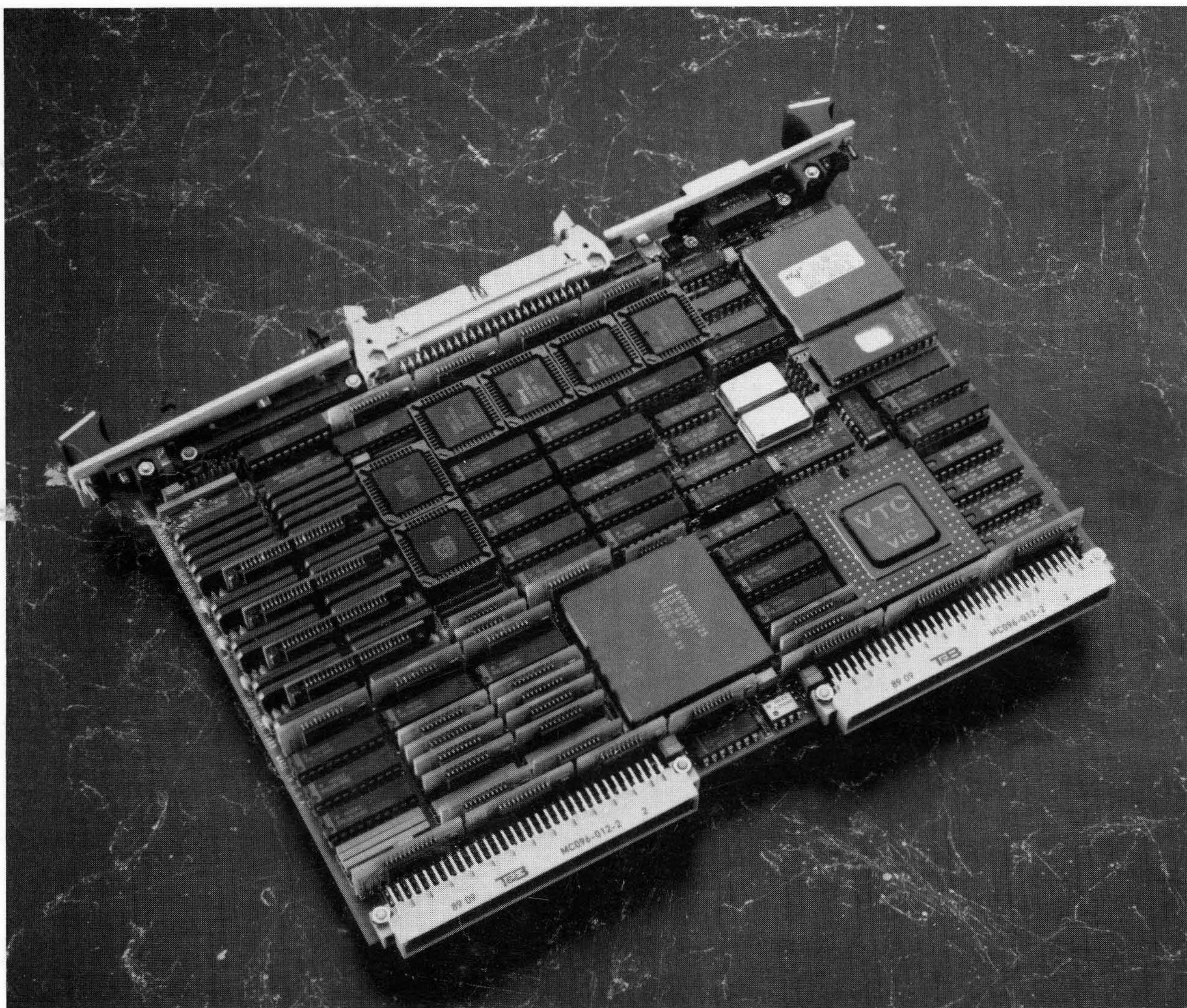
USER'S MANUAL
Revision E
July 1990

HK80/V960E

VMEbus Single-Board Computer



HEURIKON CORP.®
OPEN SYSTEMS::OPEN TOOLS

HK80/V960E

VMEbus Single-Board Computer

USER'S MANUAL
Revision E
July 1990

HEURIKON CORP.®
OPEN SYSTEMS::OPEN TOOLS

The information in this manual has been checked and is believed to be accurate and reliable. HOWEVER, NO RESPONSIBILITY IS ASSUMED BY HEURIKON FOR ITS USE OR FOR ANY INACCURACIES. Specifications are subject to change without notice. HEURIKON DOES NOT ASSUME ANY LIABILITY ARISING OUT OF USE OR OTHER APPLICATION OF ANY PRODUCT, CIRCUIT OR PROGRAM DESCRIBED HEREIN. This document does not convey any license under Heurikon's patents or the rights of others.

Heurikon is a registered trademark of Heurikon Corporation. Intel is a trademark of Intel Corporation. Ethernet is a trademark of Xerox Corporation. UNIX is a registered trademark of AT&T. VxWorks is a trademark of Wind River Systems, Inc.

REVISION HISTORY

Revision Level	Principal Changes	Date of Publication	Board Revision	80960CA Level
A (Preliminary)	First publication	November 1989	EP1	
B (Preliminary)	Expanded text and added illustrations	December 1989	EP1, EP2	
C (Preliminary)	Added power requirements and clarified text.	January 1990	EP1, EP2	
D	Extensive revision for release to production.	June 1990	P	A4
E	Added Appendix C.	July 1990	P	B1

Copyright 1990 Heurikon Corporation. All rights reserved. Portions copyright Intel Corporation 1989. Used with permission.

Table of Contents

1 — Overview

1.1 Introduction 1-1
1.2 Features 1-1
1.3 Block Diagram 1-3
1.4 Component Map 1-4
1.5 Bus Summary 1-5
1.6 Jumpers, Connectors, and Switches 1-6
1.6.1 Jumpers 1-6
1.6.2 Connectors 1-7
1.6.3 Reset 1-7
1.7 Overview of the Manual 1-7
1.7.1 Terminology and Notation 1-7
1.7.2 Additional Technical Information 1-7

2 — Getting Started

2.1 Equipment 2-1
2.2 Preliminary Considerations 2-2
2.2.1 Electrical 2-2
2.2.2 Physical 2-2
2.2.3 Environmental 2-2
2.3 Installation and Power-up 2-2
2.4 Troubleshooting and Service Information 2-3
2.5 Monitor Summary 2-5

3 — MPU Summary Information

3.1 Introduction 3-1
3.2 MPU Initialization 3-1
3.2.1 Initialization Boot Record (IBR) 3-2
3.2.2 Process Control Block (PRCB) 3-2
3.3 Byte Ordering 3-4
3.4 MPU Interrupts 3-5
3.4.1 Interrupt Structures 3-5
3.4.1.1 The Interrupt Table 3-5
3.4.1.2 The Interrupt Stack Frame 3-6
3.4.2 The Nonmaskable Interrupt (NMI) 3-6
3.4.3 Hardware Interrupts 3-7

3.4.3.1 Interrupt Priority 3-10
3.4.3.2 Interrupt Mask Register (IMSK) 3-11
3.4.3.3 Interrupt-Pending Register (IPND) 3-11
3.4.3.4 Interrupt Mapping Registers (IMAP0-IMAP2) 3-11
3.4.3.5 Interrupt Control Register (ICON) 3-13
3.4.4 Software Interrupts 3-13
3.5 MPU Faults 3-14
3.5.1 The Fault Table 3-14
3.5.2 The Fault Stack Frame 3-15
3.6 MPU DMA Support 3-16
3.6.1 HK80/V960E Implementation 3-17
3.6.2 Registers/Instructions 3-18
3.6.2.1 DMA Command Register (DMAC) 3-18
3.6.2.2 The Set-up-DMA (sdma) Instruction 3-18
3.6.2.3 Update DMA-Channel RAM Instruction (udma) 3-18
3.6.3 DMA Interrupts 3-19
3.6.4 DMA Data Alignment 3-19
3.7 MPU Trace Events 3-20
3.8 MPU Caches 3-21
3.8.1 Data RAM Cache 3-21
3.8.2 Instruction Cache 3-21
3.8.3 Register Cache 3-22
3.9 MPU Processing Modes 3-22
3.10 MPU Register Summary 3-22

4 — System Error Handling
4.1 Introduction 4-1
4.2 Error Conditions 4-1
4.2.1 Hardware Errors 4-1
4.2.2 Software Errors 4-2

5 — On-card Memory Configuration
5.1 Introduction 5-1
5.2 ROM 5-1
5.3 On-card RAM 5-3
5.4 Bus Memory 5-3
5.5 Physical Memory Map 5-4
5.6 Memory Timing 5-5
5.7 Nonvolatile RAM 5-6

6 — VMEbus Control
6.1 Introduction 6-1
6.2 VMEbus Signal Descriptions 6-2
6.3 VIC Register Map 6-5

6.4 VMEbus Interrupts 6-8
6.4.1 Interrupter Operation 6-8
6.4.2 Interrupt Handler Operation 6-9
6.4.2.1 VIC Interrupt Requests 6-9
6.4.2.2 VIC Interrupt Acknowledges 6-10
6.5 Mailbox Interface 6-13
6.6 VMEbus System Controller 6-15
6.7 VMEbus Master Interface 6-16
6.8 VMEbus Slave Interface 6-17
6.8.1 Extended Space 6-17
6.8.2 Standard Space 6-18
6.8.3 Short Space 6-20
6.9 SYSFAIL Control 6-20
6.10 VMEbus and Local Bus Watchdog Timers 6-20
6.11 VMEbus Interface 6-21
6.11.1 VMEbus Pin Assignments, P1 6-21
6.11.2 VMEbus and VSB Pin Assignments, P2 6-22

7 — VME Subsystem Bus (VSB) Control
7.1 Introduction 7-1
7.2 VME Subsystem Bus (VSB) Signal Descriptions 7-1
7.3 VSB Operation 7-3
7.4 VSB Termination 7-4
7.5 VMEbus and VSB Pin Assignments, P2 7-5

8 — User LEDs and Front Panel Interface
8.1 User LEDs 8-1
8.2 Front Panel Interface (FPI), J2 8-2

9 — CIO Usage
9.1 Introduction 9-1
9.2 Port C Bit Definition 9-1
9.3 Port B Bit Definition 9-2
9.4 Port A Bit Definition 9-3
9.5 Counter/Timers 9-4
9.6 Register Address Summary (CIO) 9-5
9.7 CIO Initialization 9-5

10 — Serial I/O
10.1 Introduction 10-1
10.2 RS-232 Pin Assignments, P5 10-1
10.3 Signal Naming Conventions (RS-232) 10-4
10.4 Connector Conventions 10-6
10.5 SCC Initialization Sequence 10-6
10.6 Port Address Summary 10-7
10.7 Serial DMA 10-7

10.8 Baud Rate Constants 10-8
10.9 RS-422 Operation 10-9
10.10 Relevant Jumpers (Serial I/O) 10-9
10.11 Serial I/O Cable Drawing 10-10

11 — Ethernet Interface
11.1 Introduction 11-1
11.2 Components 11-1
11.2.1 Network Interface Controller 11-1
11.2.2 Serial Network Interface 11-2
11.3 Ethernet Access 11-2
11.3.1 Arbiter Enable 11-3
11.3.2 Port Access 11-3
11.3.3 Channel Attention (CA) 11-4
11.3.4 Ethernet Byte Ordering 11-4
11.4 Ethernet Port Pin Assignments, P6 11-5
11.5 Transceiver Configuration 11-6

12 — SCSI Port
12.1 Introduction 12-1
12.2 SCSI DMA 12-1
12.3 Register Address Summary (SCSI) 12-2
12.4 SCSI Reset 12-2
12.5 SCSI Port Pin Assignments, P4 12-3
12.6 SCSI Termination 12-4

13 — Centronics Port
13.1 Introduction 13-1
13.2 Centronics Port Pin Assignments, P3 13-1
13.3 Centronics Control Port Address 13-3
13.4 Centronics Printer Interface Cable 13-5

14 — Real-Time Clock (RTC)
14.1 Introduction 14-1
14.2 RTC Implementation 14-1

15 — Summary Information
15.1 Software Initialization Summary 15-1
15.2 On-Card I/O Addresses 15-1
15.3 Hardware Configuration Jumpers 15-3
15.4 Power Requirements 15-5
15.5 Environmental Requirements 15-5
15.6 Mechanical Specifications 15-5

Appendix A — Code Examples

Appendix B — NV-RAM Information

Appendix C — 80960CA and 82596CA Implementation Notes and Errata

C.1 80960CA C-1

C.1.1 80960CA Step A4 C-2

C.1.1.1 Type A Errata — Features That Are Not Implemented C-2

C.1.1.2 Type B Errata — Implemented Features That Do Not Function As Desired C-3

C.1.2 80960CA Step B1 C-14

C.1.2.1 Type A Errata — Anomalies That Have Serious Consequences C-14

C.1.2.2 Type B Errata — Anomalies That Have Performance/Specification Implications C-14

C.1.2.3 Type C Errata — Anomalies That Have Definitional Implications C-15

C.2 82596CA C-16

C.2.1 Erratum 1 — FIFO Operation Failure Region C-16

C.2.2 Erratum 2 — Truncated Frame on Transmit C-16

C.2.3 Erratum 3 — Receive Unit (RU) Start When RU Active C-17

C.2.4 Erratum 4 — Command Unit (CU) Abort when CU Suspended C-18

C.2.5 Erratum 5 — Revision of SCP Bit Values C-19

Figures

Figure 1-1 HK80/V960E Block Diagram 1-3

Figure 1-2 HK80/V960E Component Map 1-4

Figure 1-3 Jumpers, Connectors, and Switches 1-6

Figure 3-1 MPU Structures and Control Table 3-3

Figure 3-2 MPU Interrupt Table 3-6

Figure 3-3 HK80/V960E Interrupt Architecture 3-9

Figure 3-4 MPU Fault Table 3-14

Figure 3-5 Fault Table Entries 3-15

Figure 5-1 ROM Capacity and Jumper Positions 5-2

Figure 5-2 ROM Positioning Diagram 5-2

Figure 5-3 Physical Memory Map 5-4

Figure 5-4 EEPROM Partitions 5-8

Figure 6-1	HK80/V960E Interrupt Architecture	6-9
Figure 6-2	VMEbus Mailbox Structure	6-14
Figure 6-3	P1 and P2 VMEbus and VSB Connectors	6-21
Figure 7-1	Location of VSB Terminators	7-5
Figure 7-2	VSB Connector, P2	7-5
Figure 8-1	Location of User LEDs	8-1
Figure 8-2	Location of Front Panel Interface, J2	8-2
Figure 10-1	RS-232 Connector, P5	10-1
Figure 10-2	Serial I/O Cable	10-10
Figure 11-1	Ethernet Connector, P6	11-5
Figure 12-1	SCSI Connector, P4	12-3
Figure 12-2	Location of SCSI Terminators	12-4
Figure 13-1	Centronics Connector, P3	13-1
Figure 13-2	Centronics Interface — Block Diagram	13-4
Figure 13-3	Centronics Printer Interface Cable	13-5
Figure 15-1	HK80/V960E Jumper Locations	15-4
Figure C-1	Erratum B-2-k Workaround	C-6
Figure C-2	Erratum 3 System Control Block Status and Control Words	C-17
Figure C-3	Erratum 4 System Control Block Status and Control Words	C-18
Figure C-4	Erratum 5 SCP Values	C-19

Tables

Table 1-1	HK80/V960E Components	1-5
Table 2-1	Power Requirements	2-2
Table 2-2	Summary of Editing Commands for the Monitor Program	2-6
Table 3-1	Little-endian and Big-endian Byte Ordering	3-4
Table 3-2	HK80/V960E Error Status Latch Encoding	3-7
Table 3-3	External Interrupt Pin Mappings	3-8
Table 3-4	Interrupt Mask Register	3-11
Table 3-5	Interrupt-Pending Register	3-11
Table 3-6	Interrupt Mapping Registers	3-12
Table 3-7	Interrupt Mapping Register Format	3-12
Table 3-8	PRCB Definition	3-12
Table 3-9	ICON Register Definition	3-13

