter timeouts    */
} ;
```

The fields in the VME control register structure are as follows:

- **addr_map [64]**

This field has 64 registers that map bits 16–21 of a Butterfly Plus physical address into bits 16–31 of a VMEbus physical address. When the VME interface receives a request to read or write VMEbus memory, it generates a 32-bit VMEbus address by passing the high-order six bits of the 22-bit Butterfly Plus address through the address map table, **addr_map**, to get the high-order 16 VME address bits and also 16 bits of address modifier information. Each entry in the address map table can be written individually through this set of 64 registers.

Bits 0–15 of an address map table entry are the VMEbus address bits 16–31 to be used for the access. Bits 16–31 of the entry are the address modifier bits. Only seven bits of the address modifier information are used.

The VME interface initializes the address map to allow VMEbus access and to convert the Butterfly Plus addresses 0x010000 through 0x3FFFFF to VMEbus addresses 0x00010000 through 0x003FFFFF. (Butterfly Plus addresses below

0x010000 are the 64-kilobytes of bank 0 memory in the VME interface.) Address modifier bits are initialized to reflect **Standard** addressing of supervisory data. (See the *Motorola VMEbus Specification* for details.)

- **rtc**

The realtime clock on the VMEbus interface is similar to the clocks in each Butterfly Plus processor node, but it is not synchronized with them. Each tick is four milliseconds. This register is intended for diagnostic use only.

- **switch_timeout**

Writing this register changes the timeout interval for Butterfly Plus switch messages. The timeout value specifies the number of 4-millisecond clock ticks to wait before timing out. Values of one or two result in unreliable time intervals. The default timeout interval value is five ticks, which corresponds to 20 milliseconds. A message that times out is discarded. The Chrysalis utility program **toset** sets the **switch_timeout** value for all VME interfaces in the Butterfly Plus, as well as for all processor nodes. Contention on a VME Node Controller where both switch ports are being used for block transfer can result in switch timeouts. For this reason, some applications require increasing **switch_timeout** to as much as 100 milliseconds with the **toset** utility program.

- **vba_timeout**

This register determines the number of 4-millisecond clock ticks before a 32-bit operation on the external bus times out. External bus timeouts should occur only as a result of a very slow VMEbus or a hardware failure. The default timeout interval is 100 ticks, which corresponds to 0.4 seconds. Upon timeout, the VME Node Controller concludes that the VME Bus Adapter is powered down or disconnected from the external bus. Its firmware begins restart recovery and increments the timeout count in the **vba_timeouts** register.

- **vba_restart_timeout**

This register determines the number of 4-millisecond clock ticks to wait before trying to restart a VME Bus Adapter that has timed out. The default value is 1250, which corresponds to five seconds.

- **bxfers[2]**

These registers contain the number of outstanding Butterfly Plus block transfers initiated by the VME interface. The value is usually zero. The registers are intended for use only by VME interface firmware. Each switch port (0, 1) has a copy of these registers.

- **rx_rm_errors[2]**
- **rx_am_errors[2]**
- **tx_rm_timeouts[2]**
- **tx_am_timeouts[2]**

These registers contain the number of receive errors or transmit timeouts logged, for request messages or answer messages, as indicated by the register name. A program can write a zero to each of these registers to clear them. The registers are also cleared to zero by a reset. Each switch port (0, 1) has a copy of this set of registers.

- **vba_timeouts**

This register contains a count of the number of times the VME Bus Adapter failed to perform an order from the VME Node Controller. The value of **vba_timeout** determines how long the VME Node Controller will wait before giving up and incrementing **vba_timeouts**.

- **intr_event[16]**

These registers specify an event handle for each distinct type of interrupt. Writing the register not only establishes the event handle, but also enables the associated interrupt, which remains enabled until the post message has been sent and then is disabled automatically. The interrupt can be re-enabled by again writing the event handle into the appropriate **intr_event** register. VME interface firmware initializes the **intr_event** table to zero (*i.e.*, no event, interrupts disabled).

Application software normally calls the Butterfly Plus operating system to obtain an event handle. The event handle in each entry of the **intr_event** table is interpreted as follows: Bits 31–24 hold the destination node for post messages. Bits 23–16 hold the sequence number. Bits 15–0 hold the object header's address. Of the 16 table entries, 10 are currently used. The **intr_event[0]** to **intr_event[6]** entries correspond to VME interrupt request

levels 1 to 7, respectively. The **intr_event[7]** entry corresponds to the VME_ACFAIL interrupt generated as power to the VME system is failing. The **intr_event[8]** entry corresponds to the VME_RESET interrupt generated when the VME system is reset. The **intr_event[9]** entry corresponds to the VME_SYSFAIL interrupt generated when the VME *system fail* line is asserted.

- **transfer_ctl**

Writing this register sets auto-increment, sequential, keepbus, holdbus, reset, and other VMEbus function attributes. The bits in this register are initialized to zero unless otherwise specified. Bits 15, 12, 10, and 7 to 0 are set to zero and unused. Bit 14, VBA_CTL_KEEP, needs is set high to keep the VMEbus for indivisible bus operations. Both VBA_CTL_KEEP and VBA_CTL_HOLD should be asserted for read-modify-write operations. For other indivisible operations, assert only VBA_CTL_KEEP. Bit 13, VBA_CTL_VRST, is set high to assert the VME reset line. Bit 11, VBA_CTL_HOLD, is set high to hold the VMEbus for read-modify-write cycles. VBA_CTL_HOLD holds the address strobe asserted. Bit 9, VBA_CTL_AINC, is set high to enable auto-increment DMA. When this bit is enabled, the VME interface increments the address it presents on the VMEbus with each bus cycle, causing successive cycles to access successive locations. When this bit is disabled, the address is not incremented; this mode can be used to access a FIFO in a device on the VMEbus. Bit 9 is initialized to 1. Bit 8, VBA_CTL_SEQ, needs to be set high for "sequential" (VMEbus block transfer) access mode. The VME specification defines a VMEbus block transfer as an operation in which an address appears on the bus only once. The master (VME interface) places the start address on the bus at the beginning of the transfer, and then asserts and removes the data strobe signal repeatedly. With each cycle, the slave accesses the next address. In a VMEbus block transfer, incrementing the address is the responsibility of the slave, so the setting of the VBA_CTL_AINC bit is irrelevant.

- **vme_int_req**

Writing this register controls the generation of interrupt requests on the VMEbus. The default is no interrupts. Data written to this 32-bit register is passed along by the VME interface to the two 7-bit registers of the SCB68154 interrupt generator chip used on the VME Bus Adapter. This 32-bit register has two fields, one to specify the interrupt level and the other to specify the interrupt vector. Bits 14–8 are the interrupt request register (RS = 1), and bits 6–0 are the interrupt vector register (RS = 0), where RS is the register select bit

on the SCB68154. Note that the SCB68154 has only a 7-bit data bus connection, and that the least significant bits are aligned. Thus, R0, bit 1 is attached to bit 0 of the **vme_int_req** register, while R1, bit 1 is attached to bit 8 of the **vme_int_req** register. Refer to a Signetics SCB68154 data sheet for more details.

- **jump_addr**

New VME interface control code can be block-transferred into a working VME Node Controller. Writing an address to this register transfers control to that address.

- **alt_paths**

These bits specify the use of alternate paths in the Butterfly Plus switch. The least significant eight bits of this register set the alternate path bits for the two switch ports. The default is to use no alternate paths. Bits 7–4 are the alternate path bits 3–0 for switch port 1. Bits 3–0 are the alternate path bits 3–0 for switch port 0.

- **vba_bus_timeout**

This register determines the timeout intervals for the bus error and bus arbiter timers on the VME Bus Adapter. Timeout intervals are specified in milliseconds with a minimum value of two and a maximum value of 255. The default for each timer is three milliseconds. Although **vba_timeout** (described earlier) applies to multiple external bus cycles during a 32-bit transfer, **vba_bus_timeout** applies to each VMEbus cycle individually. A bus error timeout, encountered while accessing the VMEbus to satisfy a Butterfly Plus read or write, propagates all the way back to the Butterfly Plus program, which receives a bus error throw exception. Bits 15–8 are for bus arbiter timeout. Bits 7–0 are for bus error timeout.

VME INTERRUPT REQUESTS

The VME interface can act as either an interrupt requester or an interrupt handler on the VMEbus. All seven VME interrupt levels are supported, as is the VME interrupt vector mechanism. In addition to the seven VME interrupt levels, there are three more VME signals that the VME interface treats as interrupts on separate levels: VME_ACFAIL, VME_RESET, and VME_SYSFAIL. Interrupts on a particular level are accepted only if an event handle has been supplied.

When handling a VME interrupt request, the VME interface uses the interrupt request level as an index into a table of Butterfly Plus event handles. The VME interface can receive vectored interrupts from the VMEbus and convert them into Butterfly Plus *post event* messages. For applications where high-speed interrupt service is important, interrupt handlers can be written using firmware in the VME interface. The Butterfly Plus generates VME interrupt requests through the VME interface by sending switch messages to manipulate the VME_INT_REQ control register located in bank 0 memory, specifying a vector and an interrupt level.

THE VMEBUS

The VMEbus is a 32-bit address, 32-bit data, asynchronous bus for connecting high performance microprocessor elements together. It is a master/slave bus with contention resolution for bus mastership. It offers non-multiplexed address and data, vectored interrupts, and direct memory access arbitration, all using asynchronous techniques. Top speed of the VMEbus is 40-megabytes-per-second for 32-bit transfers, although a typical high-speed memory card with 200-nanosecond cycle time sustains only 20-megabytes-per-second of bus bandwidth. The VME interface supports VMEbus block transfers, which access a series of contiguous addresses without ever removing the address strobe signal.

VME Address Modifier

Standard addresses are 24 bits long, in VME parlance, and extended addresses are 32 bits long. Some VMEbus memory cards respond only to one address size or the other. The VMEbus also uses a short (16-bit) address that is commonly used to access memory mapped I/O device registers on VME cards.

Besides the address bits themselves, the VMEbus conveys six address modifier bits that describe the size of an address and the type of bus access. The six address modifier bits represent 64 possible access situations, only some of which are defined. Others are reserved or application specific. Of those defined, distinctions are made such as to short, standard, or extended address length; supervisory or non-privileged context; and block transfer, program access, or data access.

A Butterfly Plus program sets address modifier bits using mapping calls in the VME library. The programmer can direct the program to set the address modifier bits with **Standard Supervisory** defaults or with user-defined codes, both of which are described in the *Motorola VMEbus Specification*.

The VME Interface as a VMEbus Device

The VMEbus specification defines several types of device behavior. This section describes which of those device types the VME interface implements and with which it interacts.

The VME interface functions only as a VMEbus master, never as a slave. It always initiates data transfers, and never responds to transfer requests initiated by other VMEbus devices. If another VMEbus device wants to transfer data into Butterfly Plus memory, it can perform one of two things: The VMEbus device can request any of the seven VMEbus interrupts, which the Butterfly Plus application program receives as an event if the VME interface is set up to handle that interrupt request, or it can set a flag in the VMEbus memory that the Butterfly Plus application program checks. There is no way for a VME device to force the transfer of data to or from Butterfly Plus memory without the cooperation of a Butterfly Plus program. The VME interface maps part of the VME address space into Butterfly Plus address space, but Butterfly Plus address space is never mapped into VMEbus address space.

A disk controller on a VMEbus typically regulates data flow between a disk and a memory, where the disk controller is a VMEbus master and the memory is a VMEbus slave. The disk controller cannot transfer data directly between a disk and Butterfly Plus memory via the VME interface. Instead, a buffer memory on the VMEbus must be used to store the data briefly. Similarly, a VME interface cannot transfer data directly between itself and a second VME interface even if both are installed on the same VMEbus; a buffer memory

must be used in between these interfaces. The use of a buffer memory would almost certainly be necessary even if VMEbus masters could access Butterfly Plus memory through the switch, since most DMA controllers cannot cope with the long access latencies the switch is likely to cause.

Besides using 32-bit addresses and data, the VME interface can also perform individual 8- or 16-bit references to a D16 device, but not 8- or 16-bit block transfers. Address modifier bits that the VME interface places on the VMEbus are totally programmable, so address modifier requirements of VME devices can usually be met.

The VMEbus specification defines several types of data transfer: single, block, read-modify-write, and unaligned. Of these transfer types, the VME interface supports all varieties of single transfers (*i.e.*, byte, word, and longword) and unaligned block transfers. (Refer to the section entitled "VME Data Alignment" for a discussion of how the VME interface handles unaligned data.) The VME interface also supports read-modify-write access in byte and word sizes only.

When a VMEbus master wants to access the bus, it functions as a bus requester. Its request is arbitrated with any other pending bus requests as the arbiter selects one requester and grants it use of the bus. The VME interface functions as VMEbus requester when it is ready to initiate a data transfer. Jumpers on the VME interface allow it to function as a simple arbiter that performs single-level arbitration on level three.

The VMEbus provides seven levels of daisy-chained interrupts. The VME interface functions both as a VMEbus interrupt requester and as an VMEbus interrupt handler for these interrupts. As an interrupt requester, the VME interface requests interrupts, drives the interrupt vector onto the VMEbus if the requested interrupt is acknowledged, and passes the interrupt acknowledge (IACK) signal along the daisy chain when it is not requesting an interrupt. As an interrupt handler, the VME interface prioritizes interrupt requests within its assigned group of interrupt request lines, requests use of the data transfer bus lines, initiates an IACK cycle when granted the bus, and initiates the appropriate interrupt service routine (by posting an event) based on the interrupt level received.

The VMEbus specification also includes six system utility signals. The VME interface monitors and responds to four of these signals: system reset, system failure, system clock, and AC fail. The VME interface contains a 16-MHz clock that drives the system clock line only if the VME interface is the VME system controller (*i.e.*, if the VME Bus Adapter card occupies VMEbus slot 1 and therefore acts as VMEbus arbiter). The VME interface ignores the VMEbus serial data and serial clock signals.

Butterfly Plus and VMEbus Block Transfers

The VMEbus specification describes a block transfer as one in which the bus master places an address on the bus only once, after which several data transfers can occur before the master releases the bus. The master expects the slave to increment the current address appropriately each time a data element is transferred, so that successive transfers access successive locations. The terms *block read* and *block write* imply the direction of data flow in a VMEbus block transfer.

A VMEbus block transfer works somewhat differently from a Butterfly Plus block transfer. Both the Butterfly Plus and the VMEbus use block transfer to reduce the overhead of transferring several pieces of data by packaging them into one transaction. As in a VMEbus block transfer, only the starting address is sent during a Butterfly Plus block transfer. However, the data in a Butterfly Plus block transfer is sent in a data packet, as opposed to the VMEbus specification, where each data element in the block is sent one at a time. Data sent to or from a VME interface by block transfer through the Butterfly Plus switch may or may not be carried forward to the VMEbus by a block transfer operation. Also, although the PNC microcode allows the Butterfly Plus switch to transfer a block of up to 64 kilobytes, system software by convention usually breaks large block transfers into smaller blocks with no more than 256 bytes in one switch message. The VME interface can accept a data block of up to one kilobyte.

Two bits in the VME interface, VBA_CTL_SEQ and VBA_CTL_AINC (described for **transfer_ctl** in the section entitled "Control Registers"), control the handling of addresses on the VMEbus when transferring a sequence of data. If VBA_CTL_SEQ is set to one, the VME interface (acting as VMEbus master) expects the slave device to sequence all accesses to its memory. Therefore, if VBA_CTL_SEQ is set and the VME interface receives a block transfer message

from the Butterfly Plus switch, the data will be transferred using a VMEbus block transfer. The Butterfly Plus message can be either a block transfer request message, which causes a read from the VMEbus, or a block transfer data message, which causes a write to the VMEbus. The VME interface acts on each transaction it receives right away, without saving up operations and later combining a batch of them into one block transfer.

If VBA_CTL_SEQ is not set (this is a default setting), the VME interface does not expect the VMEbus device to sequence the addresses, and a VMEbus block transfer will not be used. This will be true for the majority of the VMEbus slave devices. If the VME bus slave device does not sequence the addresses, then the VME interface acting as master must do so by placing a new address on the VMEbus before each piece of data is transferred. Although the VME interface remains bus master until all of the data is transferred (up to the 256 byte maximum of a Butterfly Plus switch block transfer message), this is not a VMEbus block transfer because the address is placed on the bus more than once.

If the VME interface is sequencing the transfer address, the other control bit, VBA_CTL_AINC, comes into play. When this bit is set to one (default setting), the VME interface increments the address each time it accesses the VMEbus; when the bit is cleared zero, the VME interface does not increment the address, but instead places the same address on the VMEbus again and again. Incrementing the address is appropriate when accessing successive memory locations. Retaining the same address is useful for accessing a FIFO in a VMEbus device.

Another aspect of block transfer operations is atomicity. An operation is atomic or indivisible if the entire operation is guaranteed to complete before any other processor or peripheral device can access its intermediate results. A common example is an instruction that increments a memory location, which is usually implemented as a read-modify-write operation, in which the previous value is read from memory, updated in a register, and then stored back in memory. If another processor or device can access the memory location between the reading and the writing, the operation is not atomic and presents a *race condition*, sometimes called a *multiprocessing hazard*, because the second device can operate on the data and produce incorrect final results.

Block transfers are sometimes used to provide indivisible read or write operations. In the Butterfly Plus switch, a block transfer is indivisible only if it consists of eight or fewer bytes of data. When the VME interface performs a block transfer on the VMEbus, it does not release bus mastership until the transfer is complete. However, this does not guarantee atomicity. In particular, the memory on the VMEbus might be dual-ported to another bus, or it might be accessed by some processing unit on the same circuit card as the memory.

In summary, programs perform block transfers for various reasons, generally to gain maximum performance, but occasionally for atomicity. To ensure that a Butterfly block transfer is performed, the program must call the Chrysalis operating system and ask for a block transfer (via **block_copy**, **Do_bt**, or **Start_bt**).

VME Data Alignment

The VME interface reads and writes the VMEbus only in response to messages it receives from the Butterfly Plus switch. Except for interrupts, therefore, all VME interface activity can be covered by considering each Butterfly Plus message that causes VMEbus activity. When the VME interface receives a write byte, write word, or read word message from the Butterfly Plus switch, it performs a direct access operation on the VMEbus. A byte oriented direct access operation can access any byte address. A word oriented direct access operation can make only word-aligned accesses, since the Butterfly Plus switch messages to read or write a word cannot specify an unaligned word address. The same is also true for a longword oriented direct access operation.

The VMEbus interface performs 16-bit, word aligned read-modify-write cycles on the VMEbus for mask-then-add messages from the switch. This operation is considered a direct access one. This is the only instance when the VME interface performs a read-modify-write cycle on the VMEbus.

When the VME interface receives a Butterfly block transfer message from the switch, it generates one or more VMEbus accesses. The VME interface always performs the transfers as efficiently as possible, using unaligned transfers where necessary. Each of the three types of VMEbus unaligned transfers is generated when needed: the low three bytes of a longword (BYTE0-2), the middle two bytes (BYTE1-2), or the high three bytes (BYTE1-3). All VMEbus

slave devices are required to cope with word-aligned transfers, but support for unaligned transfers is optional. Either the system must be configured so that the Butterfly Plus performs block transfers only to or from devices that support unaligned transfers, or the Butterfly Plus programmer must be careful not to perform block transfers that start or end on an odd byte address.

A programmer may wish to perform a Butterfly block transfer that crosses one or more 64-kilobyte boundaries. Neither the Butterfly Plus processor node nor the VME interface supports this as a primitive function. In particular, carries do not propagate from the VBA_LOW_ADDR register in the VME interface into the VBA_HIGH_ADDR register, where the high-order 16 address bits presented to the VMEbus originate. Use of normal Chrysalis block transfer calls (*e.g.*Start_bt, Do_bt, or block_copy) ensure that block transfers do not cross 64-kilobyte boundaries.

To simplify the design of slave hardware, the current VME specification prohibits VMEbus block transfers from crossing any 256-byte boundary. This rule was not a part of earlier versions of the specification, and many VMEbus devices respond perfectly well to VMEbus block transfers that do cross one or more 256-byte boundaries. The VME interface does not enforce the prohibition against crossing such boundaries. If the VME card cage contains devices that rely on block transfers not crossing 256-byte boundaries, it is the programmer's responsibility to meet this requirement. Note, however, that the VME interface will not normally be operating in block transfer mode (the VBA_CTL_SEQ bit is not set for non-block transfers).

PROGRAMMING THE VME INTERFACE

The VMEbus software interface is supplied by Chrysalis operating system library routines. The VME interface hardware and firmware permit parts of the address space of a VMEbus system to appear in the address space seen by the Butterfly Plus user program. Data can be transferred in either direction between Butterfly Plus memory and VMEbus devices, and from one VMEbus device to another, across the VME interface. However, VMEbus devices cannot access Butterfly Plus memory through the VMEbus address space.

To transfer data, the MC68020 executing a program (or system routines called by the program) in a Butterfly Plus processor node generates a request to transfer data to or from a particular location. The location is given as a

Butterfly Plus virtual address, since the processor executes in virtual address space. The Butterfly Plus processor node translates this virtual address into a physical address. If the access is a remote one, the processor node sends a message through the Butterfly Plus switch requesting access to that remote location. This message arrives at the VME interface, which converts the message address from a Butterfly Plus physical address to a VMEbus physical address by means of a mapping table in the interface. The mapping process generates not only a 32-bit VMEbus address, but also six address modifier bits that describe the access to the VMEbus. (VMEbus address modifier bits are described in the section entitled, "VME Address Modifier".) The VME interface next places the address and the modifier bits on the bus, and the access is performed. If the access is a write operation, it is now completed. If it is a read operation, the requested data is received by the VME interface and returned through the Butterfly Plus switch to the processor node that requested it.

As can be seen, VMEbus accesses require addresses to be translated twice, once using the memory management unit in the processor node, and once using a mapping register in the VME interface. Mapping registers are limited in quantity. The major purpose of the Chrysalis VME interface software is to manage these and other limited system resources through allocation and release. Also, separate processes can contend for use of the same resource, and the VME interface software allocates mapping registers to prevent conflict. Chrysalis greatly simplifies the Butterfly Plus system programmer'vs task, since library routines handle most of the details of VME interface operation.

Chrysalis provides two ways to access VME devices: subroutine access and mapping. These two are not mutually exclusive, and in fact, the same or different processes may freely mix use of the two methods. Both methods require the Butterfly Plus to run the VME server process, and any program that accesses VME devices to be linked with the VME library routines. Table 5–3 summarizes the two methods. The subroutine access method is simpler and will be described first.

Table 5-3
Chrysalis VME Interface Access Methods

Function	Subroutine Access Method	Mapping Method
Initialization in program	None needed	**map_vme** allocates and initializes resources: mapping registers in VME Node Controller.
Cause a data transfer	**read_vme_byte**, **read_vme_word**, **read_vme_long**, **write_vme_byte**, **write_vme_word**, **write_vme_long**, cause a single access on VMEbus.	– any MC68020 instruction, e.g., MOVE. – **block_copy** Chrysalis library call, **Do_bt**, or **Start_bt**. – **Atomic_add**, **Atomic_and**, **Atomic_ior**, **Atomic_cTa**. – not dual queue operations.
Share resources among programs	Does not apply	**remap_vme** shares VME mapping registers.
Release resources	Does not apply	**unmap_vme** deallocates resources (mapping registers).

VME Node Controller and Port Numbering

Calls to the Chrysalis VME interface take an argument, **bvme_no**, that identifies a particular VME interface in the Butterfly Plus and one of the two switch ports on that VME interface. At the hardware level, a VME interface is identified by the address(es) of the switch port(s) to which it is connected. A switch port address can be any number from zero to 255, and can vary when the machine is reconfigured.

Programmers usually prefer to deal with consecutive identification numbers that start with zero and do not change when the Butterfly Plus is recabled. The Chrysalis VME software provides a numbering scheme that associates each VME interface with two consecutive numbers, one for each of its switch

ports. Thus, in a Butterfly Plus system with three VME interfaces, **bvme_no**s of 0 and 1 correspond to the two switch ports of one VME Node Controller, 2 and 3 to the ports of another VME Node Controller, and 4 and 5 to the ports of the third VME Node Controller. The lower number designates the port that is always connected (called "port 0" when discussing VME design) and the higher number of the pair designates the optionally-connected port (*i.e.*, port 1) if it is connected, or the always-connected port if the optional port is not connected. Thus, a **bvme_no** of 3 uses the optional switch port of the second VME interface, if it is connected, and uses its primary port otherwise. The **bvme_no** parameter is used to select the port both for control actions and for actual transfer of data to and from the VMEbus.

When the Chrysalis VME interface software starts up, it determines (from the Chrysalis system) which switch ports are attached to VME interfaces. Because this discovery process always scans switch ports in the same order, a VME interface always has the same **bvme_no**, as long as the Butterfly Plus is not recabled.

In some cases, however, the programmer needs to know exactly which **bvme_no** corresponds to which VME interface. In particular, the user must be aware that if the Butterfly Plus's VME interface configuration changes (by moving, adding, powering down, or removing VME interfaces), the mapping of **bvme_no**'s to VME interfaces will change. For those situations in which the programmer must identify a particular VME interface, the Chrysalis VME interface provides the function:

```
int find_the_node (bvme_no)
        int bvme_no;
```

Subroutine Access Method

In the subroutine access method, the Butterfly Plus program calls a library routine to perform a single access on the VMEbus. Six routines are available, supporting read or write, and byte, word, or longword data size. For example, to read a word, the program calls **read_vme_word**. The arguments supplied to this call are **bvme_no** (identifying which VME interface to use), **vme_addr** (the 32-bit VMEbus address to access), and **modifier** (additional bits placed on the VMEbus to describe the transfer). If the operation is a write, the caller also supplies the data to be written. Read data is returned as the value of the call.

The subroutine access method of reading or writing VMEbus locations is easy, and is attractive for making randomly scattered accesses and for occasional small transfers such as accessing control registers in VMEbus devices. This method incurs the costs of subroutine calling and of setting up the mapping register used for the access, however. Faster response is available using the mapping method. Also, only the mapping method supports block transfers and atomic operations on the VMEbus.

Mapping Method

In the mapping method, a Butterfly Plus program calls Chrysalis VME interface routines to allocate and initialize VME mapping registers, then uses normal Butterfly Plus programming methods to access the VME address space through the window described by the mapping registers. The allocation and initialization are slower than the calls of the subroutine access method, but, once the window is set up, the mapping method offers significantly higher performance. The mapping method also entails calling the interface routines to release the mapping registers when the program is through using them. The mapping method mimics the Chrysalis primitives **Map_Obj**, which maps a portion of Butterfly Plus memory (a memory object) into the caller's address space, and **Unmap_Obj**, which removes it.

The user program calls **map_vme** to allocate and initialize memory on its processor node and mapping registers in the VME Node Controller. The **map_vme** call takes the same **bvme_no**, **vme_addr**, and **modifier** arguments used in the subroutine access method described earlier. The **map_vme** call also requires **bfly_seg** (the segment of Butterfly Plus virtual address space that will become the window into VMEbus address space), and **size** (the size of the window, in bytes). A **bfly_seg** argument value of zero implies use of the highest free segment. The **size** argument is rounded up to the next 64-kilobyte boundary. The **map_vme** call returns BFLY_ADDR, the 32-bit Butterfly Plus virtual address of the mapped memory block on the VMEbus. This BFLY_ADDR value remains valid until an **unmap_vme** call releases the VME Node Controller mapping registers used by **map_vme**.

After the **map_vme** call, the programmer uses **BFLY_ADDR** as the base address of a block of memory, **size** bytes in length, that can be manipulated by normal Butterfly Plus operations. This block of memory responds both to normal MC68020 instructions such as **MOVE**, and to Chrysalis functions invoked by the program, such as block transfers and atomic operations.

Block transfers are typically performed by calling the Chrysalis library routine **block_copy**; however, the more basic Chrysalis routines **Do_bt** and **Start_bt** are also available. The atomic operations **Atomic_add**, **Atomic_and**, **Atomic_ior**, and **Atomic_cTa** operate on VMEbus locations through the window just as they do on Butterfly Plus memory locations. They are atomic with respect to all VMEbus devices as well as to all processes executing on the Butterfly Plus. (These atomic operations apply to 16-bit quantities only.) Although most operations work on VME memory just as on Butterfly Plus memory, the VME interface and its Chrysalis software do not support any dual queue functions to locations on the VMEbus.

Mapping registers are allocated on a first come, first served basis by the Chrysalis VME server process. A throw exception occurs when mapping registers sufficient to service a **map_vme** call are not available. Each VME interface has 64 mapping registers, but two of these are reserved, leaving 62 for allocation to user windows. (Butterfly Plus physical addresses specifying mapping register 0 refer to bank 0 memory within the VME Node Controller, and cause no activity on the VMEbus. Mapping register 1 is reserved by the Chrysalis VME interface software to satisfy subroutine access method calls, which never fail from lack of a free mapping register.)

The Chrysalis VME interface provides a way to share the same window among multiple Butterfly Plus processes. The **remap_vme** call is like the **map_vme** call, but instead of using previously free mapping registers, **remap_vme** searches the mapping registers for one or more that already refer to exactly the same window as requested by the **bvme_no** (exclusive of the port bit), **vme_addr**, **modifier**, and **size** arguments. If such a mapping register is found, a usage count for that register is incremented, and the calling process shares it with other processes. The **BFLY_ADDR** returned by **remap_vme** is the same as that returned by **map_vme**, and is valid until an **unmap_vme** call releases the mapping registers.

The Chrysalis VME interface software maintains the usage count so that VME interface mapping registers are freed only when no processes are using them. Processes running on different processor nodes can share mapping registers.

If the **remap_vme** call does not find a block of mapping registers that specify the same window as the parameters of the call, a **throw** exception occurs. The programmer may want to reuse mapping registers if there are already some covering the needed window, and allocate new mapping registers otherwise. This can be accomplished by putting the **remap_vme** call in a **catch** block, and calling **map_vme** when a **throw** occurs:

```
catch
    remap_vme(parameters);
onthrow
    when (TRUE)
        map_vme(parameters);
endcatch
```

When a process is finished using its window on the VMEbus, it can release the mapping registers associated with the window by calling **unmap_vme**. This permits the re-use of those resources by the same or other processes. VME mapping registers are a Butterfly Plus systemwide resource shared among all processes. The **unmap_vme** call decrements the usage count for the mapping registers, and, if the usage count is zero, deallocates the mapping registers.

INTERRUPTS

The Chrysalis VME interface support for interrupts is independent of the method by which VMEbus data is accessed. Just as the same or different processes can mix subroutine access method calls, mapping method calls, and standard memory accesses, the same or different processes can make calls to use the VME interrupt facilities. The VME interface software places no constraints on the use of VME interrupts. Freedom from constraints also means that no protection or interlocking is enforced, so any protection or locking required is the programmer's responsibility. The VME interface software does not use the VME interrupt facility, so there is no risk of conflict between the system software and the application code.

The VME interface can request interrupts on the VMEbus at the request of Butterfly Plus processes, and it can also handle VMEbus interrupts generated by other VMEbus devices.

Requesting VMEbus Interrupts

A Butterfly Plus program generates an interrupt request by calling **write_vme_int_req**. One argument specifies which VME interface will generate the interrupt request, and a second argument determines the level of interrupt request to generate. The Chrysalis interface software places the second argument in the **vme_int_req** control register of the VME Node Controller, from where VME interface firmware transfers it to the interrupt vector register and interrupt request register of an SCB68154, and the chip places the interrupt request on the VMEbus. The low-order four bits of the interrupt vector must be 1 for a level 1 interrupt, 3 for a level 2 or 3 interrupt, 5 for a level 4 or 5 interrupt, or 7 for a level 6 or 7 interrupt.

A program may need to check on the status of interrupts, particularly to determine whether any are pending. The program obtains the current values of the registers in the SCB68154 chip by calling **read_vme_int_req**. For more information, refer to a SCB68154 data sheet.

Handling VMEbus Interrupts

When an interrupt request appears on the VMEbus (on any of the seven levels), or when any of four other conditions arise (see the section entitled "VME Interrupts"), the VME interface examines a control register associated with that request or condition. If that control register contains zero, no action is taken. If the contents are nonzero, the VME interface treats it as an event handle to post; the Butterfly Plus process owning that event will receive the event. The data posted with the event is the interrupt vector received from the VMEbus. The control register is cleared, and, if it was an interrupt request that triggered the action, the interrupt is acknowledged on the VMEbus by the VME interface.

Writing an event handle into a control register both enables the interrupt and specifies the event to post; writing zero disables the interrupt. The user's program can fill one of these registers by calling **write_vme_vector** with arguments specifying **bvme_no** (the target VME interface), **vme_int_level** (the interrupt level), and **EventHandle** (the event handle to post). See the entry **intr_event** in the section entitled "Control Registers" section for the assignment of registers to interrupt levels.

VME Interface Control Registers

The calls that exercise the VMEbus interrupt system are special cases of accessing VME interface control registers. Two low-level routines provide access to all the VME interface control registers. **read_vme_special_register** returns the value in a given control register, and **write_vme_special_register** writes a value into a control register. Application programs are unlikely to need these low-level routines for typical uses of the VME interface. The interrupt-related calls simply invoke these register access calls with appropriate arguments.

Parallel Transfers for Higher Bandwidth

Because the VME interface transfers data through the Butterfly Plus switch, higher data transfer rates between the VME and the Butterfly Plus system can be achieved if a program uses multiple switch paths in parallel. The two switch ports of each VME Node Controller can execute data transfers simultaneously. A program can establish parallel data transfers between Butterfly Plus nodes and each VME switch port. In addition, if a Butterfly Plus has several VME interfaces, they can all operate in parallel. Of course, a single VMEbus performs only one transfer at a time, but its bandwidth is greater than that of a single Butterfly Plus switch port by a factor of about 5 (in practice) to 10 (in theory). Chrysalis VME interface software allows application programs to perform simultaneous data transfers that achieve significantly higher performance.

Library Routines and the Server Process

The Chrysalis VME interface consists of two parts, library routines and a server process. The VME library routines must be linked with the user code that accesses VMEbus devices; other parts of the user's application do not need to be linked with the VME library. When the program makes calls to access the VMEbus, these library routines perform what they can by themselves, and request action from the server process as necessary, such as to allocate mapping registers.

Most library calls entail a call to the server process. The only exceptions are calls accessing VME interface special registers, such as calls manipulating the VME interrupt system. The library routines maintain a local cache of **bvme_no** to switch address correspondence; if a call accessing a VME interface special register specifies a **bvme_no** already in the cache, then no interaction with the server occurs.

Disk Data Transfer Example

A typical way the VME interface is used is to transfer data from a disk on the VMEbus into Butterfly Plus memory. The hardware I/O components include a Butterfly Plus system with a VME interface attached to a VMEbus; on the VMEbus are a disk and its controller, and a memory. The following steps describe the VMEbus disk transfer process.

1. The first thing the user program does is set up the transfer. This involves three steps. First, single word transfers are used to set up control and status registers in the disk controller. Next, the appropriate VME interface interrupt register is initialized with an event handle. The user can employ a different event handle for each of seven levels of VMEbus interrupts, so different interrupt levels generated by the disk controller can be distinguished this way. Finally, the disk transfer is started by sending a *go* command to the disk controller.
2. The disk controller transfers the data from the disk into the memory on the VMEbus. This memory is used as a buffer.

3. The disk controller sends a VMEbus interrupt to the VME interface. The VME interface acknowledges the interrupt on the VMEbus, and sends a post event message through the Butterfly Plus switch to the event owner, the user process. The user process receives the event, informing it that the data has been transferred as far as the buffer memory.
4. The user program block-transfers the data from VMEbus memory into Butterfly Plus memory (normally by a **block_copy** call). The user program has presumably already set up a window to the VMEbus memory by using the Chrysalis VME interface call **map_vme**.

List of Calls

The list below summarizes the available calls to the Chrysalis VME interface. The arguments accepted by calls to the Chrysalis VME interface library are also shown.

`int bvme_no` — Identifier of VME interface and port to use.

`int vme_addr` — A byte address on the VMEbus.

`int modifier` — The six VMEbus address modifier bits. A value of 0 defaults to 0x3d, which means standard (24-bit address) supervisory data access. This value is appropriate for simple memory access.

`byte/word/long value` — The value to be written to the VME interface or to a VMEbus device.

`int bfly_seg` — Desired segment number (0 => use highest free segment).

`int size` — Size of block to be mapped, in bytes.

`int bfly_addr` — The Butterfly virtual address (BFLY_ADDR) returned by a previous **map_vme** or **remap_vme** call.

`int vme_int_level` — A VMEbus interrupt level, 1-7.

`EH EventHandle` — An event handle to be posted upon VMEbus interrupt.

`char REGISTER` — The name of a VME interface special register.

The symbolic constants valid for the REGISTER argument, as well as the **throw** value (VME_ERR) used by the library when an error occurs, are found in the file */usr/butterfly/chrys/VERSION/include/vme.h.*

The calls:

```
byte  read_vme_byte(bvme_no, vme_addr, modifier)
word  read_vme_word(bvme_no, vme_addr, modifier)
long  read_vme_long(bvme_no, vme_addr, modifier)
```

return the specified byte, word, or longword from the specified VMEbus address.

The calls:

```
void  write_vme_byte(bvme_no, vme_addr, modifier, value)
void  write_vme_word(bvme_no, vme_addr, modifier, value)
void  write_vme_long(bvme_no, vme_addr, modifier, value)
```

write the specified byte, word, or longword value into the specified VMEbus address.

After the call:

```
BFLY_ADDR map_vme(bvme_no, vme_addr, modifier, bfly_seg, size)
```

the value of **size** will be rounded up to the next 64-kilobyte boundary. This call maps the appropriate 64-kilobyte VMEbus block(s) into the Butterfly Plus segment(s) specified. It returns the Butterfly Plus virtual address that references the beginning of the VMEbus block. This address is valid until an **unmap_vme()** call deallocates the mapping register(s) used by this call.

The call:

```
BFLY_ADDR remap_vme(bvme_no, vme_addr, modifier, bfly_seg, size)
```

searches the mapping registers for the map register group that has already mapped in the specified block. This block will have been created by a previous call to **map_vme()** with exactly the same arguments. The call maps the associated VMEbus block(s) into the Butterfly Plus segment(s) specified. It returns the Butterfly Plus virtual address that references the beginning of the VMEbus block. This address is valid until an **unmap_vme()** call deallocates the mapping register(s) consumed by this call.

The call:

```
void  unmap_vme(bvme_no, bfly_addr)
```

decrements reference counts and, if necessary, deallocates mapping registers used to access VME memory block(s) pointed to by **bfly_addr**. As a result of this call, the VMEbus is removed from the Butterfly Plus address space of the process that performs the unmap.

The call:

```
void  write_vme_int_req(bvme_no, value)
```

writes to an SCB68154's interrupt vector and interrupt request registers, causing an interrupt to appear on the VMEbus. Details of appropriate values to pass to the SCB68154 can be found in its data sheet.

The call:

```
int  read_vme_int_req(bvme_no)
```

reads an SCB68154's interrupt vector and interrupt request registers, returning the values in those registers.

The call:

```
void  write_vme_vector(bvme_no, vme_int_level, EventHandle)
```

writes **EventHandle** into the register for the VMEbus interrupt level given by **vme_int_level**. This both specifies the event the VME interface will post when that interrupt appears, and enables the handling of that interrupt by the VME interface. Writing an **EventHandle** with a value of zero disables the interrupt.

The call:

```
int  read_vme_special_register(bvme_no, REGISTER)
```

returns the value of the specified VME special register.

The call:

```
void  write_vme_special_register(bvme_no, REGISTER, value)
```

writes the specified value into the specified VME special register.

Chapter 6

Programming the Butterfly Plus

Several high-level programming languages are used to program the Butterfly Plus parallel processor. The overwhelming majority of Butterfly Plus software created to date has been written in the C programming language. Fortran–77 and Scheme programs have also been developed. All application programs run under the Butterfly Plus's Chrysalis operating system, regardless of the particular high-level language in which they are coded. This chapter provides a brief overview of some of the tools available to programmers creating applications in the Butterfly Plus environment.

Programs for the Butterfly Plus are written using a cross compiler and other software development tools on a front-end machine. Currently, C and Fortran–77 language development programs run under 4.2BSD UNIX on various front-end minicomputer systems, allowing use of the rich set of UNIX software tools available. An Ethernet or a serial line connects the Butterfly Plus parallel processor to the front-end. A typical development cycle consists of editing, compiling, and linking a program on the front-end, then downloading, running, and debugging the program on the Butterfly Plus system. A source language debugger for the C language runs on the front-end, allowing cross-network debugging of programs running on the Butterfly Plus. Figure 6–1 is a block diagram of a typical software development configuration.

An interactive command interpreter, called the Butterfly Plus shell, or **bshell**, and a terminal window manager are available to create and run programs on the Butterfly Plus parallel processor. The user accesses the system from the front-end computer, and the window manager allows rapid switching between

the front-end and Butterfly Plus system environments. The **bshell** allows application loading and execution. The window manager allows multiple processes to share a terminal for both input and output. It also provides line editing and screen management functions, such as cursor positioning.

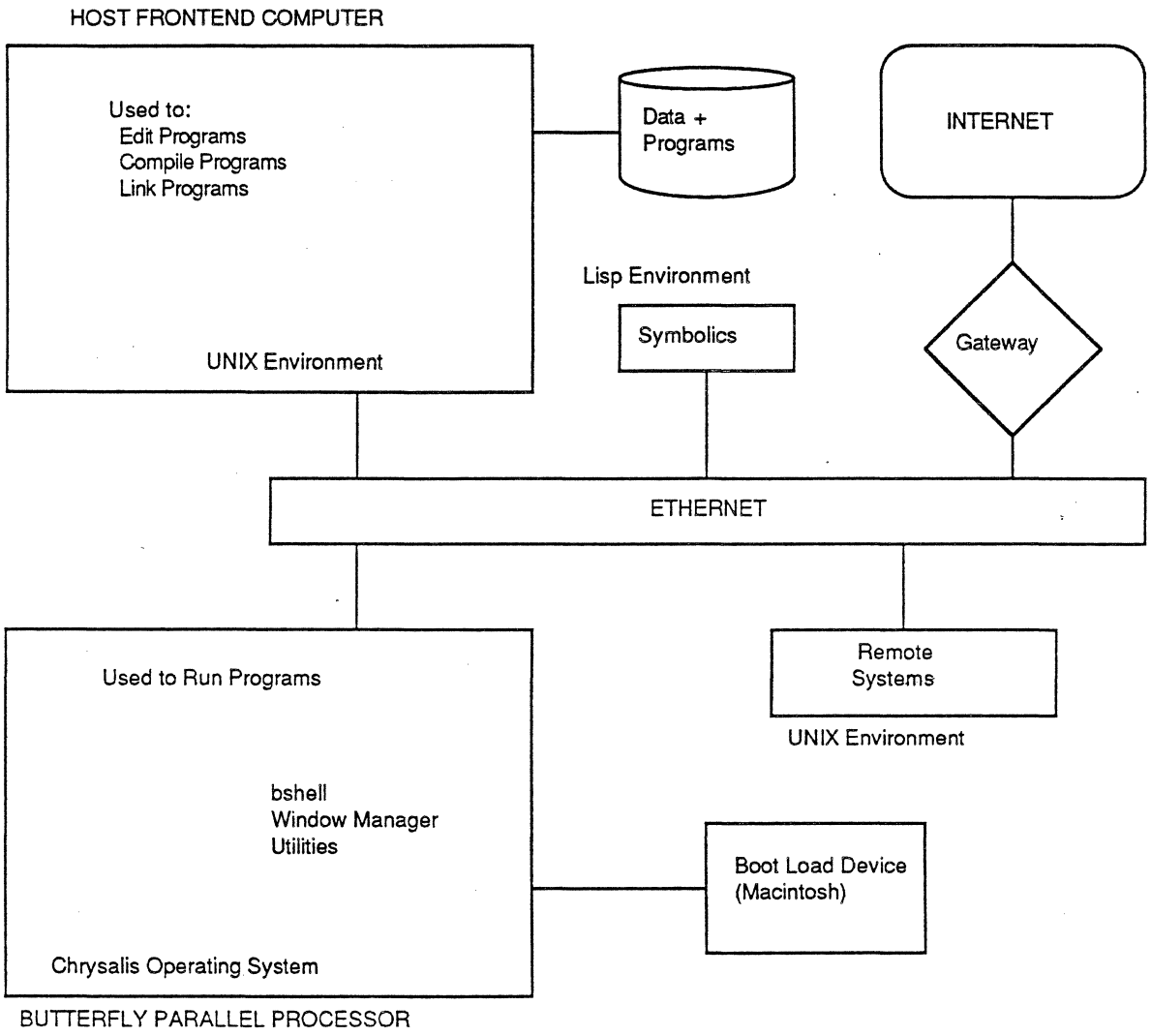


Figure 6-1
Butterfly Plus Programming Environment

CHRYSALIS OPERATING SYSTEM

The Chrysalis operating system supports applications programs on the Butterfly Plus. It provides a familiar, UNIX-like environment that supports high-level language programming without restricting users to a single language. Chrysalis has several levels. At the highest level are interactive utilities for accessing, controlling, and debugging programs on the system. These include the interactive command interpreter and window manager, as well as utilities for loading and running applications, and determining application and system status. The lower levels of Chrysalis are called as subroutines from the user's program. The higher levels provide interfaces more suitable than the lower levels for most user applications, although the lowest levels are available if needed.

Application Libraries

Application libraries manage many of the details of system resource use, such as allocating memory and setting up processes. These libraries also provide the interfaces to the rest of the operating system for application programs. Currently, the most significant application library in the Chrysalis operating system is the Uniform System library, which supports a methodology for programming the Butterfly Plus.

The Uniform System creates an environment where all processors share a common address space. Under the Uniform System, processes are created on as many processors as are available in a Butterfly Plus system configuration. Each process runs the user's application tasks on a subset of the data. The Uniform System library provides subroutines to efficiently allocate the data structures for the problem, then calls upon processors to work on the tasks specified by the programmer. Besides the Uniform System library, there are other application libraries, including:

- A buffer management package for communication applications
- A stream oriented I/O interface, similar to UNIX standard I/O, whose subroutines allow applications to receive terminal input and to perform formatted output of character strings
- A library of performance measurement tools oriented toward parallel applications

- The RAMfile System
- The X Window System.

Server Functions

Access to system resources, such as the Ethernet, is provided by server processes. The operating system's interprocess communication facilities route these requests to the appropriate server processes. Servers can execute functions on remote nodes, translate symbolic resource names to addresses, and load programs. Additional servers are available as options to support remote debugging, to provide remote file access and interface to application programs running on front-end machines, and to access networks. Remote file access allows users to read and write files on front-end machines. A remote procedure call facility allows applications running on the front-ends to communicate over the network with programs on the Butterfly Plus system.

A network server provides an interface to the Ethernet high-speed local area network and supports TCP/IP, the standard communication protocols for reliable data transport. Use of TCP/IP also lets other hosts in the DARPA Internet access the Butterfly Plus processor through gateways.

Kernel Functions

The lowest level of Chrysalis functions is provided by the kernel, a copy of which runs on each of the processors of the Butterfly Plus system. The kernel's main functions are process support, memory management, primitives for parallel processing and interprocess communication, system configuration and initialization, and I/O. Some Chrysalis kernel services are written in microcode on the processor node controller for speed; most of the rest are written in the C language. All Chrysalis kernel functions are called from the user's programs as subroutines or macro invocations.

Multiprogramming Support

Chrysalis provides multiprogramming via a process scheduler that runs on each processor. This allows multiple processes to run on each node, as well as concurrently with those on other nodes.

Multi-User Support

Chrysalis provides a multiple user environment on the Butterfly Plus. Each user is the exclusive owner of a variable number of processors. A user cannot allocate the resources of another user (*e.g.*, allocate memory on a processor owned by someone else). However, he or she can use the resource of another user if that user allows it.

Memory Management

All memory contained on each processor node is controlled through a Motorola MC68851 Paged Memory Management Unit (PMMU). The PMMU maps virtual address references using 8-kilobyte pages and a segment mapping granularity of 64-kilobytes. All processors have virtual address spaces of 4-gigabytes with page trees that are automatically generated as needed. Through this mapping scheme, programs running on a Butterfly Plus processor can reference the memory of any other processor node in the system, as well as any I/O registers and the microcoded kernel functions. Memory protection is enforced on a segment basis with kernel or user mode read, write, and execute attributes. Memory mapping is managed by the kernel and can be directly controlled by the application program, although mapping is typically done by a higher level of system software, such as the application libraries.

Another level of memory management, called object management, is provided by the Chrysalis kernel. This level revolves around objects, which are associated with areas of physical memory, or special system data structures such as processes and queues. The object system provides processor independent identifiers, called object handles, for areas of memory or for these system structures. The object handles can be passed among Butterfly Plus processors through the interprocess communication facilities. Other Chrysalis kernel calls can then map the objects into virtual address space, where the objects can be manipulated by programs.

Synchronization Primitives

Objects provide many of the primitives for interprocess communication and for parallel process synchronization and locking. Two of the most important of these primitives are the dual queue and the event. A dual queue is an interlocked data queue that can be used as a lock or for passing data, such as object handles, between processes. Events cause processes to be either suspended or

scheduled to run through wait and post operations. Together, dual queues and events are used to build higher level synchronization and interprocess communication mechanisms, such as locks, semaphores, and remote procedure calls. In addition, a set of microcoded atomic memory operations (adds and logical operations) is provided to implement simpler forms of locks and other multiprocessor synchronization facilities. Table 6–1 lists the atomic memory operations.

Table 6-1
Atomic Memory Operations

Atomic_add	Indivisibly add to a memory location.
Atomic_and	Indivisibly AND to a memory location.
Atomic_cTa	Clear then add to a memory location.
Atomic_ior	Indivisibly OR to a memory location.

Input/Output Support

The I/O functions include support for the Multibus and VMEbus hardware interfaces, a driver for the Multibus Ethernet controller, TCP/IP, remote file access, and asynchronous terminal support. The programmer's interface to the I/O system is similar to that of UNIX (**open**, **close**, **read**, **write**, etc.).

SOFTWARE DEVELOPMENT ENVIRONMENTS

The development of parallel processor programming methodologies is an active research area. The Butterfly Plus hardware and software environment represents an excellent vehicle for pursuing this research.

To date, three distinct approaches to programming the Butterfly Plus have seen widespread use. One approach is based on the notion of cooperating sequential processes as described by Dijkstra, Hoare and others. The second is the Uniform System approach developed at BBN Laboratories. The Uniform System approach emphasizes the tasks that comprise an application and tends to de-emphasize the notion of processes. The basis of the third approach to programming the machine is the Butterfly Plus Scheme implementation. Combinations of two or more of these approaches are possible, as are completely different approaches.

The three following sections provide a brief review of the familiar cooperating sequential processes approach, describe the Uniform System approach in some depth, and outline the Butterfly Plus Scheme environment. In addition, a fourth section describes the RAMFile system, a subroutine library that lets applications programmers structure Butterfly Plus memory as if it were an I/O file, thus increasing the ease of RAM data access.

Cooperating Sequential Processes

To structure an application that uses cooperating sequential processes, the programmer decomposes the application into a moderately sized collection of loosely coupled processes that exchange control signals or data from time to time. Chrysalis supports this programming methodology by providing a set of mechanisms for synchronizing and controlling processes and for interprocess communication. Using this method to program an application is similar to programming a multiprocess application on a uniprocessor based machine. The principal difference is that the processes actually run concurrently on the multiple processors of the Butterfly Plus, whereas the processes must run in an interleaved fashion on a uniprocessor. Cooperating sequential processes often make limited use of shared memory, as well as message passing.

Communication and networking applications developed at BBN for the Butterfly Plus use the cooperating sequential process approach. The Chrysalis server functions also use this approach. In the latter case, the cooperating processes are the Chrysalis server processes and the application programs that call upon the servers. Communication between the application programs and the server processes occurs within Chrysalis subroutines. All of this is transparent to the application programmer.

The Uniform System

The Uniform System approach has proven to be particularly effective for applications containing a few frequently repeated tasks, such as those used for most scientific computing. It has also been used successfully in applications with less homogeneous task structures.

Beyond the usual concerns of programming, there are two key considerations specific to the Butterfly Plus: storage management and processor management. The goal of storage management is to keep all the memory modules in

the machine equally busy, thereby preventing the slowdown that occurs when many processors attempt to access a single memory module. The goal of processor management is to keep all the processors equally busy, thereby preventing the inefficiency that occurs when some processors are overloaded while others sit idle without work to do.

Uniform System Memory Management

The Butterfly Plus switch provides low-delay, high-bandwidth access to all of the memory in the machine. To help the programmer take advantage of this common memory, the Uniform System implements a large shared memory for application programs, and provides the means to spread application data uniformly across the memories of the machine. Chrysalis in turn provides memory mapping operations that enable processes to manage their address spaces, and hence the memory they access. Two or more processes can share memory by mapping the same memory module.

There are two ways to share memory among processes. One is to isolate processes from one another by mapping memory so that only a relatively small subset of each process's address space is accessible to other processes. The shared subset of memory can be changed at any time and is often different for different groups of processes. Although this method facilitates debugging by limiting the number of processes likely to have changed a particular data structure, the Uniform System uses a different approach. Under the Uniform System, a single large block of memory is shared by mapping the block into the address space of each processor. This method frees the application programmer from the need to manipulate memory maps, and simplifies programming by implementing a large shared address space for application programs. Data that must be shared by two or more processors is allocated without regard to which processors will be using it. The code, stack, and global variables local to individual processes are kept locally, and are not fetched across the Butterfly Plus switch.

Collectively, the local memories of all the Butterfly Plus processor nodes form the shared memory of the machine. The large shared memory an application program sees is physically implemented by a collection of separate local memories. If all the shared data used by an application were to be physically located in a single memory, contention for that memory (as many processors attempt to access the data) would force the processors to proceed serially,

thereby slowing program execution. Since the aggregate memory bandwidth of the machine is very large, slowdowns due to memory contention can be reduced by scattering application data uniformly across the physical memories of the machine. When many processors access scattered data, their references tend to be distributed across the memories, enabling the processors to use the full memory bandwidth of the machine. The Uniform System library provides a memory allocator that scatters data structures in a way that allows straightforward addressing conventions. The library also supports a set of more specialized techniques that can be used if the allocator is either inappropriate or ineffective.

To summarize, the Uniform System approach to memory management is based on two principles:

- Use of a single large address space shared by all processes to simplify programming
- Scattering application data uniformly across all memories of the machine to reduce possible memory contention.

This memory management strategy has two slight disadvantages because of the slower access to remote memory and, to a lesser extent, because of possible contention in the switch and at the memories. This results in an increase in execution time of about 4% to 8%. The benefit of this memory management strategy is that the programmer can treat all processors as identical workers, each able to do any application task, since each has access to all application data. This greatly simplifies programming the machine.

Uniform System Processor Management

Optimum utilization of the Butterfly Plus's unique parallel architecture requires efficient processor management. Such management requires identification of the parallel structure inherent in a chosen algorithm, and control of the processors to achieve the determined parallelism.

Determining Parallel Structure

In many applications the parallel structure is obvious. In others, the structure is less clear and may require reworking algorithms. Occasionally, an algorithm is inherently serial, and cannot be structured to take advantage of parallel processing. The following guidelines will help in optimizing applications for the parallel environment.

- Start with the best existing algorithm. A Butterfly Plus system with **p** processors can do no more than speed up an algorithm by a factor of **p**. Speeding up a poor algorithm may not overcome its inefficiencies. For example, it may take an n^2 parallel sort longer to run on a 128-processor Butterfly Plus than it takes an $n \log n$ sort to run on a single processor.
- Try to do the same number and kind of steps as the best algorithm. The order of steps in an algorithm can often be manipulated to achieve parallelism. This procedure may involve adding logic in the form of simple locks to ensure the atomicity of selected operations.
- Look for parallel structure at all levels and in all sizes: the more the better. If necessary, the programmer can usually combine small tasks at a later stage into larger, more manageable sizes; it is often more difficult to divide a task at a later stage into smaller ones. For example, if an application requires fast Fourier transformations (FFTs) on many different channels, the programmer should plan to exploit both the parallelism inherent in an individual FFT and the parallelism due to different channels.

The Butterfly Plus can work very efficiently with individual tasks that are a few milliseconds in length; if necessary, it can also work on tasks in the hundreds of microseconds. For shorter tasks, various overheads begin to affect the quality of the performance.

There are two strategies for determining the most efficient number of concurrent operations to have at any stage in the processing. One strategy recommends a relatively static approach, using exactly **n** concurrent tasks for **n** processors. The other strategy recommends using many more than **n** tasks, typically ten times as many or more. Both strategies attempt to deal with end effects; that is, the processor idle time that occurs toward the end of a stage when some processors have finished and others are still working. The first approach minimizes the effect by explicit construction. Here the programmer attempts to manipulate the work so that all processors finish at approximately

the same time. The second approach allocates tasks to processors dynamically in an attempt to balance the load. As a processor finishes a task, it is assigned the "next" task ready for execution. This approach relies on having a large number of tasks relative to processors to minimize idle time. Some idle time occurs at the end of the problem, but this is generally acceptable since it is small relative to the total program execution time.

The Uniform System encourages the dynamic approach. For many applications the dynamic approach is simpler and more reliable, since it is not necessary to know in advance how long an individual piece of work will take. Furthermore, it is adaptable to varying numbers of processors and sizes of problems.

Controlling Parallelism: Structuring the Application

After determining what processing is to occur in parallel, the programmer must then manipulate the Butterfly Plus to make this parallelism happen. There are several ways to do this. The Chrysalis kernel provides a rich collection of relatively low-level operations for starting processes on various processors and for communicating among them. The Uniform System provides a higher level abstraction for managing the processors, one that is natural and efficient for a large class of applications.

The Uniform System treats processors as a group of identical workers, each able to do any task. To use the Uniform System, a programmer must structure an application into two parts; a set of subroutines that perform various application tasks, and, one or more "generators" that identify the "next" task for execution.

To illustrate this, consider matrix multiplication as an example. One way to structure a matrix multiplication program would be to write a routine that computes the dot product of a row and a column, and to ensure that the routine for the dot product task gets called once for each element of the result matrix, using the appropriate row and column of the operand matrices as parameters.

Usually a well designed program is structured as a set of subroutines to improve program modularity, whether or not it is intended for parallel execution. There is normally a subroutine for each task type, with each subroutine taking arguments that define individual tasks in terms of subsets of the program data to be operated on. To use the Uniform System, the programmer simply ensures that these subroutines correspond to the tasks he wants to do in parallel. In the matrix multiplication example, there is a single task type, computing dot products, and corresponding to that task type, the dot product routine, whose row and column parameters specify particular tasks.

The second part of the application code comprises one or more subroutines able to identify the next task for execution. Such a subroutine is called a generator, since its function is to generate tasks. In a serial program the generator function is usually embedded in the control structure of the program (*e.g.*, do this, then do that, then do ten of these). For parallel processing under the Uniform System, the programmer is expected to make generation of the next task explicit. In the matrix multiplication example, the task generator would be responsible for generating a call on the dot product routine for each element in the result matrix.

It is helpful to think of the generator concept in terms of three procedures (generator activator, worker procedure, and task generator) and task descriptor data structure. Figure 6–2 illustrates how the Butterfly Plus task generator works.

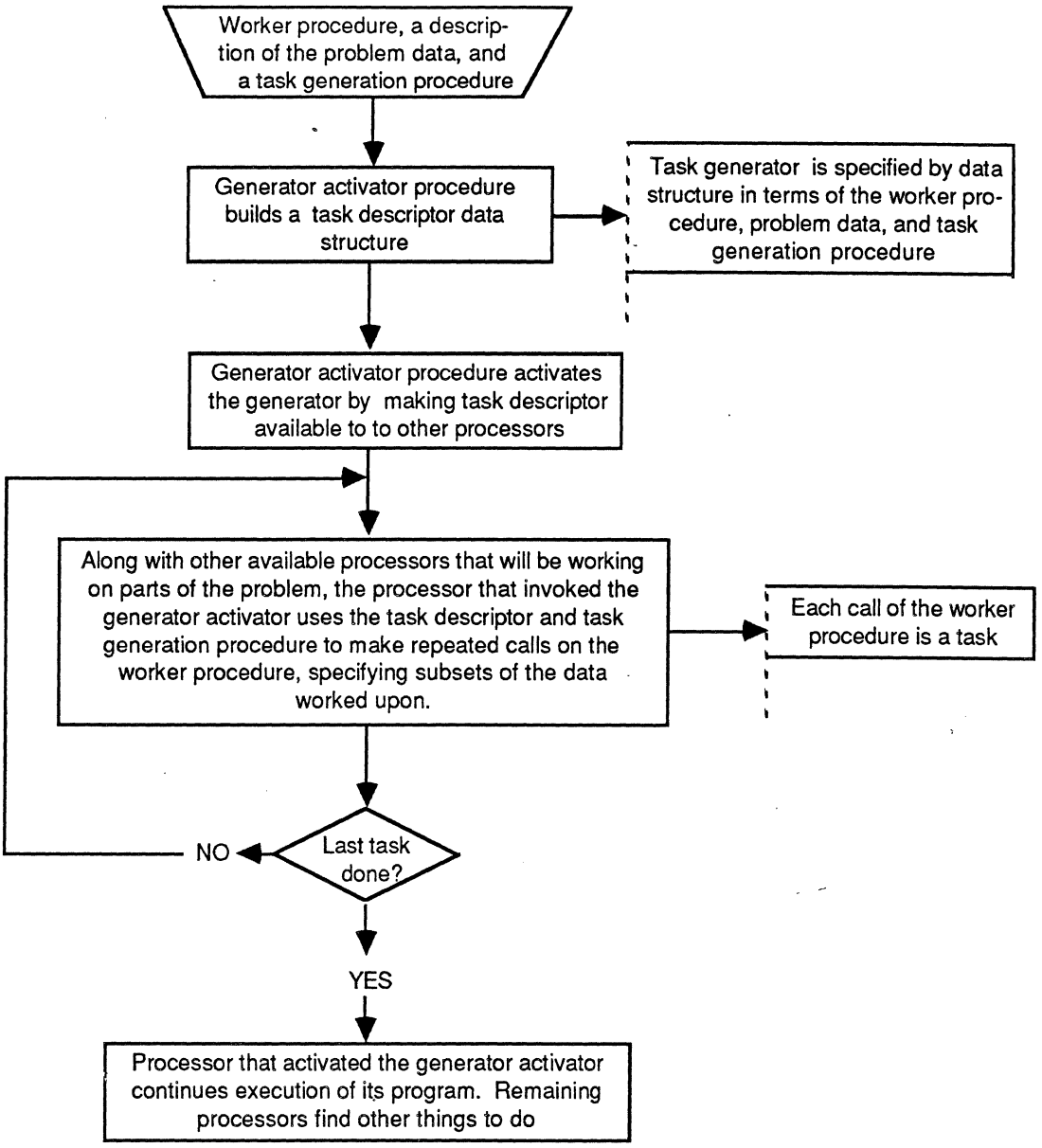


Figure 6-2
Butterfly Plus Task Generator

In the matrix multiplication example, the worker procedure is the dot product routine, and the data is the operand and result matrices. The dot product worker routine is called once for each combination of row and column index; these indices are stored in the task descriptor and incremented indivisibly each time the task generation procedure is executed by a processor.

Conceptually, the generator notion is similar to the various "map" functions in the Lisp language. What is unique about the Uniform System is that it achieves parallel operation by using processors, as they become available, to execute the various calls upon the worker procedure. Task generation and the processor management associated with it are implemented in a distributed fashion in the sense that each processor performing tasks participates in their generation.

Often the required generator is quite simple. In the matrix multiplication example, where a dot product is computed for every element in the result matrix, the generator can find the next task by incrementing row and column counters that identify the element in the result matrix to be computed next. Occasionally a generator must be more complex. A generator that selects the next node to process in an alpha-beta tree walk, for example, would rely heavily on using the most up to date information about the tree processing state. Occasionally a generator involves a simple queue, in which case it would operate much like a process scheduler found in many time sharing systems; the next task for execution would be the one at the front of the queue. In general, though, a large number of applications can be constructed from a small set of generators. The Uniform System library includes a collection of commonly used generators.

The Uniform System library provides a way to bind task generation procedures to worker procedures. The basis for this binding mechanism is a "universal" generator activator procedure. To use this universal generator activator procedure directly, application programs specify both a worker procedure and a task generation procedure. The library also includes a set of generator activator procedures that embody many commonly used task generation procedures. When an application program calls one of these "specific" generator activator procedures, it specifies only the worker procedure. In response, the generator activator passes its associated task generation procedure and a task descriptor to the universal generator activator along with the specified worker procedure.

Often an algorithm requires multiple, perhaps nested, instances of generators. As long as the algorithm does not depend on the order of task generation among different generators, the programmer is free to make multiple calls to task generators to start the system working on all of them at once. If the algorithm does depend on the order, the programmer must either provide a task

generation procedure to properly answer the question about what to do next, or carefully manage the use of existing generator activator procedures to ensure that the algorithm's ordering requirements are met.

The Uniform System approach to processor management offers three important benefits:

- The generator mechanism is very efficient. It is implemented using one process per processor in a way that ensures no unnecessary context swaps occur. Each processor executes a tight loop consisting of "generate next task – execute next task." The programmer supplies both the task generation and worker procedures, usually finding an appropriate generator activator procedure in the library. Both the task generation and the worker procedures execute at the application level. As a result, once a generator gains control of a processor, the Chrysalis kernel need not be involved until the generator has exhausted all the work it knows how to find.
- Programs that use the Uniform System task generation mechanism to exploit parallelism are insensitive to the number of processors. It is thus possible to debug programs on small configurations and run them on larger ones. If an application expands to exceed the capacity of its current configuration, it can be moved without modification to a larger one. Perhaps more importantly, programs can run on "reduced" configurations if necessary, such as when some machine processors have been removed for repair.
- The load can be balanced dynamically. Whenever a processor becomes free, a generator identifies the next task to be executed. Since the task generation procedures are supplied by the application programmer, the task choice can be based on the current state of the computation and the requirements of the application.

Programming With Butterfly Plus Scheme

Butterfly Plus Scheme provides the Butterfly Plus user with a complete multiprocessor Lisp environment. This environment includes not only Butterfly Plus based Lisp language support with extensions for multiprocessing, but also many user interface features and program development aids in a Symbolics 3600 serving as a front-end system.

Butterfly Plus Scheme is a shared memory, multiprocessor Lisp based on an extended version of the C Scheme dialect of the Lisp language. The simplicity and power of Scheme make it particularly suitable as a testbed for exploring parallel symbolic computation. Butterfly Plus Scheme supports the evaluation of multiple Lisp expressions simultaneously in the context of a single, uniformly mapped, global Lisp heap (the space in which all Lisp objects reside).

Rather than implement a loosely coupled set of separate Lisps that communicate via some external message passing protocol, we have chosen to capitalize on the Butterfly Plus parallel processor's shared memory architecture by providing a single Lisp heap mapped identically by all processors. Data structures of arbitrary complexity are easily communicated from one context to another by simply transmitting a pointer rather than by copying the whole data structure. This architecture allows any such Lisp object to be shared and communicated among any or all Lisp computations (concurrent or otherwise).

Butterfly Plus Scheme uses the **future** mechanism as its basic task building construct. This construct arranges for the evaluation of an argument expression and returns immediately to the caller, supplying as its result a place holder for the ultimate value of the expression. For example,