Table 3-10	80960CA Fault Types and Subtypes	3-16
Table 3-11	80960CA DMA Channels on the HK80/V960E	3-17
Table 4-1	HK80/V960E Error Status Latch Encoding	4-2
Table 5-1	ROMINH Value and ROM Addresses	5-1
Table 5-2	HK80/V960E Memory Space	5-3
Table 5-3	80960CA Clock Cycles for Zero Wait States	5-5
Table 5-4	RAM Access Time Required for the HK80/V960E	5-6
Table 5-5	Nonvolatile RAM Addresses	5-7
Table 6-1	VIC Register Map	6-6
Table 6-2	VIC Interrupt Lines and Associated Acknowledge Addresses	6-10
Table 6-3	Interrupt Priorities	6-12
Table 6-4	Mailbox Enable	6-14
Table 6-5	HK80/V960E "Short" Space Slave Mapping on VMEbus (Mailbox)	6-15
Table 6-6	Bus Control Jumpers	6-16
Table 6-7	Relationship of Physical Addresses to VMEbus and VSB Memory Regions	6-16
Table 6-8	Slave "Extended" Space Enable	6-17
Table 6-9	Slave "Extended" Space Slave Mapping on VMEbus	6-18
Table 6-10	Slave "Standard" Space Enable	6-19
Table 6-11	HK80/V960E "Standard" Space Slave Mapping on VMEbus	6-19
Table 6-12	VMEbus Connector Pin Assignments, P1	6-21
Table 6-13	VMEbus and VSB Connector Pin Assignments, P2	6-23
Table 7-1	VSB Release Modes	7-4
Table 7-2	VSB Arbiter Enable	7-4
Table 7-3	VSB Terminations	7-5
Table 7-4	VMEbus and VSB Connector Pin Assignments, P2	7-6
Table 8-1	User LEDs — Addresses	8-1
Table 8-2	Front Panel Interface Connector Pin Assignments, J2	8-2
Table 8-3	J2 Interrupt and Reset Signals	8-3

Table 9-1	HK80/V960E "Standard" Space Slave Mapping on VMEbus	9-2
Table 9-2	HK80/V960E "Short" Space Slave Mapping on VMEbus	9-3
Table 9-3	Slave "Extended" Space Slave Mapping on VMEbus	9-4
Table 9-4	CIO Register Addresses	9-5
Table 10-1a	Serial Port Pin Assignments, P5 — Port A	10-2
Table 10-1b	Serial Port Pin Assignments, P5 — Port B	10-2
Table 10-1c	Serial Port Pin Assignments, P5 — Port C	10-3
Table 10-1d	Serial Port Pin Assignments, P5 — Port D	10-3
Table 10-2	RS-232 Signal Naming Conventions	10-5
Table 10-3	RS-232 Reversal Cable	10-5
Table 10-4	SCC Initialization Sequence	10-7
Table 10-5	SCC Register Addresses	10-7
Table 10-6	Baud Rate Constants	10-8
Table 10-7	Relevant Jumpers — Serial I/O	10-9
Table 11-1	Ethernet Accesses	11-2
Table 11-2	82596CA Port Accesses	11-3
Table 11-3	82596CA Port Access Definition	11-4
Table 11-4	Ethernet Byte Ordering	11-5
Table 11-5	Ethernet Connector Pin Assignments, P6	11-5
Table 11-6	Transmit Differential Line Configuration, J11	11-6
Table 12-1	SCSI Register Address Summary	12-2
Table 12-2	SCSI Pin Assignments, P4	12-3
Table 13-1	Centronics Pin Assignments, P3	13-2
Table 13-2	Centronics Control Addresses	13-3
Table 13-3	Centronics Data/Status Addresses	13-3
Table 14-1	RTC Accesses	14-1
Table 15-1	Address Summary	15-1
Table 15-2	Jumper and Terminator Configurations	15-3
Table 15-3	HK80/V960E Power Requirements	15-5
Table 15-4	Mechanical Specifications	15-5
Table B-1	EEPROM Addresses	B-1
Table C-1	Guide to 80960CA versions	C-1
Table C-2	Erratum B-7-n — Current Device Operation	C-12
Table C-3	Errant and Correct Fault Types and Subtypes	C-13

1

Overview

1.1 INTRODUCTION

The HK80/V960E is a 32-bit single-board computer based on the Intel 80960CA microprocessor and the Intel 82596CA Ethernet coprocessor. The HK80/V960E also has four RS-232 serial ports, a SCSI port, a Centronics port, mailbox interrupt support, a real-time clock, and VMEbus/VSB compatibility.

1.2 FEATURES

MPU

The microprocessor is an Intel SuperScalar 80960CA RISC chip operating at 25 MHz or 33 MHz. The 80960CA has a 32-bit internal architecture with 32-bit address and data paths, a 4-Gbyte addressing range, 1 Kbyte of static data RAM, a 1-Kbyte instruction cache, and a programmable register cache. The 80960CA also has a 4-channel, 32-bit DMA controller and high-speed interrupt controller.

Ethernet

The Ethernet interface consists of an Intel 82596CA 32-bit LAN coprocessor for CSMA/CD MAC, 10BASE5 IEEE-802.3 communications. The coprocessor has transmit and receive FIFOs and on-chip DMA with 116 Mbyte/sec bus bandwidth. The coprocessor provides network management and self-test diagnostics.

RAM

The HK80/V960E has 2- or 8-Mbyte RAM capacity and one parity bit per byte (optional). RAM uses 256K x 4 or 1024K x 4 DRAMs. The HK80/V960E uses hardware logic for refresh.

EPROM

The HK80/V960E has one ROM socket with a 1-Mbyte capacity.

NV-RAM

The HK80/V960E has nonvolatile static RAM in an 8K x 8 configuration for user-definable and system parameters. The internal EEPROM has 100-year retention and 10,000 store cycle lifetime.

VMEbus The HK80/V960E uses the VTC VIC068 intelligent VMEbus controller/arbiter for the VMEbus, which uses a 32-bit address bus with 24- or 32-bit address modes (4-Gbyte range) and a 32-bit data bus with 8-, 16-, or 32-bit board compatibility. There are seven bus interrupts.

VSB The VME subsystem bus provides high-speed local memory expansion. The VSB supports secondary bus masters.

Serial I/O The HK80/V960E has four serial I/O ports via two Z85C30 SCCs. There are separate baud rate generators for each port and asynchronous and synchronous modes. The RS-232C interface is standard; RS-422 is optional.

SCSI The HK80/V960E uses an ANSI X3T9.2-compatible controller (WD33C93A) for a SCSI interface. The SCSI interface supports up to eight disk drive controllers or other devices, and provides synchronous protocol support.

Centronics There is one 8-bit parallel port for a Centronics type of printer or other device.

LEDs There are four user LEDs under software control and two Ethernet LEDs.

CIO The HK80/V960E uses a Zilog Z8536 counter/timer and parallel I/O unit that has three 16-bit counter/timers. There are three parallel ports for on-card control functions.

Mailbox The mailbox allows remote control of the HK80/V960E via specified VMEbus addresses.

FPI The front panel interface allows remote display of system status.

RTC The HK80/V960E has a real-time clock with battery backup.

1.3 BLOCK DIAGRAM

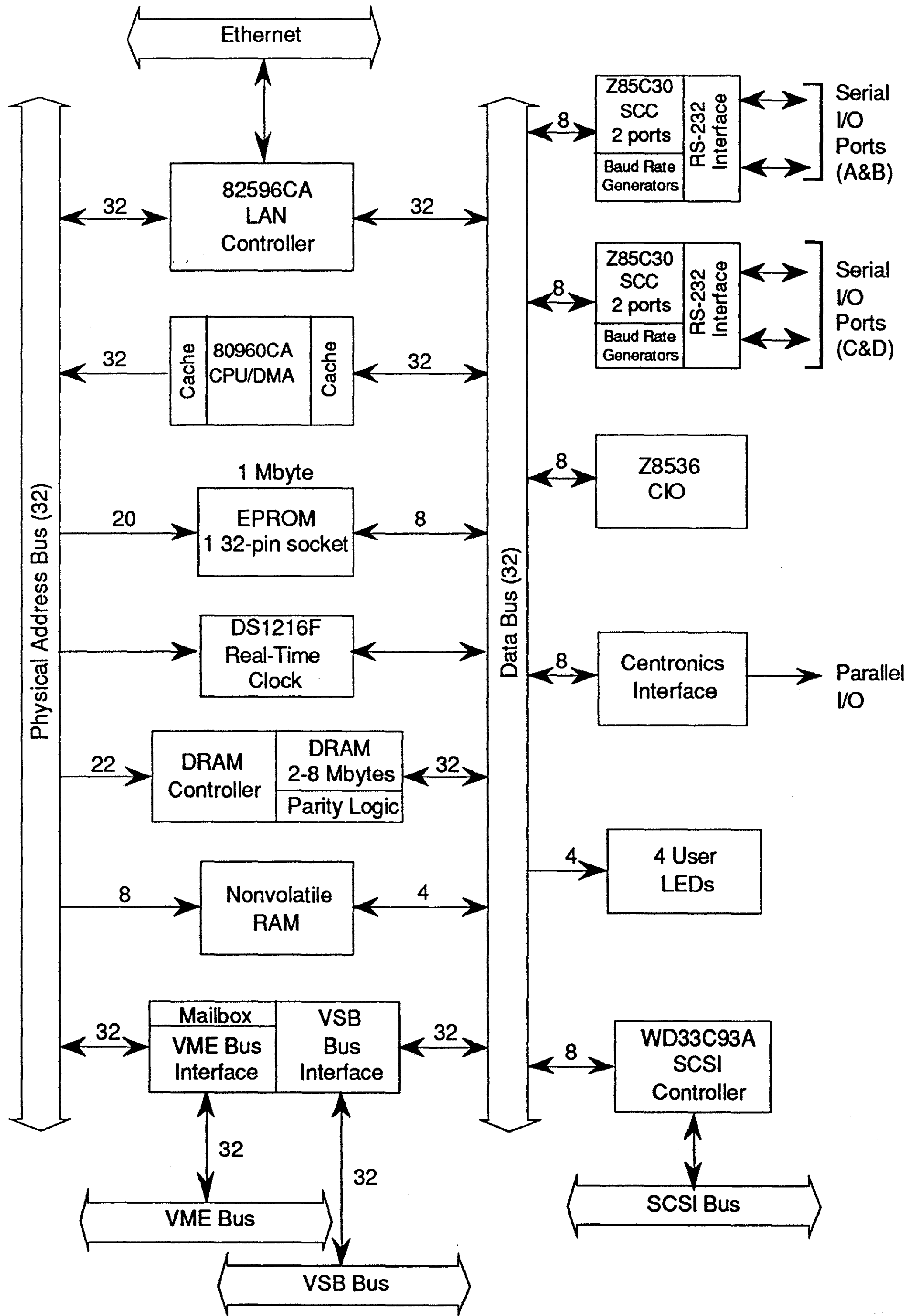


FIGURE 1-1. HK80/V960E block diagram

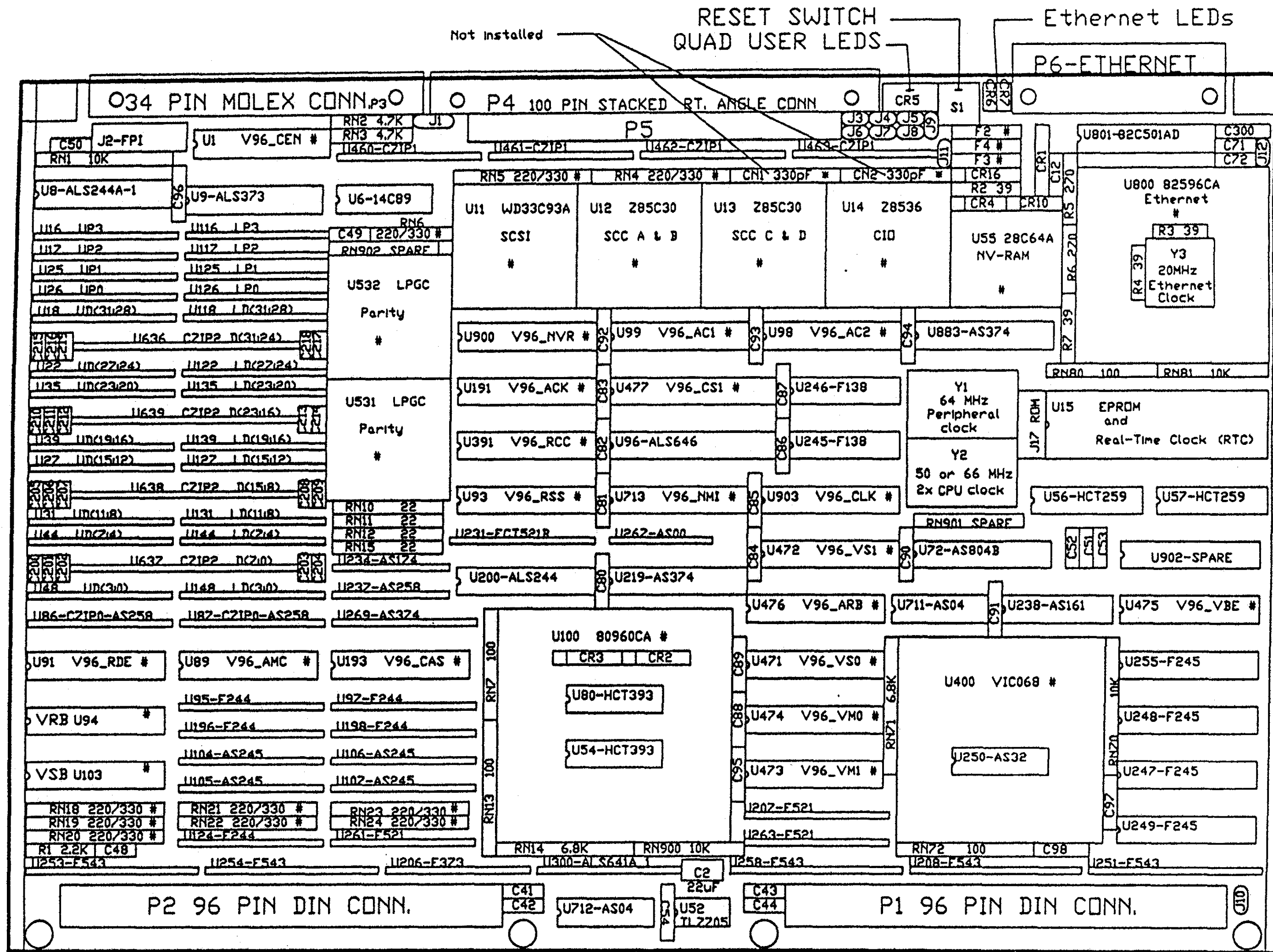


FIGURE 1-2. HK80/V960E component map

TABLE 1-1
HK80/V960E components

Component Number	Function
U100	80960CA
U800	82596CA
U400	VIC068
U16,U17,U18,U22,U25,U26,U27, U31,U35,U39,U44,U48,U116, U117,U118,U122,U125,U126, U127,U131,U135,U139,U144, U148	RAM
U15	EPROM and RTC
U55	EEPROM (NV-RAM)
U12,U13	Serial I/O
CR5	User LEDs
CR6,CR7	Ethernet LEDs
U14	CIO
U11	SCSI
P1	VMEbus
P2	VMEbus and VSB
P3	Centronics
P4	SCSI
P5	Serial I/O
P6	Ethernet
J2	Front Panel Interface
S1	Reset Switch

1.5 BUS SUMMARY

The VMEbus offers high throughput for data transfers between boards or sub-systems on the VMEbus, and is the main conduit for transferring system level information between processor subsystems. The VME subsystem bus (VSB) allows high-speed local communications among a set of VME boards without using the the VMEbus.

1.6 JUMPERS, CONNECTORS, AND SWITCHES

1.6.1 Jumpers

Twelve jumpers are used to configure the HK80/V960E for the following selections:

- Serial port selection and power — eight jumpers
- SCSI bus power — one jumper
- HK80/V960E as system controller — one jumper
- Ethernet transceiver type — one jumper
- ROM size — one set of jumpers

Refer to Figure 15-1 for detailed descriptions of jumpers.