```
(future <lisp-expression>)
```

records the fact that a request has been made for the evaluation of the expression <lisp-expression>, and causes the system to commit resources to that evaluation when they become available. Control returns immediately to the caller, returning a new type of Lisp object called an "undetermined future." The "undetermined future" object serves as a place holder for the value that the evaluation of <lisp-expression> will ultimately produce. That place holder can be manipulated in standard ways. For example, it can be stored as the value of a symbol, made a member of a list, used as a procedural argument, etc. As long as such manipulations do not require the value of the **future** expression before it is available, the process(es) performing the manipulations proceed without interruption. Should an attempt be made to reference the value prior to its determination, the referencing process is automatically suspended until the value arrives. The **future** construct is thus a good abstraction for the creation of value producing computations and for synchronizing the consumers with the producers of values.

In addition to **future**, constructs are available for mutual exclusion, task distribution, user defined scheduling policies, and task cancellation (permitting the use of "speculative" computations, whose results may not be useful, depending on the future evolution of the entire algorithm).

The garbage collector is of the parallel stop-and-copy variety, with pure, constant, and collectible spaces. This enables efficient garbage collection and caching of code and constant data. The garbage collector distributes noncachable data items as uniformly as possible, so as to make good use of the Butterfly Plus's memory bandwidth, and avoid memory "hot spots."

Butterfly Plus Scheme User Interface

User interface and program development facilities reside in a front-end Symbolics 3600 series Lisp machine that communicates with the Butterfly Plus using standard Internet protocols. These facilities are a combination of adapted versions of existing Lisp machine tools and features, such as the structure inspector and the menu package, together with original software providing I/O, graphics, and debugging and performance analysis capabilities. The interface lets the user control and communicate with tasks running on the Butterfly Plus, and provides a continuously updated display of overall system status and performance. Special Butterfly Plus Scheme interaction windows are provided, associated with tasks running on the Butterfly Plus. These windows are easily selected, moved, resized, or folded up into task icons. There is also a Butterfly Plus Scheme mode provided for the ZMACS editor, which connects the various evaluation commands (*e.g.*, **evaluate region**) to an evaluation service task running in the Butterfly Plus Scheme system. A version of the Lisp machine based structure inspector is also being adapted for examining task and data structures on the Butterfly Plus.

Each task can create an interaction window on the Lisp machine. The first time an operation is performed on one of the standard input or output streams, a message is sent to the Lisp machine and the associated window is created. Output is directed to this window, and input that is typed while the window is selected can be read by the task. This multiple window approach makes it possible to use standard system utilities like the trace package and the debugger.

A pane at the top of the screen displays the system state. The system state information is collected by a Butterfly Plus process separate from the Butterfly Plus Scheme system, but has shared memory access to important interpreter data structures. The major feature of this pane is a horizontal rectangle broken vertically into slices. Each slice shows the state of a particular processor. If the top half of the slice is black, the processor is running; if it is gray, the processor is garbage collecting; and if it is white, the processor is idle. The bottom half of each slice is a bar graph that shows how much of each processor's portion of the heap is in use. The status pane also shows, in graphic form, the number of tasks awaiting execution. This display makes performance problems like task starvation easy to recognize. It is also possible to display graphs of any of the metering information provided by the Butterfly Plus.

A simple read-eval-print LISP interface is also available to support ASCII terminals.

The RAMFile System

The Butterfly Plus RAMFile system gives programmers a convenient method for using Butterfly Plus memory effectively. The system creates large data objects, called RAMFiles, within Butterfly Plus random access memory. The objects look like I/O files to the programmer, and data stored in the RAMFiles can be manipulated using a set of UNIX-like I/O primitives. The RAMFile thus provides UNIX I/O functionality at memory speeds. In addition, the RAMFile supports a concurrent read, exclusive write model, enabling user processes to exploit parallelism in their I/O operations while maintaining data integrity.

The RAMFile system is implemented as a library of subroutines layered above the Chrysalis operating system. These subroutines let users access and manipulate Butterfly Plus memory easily. Using a set of RAMFile routines very similar to UNIX file I/O subroutines, the programmer can call and specify operations such as open, read, write, seek, and the like. The routines operate on Butterfly Plus random access memory precisely as if Butterfly Plus memory were an I/O device, such as a disk. The files thus accessed can span any number of processor nodes; they need not be limited to a single local or remote node. Any Butterfly Plus program can use the RAMFile system.

RAMFiles are composed of a number of data segments, or sectors, each the same size, that correspond directly to Chrysalis memory objects. These sectors are dynamically allocated by the RAMFile system to accommodate user write requests. The memory used to house the sectors is reclaimed by the system after file truncation or deletion.

There are ten RAMFile routines that can be used as primitives to access memory in the RAMFile system.

RF_access

The **RF_access** primitive looks at a given RAMFile path and checks to see if the specified file is there. In addition, it determines the access characterstics of the file (e.g., whether the file may be written to or if it is read-only).

RF_creat

The **RF-creat** routine creates a new RAMFile by obtaining the necessary memory objects, setting up an object called the RAMFile descriptor, and assigning a specified filename. The **RF-creat** automatically distributes segments of the RAMFile created over as many different processors as it can, thus allowing greater parallelization. In addition to creating a new RAMFile, **RF_creat** can also be used to rewrite an existing RAMFile.

RF_open

The **RF_open** primitive creates a private copy of the global RAMFile descriptor on a local processor node. The **RF_open** routine can be run on any number of processor nodes, thus allowing multiple processes to perform parallel reads and writes on individual sectors of the RAMFile simultaneously. RAMFile system protocols guarantee that competing processes do not access the same segment at the same time, thus ensuring data integrity.

RF_read and RF_write

The RAMFile read and write routines transfer data from and to the specified RAMFiles. Sector index and lock tables locate the appropriate portion of the RAMFile that the operation is to be performed on, and lock the sector for the

duration of the transfer. The system uses the Chrysalis **block_copy** call to perform the read or write transfer.

RF_lseek

The **RF_lseek** primitive adjusts the current position field of the RAMFile descriptor. Current position is stored as a byte offset into the RAMFile. The **RF_read** and **RF_write** primitives use the value in the current position field to locate exactly where data should be transferred into or out of the file.

RF_close

The **RF_close** primitive releases the local process's private copy of the RAMFile descriptor and unmaps the global descriptor object from the local processor node.

RF_stat, RF_fstat

The **RF_stat** and **RF_fstat** routines obtain information for the programmer. **RF_stat** obtains information about the specified RAMFile path, such as owner's ID, file size and the like. **RF_fstat** obtains the same kind of information about an open file referenced by the RAMFile descriptor (such as would be obtained by an **RF_open** call).

RF_truncate, RF_ftruncate

These two primitives let the programmer truncate a file in the RAMFile system to a specified length. Any data in the file beyond the specified length is lost. The only difference between the two routines is that the **RF_ftruncate** call operates only on files that are open for writing at the time of the call.

RF_unlink

The **RF_unlink** call removes the entry for a specified file and file path from the RAMFile directory. All resources associated with the file are reclaimed by the system. If the file is still open in any process when the **RF_unlink** executes, the directory entry is purged, but the actual resource reclamation is delayed until the process closes the file.

Appendix A

PNC Microcode Functions

This appendix contains detailed information about the PNC microcode functions and registers that, although of little interest to the typical Butterfly user, might prove valuable to programmers who need to understand system calls and performance issues at the microcode level. Procedures for using the block transfer facility, event posting, dual queue handling, and other kernel functions all require a program to interact with the processor node controller. They sometimes involve the use of special registers that are not normally accessed by application programs, and they generally rely on special, low level procedures for memory management. Parameter blocks passed to those processor node controller functions that require them are normally part of the process control block (**pnc_pblk**) of the calling process.

BLOCK TRANSFER FACILITY

The microcoded block transfer capability copies a block of data from one location to another, generally for the purpose of moving the data into a processor's local memory. The processor invokes this facility by writing the physical address of a parameter block with the following format into the location **btran** at address FFF70000.