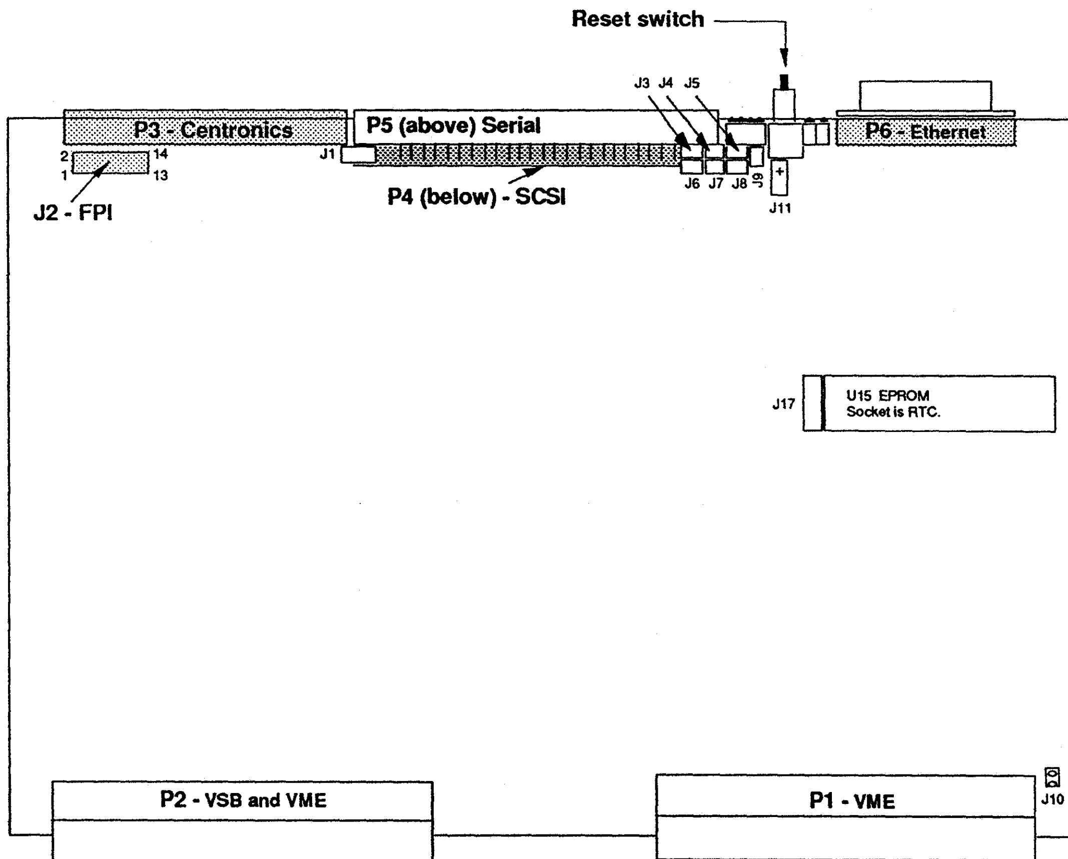


FIGURE 1-3. Jumpers, connectors, and switches.

1.6.2 Connectors

The HK80/V960E has seven ports:

- P1 and P2 — Standard 96-pin VMEbus and VSB connectors
- P3 — 34-pin parallel port connector (Centronics interface)
- P4 — Standard 50-pin SCSI connector
- P5 — 50-pin serial port connector (four RS-232 ports)
- P6 — Standard 15-pin Ethernet port connector
- J2 — 14-pin front panel interface connector

1.6.3 Reset Switch

This switch resets the HK80/V960E and also resets the VMEbus if the HK80/V960E is the VME system controller.

1.7 OVERVIEW OF THE MANUAL

- Chapters 1 and 2 contain introductory material.
- Chapters 3 through 14 describe board components and interfaces.
- Chapter 15 contains summary information, including on-card I/O addresses and a jumper diagram.

1.7.1 Terminology and Notation

Throughout this manual *byte* refers to 8 bits; *short* refers to 16 bits; *word* and *long word* refer to 32 bits; and *quad word* refers to 4 long words (that is, 128 bits).

Hexadecimal numbers are subscripted with a *16* and binary numbers with a *2*.

The word "CAUTION" is used to label procedures that must be taken to prevent damage to the board.

1.7.2 Additional Technical Information

This manual describes Heurikon's implementation of the intelligent components of this board. Further information on basic operation and programming can be found in the following documents:

- For details on the MPU, read the *Intel 80960CA User's Manual,* Intel publication number 270710-001 (Santa Clara: CA: Intel Corporation, 1989).
- For information on the Ethernet interface, read *Intel 82596CA User's Manual* and the *Intel 82C501AD Data Sheet.*
- For details on the VME interface, read the *VIC068 VMEbus Interface Controller Specification* (Bloomington, MN: VTC Incorporated, 1989) and the *VMEbus Specification C.1* (Motorola, 1985).
- For details on the VME Subsystem Bus, read *Parallel Sub-System Bus of the IEC 821 Bus, Revision C* (International Electromechanical Commission, 1986).
- For details on the serial interface, read *EIA Standard RS-232-C* (Washington, DC: Electronic Industries Association, 1969) and *Z8030 Z-BUS SCC/Z8530 SCC Serial Communications Controller Technical Manual* (Campbell, CA: Zilog, Inc., 1989).
- For information on the real-time clock, read *DS1216F Dallas Semiconductor Clock Module Data Sheet*
- For information on the SCSI interface, read the *WD33C93A Technical Specification.*

Feel free to contact our Customer Support Department at 1-800-327-1251 if you have questions. We are prepared to answer general questions and provide help with documentation and specific applications.

2

Getting Started

2.1 EQUIPMENT

You need the following equipment to install the Heurikon HK80/V960E:

- Heurikon HK80/V960E microcomputer board
- VME card cage and power supply
- Serial interface cable (RS-232)
- CRT terminal
- Heurikon EPROM, which includes both monitor and bootstrap

CAUTION: **All semiconductors should be handled with care. Static discharges can easily damage the components on the HK80/V960E. Keep the board in an antistatic bag whenever it is out of the system chassis and *do not handle the board* unless absolutely necessary. Ground your body before touching the HK80/V960E board.**

CAUTION: **High operating temperatures will cause unpredictable operation and could damage the HK80/V960E. Because of the high chip density, fan cooling is required for all configurations, even when cards are placed on extenders.**

CAUTION: **Do not install the board in a rack or remove the board from a rack while power is applied, at risk of damage to the board.**

For basic operation, programming information, and a basic understanding of the intelligent components of this board, the following documents are essential:

- *Intel 80960CA User's Manual*
- *VTC VIC VMEbus Interface Controller Specification*
- *Intel 82596CA User's Manual* and the *Intel 82C501AD Data Sheet*

Contact us or the vendors for these documents.

2.2 PRELIMINARY CONSIDERATIONS

2.2.1 Electrical

If you are adding the HK80/V960E to an enclosure, the power supply must be sufficient for the additional board, as shown in Table 2-1.

TABLE 2-1
Power requirements

Voltage	Current	Usage
+5	9.0 A	All logic
+12	1.0 A	RS-232 interface and Ethernet
-12	1.0 A	RS-232 interface

Note: All of the "+5" and "Gnd" pins on P1 ***and*** P2 *must* be connected to ensure proper operation.

2.2.2 Physical

The board is a single-height VMEbus board (9.187" W x 6.299" H x 0.6" D) that occupies one slot in a VMEbus card cage.

2.2.3 Environmental

As with any printed circuit board, be sure that air flow to the board is adequate. Recommended air flow rate is about 2-3 cubic feet per minute, depending on card cage constraints and other factors. Operating temperature is specified at 0° to 55° C ambient, as measured at the board.

CAUTION: **High operating temperatures will cause unpredictable operation and could damage the HK80/V960E. Because of the high chip density, fan cooling is required for all configurations, even when cards are placed on extenders.**

2.3 INSTALLATION AND POWER-UP

All products are fully tested before they are shipped from the factory (please contact us if you would like to have current

information on MTBF [mean time between failures]). When you receive your HK80/V960E, follow these steps to ensure that the system is operational:

1. Visually inspect the board(s) for components that could have become loose during shipment. Visually inspect the chassis and all cables. Be sure all boards are seated properly in the VME card cage. Be sure all cables are securely in place. Power requirements are shown in Table 2-1.

2. Connect a CRT terminal to serial port B (port A for the VxWorks operating system), via connector P5. If you are making your own cables, refer to the drawing in section 10.11. Set the terminal as follows:

   - 9600 baud, full duplex
   - Eight data bits (no parity)
   - Two stop bits for transmit data
   - One stop bit for receive data
   - If your terminal does not have separate controls for transmit and receive stop bits, select one stop bit for both transmit and receive.

3. Turn the system on.

4. Push the system RESET button. A sign-on message and prompt from the monitor should appear on the screen. If not, check your power supply voltages and CRT cabling.

5. Now is the time to read the monitor manual and the operating system literature. Short course: type **help** to view a list of monitor commands, or type **bootrom** to boot the operating system, if an operating system is accessible.

6. Reconfigure the jumpers, etc., as necessary for your application. See section 15 for a summary of I/O device addresses and configuration jumpers.

2.4 TROUBLESHOOTING AND SERVICE INFORMATION

In case of difficulty, use this checklist.

CAUTION: **Always be sure you are grounded when you touch the HK80/V960E.**

1. Be sure the system is not overheating.

2. Inspect the power cables and connectors.

3. If the monitor program is executing, run the diagnostics by using the monitor **testmem** command.

4. Check your power supply for proper DC voltages. If possible, use an oscilloscope to look for excessive power supply ripple or noise. Note that the use of P2 is required to meet the power specifications.

5. Check the chips to be sure they are firmly in place. Look for chips with bent or broken pins. In particular, check the EPROM.

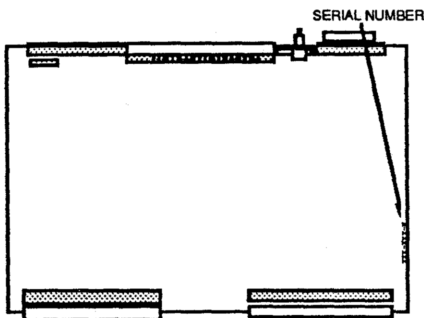
6. Check your terminal switches and cables. Be sure the P5 connector is on properly. If you have made your own cables, pay particular attention to the cable drawings in sections 10.11 and 13.4.

7. Check the jumpers to be sure your board is configured properly. All jumpers should be in the "standard configuration" positions shown in section 15.3. Check the EPROM jumpers, especially.

8. Since the HK80/V960E monitor uses its on-card non-volatile RAM (NV-RAM) to configure and set the baud rates for its console port, the lack of a prompt might be caused by incorrect terminal settings, an incorrect configuration of the NV-RAM, or a malfunctioning NV-RAM. Another possible cause is that the autoboot parameters are set in NV-RAM so that the monitor is trying to autoboot something. Try pressing the **H** character a few times after a reset. If the prompt comes up, the NV-RAM was most likely configured to autoboot. For more information about the way that the NV-RAM configures the console port baud rates, refer to the summary at the end of this manual (Appendix B).

9. After you have checked all of the above items, call our Customer Service Department for help. Please have the following information handy:

- The monitor program revision level. The revision level can be found on the display screen as part of sign-on message and on the EPROM label.
- The HK80/V960E p.c.b. serial number (inscribed along the card edge)
- The serial number of the operating system



If you plan to return the board to Heurikon for service, contact our Factory Service Department at 1-800-327-1251 to obtain a **Return Merchandise Authorization** (RMA) number. Be prepared to provide the items listed above, plus your purchase order number and billing information if your HK80/V960E is out of warranty. If you return the board, be sure to enclose it in an antistatic bag such as the one in which it was originally shipped. Send it prepaid to:

Heurikon Corporation
Factory Service Department
8310 Excelsior Drive
Madison, WI 53717

Please put the RMA number on the outside of the package so we can handle your problem most efficiently. Heurikon cannot accept material received without an RMA number.

2.5 MONITOR SUMMARY

An optional EPROM-based debug-monitor/bootstrap for the HK80/V960E is available. General features and functions include the ability to:

- Manually download data or 80960CA program code.
- Check the processor, memory, VME, VSB, and I/O devices.
- Execute a bootstrap (for example, boot an operating system).
- Disassemble 80960CA program code.

The monitor uses the area between 400_{16} and 10000_{16} for stack and uninitialized-data space. Any *writes* to that area can cause unpredictable operation of the monitor. The monitor initializes this area (that is, writes to it) to prevent parity errors, but it is the programmer's responsibility to initialize any other memory areas that are accessed.

Help

Type **help** to read a summary of monitor commands, or just type the command name to view selections. Each command may be typed with the shortest number of characters that uniquely identifies the command.

Command editor

The monitor provides a command line editor that uses typical UNIX® *vi* editing commands. You can edit any command line you type. First press the ESC key to invoke the editor. Press Enter or Return to send a carriage return <cr>, which executes the current command and exits the editor. A summary of the editor commands is shown in Table 2-2.

TABLE 2-2
Summary of editing commands for the monitor program

Key	Function
<ESC>	At the monitor prompt, invokes the editor.
<cr>	Once the editor is invoked, causes the current command to be executed and the editor to be "exited."
k	Scroll "backward" through command list.
j	Scroll "forward" through command list.
h	Move cursor "left" in command line.
l	Move cursor "right" in command line.

Other *vi*-like commands that can be used are **x, i, a, A, $, 0, w, cw, dw, r,** and **e.**

3

MPU Summary Information

3.1 INTRODUCTION

This section details some of the important features of the 80960CA MPU chip and, in particular, items that are specific to its implementation on the Heurikon HK80/V960E.

Refer to the 80960CA user's manual for more information on the processor's implementation of the features described in this section.

3.2 MPU INITIALIZATION

After the HK80/V960E is powered up (or after an HK80/V960E reset), the 80960CA begins its initialization. It uses an initial memory image (IMI) to establish its state. The IMI contains the initialization boot record (IBR), the process control block (PRCB), and the system data structures. The 80960CA reads in the IBR and PRCB, does the specified configuration, and then starts execution of the user program specified in the IBR.

The 80960CA may be reinitialized by software (via the ASM960 **sysctl** instruction). When reinitialization takes place, a new PRCB and a reinitialization instruction pointer are specified. Reinitialization is useful for relocating data structures from ROM to RAM after initialization.

Refer to Figure 3-1 below for a general overview of the 80960CA structures. For more details of these structures, refer to the 80960CA user's manual.

3.2.1 Initialization Boot Record (IBR)

The 80960CA internally defines the base of the IBR to be at FFFF,$FF00_{16}$ (which is why ROM needs to be in this area at power-up). The IBR is the primary data structure (12 long words) required to initialize the 80960CA.

3.2.2 Process Control Block (PCRB)

The PRCB contains pointers to system data structures, and also contains information used to configure the processor at initialization (Fig. 3-1).

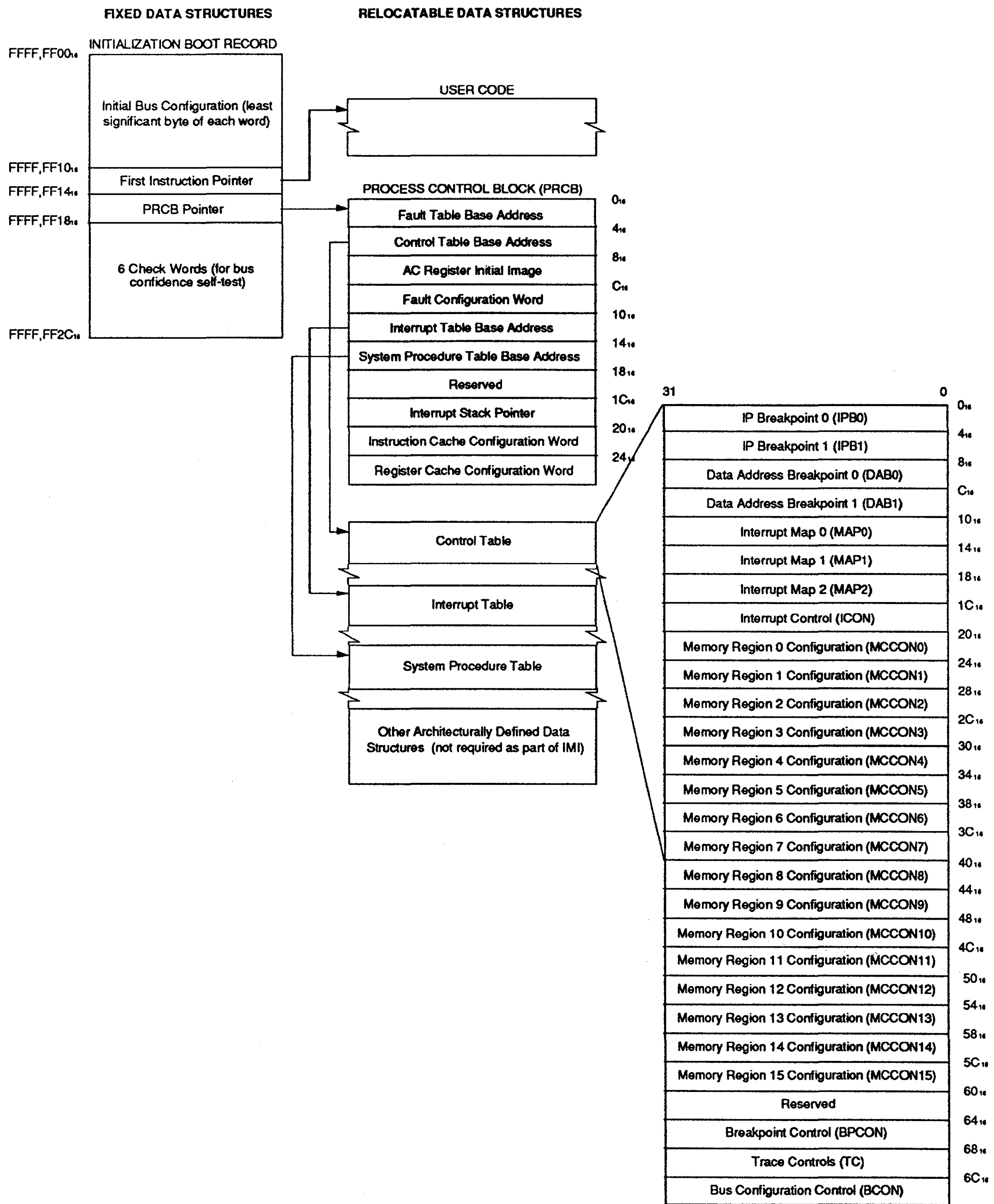


FIGURE 3-1. MPU structures and control table

Adapted from 80960CA User's Manual, *1989, pages E-13 and E-19. Used by permission.*

3.3 BYTE ORDERING

The 80960CA supports both little-endian (Intel) and big-endian (Motorola) byte ordering. The byte ordering determines which memory location stores the least significant byte of the operand. For little-endian systems, the *least* significant byte is stored at the lowest byte address. For big-endian systems, the *most* significant byte is stored at the lowest address. The number of bytes per operand depends on the data type. For example, if a Motorola (big-endian) processor writes the long word 12345678_{16} to location 0, the HK80/V960E (in little-endian mode) reading a byte from location 0 sees 78_{16}. From location 1, it sees 56_{16}, from location 2 it sees 34_{16}, and from location 3 it sees 12_{16} (see Table 3-1).

TABLE 3-1
Little-endian and big-endian byte ordering

	Long Word Written by a Big-endian Processor: Location 0 D31 – D0 12345678_{16}							
	Read by HK80/V960E in Little-endian Mode				**Read by HK80/V960E in Big-endian Mode**			
Byte	Location 0 ($A_1A_0=00_2$) D7 – D0 = 78_{16}	Location 1 ($A_1A_0=01_2$) D15 – D8 = 56_{16}	Location 2 ($A_1A_0=10_2$) D23 – D16 = 34_{16}	Location 3 ($A_1A_0=11_2$) D31 – D24 = 12_{16}	Location 0 ($A_1A_0=00_2$) D31 – D24 = 12_{16}	Location 1 ($A_1A_0=01_2$) D23 – D16 = 34_{16}	Location 2 ($A_1A_0=10_2$) D15 – D8 = 56_{16}	Location 3 ($A_1A_0=11_2$) D7 – D0 = 78_{16}
Word	Location 0 ($A_1A_0=00_2$) D15 – D0 = 5678_{16}		Location 2 ($A_1A_0=10_2$) D31 – D16 = 1234_{16}		Location 0 ($A_1A_0=00_2$) D31 – D16 = 1234_{16}		Location 2 ($A_1A_0=10_2$) D15 – D0 = 5678_{16}	
Long Word	Location 0 ($A_1A_0=00_2$) D31 – D0 = 12345678_{16}				Location 0 ($A_1A_0=00_2$) D31 – D0 = 12345678_{16}			

The 80960CA uses little-endian byte ordering internally. From the 80960CA's point of view, all of the memory regions (there are 16), including the on-chip data RAM, may be individually configured as big-endian or little-endian via the memory configuration registers (MCON0-MCON15) of the 80960CA. Data and instructions may be located in either big- or little-endian regions.

The HK80/V960E user's manual is valid for a little-endian implementation. That is, the device addresses are correct for little-

endian, but some data regions (such as VMEbus) may be configured either way. Compilers that are currently used only support little-endian code generation.

Please refer to the 80960CA user's manual for further details, or contact Heurikon regarding implementation possibilities.

3.4 MPU INTERRUPTS

The 80960CA interrupt controller manages three types of interrupts:

1. Twelve hardware interrupt sources coming from eight external interrupt pins and the four internal DMA interrupt sources.
2. A single, nonmaskable interrupt (NMI) pin that indicates serious system failures.
3. Software interrupts that can be posted directly by a user's program or by another processor.

This section describes the hardware and NMI interrupts, the data structures used for interrupt handling, and the method by which these data structures are used by the interrupt handler.

3.4.1 Interrupt Structures

3.4.1.1 The Interrupt Table

The interrupt table is a 1028-byte table that is referenced by software and hardware interrupts (Fig. 3-2). The table base is described in the process control block (PRCB), which is read at power-up or during processor reinitialization. The interrupt table must be long word aligned. Vectors 0-7 are not defined in the 80960CA architecture; the locations are used by the interrupt controller to control pending software interrupts. The first 36 bytes of the interrupt table used for software interrupts are described in the 80960CA user's manual. The remainder of the table describes the address of the interrupt handler for vectors 8-255 (8_{16}-FF_{16}).

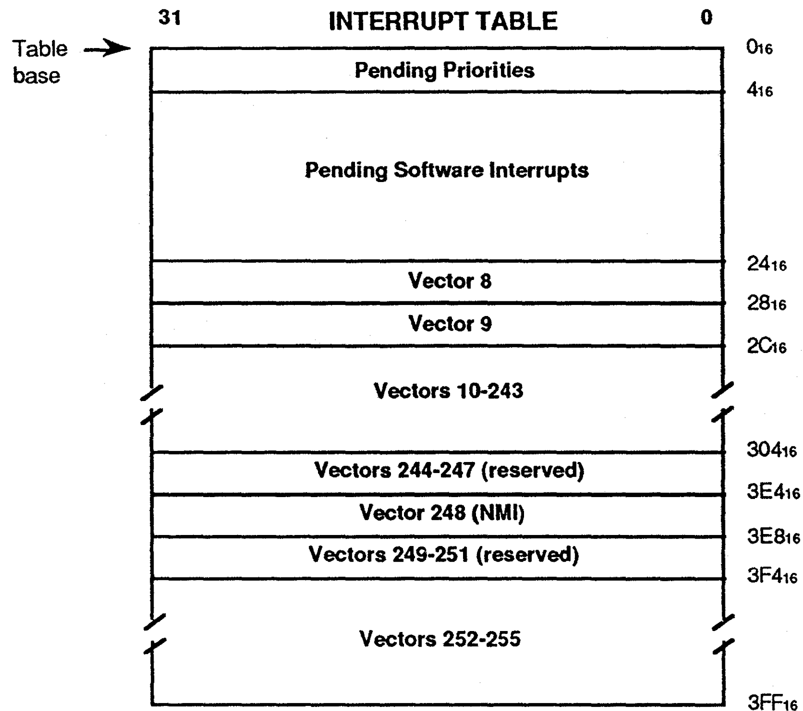


FIGURE 3-2. MPU interrupt table

The address of the long word location associated with any particular vector can be calculated by multiplying the vector by 4 and adding the result to the table base plus 4.

The C expression below can be used to write the address of an interrupt handler into the interrupt table for a given vector.

```
*((unsigned long *) (INT_TABLE_BASE + 4 + (Vector << 2))) = Intr_Handler();
```

3.4.1.2 The Interrupt Stack Frame

When an interrupt is serviced, an interrupt record containing the vector number and control registers is written on the interrupt stack. A stack frame containing the return instruction pointer is also allocated on the interrupt stack. The interrupt stack pointer is loaded from the PRCB during initial power-up and during reinitialization.

3.4.2 The Nonmaskable Interrupt (NMI)

The NMI interrupt is caused by the assertion of a dedicated external interrupt pin. The NMI is always vectored to the interrupt table entry for vector 248 (byte offset $3E4_{16}$ from the table base) and has a priority of 31. Either of the following two conditions can cause an NMI:

1. **Bus error** — A bus error occurs either when a bus access was not acknowledged before the bus watchdog timer expired (time-out time is programmable via the VIC chip, as described in section 6-10), *or* when an illegal bus access was requested (for example, a 32-bit request from an 8-bit port.)

2. **Parity error** — A parity error occurs when the RAM interface detects bad parity read from memory. This can happen for a "true" parity error or if uninitialized RAM is read.

For both error conditions, the cycle in which the error occurred is terminated, and then the NMI interrupt handler is serviced. It is the responsibility of the interrupt handler to determine the cause of the interrupt. When an NMI occurs, the interrupt service routine ***must*** read the status latch to remove the interrupt. The status latch is an 8-bit port that removes the NMI interrupt request and provides a 3-bit code that indicates the cause of the failure. If the NMI service routine fails to read the status latch, the program will hang indefinitely in the service routine; that is, the hardware will not remove the NMI signal. The encoding of the status latch is described in Table 3-2. Note that only the lowest three bits are defined. All others are undefined. The status latch is located at address $0210,0000_{16}$ and should be read as a byte port.

TABLE 3-2
HK80/V960E error status latch encoding

Port address: $0210,0000_{16}$. Size: Byte. Type: Read.				
D2	**D1**	**D0**	**Failure Type**	**Owner of Local Bus**
0	0	0	Bus error	Unknown
0	0	1	Parity error	Unknown
0	1	0	Bus error	82596CA (Ethernet)
0	1	1	Parity error	82596CA (Ethernet)
1	0	0	Bus error	VIC068 (VME slave access)
1	0	1	Parity error	VIC068 (VME slave access)
1	1	0	Bus error	80960CA (MPU)
1	1	1	Parity error	80960CA (MPU)

An example program can be found in Appendix A.

3.4.3 Hardware Interrupts

There are 12 possible sources for hardware interrupts — four internal DMA channel interrupts and eight external I/O interrupts. Table 3-3 shows the connections of external interrupt pins

to external devices, and Figure 3-3 shows the HK80/V960E interrupt architecture.

TABLE 3-3
External interrupt pin mappings

80960CA Interrupt Pin	Connected To Device:
XINT7	CIO (counter timer)
XINT6	SCSI
XINT5	SCC ports A and B
XINT4	SCC ports C and D
XINT3	Ethernet
XINT2	VME VIC068
XINT1	
XINT0	

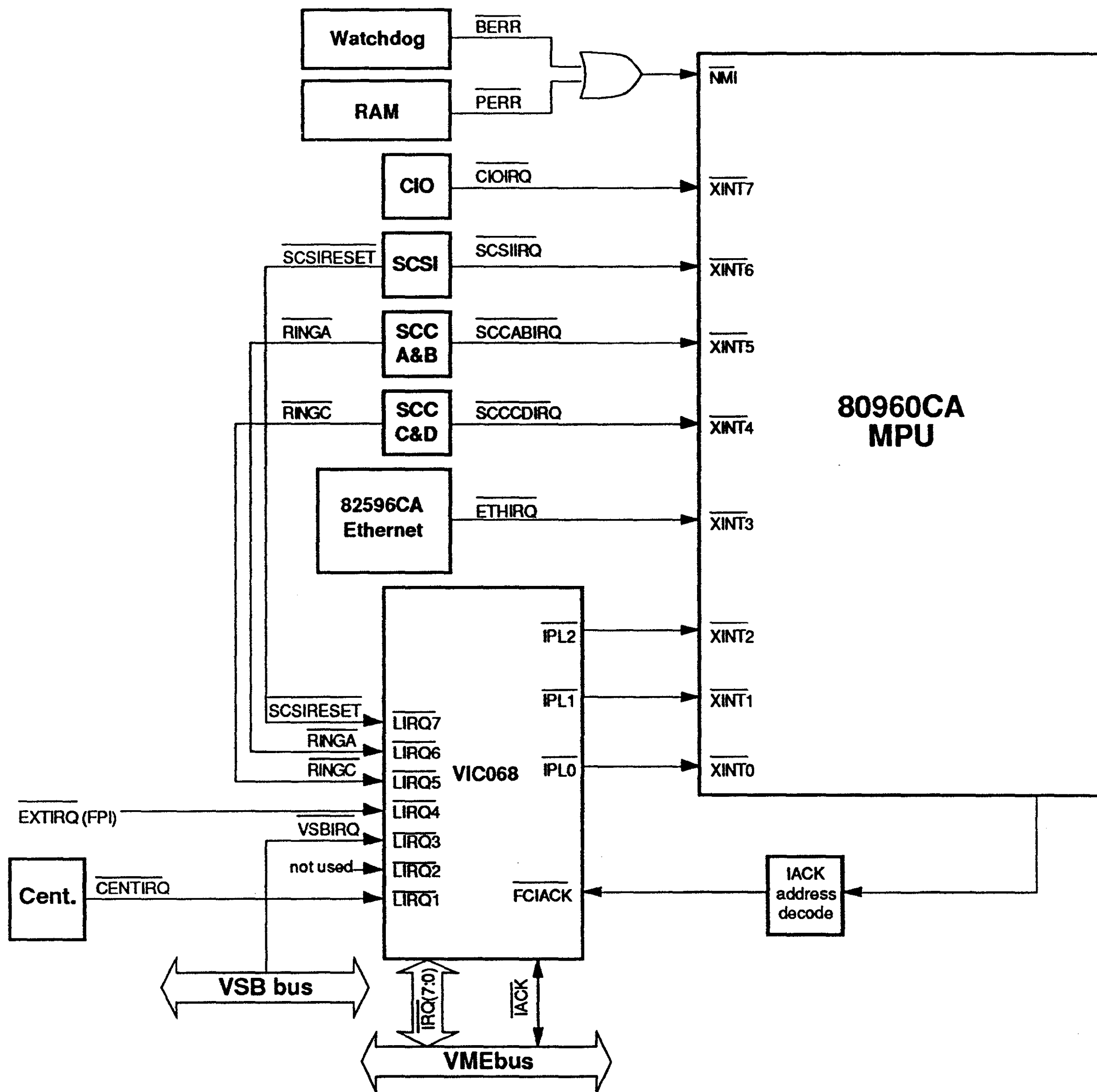


FIGURE 3-3. HK80/V960E interrupt architecture

The 80960CA interrupt controller can operate in one of three modes — dedicated, expanded, or mixed. The HK80/V960E supports only the *dedicated* mode and must be initialized, via the interrupt control register *ICON*, to this state to function properly.

The 80960CA is an extremely flexible architecture that allows the programmer to control and configure external interrupts through several registers. The programmer can control the following functions:

1. Individual mask bits are provided for each hardware interrupt by writing to the special function register one (sf1), also called the interrupt mask register (IMSK).

2. Individual hardware interrupt(s) can be detected in software by using the special function register zero (sf0), also called the interrupt-pending register (IPND).

3. Each hardware interrupt can be mapped to one of 16 priority levels. The priority levels are set in the interrupt map (IMAP) registers loaded from the processor control table specified in the PRCB. (Refer to the 80960CA user's manual for details on the PRCB.)

4. Each hardware interrupt can be programmed as either level-sensitive or edge-sensitive.

5. Interrupts can be programmed either to debounce the interrupts for several clocks or to respond immediately for faster response times.

6. Interrupts can be cached in the internal data RAM for faster response times.

All of these functions are controlled through several registers. The next few sections describe them, and a suggested register initialization is given when applicable.