```
struct btrctrl              /* block transfer control block       */
{
 char  *bt_to ;             /* physical address of destination    */
 short unsigned bt_len ;    /* length of transfer (bytes minus 1) */
 char  *bt_from ;           /* physical address of source         */
} ;
```

A microcode block transfer cannot cross a 64-kilobyte physical address boundary. Also, a block cannot be transferred to or from the I/O space (including Multibus memory) or special function (FP) space. During transfers, a bus error will occur if the source processor is disabled or unreachable. On the other hand, a disabled or unreachable destination processor, does not generate an error. The transfer simply never completes. Higher level software is responsible for timing out incomplete transfers.

There can be one, two, or three processors involved in a block transfer. The source and destination processors will be quite busy during the block transfer, since in servicing the switch their processor node controllers will try to use about three quarters of their total memory bandwidth. I/O transfers have priority over block transfers, but processor memory access requests do not, so the processors run much slower than normal when a block transfer is being performed. If the processor that requested the block transfer is not involved in passing the data, it simply continues normally (unless it performs another switch operation that interacts with the block transfer). Block transfer completion can be monitored by testing the memory location **bxfers** (FE00004A), which contains the number of transfers started from the processor node that have not yet completed.

Because block transfers can use so much switch and processor bandwidth, it is important to consider their impact on other switch traffic and system latencies. The most common result of excessive contention in the switch is that some messages are delayed past the timeouts that normally detect broken or missing hardware. These timeouts are currently set to about 10 milliseconds. If they are exceeded, level 7 interrupts and possibly bus errors result. A system utility program, **toset**, can be used to modify timeout length.

POSTING EVENTS

Events are the main process control primitive in the Chrysalis operating system. Events can be posted to notify a process of some interesting happening in the machine. To post an event, the processor writes the physical address of a parameter block into **PNC_post** at address FFF78000. The parameter block has the following format:

```
struct PNC_pb
{
EH          p_handle ;      /* event handle             */
long        p_postdata ;    /* stored into event block */
short bits  p_reply ;       /* binary reply code        */
} ;
```

The event handle designates an event block that can either be in the processor's local memory or on another processor node. Switch errors and event handles specifying an invalid processor node generate bus errors (as do remote read errors).

The binary reply code in the parameter block reports on the result of the posting operation. If the event handle is invalid, the reply code will be either 0100 (hex) to signal a bad event pointer, or 0200 (hex) to indicate a bad sequence number. Otherwise, the reply code is simply the old value of the e_flags byte from the event block; that is, bit 0 is set to a value of one if the event was already posted or bit 1 is set to a value of one if the event was already posted more than once. In addition, bit 2 is set to a value of one if the event cannot be posted more than once. The two flags in bits 0 and 1 reflect the state of the event block before posting the event. If an event has been posted and cannot be posted more than once, any further attempt to post the event will fail, and the reply code will have bit 2 set to a value of one along with either bit 0 or bit 1 set to a value of one, depending on whether the same attempt to post has already failed once. Consequently, reply codes up to and including four are normal, and reply codes greater than four indicate various error conditions.

The event handling mechanism assumes that the event block and process control block have the following layout.

```
struct  EBLOCK                  /* event block                      */
{
struct  EBLOCK  *e_next ;       /* link to next event block         */
  char    e_type  ;             /* block type BT_EVNT = 2           */
  char    e_flags ;             /* flag bits (see above)            */
  char    e_spare0 ;            /* spare byte                       */
  char    e_seqno ;             /* block sequence number            */
  long    e_prot ;              /* protection bits (reserved)       */
  PH      e_owner ;             /* handle of owning process         */
  long    e_data ;              /* data passed with post            */
} ;

struct  PCB                     /* process control block            */
{
struct  PCB  *p_next ;          /* link to next process block       */
  char    p_type ;              /* block type                       */
  char    p_state ;             /* process state flags              */
  char    p_spare0 ;            /* spare byte                       */
  char    p_seqno ;             /* block sequence number            */
  long    p_prot ;              /* protection bits (reserved)       */
  EH      p_erreh ;             /* error handler event handle       */
struct  q_ctrl  *p_que          /* current scheduler queue          */
struct  q_ctrl  p_eventq ;      /* posted event block queue         */
}  ;                            /* other fields not used by post */
```

Table A-1 lists the values and meanings of the bits in **p_state**.

Table A-1
p_state Bits

Mnemonic	Code Value	Meaning
p_frozen	8	Process cannot become runable.
p_run	4	Process is runable.
p_posted	2	Process is posted.
p_onque	1	Process cannot be put on a scheduler queue now.

In addition to these data structures, the post event microcode also depends on **CurQ**, a pointer to the currently active scheduler queue, and on the layout of the array of scheduler queues, to decide whether to invoke the scheduler. The algorithm used to post an event is as follows:

1. Check the block type in the specified event block.
2. Check the sequence number in the specified event block.
3. Test whether the event block is already posted; if so set the **posted-more-than-once** flag, then return the **was-posted** or the **illegal-multiple-post** reply.
4. Set the **event-posted** flag; store user data in the event block.
5. Enqueue the event on the owner's event queue.
6. Set the **process-posted** flag and the **process-on-queue** flag.
7. If the **process-on-queue** flag was set, return the normal reply.
8. Compare the priority of the owner's queue with the priority of **running-process-queue** and set interrupt level 1 if a context switch is needed.
9. Enqueue the process on the proper scheduler queue.
10. Return the normal reply.

For the preceding algorithm to work, both the process and the event (but not the poster of the event) must be on the same processor node. Chrysalis enforces this by not allowing the disowning of events.

DUAL QUEUE FUNCTIONS

Dual queue functions require a parameter block with the following format:

```
struct  PNC_dqpb            /* dual queue handling functions  */
{
  QH        dq_handle ;     /* queue handle                   */
  long      dq_datum ;      /* datum or event handle          */
  char bits dq_code ;       /* reply code, see dRC_...        */
  char bits dq_flgs ;       /* return code flags, see q_flags */
} ;
```

The **dq_handle** is always overwritten in one of three ways: by a copy of the user data, by the original data itself, or by an event handle extracted from the dual queue. If an event handle is returned, the processor can post the event immediately, using the same parameter block.

The dual queue can be anywhere in memory, on any processor node. If the processor number in the handle is invalid, the resulting switch error will generate a bus error. Dual queue functions include enqueue, dequeue, stack, priority-dequeue, fetch (an event), and poll (for data). They are described in Table A-2.

Table A-2
Dual Queue Functions in PNC Microcode

Function Name	Address	Description
dq_poll	FFF7A000	Poll dual queue.
dq_deq	FFF7A020	Dequeue from dual queue.
dq_pdeq	FFF7A060	High-priority dequeue.
dq_fetch	FFF7A080	Fetch an event from a dual queue.
dq_enq	FFF7A0A0	Enqueue on dual queue.
dq_stack	FFF7A0E0	Stack data on dual queue.

It is important to distinguish between the enqueue and dequeue functions as requested by the processor, and the Chrysalis system calls using these functions. It is also important to distinguish between the dequeue function and the queue extract operation. If data is available, the dequeue function invokes the extract operation; otherwise dequeue uses the insert operation. The same is true for the the enqueue operation, which can perform either an insert or an extract.

Dual queues have two sections, a queue header and a ring buffer. The ring buffer can hold either data entries or event handles, or it can be empty. A flag bit in the queue header distinguishes data queues from event queues. A queue can be frozen by its owner (for debugging purposes, etc.) by setting a flag bit. If frozen in this way, all dual queue requests return the "frozen" reply code.

When dual queues are used as locks, an empty queue represents a free lock. To release a lock, the process holding the lock performs an enqueue to store something on the queue (usually its process handle is stored as a debugging aid). Anyone who needs the lock can perform a dequeue to wait for the lock, or else poll to test the lock without waiting. Setting the lock flag bit limits the queue to a maximum of one data entry at a time, and thus prevents inadvertently freeing a lock more than once.

Table A-3 lists the PNC dual queue return codes.

Table A-3
Dual Queue Return Codes

Return Code	Value	Type	Meaning
dRC_FRZN	00	Reply	Queue is frozen.
dRC_BQH	01	Error	Bad queue handle.
dRC_SNE	02	Error	Sequence number error.
dRC_MUNLK	04	Reply	Attempt to perform multiple unlocks (enq).
dRC_FULL	08	Reply	Queue is full (enq/stack/deq).
dRC_EMPTY	10	Reply	Queue is empty (fetch/poll).
dRC_EXTRT	20	Reply	Extracted a queue entry.
dRC_INSRT	40	Reply	Inserted a queue entry.

The queue header layout is:

```
struct DUALQUE
{
struct   DUALQUE  *q_next ; /* link to next queue header      */
  char   q_type ;           /* block type BT_DQH = 6          */
  char   q_flags ;          /* flag bits (see below)          */
  char   q_spare0 ;         /* spare byte                     */
  char   q_seqno ;          /* block sequence number          */
  long   q_prot ;           /* protection bits (reserved)     */
  EH     q_ownevnt ;        /* owner's event handle           */
  short  q_exadd ;          /* extract address                */
  short  q_inadd ;          /* insert address                 */
  short  q_endadd ;         /* end-of-queue address (last+4)  */
  short  q_stadd ;          /* start-of-queue address         */
  char   q_spare1 ;         /* spare byte                     */
  char   q_queueh ;         /* queue address bits 19.16       */
  long   q_lastext ;        /* at time of last extract: element
                               that would have been inserted */
  long   q_exttime ;        /* time of last extract           */
  long   q_instime ;        /* time of last insert            */
} ;
```

Table A-4 lists the values and meanings of the bits in **q_flags**.

Table A-4
q_flags Bits

Mnemonic	Code Value	Meaning
q_frozen	8	Queue is frozen by owner.
q_lock	4	Queue can have only one data entry.
	2	Must be zero!
q_events	1	Queue has events (otherwise has data).

The memory for the ring buffer must fit entirely within a single 64-kilobyte physical memory block. The physical address of the first word is stored in **q_queueh** and **q_stadd**. Copies of **q_stadd** are stored in **q_exadd** and **q_inadd**. The ring buffer length must be an exact multiple of four bytes. **q_stadd** plus the ring length is stored in **q_endadd**. To determine the size of the ring buffer, it is necessary to determine how many objects can be present and how many processes can be waiting in the dual queue. The queue must be large enough to hold the larger of these two values. Each queue entry is four bytes long and space for one entry is unused. As a result, the minimum ring length is eight bytes.

The dual queue algorithm is as follows:

1. Check the block type in the specified queue header.
2. Check the sequence number in the specified queue header.
3. If the dual queue is frozen, return the frozen reply.
4. Decode the function requested and the queue state to decide whether to try to extract an entry or to insert an entry.

The algorithm to extract an entry is:

1. If queue is empty, reverse queue state and try to insert instead.
2. If last entry in ring, reset extract address to start of ring.
3. Save the user data for debugging purposes.
4. Extract an entry.

5. Update the **time-of-last-extract**.
6. Return entry with the **extracted something** reply code.

The algorithm to insert an entry is:

1. If data queue is not empty and this is a lock, return the **multiple unlock** reply code.
2. Test if this is a polling or fetch request; if so, return the **empty** reply code.
3. Test if inserting at the beginning or end of the queue.
4. Update the appropriate ring pointer, unless the queue was full; otherwise return the **queue full** reply code.
5. Insert the user data.
6. Update the **time-of-last-insert**.
7. Return the **inserted something** reply code.

INTERRUPT CONTROL REGISTER

The PNC can issue an MC68020 interrupt request by modifying bits in the 8-bit interrupt control register, INTr. The interrupt control register (INTr) contains independent bits, some of which can be set directly by the PNC. Therefore, non-interruptable methods for setting and clearing these bits are necessary. Table A-5 shows the INTr layout. Table A–6 lists the functions of the interrupt control register.

Table A-5
Interrupt Control Register Bit Assignments

Bit	Name	Active	Description
7	**nliteLED**	low	Red LED on PNC board.
6		high	Stop the switch randomizer clock.
5	**niR5**	low	Level 5 interrupt request.
4	**niR2**	low	Level 2 interrupt request.
3	**niR1**	low	Level 1 interrupt request.
2	(Reserved)		Used by the PNC – do not change.
1	(Reserved)		Used by the PNC – do not change.
0	**niR7**	low	Level 7 interrupt request.

Table A-6
Interrupt Register Functions

Function Name	Address	Description
rdint	FFF74000	Read INTr value (Read-Only).
andint	FFF74000	AND to INTr (Write-Only).
iorint	FFF74020	OR to INTr (Write-Only).

Data written into the variable **andint** at address FFF74000 is bitwise ANDed into the INTr, whereas data written into **iorint** at address FFF74002 is bitwise ORed into the INTr. These functions allow bits to be set or cleared without concern for possible errors from interrupt problems. A processor IOR-to-memory instruction would not work since the PNC might modify the INTr between the read and rewrite cycles. Both **andint** and **iorint** are write-only registers. A program can read the INTr by reading **rdint** at address FFF74000. These registers should be accessed as C shorts (16-bits), even though only the low byte is used.

All eight bits are saved in the low-order byte of an ALU register. An attempt to read the INTr actually references that register instead. The ALU register is also used to restore the real INTr after certain protection-checking functions in the microcode.

The microcode reset code turns on the red LED located on the processor node card. A ROM bootstrap program, the Ultra-Simple Debugger (USD) turns off the light to indicate that the node is in service. If the node goes out of service the light will be turned on to indicate which processor node is having the problem.

Bits 1 and 2 are used exclusively by the microcode; do not change them. Bits 0, 3, 4, and 5 are control interrupts to the processor. If any of these bits are cleared, an interrupt is caused on that interrupt level. Except for level 7 interrupts, the operating system must logically clear (really set) the appropriate interrupt request bit in each interrupt service routine, otherwise the interrupt will immediately recur when the routine terminates. Level 7 requests are automatically cleared by the PNC (or the processor) just as the interrupt routine is being initiated, thus the level 7 handler must be written to be reentrant. Level 7 events occuring while the handler is running will cause nested interrupts; otherwise such events might be lost completely.

Interrupt requests can be lost if a second request occurs before the previous request at the same level has been serviced. You are safe if you do not initiate any action that can result in an interrupt until you have logically cleared the corresponding interrupt bit.

There is a microcode routine that provides an interrupt vector to the processor for all external interrupt conditions. The processor requests a vector for a specific interrupt level. Levels 1, 2, 5, 6, and 7 are assigned in the microcode to coincide with the processor interrupt auto-vector of the corresponding level. Levels 3 and 4 use an interrupt vector provided by an I/O board (if valid, otherwise the auto-vector is used). The full 16-bit interrupt vector supplied to the processor is stored by the microcode in the **intVEC** array in main memory, with one entry per interrupt level (1–7). Currently, this array is of little interest except for I/O interrupts.

ATOMIC CLEAR-THEN-ADD

The PNC supports a 16-bit clear-then-add-to-memory function, which can be used as an atomic add, atomic, and or atomic or function on any 16-bit memory location (which must be 16-bit aligned). The function first saves the old value of the location (to be returned to the caller as **oldvalue**), ands the **mask** value to the location, then adds the **incr** value to the location. All of

these operations are guaranteed to complete before any other memory operations are permitted, and so are atomic.

The clear-then-add function is triggered by writing the physical address of a control block to the location **addmem** (at location FFF76000). The caller can then read the result (previous value of the location) from the **oldvalue** entry of the control block.