3.4.3.1 Interrupt Priority

The interrupt controller assumes a unique priority for each vector in the table. Vector 256 has the highest priority and vector 8 has the lowest priority.

At all times, the processor is executing at one of 31 priorities, which are encoded by five bits of the processor's control word. The priority level can be read or modified with the ASM960 **modpc** instruction. When an interrupt is detected, its priority is compared with the priority of the currently running program. If the interrupt's priority is greater than the processor's current priority, the interrupt handler is serviced, and the processor's priority is modified to the higher level. When multiple interrupt requests are pending at the same priority level, the highest vector number is serviced first. If the interrupt priority is less than or equal to the 80960CA's priority, the processor does not service the request.

The priority of an interrupt is calculated by shifting the vector number right by three bits. Vectors 8-15 are priority 1, vectors 16-23 are priority 2, and vectors 248-255 are priority 31. Priority 0 is not defined in the 80960CA architecture.

3.4.3.2 Interrupt Mask Register (IMSK)

The interrupt mask register (IMSK), which is special function register 1 (sf1), allows masking of any of the twelve hardware interrupts. The format of this register and the device associated with each bit are described in Table 3-4. Writing a 1 enables the interrupt. Writing a 0 disables the interrupt.

TABLE 3-4
Interrupt mask register

11	10	9	8	7	6	5	4	3	2	1	0
DMA Ch. 3	DMA Ch. 2	DMA Ch. 1	DMA Ch. 0	XINT7 CIO	XINT6 SCSI	XINT5 SCC A&B	XINT4 SCC C&D	XINT3 Ethernet	XINT2 VIC Level 2	XINT1 VIC Level 1	XINT VIC Level 0

3.4.3.3 Interrupt-Pending Register (IPND)

The format of the interrupt-pending register (IPND), which is special function register 0 (sf0), is the same as the mask register. When it is read, the register indicates a pending interrupt with a 1. The interrupt-pending register can also be used to generate interrupts by writing a 1 to the associated bit. This register *must* be cleared after every interrupt acknowledge. The format of this register and the device associated with each bit are described in Table 3-5.

TABLE 3-5
Interrupt-pending register

11	10	9	8	7	6	5	4	3	2	1	0
DMA Ch. 3	DMA Ch. 2	DMA Ch. 1	DMA Ch. 0	XINT7 CIO	XINT6 SCSI	XINT5 SCC A&B	XINT4 SCC C&D	XINT3 Ethernet	XINT2 VIC Level 2	XINT1 VIC Level 1	XINT VIC Level 0

3.4.3.4 Interrupt Mapping Registers (IMAP0-IMAP2)

Three interrupt map registers (IMAP0-2) are used to determine the priority of hardware interrupts. Each interrupt source is associated with a 4-bit value in the register. Table 3-6 shows the relationship between the value written, the interrupt vector, the priority of the interrupt, and the location for caching the vector. A suggested setting is also included. The interrupt map registers are loaded at reset from the PRCB or by the ASM960 **sysctl** instruction.

TABLE 3-6
Interrupt mapping registers

Value in IMAP	Interrupt Priority	Associated Vector Number	Internal RAM Address (if cached)	Suggested Source
1111_2	30	242 ($F2_{16}$)	$3C_{16}$	—
1110_2	28	226 ($E2_{16}$)	38_{16}	—
1101_2	26	210 ($D2_{16}$)	34_{16}	—
1100_2	24	194 ($C2_{16}$)	30_{16}	DMA channel 3
1011_2	22	178 ($B2_{16}$)	$2C_{16}$	DMA channel 2
1010_2	20	162 ($A2_{16}$)	28_{16}	DMA channel 1
1001_2	18	146 (92_{16})	24_{16}	DMA channel 0
1000_2	16	130 (82_{16})	20_{16}	CIO counter/timer
0111_2	14	114 (72_{16})	$1C_{16}$	SCSI
0110_2	12	98 (62_{16})	18_{16}	SCC ports A&B
0101_2	10	82 (52_{16})	14_{16}	SCC ports C&D
0100_2	8	66 (42_{16})	10_{16}	Ethernet
0011_2	6	50 (32_{16})	$0C_{16}$	VIC level 2
0010_2	4	34 (22_{16})	08_{16}	VIC level 1
0001_2	2	18 (12_{16})	04_{16}	VIC level 0

The format of the interrupt mapping registers is outlined in Table 3-7.

TABLE 3-7
Interrupt mapping register format

Register	Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
IMAP2	DMA ch. 3	DMA ch. 2	DMA ch. 1	DMA ch. 0
IMAP1	XINT7 CIO	XINT6 SCSI	XINT5 SCC ports A&B	XINT4 SCC ports C&D
IMAP0	XINT3 Ethernet	XINT2 VIC level 2	XINT1 VIC level 1	XINT0 VIC level 0

The definition in the PRCB shown in Table 3-8 would initialize the interrupts as suggested in Table 3-6.

TABLE 3-8
PRCB definition

IMAP2:	.word	$0000CBA9_{16}$	# Interrupt control register 2
IMAP1:	.word	00008765_{16}	# Interrupt control register 1
IMAP0	.word	00004321_{16}	# Interrupt control register 0

3.4.3.5 Interrupt Control Register (ICON)

The interrupt control register (ICON) is a collection of bit fields that are used to configure the interrupt controller. The bits are defined in Table 3-9. This register is read by the processor at reset from the PRCB or loaded by using the ASM960 **sysctl** instruction.

TABLE 3-9
ICON register definition

Register bits	Definition	Suggested setting
1-0	Selects interrupt controller mode.	0 — Dedicated mode
9-2	Indicates if level- or edge-sensitive for XINT7-0	0 — Level, active low, respectively.
10	Global interrupt enable	0 — Enable
12-11	Determines interrupt mask operation.(See 80960CA user's manual.)	0 — Mask unchanged
13	Enables caching of all vectors.	0 — Enable
14	Sample mode for interrupts.	1 — Fast, no debounce
15	DMA suspension on interrupt.	1 — Yes

The following PRCB definition is derived from the suggested setting in Table 3-9:

```
ICON:    .word    0000C000₁₆    # Interrupt config register
```

3.4.4 Software Interrupts

Interrupts may be requested directly by a user program. This mechanism may be useful for requesting and prioritizing low-level tasks in a real-time application. Software can request interrupts in the following two ways:

1. With the **sysctl** instruction
2. By the 80960CA (or another processor) posting an interrupt in the pending-priorities/pending-interrupts fields of the interrupt table (see Figure 3-2).

Refer to the 80960CA user's manual for details.

3.5 MPU FAULTS

During processor execution, numerous conditions can cause the processor to follow an alternate execution thread or to calculate incorrect results. These conditions are considered "fault conditions." Examples of fault conditions are division by zero, invalid operands, protection violations, and trace faults.

This section briefly describes the data structures used for handling faults and the faults defined in the 80960CA architecture. For a detailed description of faults, refer to the 80960CA user's manual.

3.5.1 The Fault Table

The fault table is a 256-byte table that provides a pathway to fault-handling procedures. The fault table base address is defined in the PRCB. The fault table must be long word aligned. There is one 8-byte entry for each fault type in the table. The processor uses each entry to determine the location and type of fault handling procedure to use. Figure 3-4 shows how the 80960CA fault table is organized.

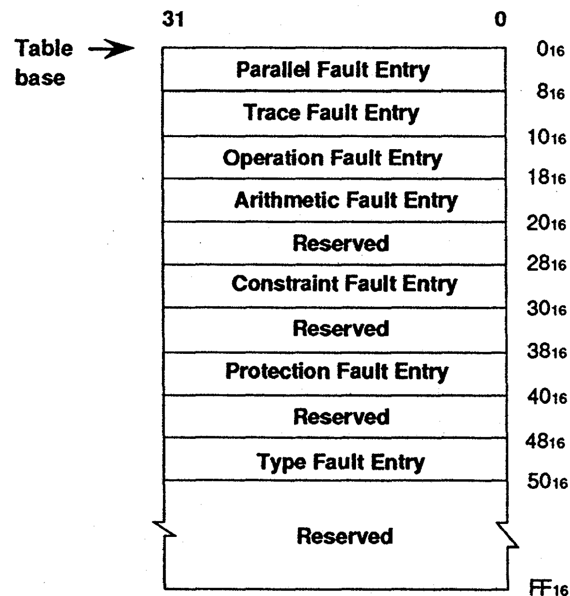


FIGURE 3-4. MPU fault table

Two types of fault table entries are allowed: a local-call entry and a system-call entry. Both entry types are two long words in length. Figure 3-5 shows the format for both entry types.

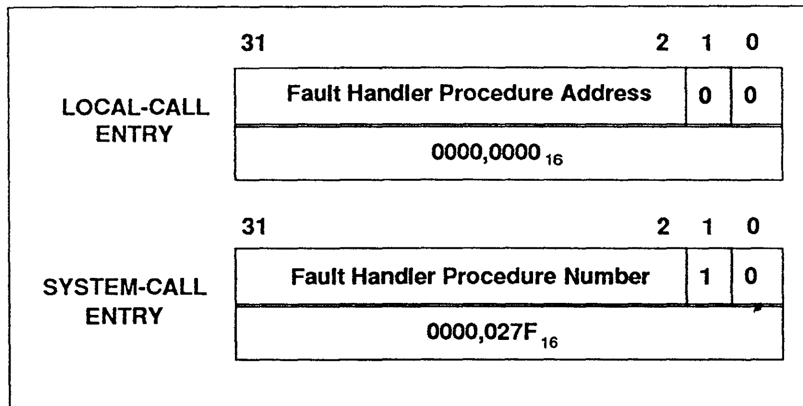


FIGURE 3-5. Fault table entries

The local-call entry provides an instruction pointer to the fault handling procedure. The system-call entry provides fault handling through the system procedure table. The system procedure table is described in the 80960CA user's manual.

3.5.2 The Fault Stack Frame

When a fault is detected, the processor allocates a new set of registers on the currently active stack and creates a fault record on the stack. The fault record contains the processor's control registers, the address of the faulting instruction, and one long word encoded with the type of fault. Table 3-10 shows the types of faults defined in the 80960CA architecture.

TABLE 3-10
80960CA fault types and subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number/Bit Position**	**Name**	
0_{16}	Parallel	$2_{16} - FF_{16}$	Indicates number of faults that occur in parallel.	$XX00,XX02_{16}$ $XX00,XXFF_{16}$
1_{16}	Trace	Bit 1	Instruction Trace	$XX01,XX02_{16}$
		Bit 2	Branch Trace	$XX01,XX04_{16}$
		Bit 3	Call Trace	$XX01,XX08_{16}$
		Bit 4	Return Trace	$XX01,XX10_{16}$
		Bit 5	Prereturn Trace	$XX01,XX20_{16}$
		Bit 6	Supervisor Trace	$XX01,XX40_{16}$
		Bit 7	Breakpoint Trace	$XX01,XX80_{16}$
2_{16}	Operation	1_{16}	Invalid Opcode	$XX02,XX01_{16}$
		2_{16}	Unimplemented	$XX02,XX02_{16}$
		3_{16}	Unaligned	$XX02,XX03_{16}$
		4_{16}	Invalid Operand	$XX02,XX04_{16}$
3_{16}	Arithmetic	1_{16}	Integer Overflow	$XX03,XX01_{16}$
		2_{16}	Arithmetic Zero-Divide	$XX03,XX02_{16}$
4_{16}	Reserved (Floating Point)			
5_{16}	Constraint	1_{16}	Constraint Range	$XX05,XX01_{16}$
		2_{16}	Privileged	$XX05,XX02_{16}$
6_{16}	Reserved			
7_{16}	Protection	Bit 1	Length	$XX07,XX01_{16}$
$8_{16} - 9_{16}$	Reserved			
A_{16}	Type	1_{16}	Type Mismatch	$XX0A,XX01_{16}$
$B_{16} - F_{16}$	Reserved			

SOURCE: 80960CA User's Manual, 1989, p. 7-3.

3.6 MPU DMA SUPPORT

Refer to the 80960CA user's manual for more detail of the processor's implementation of the features described in this section.

The CPU (80960CA) has an on-chip DMA controller, which can manage four independent channels of DMA concurrently. All channels support the following:

- Standard multi-cycle transfers with byte-assembly
- Multiple operand size combinations, for example:

  8 to 8 bits

  8 to 32 bits

  32 to 8 bits

  32 to 32 bits

  128 to 128 bits (that is, burst mode)
- Memory-to-memory transfers (block mode, synchronized), in which the source and destination can be any combination of internal data RAM (cache) or external memory
- Memory-to-device transfers (demand mode, synchronized)
- Device-to-memory transfers (demand mode, synchronized)
- Chained DMA transfers (source and/or destination)
- Burst DMA transfers (using the 128-byte quad transfer mode)
- Fixed or rotating channel priority

3.6.1 HK80/V960E Implementation

In the HK80/V960, all four channels are dedicated to on-card devices (Table 3-11). The channels can still be used for memory-to-memory transfers.

TABLE 3-11
80960CA DMA channels on the HK80/V960E

80960CA DMA Channel	Device	Section
0	SCC port D	10.7
1	SCC port C	10.7
2	SCC port A	10.7
3	SCSI	12.2

3.6.2 Registers/Instructions

Multiple registers and instructions are associated with the 80960CA DMA. Their descriptions and access methods are described below.

3.6.2.1 DMA Command Register (DMAC)

This register is specified as special function register 2 (sf2) in Intel ASM960 assembler. Refer to the 80960CA user's manual for details. The register contains the following:

- The enable for the channels
- The status of the channels during and after a transfer
- The channel priority mode (fixed or rotating)
- The DMA throttle, which selects the maximum ratio of DMA/CPU clocks

3.6.2.2 The Set-up-DMA (sdma) Instruction

ASM960 assembler

SYNTAX: **sdma** *op1, op2, op3*

op1 specifies channel number (0 – 3).

op2 specifies the DMA control word, which includes:

- Transfer type
- Operand size
- Demand or block mode
- Chaining select
- Termination conditions

op3 This quad-aligned register must be the first of three consecutive registers, where:

op3 = byte count

op[*3+1*] = source address

op[*3+2*] = destination address

3.6.2.3 Update DMA-Channel RAM Instruction (udma)

ASM960 assembler

SYNTAX **udma**

This command causes the current status of the DMA channels to be written to the dedicated DMA RAM, which is between $0000{,}0040_{16}$ and $0000{,}00C0_{16}$.

3.6.3 DMA Interrupts

There is a dedicated interrupt for each DMA channel (0 – 3). Refer to section 3.4 ("MPU Interrupts") and the 80960CA manual for a detailed discussion of DMA interrupts. Take special note of the following 80960CA registers:

- Interrupt Control Register (ICON)
- Interrupt Mapping Register (IMAP2)
- Interrupt Mask Register (IMSK), ASM960 syntax — sf1
- Interrupt Pending Register (IPND), ASM960 syntax — sf0

Interrupt priorities may be user-defined; Table 3-6 and Table 3-8 show the recommended set-up.

3.6.4 DMA Data Alignment

In many cases, the DMA controller in the 80960CA may perform operations on source and destination data that are not aligned in memory. In other words, the source and destination addresses do not need to be aligned to a module memory boundary, or aligned with respect to one another. Alignment restrictions are as follows:

For all DMA where the *source* and *destination* addresses increment (except "quad" transfers), there are no alignment restrictions.

Source, *destination*, and *byte-count* must be quad-word aligned for "quad" transfers.

If the *source* address is fixed (rather than incrementing), the *source* address must be aligned.

If the *destination* address is fixed (rather than incrementing), the *destination* address must be aligned.

In general, aligned DMA transfers perform better than non-aligned transfers.

Many nonaligned cases execute byte-long bus requests to load or store data at the nonaligned address. For example, a non-aligned 16- to 8-bit transfer would revert to 8- to 8-bit.

Consult the 80960CA user's manual for further details.

3.7 MPU TRACE EVENTS

The 80960CA architecture provides facilities for monitoring the activity of the processor through the generation of "trace events." A trace event indicates a condition in which the processor has just executed (or is about to execute) a particular instruction.

When the processor detects a trace event, it generates a trace fault and makes an implicit call to the fault-handling procedure for trace faults. This procedure can be used to call debugging software to display or analyze the state of the processor when the trace event occurred.

Tracing is enabled by the trace-enable bit in the process-controls register (pc) and a set of trace-mode bits in the trace-controls register (tc). Alternately, the **mark** and **fmark** instructions can be used to generate trace events explicitly from a program.

Also provided are four hardware "breakpoint" registers that generate trace events and trace faults. Two registers are dedicated to trapping on instruction execution addresses, while the remaining two registers can trap on the addresses of various types of data accesses.

Trace modes are summarized below:

Instruction	Traps on every instruction.
Branch	Traps on every **branch** instruction.
Call	Traps on every **call** instruction.
Return	Traps on every **return** instruction.
Prereturn	Traps before every **return** instruction.
Supervisor	Traps on every **call-system** instruction.
Breakpoint	Traps on **breakpoints** specified in breakpoint registers.

The trace registers are summarized below.

tc	Trace controls register
IPB0-IPB1	Instruction address breakpoint registers
DAB0-DAB1	Data address breakpoint registers
BPCON	Hardware breakpoint control registers

Refer to the 80960CA user's manual and section 3.5 ("MPU Faults") of this manual for details about faults and tracing. If you are using the Heurikon HK80/V960E monitor EPROM, refer to Appendix A for details about our implementation of tracing.

3.8 MPU CACHES

The 80960CA supports three caching mechanisms: data RAM, instruction cache, and register cache. These are briefly described below.

Refer to the 80960CA user's manual for further details.

3.8.1 Data RAM Cache

One Kbyte of user-visible high-speed (528 Mbytes/sec at 33 MHz) internal data RAM is integrated on the 80960CA on an internal 128-bit bus, which is mapped into the first 1 Kbyte of address space on the HK80/V960: 0000_{16} – 0400_{16}. Allocated correctly, this resource can be used to dramatically increase the performance of critical application algorithms.

Data RAM is accessed by loads, stores, or DMA transfers. Instruction fetches to these addresses will cause an "operation-unimplemented" fault to occur. Some of the data RAM may optionally be used to store DMA status, cached interrupt vectors, and cached local registers. Application software may use the data RAM.

3.8.2 Instruction Cache

The 80960CA contains a 1-Kbyte two-way set associative instruction cache, which is organized into two sets of 16 eight-word lines. Each line is composed of four two-word blocks.

The instruction cache enhances the 80960CA's performance by reducing the number of instruction fetches from slower external RAM, resulting in fast execution of cached code, and also provides more bus bandwidth for data operations to external memory.

The instruction cache may be enabled or disabled via the "Instruction Cache Configuration Word" at initialization in the PRCB or by using the ASM960 **sysctl** instruction. See the 80960CA user's manual for details.

3.8.3 Register Cache

At initialization, the "Register Cache Configuration Word" is used to specify the number of register sets (0 to 15) that may be cached on-chip. The local register set is saved to the local register cache when a "call" is made. When the cache is full, the oldest set of local registers is flushed to the stack in external memory.

3.9 MPU PROCESSING MODES

The capability of a separate "user" and "supervisor" execution mode by the 80960CA creates a code and data protection mechanism referred to as the "user-supervisor protection model". This mechanism may be used to restrict access to all or parts of the operating system (kernel) by application code.

Refer to the 80960CA user's manual for further details.

3.10 MPU REGISTER SUMMARY

The 80960CA consists of the following registers and structures. Refer to the 80960CA user's manual for details (a summary can be found in the appendix to the 80960CA user's manual).

REGISTERS	DESCRIPTION
g0-g15	Sixteen 32-bit global registers **g0-g14** General purpose **g15** Frame pointer (FP)
r0-r15	Sixteen 32-bit local registers, which provide local storage for each active procedure, where: **r0** Previous frame pointer (pfp) **r1** Stack pointer (sp) **r2** Return instruction pointer (rip) **r3-r15** General purpose
sfr0-sfr2	Three special function registers whose meanings are: **sf0** Interrupt-pending register (IPND) **sf1** Interrupt mask register (IMSK) **sf2** DMA command register (DMAC)
pfp	Previous frame pointer (r0)
sp	Stack pointer (r1)

rip	Return instruction pointer (r2)
ac	Arithmetic controls register
pc	Process controls register
ICON	Interrupt control register
IPND	Interrupt-pending register (sf0)
IMSK	Interrupt mask register (sf1)
IMAP0-IMAP2	Interrupt mapping registers
DMAC	DMA command register (sf2)
DMACW	DMA control word: accessed via assembler: **sdma**
tc	Trace controls register
IPB0-IPB1	Instruction address breakpoint registers
DAB0-DAB1	Data address breakpoint registers
BPCON	Hardware breakpoint control register
MCON0-MCON15	Memory configuration registers
BCON	Bus configuration register

CONTROL STRUCTURES	DESCRIPTION
IMI	Initial memory image
IBR	Initialization boot record
PRCB	Process control block

4

System Error Handling

4.1 INTRODUCTION

Many events can cause either a hardware or software error. The responses to those error conditions are carefully controlled. This section describes the error types and sources.

4.2 ERROR CONDITIONS

4.2.1 Hardware Errors

Hardware errors are errors that are detected in the hardware logic of the HK80/V960E.

The following error conditions might arise during MPU cycles:

CONDITION	DEFINITION
Parity Error	Incorrect parity was detected during a read cycle from on-card RAM memory. This might result from a true parity error (RAM data changed) or because the memory location was not initialized prior to the read and it contained garbage. Parity errors generate a nonmaskable interrupt.
Bus Error	The bus error occurs when an access has timed out before the cycle has been acknowledged. All on-card accesses and accesses to either the VMEbus or the VSB bus are timed by the VME interface controller (VIC). The timeout period is programmable and enabled in the VIC (see section 6.10).

Accesses to nonexistent locations on the VMEbus or undefined on-card I/O can cause the bus to hang indefinitely if no watchdog timer is enabled.

Bus errors generate a nonmaskable interrupt.

Bus errors and parity errors assert the nonmaskable interrupt pin and then terminate the cycle normally. The processor then traps to the NMI exception routine. When read, the error status latch removes the nonmaskable interrupt and provides a 3-bit code that indicates the bus master at the time of failure and the source of the failure (Table 4-1).

TABLE 4-1
HK80/V960E error status latch encoding

Port address: 0210,0000$_{16}$. Size: Byte. Type: Read.

D2	D1	D0	Failure Code	Owner of Local Bus
0	0	0	Bus error	Unknown
0	0	1	Parity error	Unknown
0	1	0	Bus error	82596CA (Ethernet)
0	1	1	Parity error	82596CA (Ethernet)
1	0	0	Bus error	VIC068 (VME slave access)
1	0	1	Parity error	VIC068 (VME slave access)
1	1	0	Bus error	80960CA (MPU)
1	1	1	Parity error	80960CA (MPU)

4.2.2 Software Errors

Software errors are errors that are detected by the 80960CA and are all handled through the fault table as described in section 3.5. For a detailed description of methods the processor uses to deal with these errors, refer to section 3.5 ("MPU Faults") and the 80960CA user's manual.

CONDITION	SUBTYPE AND DEFINITION
Operation Faults	An operation fault indicates that the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics. ***Invalid opcode:*** The processor has detected an undefined opcode or addressing mode.

Unimplemented: The processor has attempted to execute an instruction that was fetched from on-chip RAM.

Unaligned: The processor has attempted to access an unaligned word or group of words in memory. This error can be disabled in the fault configuration word.

Invalid operand: The processor has attempted to execute an instruction for which one or more of the operands have special requirements that are not satisfied.

Arithmetic Faults

An arithmetic fault indicates that the processor has encountered a problem while attempting to execute an arithmetic operation.

Integer Overflow: The result of an integer instruction overflows the destination. This error can be disabled in the arithmetic controls register.

Zero Divide: A zero divide indicates that the divisor operand of a divide operation is zero.

Type Fault

The ***Type Mismatch*** fault indicates the processor has attempted to perform an illegal operation of an architecturally defined data type or a typed data structure. From user mode, attempts to execute the **modpc** instruction and attempts to access on-chip data RAM or a special function register generate this fault.

Protection fault

The ***Length*** fault indicates that the index in a call's instruction points to an entry beyond the extent of the system procedure table.

Constraint fault

The ***Privileged*** fault is generated when a program or procedure attempts to use a supervisor-only instruction from user mode. Privileged instructions are **sdma, udma,** and **sysctl.**

Parallel fault

The ***Parallel*** fault indicates that one or more faults occurred when the processor was executing instructions in parallel by different execution units. (Multiple faults can occur simultaneously because the processor can execute multiple instructions in parallel.) If this happens, a fault record is created for each fault that occurs.

5

On-card Memory Configuration

5.1 INTRODUCTION

The Heurikon HK80/V960E microcomputer accommodates a variety of RAM and ROM configurations. There is a single ROM socket for PROM, EPROM or EEPROM, 24 ZIP RAM positions, and a nonvolatile RAM. Off-card memory may be accessed via the VMEbus or the VSB.

5.2 ROM

The HK80/V960E's ROM is accessible during the initial power-up sequence until the ROMINH bit is set. When the ROMINH bit is cleared (reset state) ROM is mirrored throughout the highest 1 Mbyte of memory ($FFF0,0000_{16}$ – $FFFF,FFFF_{16}$). When the ROMINH bit is set, the highest 1 Mbyte becomes the VME extended memory space (see Table 5-1). Although execution out of ROM is impossible when ROMINH is set, it is still possible to access the real-time clock module.

TABLE 5-1
ROMINH value and ROM addresses

Port address : $0200,0040_{16}$. Size: Long. Type: Write.

D0	ROM Address Space
0	F000,0000 – FFF0,0000 is reserved. $FFF0,0000_{16}$ – $FFFF,FFFF_{16}$ is ROM.
1	$F000,0000_{16}$ – $FFFF,FFFF_{16}$ is VME extended space.

Associated with the ROM socket is a set of jumpers that must be set according to the type of ROM being used. The HK80/V960E supports EPROM sizes from 64 Kbit to 8 Mbit (2764 – 27080). The ROM size and associated configuration are shown in Figure 5-1:

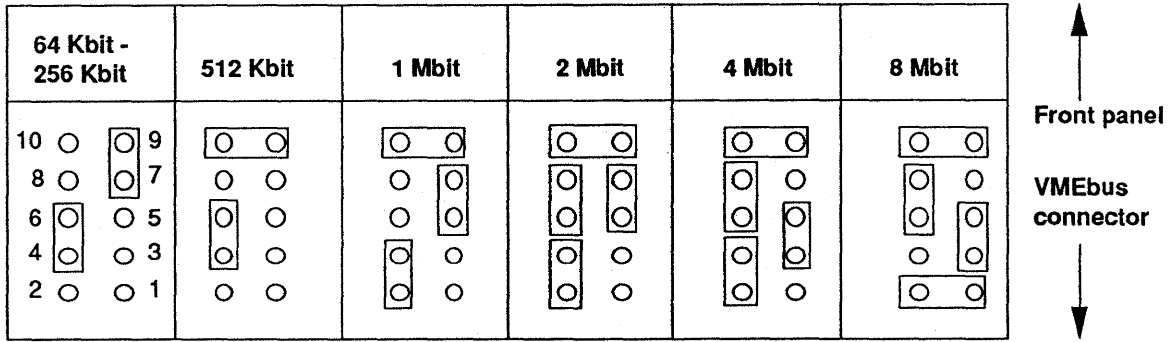


FIGURE 5-1. ROM capacity and jumper positions

Jumper J17 is for ROM (U15). See section 15.3 for help in locating the jumpers.

The ROM socket has 32 pins. When using a 28-pin device, justify it so that socket pins 1, 2, 31, and 32 are empty. Twenty-four-pin devices are not supported. The ROM access time must be ≤250 nanoseconds.

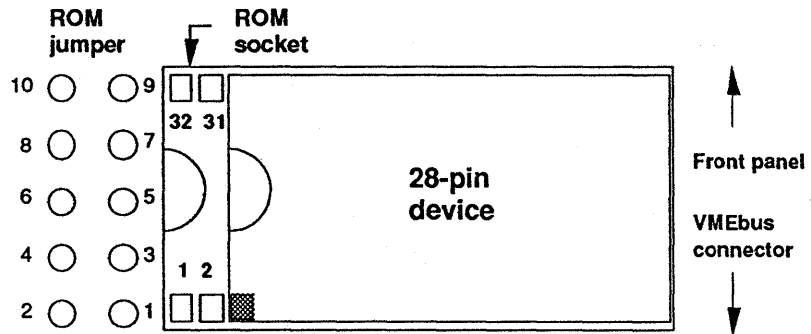


FIGURE 5-2. ROM positioning diagram

Note: If you make your own ROMs, keep in mind that no matter what size the ROM is (or where it is located), some part of it should be at the IBR (Initialization Boot Record) address $FFFF,FF00_{16}$. This address is a function of the 80960CA and cannot be changed. Therefore, the ROM must contain the IBR and it must be at this address (mirror or otherwise) from the 80960CA's point of view. Refer to the 80960CA user's manual for further details.

5.3 ON-CARD RAM

The HK80/V960E uses 24 ZIP RAM packages. Standard memory configurations are 2 or 8 Mbytes. On-card RAM occupies physical addresses starting at $0000{,}0400_{16}$. The first 400_{16} of memory is the on-chip DATA RAM of the 80960CA. The memory spaces are given in Table 5-2.

TABLE 5-2
HK80/V960E memory space

Memory Size	Memory Address Space
1-Kbyte data cache	$0000{,}0000_{16}$ – $0000{,}0400_{16}$
2 Mbytes	$0000{,}0400_{16}$ – $0020{,}0400_{16}$
6 Mbytes (optional)	$0020{,}0400_{16}$ – $00080{,}0400_{16}$

5.4 BUS MEMORY

See sections 6 and 7 for details concerning the VME/VSB bus interface.

5.5 PHYSICAL MEMORY MAP

See section 15.2 for an I/O device address summary.

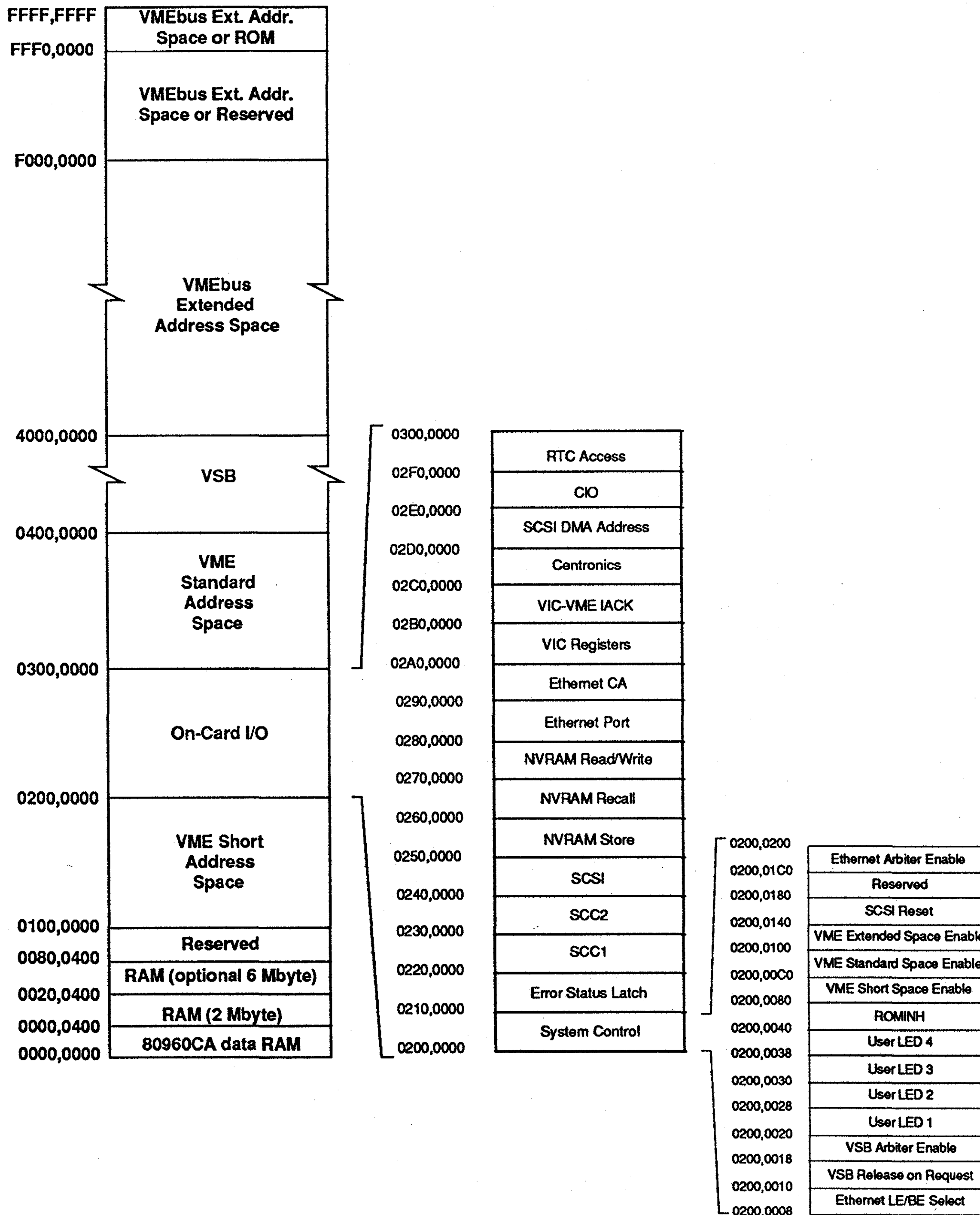


FIGURE 5-3. Physical memory map

5.6 MEMORY TIMING

The HK80/V960E memory logic has been carefully tuned to give optimal memory cycle times under a variety of conditions.