```
struct PNC_am {
  short *am_addr;  /* physical address of target location */
  short mask;   /* value to and to target location */
  short incr;   /* value to add to target location */
  short oldvalue;  /* previous value of target location */
  }
```

LOCAL BANK 0 MEMORY ACCESS

The first 64-kilobytes of local memory are accesible through the PNC at locations FFF00000-FFF0FFFF. Since this memory is accessed through the PNC, the access time is much slower than normal direct access. This function is only present to simplify some difficult problems that occur during power on initialization, and should not normally be used otherwise.

OTHER KERNEL FUNCTIONS

This section describes PNC microcode functions that support the operating system kernel, and involve kernel mode access to control blocks or other data located in physical memory in the range 0–64 kilobytes. The manipulation of true virtual addresses is not possible in these functions, and only very limited error checking is provided. To initiate one of these functions, the supervisor provides the 24-bit physical address of a parameter block in local memory via a **movl** instruction. The parameter block can have various formats, depending on the function involved. Sometimes results are returned in the parameter block, and sometimes various data structures are modified. If errors are possible, error codes are returned in the parameter block. The functions run as uninterruptable units, which is their primary reason for existence; their speed advantage, though possibly significant, is secondary. Table A–7 lists the available functions. The particular function to be performed depends on where the address of the parameter block is stored. The locations are write-only.

Table A-7
Microcode Functions

Function Name	Address	Description
PNC_enq	FFF77000	Enqueue block on the end of a queue.
PNC_deq	FFF77004	Dequeue (pop) block from the beginning of a queue.
PNC_push	FFF77008	Push block onto the beginning of a queue.
PNC_rem	FFF7700C	Remove specified block from anywhere on a queue.
PNC_cTx	FFF77010	Clear-then-XOR specified bits in memory word; save old value.
PNC_cTa	FFF77014	Clear-then-add specified bits in memory word; save old value.

The PNC microcode occupies approximately 1,200 words of the available four kilowords of 64-bit micromemory. The functions described here are the lowest level facilities of the processor node. Most of these functions will never be invoked by application programmers, who use higher level operating system functions.

Enqueue, Dequeue, Push, and Remove

In microcode function control blocks, some parameters are returned by the microcode and others are supplied by the caller. The queuing functions expect queued items to have the address of the next element on the queue (or 0 if this is the last element) at byte offset 2, and arbitrary data at offset 4. The microcode uses only the word at offset 2. The enqueue and push functions set and maintain the contents of that word; dequeue and remove use it, but do not modify it. The data in the word at byte offset 0 is established and used only by the processor.

The queue header has the address of the first element on the queue (or 0 if the queue is empty) at byte offset 2 and the address (0–FFFF) of the last element on the queue (or arbitrary data if the queue is empty) at offset 6. The parameter block for the enqueue/push functions has the address of the queue header at byte offset 2 and the address of the element to add at offset 6. The

parameter block for the dequeue(pop)/remove functions has the address of the queue header at byte offset 4 and returns the address of the element that was removed from the queue (or 0 if the queue was empty) at byte offset 6. (For remove functions, this address is supplied by the processor, and is cleared if the element is not found.) No validity checking is attempted for these parameters.

Clear-Then-XOR and Clear-Then-Add

The PNC clear-then-XOR (**PNC_cTx**) and clear-then-add (**PNC_cTa**) functions are used to modify a word in memory and read back the old value at the same time. **PNC_cTx** and **PNC_cTa** operate on the first 64-kilobytes of local F8 memory only. Use of these functions avoids problems in handling interrupts. The old word is read from memory and stored in the parameter block. Bits specified in the parameter block mask word are then cleared from the value of the old word, the parameter block data word is XORed (or added) to the result, and the word is stored back into memory. Parameter blocks for either of these functions contain the address of the word to modify at byte offset 2, a word containing the mask of bits to clear at offset 4, and a word containing a mask of bits to XOR or add to memory at offset 6. Both functions return the old value of the specified word at offset 8. The functions can be used to exchange bits, bytes, or entire 16-bit words. They can also be used to XOR, AND, IOR, add to memory, etc.

The misc Register

The **misc** register in the PNC has three sections, the local node match register, a switch path enable selection section, and a switch message header checksum section. Bits 15–8 contain the node number used to match "local" addresses, bits 7–4 are used for path enable selection, and bits 3–0 for the header checksum. The processor can only set this register as a whole – it cannot set one field that is independent of the others. The operating system therefore keeps a copy for use in updating the fields separately. The **misc** register is available at address FFF72000.

Interprocessor Interrupts and Resets

To request a remote processor level 7 interrupt, store the processor number, as a byte, into **rrint7** (address FFF71000). This operation is not really useful as

a high bandwidth signaling mechanism, since the remote processor's supervisor pushdown list could overflow if too many level 7 interrupts arrived at once. To request a remote processor level 5 interrupt, store the processor number, as a byte, into **rrint5** (address FFF71004). To request a remote processor level 2 interrupt, store the processor number, as a byte, into **rrint2** (address FFF71006).

To request a remote processor reset, store the processor number, as a byte, into **rreset** (address FFF71002). This method of reset, however, should be used only in emergencies, as it will completely invalidate the state of the remote processor and its PNC in the course of restarting the remote operating system. Although main memory is preserved across the reset, the operating system re-initializes parts of memory during the restart; therefore, important debugging information can be lost. The reset always succeeds (unless there is a hardware malfunction). The identity of the processor that last reset a given processor can be read, as a byte, from **rrpnn** in the processor that was reset (address FFF71000).

PNC Status Register

The PNC Status Register includes various bits that are useful for diagnosis of bus errors, resets, switch errors, etc. All of the bits except **PNC_pwr_low** are cleared automatically when the PNC status register is read, so they must be processed at once. Table A-8 lists the PNC Status Register bit assignments. The register is named PNCsRC, and is at location FFF75000.

When power starts to fail, the processor gets a level 7 interrupt; the handler must check **PNC_pwr_int** and take appropriate action. A second bit, **PNC_pwr_low**, is also set and is not cleared; it prevents multiple level 7 interrupts. When the processor has done what it can about the situation, it should halt (the way to halt is to make the supervisor push-down list inaccessible, then cause a bus error). The microcode will set **PNC_halt_reset**, clear all the other bits, put memory in auto-refresh mode, and start to reset. Reset cannot succeed until power is normal again. If power goes all the way down and comes back later, the power supply will initiate a new-power type reset, which will clear **PNC_halt_reset**.

Table A-8
PNC Status Register (PNCsRC at FFF75000) Layout

Name	Hex Value	Description
PNC_pwr_low	8000	PNC power is low.
PNC_pwr_int	4000	PNC power just went low.
		Generates a level 7 interrupt request.
	2000	Unassigned.
PNC_rmPerr_int	1000	Remote memory read reply had an error.
		Generates a level 7 interrupt request.
PNC_rep_long	0800	Waited too long to receive a reply.
		Generates a bus error.
PNC_req_long	0400	Waited too long to send a request.
		Generates a bus error.
PNC_unex_reply	0200	Receiver got an unexpected reply message.
PNC_ack_hce	0100	Error detected in header checksum.
PNC_ack_err	0080	Error detected in acknowledgement message.
PNC_req_err	0040	Error detected in request message.
PNC_vlm_err	0020	Error detected in variable length message.
	0010	Unassigned.
PNC_unimpl_fnc	0008	Unimplemented **misc** function.
PNC_unimpl_int	0004	Unimplemented processor request.
		Generates a bus error.
PNC_halt_reset	0002	Processor halted, initiating a reset.
PNC_unspec_int	0001	Unspecified microinterrupt occurred.
		Generates a bus error.

PROCESSOR NODE NUMBER

The PNC maintains the processor node number in two different registers. One copy of this number is maintained in the upper byte of the **misc** register (*q.v.*), which controls which memory accesses are considered to be "local". The other copy is kept in the switch transmit interface, and is the number used in sending switch messages. Normally, these two registers must contain the same value. The switch interface node number is an 8-bit register called **pnn**.

It may be read or written at location FFF73000. Correct setting of **pnn** and **misc** is normally the responsibility of the power on initialization code in the boot EPROM. Users should avoid changing these values. (Note that readers of **pnn** should be aware that the C programming language often sign-extends 8-bit variables, although node numbers are normally considered to be unsigned values.)

PNC WRITABLE CONTROL STORE CONTROL

The PNC microcode control store is downloadable from the MC68020. The initial microcode is loaded into the PNC from the boot EPROM by the power up initialization code. The operating system may load its own custom version of the microcode during system boot. The PNC must be disabled while microcode is being loaded (this is its state after a reset). To load the PNC microcode, the following steps must be followed:

1. Reference PNC0dis to stop the PNC.
2. Reference PNC0dis to set the load address to 0.
3. Write the high half (32 bits) of the microcode word to WCShigh.
4. Write the low half (32 bits) of the microcode word to WCSlow.
5. Reference WCSincr to advance to the next microcode address.
6. Repeat steps 3-5 until all the microcode has been loaded.
7. Reference PNC0dis to reset the microcode address to 0 (start location).
8. Reference PNCenb to enable the PNC, which now starts executing microcode from location 0.

Table A–9 lists the PNC writable control store registers.

Table A-9
PNC Writable Control Store Control Registers

Name	Address	Function
PNC0dis	FFF88000	Disables PNC, sets uaddr=0.
WCSincr	FFF88004	Increments uaddr for WCS loading.
PNCenb	FFF88008	Enables PNC (at whatever uaddr was last set).
WCSlow	FFF88010	Write-only low 32-bits of current uaddr.
WCShigh	FFF88014	Write-only high 32-bits of current uaddr.

REAL-TIME CLOCK

The PNC maintains a 32-bit elapsed-time clock on every processor node. This clock has a resolution of 62.5 microseconds, and wraps around approximately every three days. This clock may be read or written (as a 32-bit quantity only) as **rtc**, at location FFF7B000. (Note that setting the real-time clock reliably can be difficult. Contact Butterfly Software Support if you need to know how to do this correctly. This is normally performed by operating system initialization.)

INTERVAL TIMER

The PNC maintains a 16-bit interval timer (related to **rtc**), which is available for use by the operating system. Writing to two locations controls the timer: writing a 16-bit value to **sItime** (at location FFF7C000) sets the interval timer to the value, if less than the current remaining time, or does nothing if longer. The location **iItime** (at FFF7C0002) behaves similarly, but will only increase the timer value. When the timer expires, the PNC requests a level-2 interrupt to the MC68020, and resets the timeout value to its maximum (about 4 seconds). The timer value is maintained in 62.5 microsecond ticks (as is maintained by **rtc**).

DIAGNOSTIC UART

The processor node contains a 2681 DUART and RS-232 drivers to communicate with a diagnostic console. The address of the DUART is FFF80000, with the DUART's 8-bit wide registers being accessed as the upper byte of

successive 16-bit locations. See the Signetics SCN2681 data sheet for programming information. This data sheet may be found in the *Signetics MOS Microprocessor Data Manual 1983*.

LOCAL MEMORY CONTROL REGISTERS

Four local memory control registers are located on each processor node. While not controlled by the PNC, these registers are used to control the onboard memory. For historical reasons, they are given as offsets from a base location (mCr, at FFF84000). Except for MCR_PAR_CLR, the value read or written to the location is irrelevant; only the reference to the address is important. The sign bit of MCR_PAR_CLR is set if the memory system has had a parity error since the last MCR_PAR_CLR access. Referencing MCR_PAR_CLR resets the parity error latch. The memory system is automatically disabled and placed into auto-refresh mode by a node reset. The boot EPROM enables the memory system during power-up initialization. Table A–10 lists these control registers.

Appendix B

Physical Memory Map

Physical addressses on the processor node card are decoded accorded to the following table. Note that the signal PA32 is true when bits <31...24> of the logical address all ones.

Device	Physcial Address					Comments
	32	31 ... 24	23 ... 16	15 ... 0	fc<2 ... 0>	
Address Trap	x	x	x		3	
Interrupt Acknowledge	x	x	x		7	
EPROM	0	x	0		6 (supervisor program)	Boot Vector
	0	xxxxxxxx	00000000	00xxxxxx	xxxxxxxx	
Local RAM	0	pn	x		!3 & !7	
Remote RAM	0	!pn	x		!3 & !7	
EPROM	1	x	FC-FD		!3 & !7	PROM Execution
	1	xxxxxxxx	111111xx	xxxxxxxx	xxxxxxxx	
UART	1	x	F8	0xxx	!3 & !7	
	1	xxxxxxxx	11111000	00xxxxxx	xxxrrrrx	Registers decoded in chip
Memory Control	1	x	F8	1xxx	!3 & !7	
Refresh On	1	xxxxxxxx	11111000	01xxxxxx	00xxxxxx	
Read Error Register	1	xxxxxxxx	11111000	01xxxxxx	01xxxxxx	
Refresh Off	1	xxxxxxxx	11111000	01xxxxxx	10xxxxxx	
Control Store	1	x	F8	2xxx	!3 & !7	
Zero-Disable	1	xxxxxxxx	111110xx	10xxxxxx	xxx000xx	
Increment	1	xxxxxxxx	111110xx	10xxxxxx	xxx001xx	
Enable	1	xxxxxxxx	111110xx	10xxxxxx	xxx010xx	
Write Low	1	xxxxxxxx	111110xx	10xxxxxx	xxx100xx	
Write High	1	xxxxxxxx	111110xx	10xxxxxx	xxx101xx	
Read Low	1	xxxxxxxx	111110xx	10xxxxxx	xxx110xx	
Read High	1	xxxxxxxx	111110xx	10xxxxxx	xxx111xx	
PNC Functions	1	x	F7		!3 & !7	d = dispatch address
	1	xxxxxxxx	xxxx0111	ddddxxxx	xxxxxxxx	
BIO Functions	1	x	F6		!3 & !7	
	1	xxxxxxxx	xxxx0110	xxxxxxxx	xxxxxxxx	
Multibus Memory Unswapped	1	Multibus bits 23 - 16	E5		!3 & !7	
Multibus Memory Swapped	1	Multibus bits 23 - 16	F5		!3 & !7	
	1	aaaaaaaa	xxxs0101	aaaaaaaa	aaaaaaaa	a = Multibus address s = swap
Multibus I/O Unswapped	1	x	E4		!3 & !7	
Multibus I/O Swapped	1	x	F4		!3 & !7	
	1	xxxxxxxx	xxxs0100	aaaaaaaa	aaaaaaaa	a = Multibus address s = swap
Local Memory Bank 0	1	x	F0		!3 & !7	

! indicates not used.

Bibliography

1. *Bipolar Microprocessor Logic and Interface Data Book*, Advanced Micro Devices, Inc. (901 Thompson Place, P.O. Box 453, Sunnyvale, Calif. 94086, 1981).

2. *Butterfly Parallel Processor Chrysalis Programmer's Manual, Version 4.0*, BBN Advanced Computers Inc. (Cambridge, Mass. 02238, 1987).

3. *EXOS 201 Ethernet Front-End Processor for Multibus Systems Reference Manual, Revision B*, Excelan, Inc. (2180 Fortune Drive, San Jose, California, 95131, 1985). Publication Number: 42000006-00.

4. Harbison, Samuel P. and Steele, Guy L., Jr. *C: a Reference Manual*, Prentice-Hall, Inc. (Englewood Cliffs, N.J., 07362, 1987).

5. *IEEE P1014/Draft 1.2 and IEC 821 Bus Standards*, The Institute of Electrical and Electronics Engineers, Inc. (345 East 47th Street, New York, NY, 10017, 1983).

6. *IEEE Standard Microcomputer System Bus (IEEE Std 796-1983)*, The Institute of Electrical and Electronics Engineers, Inc. (345 East 47th Street, New York, NY, 10017, 1983).

7. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc. (Englewood Cliffs, N.J. 07362, 1978).

8. *MC68020 32-Bit Microprocessor User's Manual, Second Edition*, Prentice-Hall, Inc. (Englewood Cliffs, N.J. 07632., 1985, 1984).

9. *MC68851 Paged Memory Management Unit User's Manual, First Edition*, Prentice-Hall, Inc. (Englewood Cliffs, N.J. 07362, 1986).

10. *MC68881 Floating-Point Coprocessor User's Manual, First Edition*, Motorola (1985).

11. *MM-9000D/1M, Revision E Users Guide, October 1985*, Micro Memory Inc. (9540 Vassar Ave., Chatsworth, California, 91311, 1985).

12. *MOS Microprocessor Data Manual 1983*, Signetics Corporation (811 East Arques Ave., P.O. Box 409, Sunnyvale, Calif. 94086, 1983).

13. *Programming the Butterfly Plus*, BBN Advanced Computers Inc. (Cambridge, Mass. 02238, 1987).

14. Steele, Guy L., Jr. *Common Lisp: The Language*, Digital Press (Bedford, Mass. 01730, 1984).

15. *The VMEbus Specification Manual, Revision C.1, October 1985, Second Printing*, Motorola, (1985).

Index