The base cycle time for an 80960CA is two clock cycles for a RAM read or write and one clock cycle for subsequent burst cycles. Although the 80960CA cannot perform memory accesses any faster than this, it can be made to perform slower accesses. Table 5-3 shows the base cycle times for the 80960CA for accesses with no wait states:

TABLE 5-3
80960CA clock cycles for zero wait states

Cycle	Number of Clocks
Reads	2
Writes	2
Burst Read (4 accesses)	5
Burst Write (4 accesses)	5

The HK80/V960E utilizes several features to provide the memory bandwidth the processor requires.

Reads The HK80/V960E provides a bank interleave memory structure that allows the concurrent access of adjacent long words in memory. The bank interleaving of read cycles allows the processor to achieve no wait states on the second, third, and fourth accesses of a burst read cycle at 33 MHz.

Writes The HK80/V960E also performs write posting of memory write cycles. This allows the processor to terminate the write cycle early, permitting the memory to complete the write cycle. With the combination of bank interleaving and write posting, the HK80/V960E can achieve no wait state cycles for writes and burst writes.

The use of bank interleaving and write posting provides the HK80/V960E with nearly no-wait-state performance. The only wait state occurs on read cycles and the first cycle of a read burst cycle.

Table 5-4 describes the expected wait states for the HK80/V960E:

TABLE 5-4
RAM access time required for the HK80/V960E

Cycle	Total Clock Cycles	Wait States
Reads	4	2
Writes	2	0
Burst Read (4 accesses)	7	2-0-0-0
Burst Write (4 accesses)	5	0-0-0-0

The HK80/V960E will provide either 2 or 8 Mbytes of memory using 70-nanosecond DRAMs.

There are two other sources of wait states that DRAM architectures can exhibit:

1. When a refresh must be performed and the DRAM controller is unable to perform the refresh during non-RAM cycles. This happens so infrequently that any performance degradation is usually unnoticeable.

2. When the processor is required to perform back-to-back memory cycles with no delays, which rarely occurs because of the instruction cache and data RAM. In such a case, single reads and writes would require five clock cycles, and burst reads and writes would require eight clock cycles.

While the above information is important in comparing the relative performance of DRAM designs, the performance of individual DRAM designs has much less impact on overall system performance than one might expect. The reason for this is that the internal cache and data RAM built into the 80960CA chip helps to decouple the processor from slower speed memories such as DRAMs.

To summarize, the higher the cache hit rates, the less impact external memory has on system performance.

5.7 NONVOLATILE RAM

A particularly useful feature of the HK80/V960E is its nonvolatile RAM (NV-RAM), which allows precious data and system configuration information to be stored and recovered across power cycles. The NV-RAM is configured as 8 Kbytes of 8-bit words (low byte of every other long word), of which 6 Kbytes are user-accessible (Table 5-5).

TABLE 5-5
Nonvolatile RAM addresses

Address	Mode	Size	Function
$0270{,}0000_{16}$ - $0271{,}0000_{16}$	Read	8 Kbytes	Readable portion of nonvolatile memory
$0270{,}0000_{16}$ - $0270{,}C000_{16}$	Write (read-modify-write)	6 Kbytes	Writeable portion of nonvolatile memory

To avoid destruction of nonvolatile memory by an errant program, a read-modify-write cycle is required to write the nonvolatile memory (ASM960 atomic modify **atmod**, ASM960 atomic add **atadd**). Examples of modifying nonvolatile memory can be found in Appendix A.

Physically, the nonvolatile memory is an 8-Kbyte-by-8-bit EEPROM (or equivalent). Reads from nonvolatile memory take about 300 nanoseconds. Writes to nonvolatile memory are much more time consuming; they take about 10 milliseconds. A write can be verified by continually reading the location until the expected value is returned. To reduce the write delays, the nonvolatile memory supports a burst mode, which allows the writing of a 32-byte block of memory at one time. The chip is rated for 10,000 write cycles per location.

The nonvolatile memory device has been partitioned into three sections, which are outlined in Figure 5-4 and further described in this section.

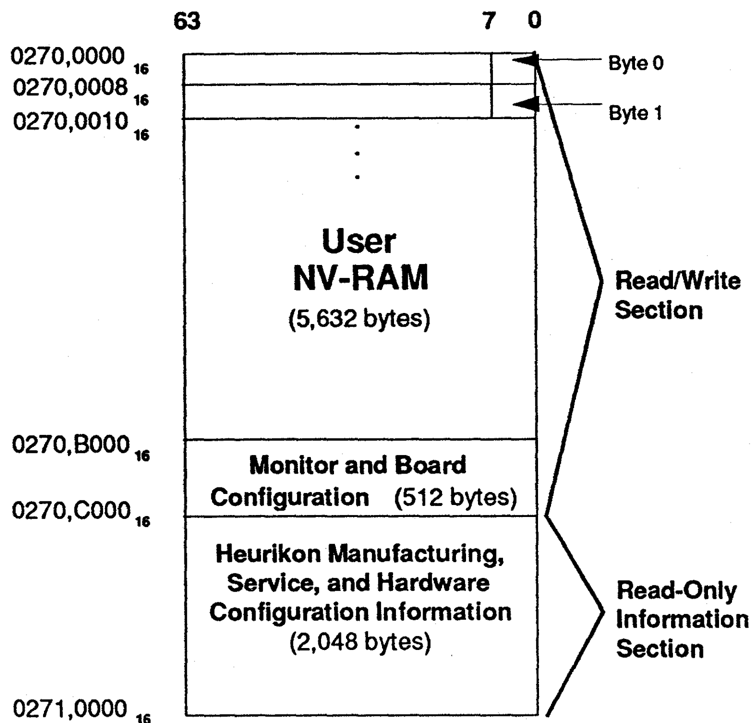


FIGURE 5-4. EEPROM partitions

The first section (5,632 bytes) is the user-configurable read/write section of the nonvolatile memory. This section can be modified by a user's application with no effect on the other sections.

The second section consists of 512 bytes of information for configuring the monitor and board upon reset. This section should not be modified by a user's application.

The last quarter (2,048 bytes) of the nonvolatile memory is reserved for Heurikon's use and contains manufacturing information, service information, and hardware configuration information. This region is hardware write protected and can only be written by Heurikon Corporation.

The HK80/V960E addresses nonvolatile memory so that 1 byte is mapped to every other long word location. On this basis, the first byte is located at 0270,0000, the second byte is at 0270,0008, the third byte at 0270,0010, and so forth.

We recommend that you use a function that reads portions of the nonvolatile memory into contiguous memory buffers for easy manipulation. See Appendix B for the definitions and contents of the Heurikon-defined structures. Also see Appendix A for programming examples for maintaining the nonvolatile memory structures.

6

VMEbus Control

6.1 INTRODUCTION

The HK80/V960E has a VMEbus interface that conforms to the specifications set forth in the following section. The VMEbus interface consists of the VIC068 VMEbus Interface Controller (VIC) and required support circuitry to perform all VMEbus functions. The control logic for the VMEbus allows numerous bus masters to share the resources on the bus. Up to 21 boards may be used on the VMEbus.

Please refer to the *VIC068 VMEbus Interface Controller Specification* from VTC for a detailed description of the VIC.

The HK80/V960E VME interface has the following features:

Address The VMEbus interface uses 32 address lines for a total of 4 Gbytes of VMEbus address space. Supported are the "short," "standard," and "extended" address modes, which use 16, 24, and 32 address lines, respectively.

Data The VMEbus interface uses 32 data lines to support 8-, 16-, 24-, or 32-bit data transfers.

Interrupts The VIC handles the seven VMEbus interrupts and multiple local interrupts.

Mailbox The mailbox consists of a collection of 8-bit registers that can be used for interprocessor communications over the VMEbus.

System Controller The HK80/V960E may be configured as the VMEbus system controller, and would perform the necessary system controller functions of SYSCLK, BCLR, SYSRESET, bus watchdog, and bus arbiter.

VSB

The HK80/V960E supports the VME subsystem bus (VSB) expansion interface, which allows high speed 8-, 16-, 24-, 32-bit data transfers without the need for the VME bus. See section 7 for details.

6.2 VMEbus SIGNAL DESCRIPTIONS

VME signals, described below, are defined on P1 and part of P2. VSB is defined on the rest of P2; VSB signal descriptions are described in section 7.

Refer to the Motorola VMEbus specification, revision C.1, for detailed usage of VME signals. All signals are bidirectional unless otherwise stated.

The following signals on connectors P1 and P2 are used for the VMEbus interface. For a complete listing of the pins, refer to section 6.11.

A01-A15

ADDRESS bus (bits 1-15). Three-state address lines that are used to broadcast a *short* address.

A16-A23

ADDRESS bus (bits 16-23). Three-state address lines that are used in conjunction with A01-A15 to broadcast a *standard* address.

A24-A31

ADDRESS bus (bits 24-31). Three-state address lines that are used in conjunction with A01-A23 to broadcast an *extended* address.

ACFAIL*

AC FAILURE. This signal is an input to the HK80/V960E and may be used to generate an interrupt to the 80960CA by programming the VIC accordingly.

AM0-AM5

ADDRESS MODIFIER (bits 0-5). Three-state lines that are used to broadcast information such as address size and cycle type.

AS*

ADDRESS STROBE. A three-state signal that indicates when a valid address has been placed on the address bus.

BBSY*

BUS BUSY. An open-collector signal driven low by the current MASTER to indicate that it is using the bus. When the MASTER releases this line, the resultant rising edge causes the ARBITER to sample the bus request lines and grant the bus to the high-

est priority requester. Early release mode is supported.

BCLR* BUS CLEAR. A totem-pole signal generated by the ARBITER to indicate when there is a higher priority request for the bus. This signal requests the current MASTER to release the bus.

BERR* BUS ERROR. An open-collector signal generated by a SLAVE or BUS TIMER. This signal indicates to the MASTER that the data transfer was not completed.

BG0IN*-BG3IN* BUS GRANT (0-3) IN. Totem-pole signals generated by the ARBITER and REQUESTERS. Bus-grant-in and bus-grant-out signals form bus grant daisy chains. An input to the HK80/V960E, the bus-grant-in signal indicates that the HK80/V960E may use the bus.

BG0OUT*-BG3OUT* BUS GRANT (0-3) OUT. Totem-pole signals generated by REQUESTERS. An output from the HK80/V960E, the bus-grant-out signal indicates to the next board in the daisy-chain that it may use the bus.

BR0*-BR3* BUS REQUEST (0-3). Open-collector signals generated by REQUESTERS. Assertion of one of these lines indicates that some MASTER needs to use the bus.

D00-D31 DATA BUS. Three-state bidirectional data lines used to transfer data between MASTERS and SLAVES.

DS0*, DS1* DATA STROBE ZERO, ONE. A three-state signal used in conjunction with LWORD* and A01 to indicate how many data bytes are being transferred (one, two, three, or four). During a write cycle, the falling edge of the first data strobe indicates that valid data are available on the data bus.

DTACK* DATA TRANSFER ACKNOWLEDGE. An open-collector signal generated by a SLAVE. The falling edge of this signal indicates that valid data are available on the data bus during a read cycle, or that data have been accepted from the data bus during a write cycle. The rising edge indicates when the SLAVE has released the data bus at the end of a READ CYCLE.

IACK* INTERRUPT ACKNOWLEDGE. An open-collector or three-state signal used by an INTERRUPT HANDLER acknowledging an

interrupt request. It is routed, via a backplane signal trace, to the IACKIN* pin of slot 1, where it forms the beginning of the IACKIN*-IACKOUT* daisy-chain.

IACKIN* INTERRUPT ACKNOWLEDGE IN. A totem-pole signal and an input to the HK80/V960E. The IACKIN* indicates that the board may respond to the INTERRUPT ACKNOWLEDGE CYCLE that is in progress.

IACKOUT* INTERRUPT ACKNOWLEDGE OUT. A totem-pole signal and an output from the HK80/V960E. The IACKIN* and IACKOUT* signals form a daisy-chain. The IACKOUT* signal indicates to the next board in the daisy-chain that it may respond to the INTERRUPT ACKNOWLEDGE CYCLE in progress.

IRQ1*-IRQ7* INTERRUPT REQUEST (1-7). Open-collector signals, generated by an INTERRUPTER, which carry interrupt requests. When several lines are monitored by a single INTERRUPT HANDLER, the highest numbered line is given the highest priority.

LWORD* LONG WORD. A three-state signal used in conjunction with DS0*, DS1*, and A01 to select which byte location(s) within the 4-byte group are accessed during the data transfer.

RESERVED RESERVED. A signal line reserved for future VMEbus enhancements. This line must not be used.

SERCLK SERIAL CLOCK. A totem-pole signal that is used to synchronize the data transmission on the VMEbus. This signal is not implemented on the HK80/V960E.

SERDAT* SERIAL DATA. An open-collector signal that is used for VMEbus data transmission. This signal is not implemented on the HK80/V960E.

SYSCLK SYSTEM CLOCK. A totem-pole signal that provides a constant 16-MHz clock signal that is independent of any other bus timing. This signal is driven if the HK80/V960E is a system controller.

SYSFAIL* SYSTEM FAIL. An open-collector signal that indicates a failure has occurred in the system. It is also used at power-on to indicate that at least one VMEbus board is still in its power-on initialization phase. This signal

may be generated by any board on the VMEbus. The VIC drives this signal low at power-up and may be programmed to generate an interrupt if asserted by another board in the system. Details are given in section 6.9.

SYSRESET* SYSTEM RESET. An open-collector signal that, when asserted, causes the system to be reset.

WRITE* WRITE. A three-state signal generated by the MASTER to indicate whether the data transfer cycle is a *read* or a *write*. A high level indicates a read operation; a low level indicates a write operation.

+5V STDBY +5 Vdc STANDBY. This line supplies +5 Vdc to devices requiring battery backup. This signal is not used on the HK80/V960E.

6.3 VIC REGISTER MAP

The base address of the VIC chip is $02A0,0000_{16}$.

Table 6-1 shows the VIC register offsets from the base. Please refer to the *VIC068 VMEbus Interface Controller Specification* from VTC for a detailed description of the VIC registers.

TABLE 6-1
VIC register map

Offset Address	Acronym	Register Name
0_{16}	VIICR	VMEbus Interrupter Interrupt Control Register
4_{16}	VICR1	VMEbus Interrupter Control Register 1
8_{16}	VICR2	VMEbus Interrupter Control Register 2
C_{16}	VICR3	VMEbus Interrupter Control Register 3
10_{16}	VICR4	VMEbus Interrupter Control Register 4
14_{16}	VICR5	VMEbus Interrupter Control Register 5
18_{16}	VICR6	VMEbus Interrupter Control Register 6
$1C_{16}$	VICR7	VMEbus Interrupter Control Register 7
20_{16}	DSICR	DMA Status Interrupt Control Register
24_{16}	LICR1	Local Interrupt Control Register 1
28_{16}	LICR2	Local Interrupt Control Register 2
$2C_{16}$	LICR3	Local Interrupt Control Register 3
30_{16}	LICR4	Local Interrupt Control Register 4
34_{16}	LICR5	Local Interrupt Control Register 5
38_{16}	LICR6	Local Interrupt Control Register 6
$3C_{16}$	LICR7	Local Interrupt Control Register 7
40_{16}	ICGSICR	ICGS Interrupt Control Register
44_{16}	ICMSICR	ICMS Interrupt Control Register
48_{16}	EGICR	Error Group Interrupt Control Register
$4C_{16}$	ICGSIVBR	ICGS Interrupt Vector Base Register
50_{16}	ICMSIVBR	ICMS Interrupt Vector Base Register
54_{16}	LIVBR	Local Interrupt Vector Base Register
58_{16}	EGIVBR	Error Group Interrupt Vector Base Register
$5C_{16}$	ICSR	Interprocessor Communications Switch Register
60_{16}	ICR0	Interprocessor Communications Register 0
64_{16}	ICR1	Interprocessor Communications Register 1
68_{16}	ICR2	Interprocessor Communications Register 2
$6C_{16}$	ICR3	Interprocessor Communications Register 3

Continues.

TABLE 6-1 — *Continued.*
VIC register map

70_{16}	ICR4	Interprocessor Communications Register 4
74_{16}	ICR5	Interprocessor Communications Register 5
78_{16}	ICR6	Interprocessor Communications Register 6
$7C_{16}$	ICR7	Interprocessor Communications Register 7
80_{16}	VIRSR	VMEbus Interrupt Request and Status Register
84_{16}	VIVR1	VMEbus Interrupt Vector Register 1
88_{16}	VIVR2	VMEbus Interrupt Vector Register 2
$8C_{16}$	VIVR3	VMEbus Interrupt Vector Register 3
90_{16}	VIVR4	VMEbus Interrupt Vector Register 4
94_{16}	VIVR5	VMEbus Interrupt Vector Register 5
98_{16}	VIVR6	VMEbus Interrupt Vector Register 6
$9C_{16}$	VIVR7	VMEbus Interrupt Vector Register 7
$A0_{16}$	TTR	Transfer Timeout Register
$A4_{16}$	LBTR	Local Bus Timing Register
$A8_{16}$	BTDR	Block Transfer Definition Register
AC_{16}	VICR1	VMEbus Interface Configuration Register 1
$B0_{16}$	ARCR	Arbiter and Requester Configuration Register
$B4_{16}$	AMSR	Address Modifier Source Register
$B8_{16}$	BESR	Bus Error Status Register
BC_{16}	DMASR	DMA Status Register
$C0_{16}$	SS0CR0	Slave Select 0 Control Register 0
$C4_{16}$	SS0CR1	Slave Select 0 Control Register 1
$C8_{16}$	SS1CR0	Slave Select 1 Control Register 0
CC_{16}	SS1CR1	Slave Select 1 Control Register 1
$D0_{16}$	RCR	Release Control Register
$D4_{16}$	BTCR	Block Transfer Control Register
$D8_{16}$	BTLR0	Block Transfer Length Register 0
DC_{16}	BTLR1	Block Transfer Length Register 1
$E0_{16}$	SYSRR	System Reset Register
$E4_{16}$		Undefined
$E8_{16}$		Undefined
EC_{16}		Undefined
$F0_{16}$		Undefined
$F4_{16}$		Undefined
$F8_{16}$		Undefined
FC_{16}		Undefined

6.4 VMEbus INTERRUPTS

VMEbus interrupt generation and handling capability is provided by the VIC chip. The following features are included:

- Conformance to the Motorola VMEbus specification revision C.1
- The capability to interrupt other boards on the VMEbus using any of the seven VMEbus interrupt levels
- The capability to generate interrupts on multiple levels at the same time
- The capability to intercept VMEbus, VIC, and on-board interrupts and provide an interrupt to the MPU
- Capability to provide vectors for VIC and local interrupts
- A timer interrupt

The seven VMEbus interrupts are monitored and controlled by the VIC chip (as shown in Fig. 6-1). An interrupt to the 80960CA can be generated when a desired bus interrupt signal is on. There are two functions described below. The *interrupter* generates bus interrupts; the *interrupt handler* receives interrupts from the bus.

For details on the VIC processor, read the *VIC068 VMEbus Interface Controller Specification* by VTC Incorporated.

6.4.1 Interrupter Operation

The VIC may assert interrupt requests on the VMEbus at all of the seven interrupt levels. It may generate interrupt requests on multiple levels simultaneously.

Interrupt generation is programmed through the VMEbus interrupt request/status register (VIRSR) of the VIC processor. This register allows each interrupt to be set and reset by writing a 1 or a 0 to the corresponding bit in the register.

The VIC068 also includes seven VMEbus interrupt vector registers (VIVR1-VIVR7) that must be initialized before the interrupt is turned on. When a VMEbus interrupt is acknowledged, an internal interrupt can be generated to complete the handshake without polling the VMEbus interrupt request and status register (VIRSR) for the acknowledge of an interrupt. The local interrupt for acknowledges is programmed using both the VMEbus interrupter interrupt control register (VIICR) and the error group interrupt vector base register (EGIVBR).

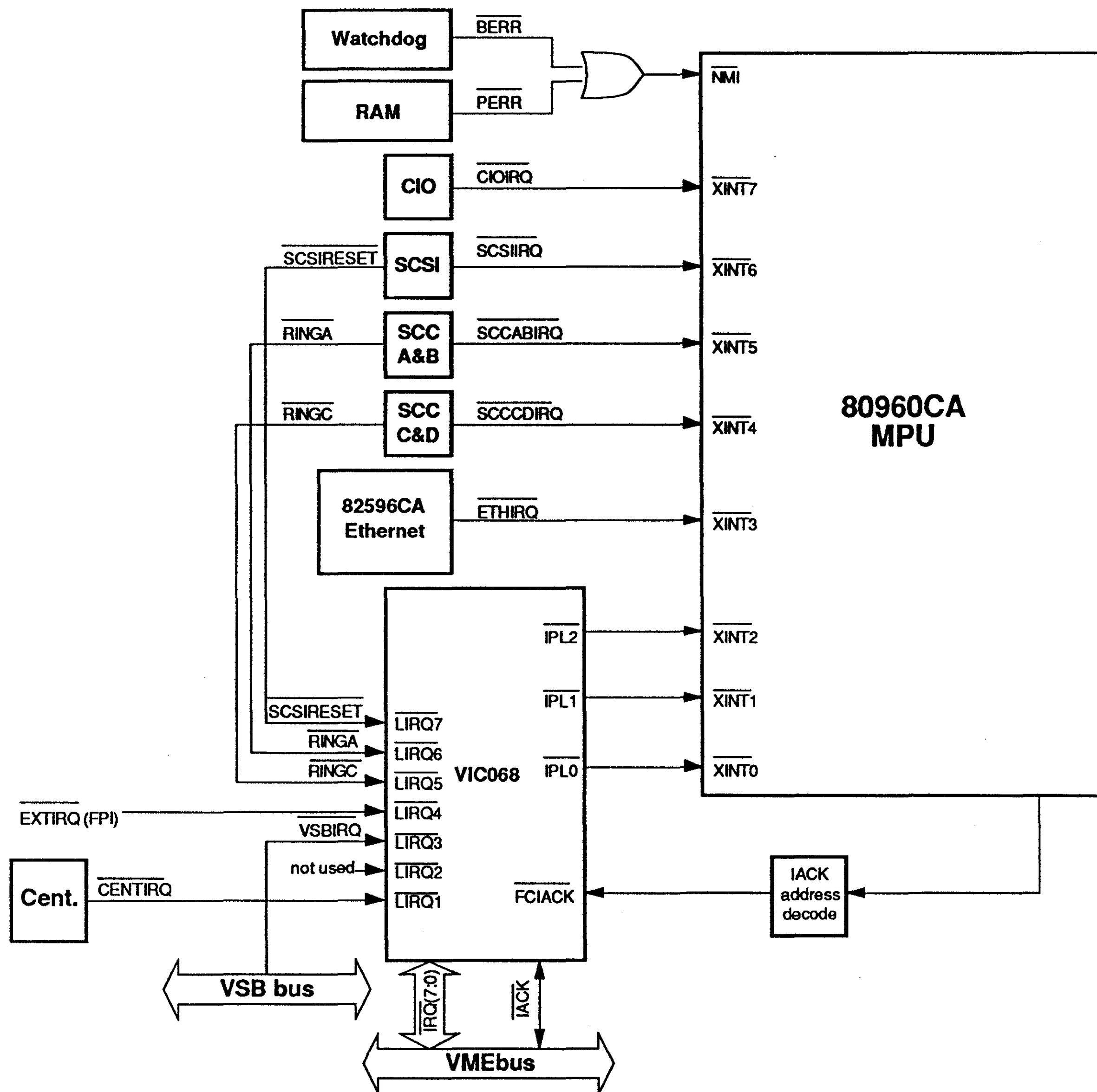


FIGURE 6-1. HK80/V960E interrupt architecture

6.4.2 Interrupt Handler Operation

The VIC controller handles all VMEbus interrupts (IRQ1*-IRQ7*) and some local interrupts (see Table 6-3).

6.4.2.1 VIC Interrupt Requests

VIC interrupts are presented to the 80960CA on three lines, IPL0-IPL2. The VIC chip can be programmed to present one of

seven priority levels on the IPL lines. The 80960CA interrupt controller treats these lines as dedicated interrupt requests.

Note: Because the IPL lines are interpreted by the 80960CA as nonencoded interrupt requests, the VIC *must never* be programmed to assert two IPL lines for any interrupt source. In other words, the VIC should be programmed to drive IPL2-0 values of 01_{16}, 02_{16}, and 04_{16} corresponding to IPL0, IPL1, and IPL2, respectively. Programming any other values into the VIC registers will result in unpredictable program behavior that might prove very difficult to debug.

6.4.2.2 VIC Interrupt Acknowledges

The VIC indicates an interrupt condition to the processor on either IPL2, IPL1, or IPL0 and is received by the processor on XINT2, XINT1, and XINT0, respectively. When the 80960CA has detected an interrupt and the correct interrupt handler is executed, it is the responsibility of the interrupt handler to remove the VIC interrupt request by reading the interrupt vector. One interrupt acknowledge address is associated with each interrupt line (shown in Table 6-2). These IACK addresses are used for on-card interrupt sources from the VIC and VME interrupts. Thus, during a VME interrupt cycle, reading these addresses will cause the VIC to fetch the STATUS/ID and perform an IACK cycle on the VMEbus.

TABLE 6-2
VIC interrupt lines and associated acknowledge addresses

IPL(2:0)	Interrupt Source	Interrupt Acknowledge Address
01_{16}	IPL0	8-bit vector (STATUS/ID) at $02B0,0010_{16}$
02_{16}	IPL1	8-bit vector (STATUS/ID) at $02B0,0004_{16}$
04_{16}	IPL2	8-bit vector (STATUS/ID) at $02B0,0008_{16}$

Reading an interrupt acknowledge address causes an 8-bit vector to be read and the interrupt to be removed. An attempt to read a vector when no interrupt is present results in a bus error.

The VIC can be programmed to generate interrupts to the 80960CA for the following sources:

- **Error Group Interrupts:** Refer to the VIC error group control register (EGICR) and the error group interrupt vector base register (EGIVBR).

ACFAIL: If a power failure module is installed on the VMEbus backplane, the VIC may be programmed to generate an interrupt if a power failure occurs (that is, VMEbus ACFAIL* asserted).

SYSFAIL: The VIC may be programmed to generate an interrupt when a system failure is indicated (that is, VMEbus SYSFAIL* asserted).

Arbitration timeout: When the VIC times out on arbitration, the VIC can be programmed to generate an interrupt.

Write posted cycle failure: If a write cycle that was posted by the processor fails, the processor is notified by this interrupt.

- **Local Interrupts (LIRQ7-LIRQ0)**: The VIC can be programmed to generate interrupts for SCSI Resets, Ring Detection on SCC ports A and C, front panel interrupt requests, and VSB interrupt requests. Refer to the VIC local interrupt control registers (LICR1-LICR7) and local interrupt vector base register (LIVBR).

- **ICGS Group Interrupts**: The interprocessor communications global switches (ICGS) allow other VMEbus boards to interrupt the HK80/V960E for global events. Refer to the VIC ICGS interrupt control register (ICGSICR) and ICGS interrupt vector base register (ICGSIVBR).

- **ICMS Group Interrupts**: The interprocessor communications module switches (ICMS) allow other VMEbus boards to interrupt the HK80/V960E for HK80/V960E-specific events. Refer to the VIC ICMS interrupt control register (ICMSICR) and ICMS interrupt vector base register (ICMSIVBR).

- **VMEbus Interrupts (IRQ7-IRQ0)**: The VIC can be programmed to receive and generate the seven VMEbus interrupts with STATUS/ID (vector) information. Refer to the VIC VMEbus interrupt control registers (VICR1-VICR7) and VMEbus interrupt vector registers (VIVR1-VIVR7).

- **DMA Status/Complete Interrupt**: If this interrupt is enabled, the VIC generates an interrupt if either the DMA completes, or a BERR occurs (local *or* VME), during the DMA transfer. Refer to the DMA status interrupt control register (DSICR) and error group interrupt vector base register (EGIVBR).

- **VME interrupter handshake**: When a VMEbus interrupt generated by the HK80/V960E is acknowledged, this interrupt can be used to indicate the acknowledge has taken place. Refer to the VMEbus interrupter interrupt control register (VIICR) and the error group interrupt vector base register (EGIVBR).

Interrupts are internally prioritized, as shown in the following table.

TABLE 6-3
Interrupt priorities

Rank	Interrupt
19	LIRQ7 (SCSI Reset)
18	Error Group Interrupt
17	LIRQ6 (Ring Detect Port A)
16	LIRQ5 (Ring Detect Port C)
15	LIRQ4 (External Interrupt FPI)
14	LIRQ3 (VSB Interrupt Request)
13	LIRQ2 (not used)
12	LIRQ1 (Centronics Interrupt Request)
11	ICGS Group Interrupt
10	ICMS Group Interrupt
9	IRQ7 (VME Interrupt Request Level 7)
8	IRQ6 (VME Interrupt Request Level 6)
7	IRQ5 (VME Interrupt Request Level 5)
6	IRQ4 (VME Interrupt Request Level 4)
5	IRQ3 (VME Interrupt Request Level 3)
4	IRQ2 (VME Interrupt Request Level 2)
3	IRQ1 (VME Interrupt Request Level 1)
2	DMA Status/Complete Interrupt
1	VME Interrupt Acknowledged

Vector base registers are provided for each of the following groups of interrupts:

- ICGS
- ICMS
- Local interrupts
- Error interrupts

If the interrupt source is a VME interrupt, then the VIC latches the STATUS/ID (vector) onto the local bus during the local IACK cycle.

6.5 MAILBOX INTERFACE

Interprocessor communication (also known as mailbox) is provided by the VMEbus Interface Controller (VIC) processor. This section provides a brief description of the interprocessor communications facilities of the HK80/V960E. For a detailed description, read the VIC068 specification.

The mailbox interface consists of a collection of 8-bit registers and memory locations that can be used for communications with the HK80/V960E through the VMEbus. A description of the VIC registers follows:

VIC REGISTER	DESCRIPTION
ICR4-ICR0	Five general-purpose, dual-port "Interprocessor Communications" registers, which can be accessed from the VMEbus and from the HK80/V960E local bus. These registers are 8 bits wide and each has an associated semaphore bit in ICR7.
ICR5-ICR7	Four global switch registers. **ICR5** ICR5 is a VIC-specific register that specifies the revision level of the VIC. **ICR6** ICR6 is read only from the VMEbus and provides the status of the HK80/V960E. **ICR7** ICR7 is a dual-port register accessible from the VMEbus and the HK80/V960E local bus. This register provides semaphore bits for ICR0-ICR4, status of the HK80/V960E, and a means for remote resetting of the HK80/V960E.
ICGS0-ICGS3	Four interboard communications "global switch" registers, which are used to generate interrupts for global events.
ICMS0-ICMS3	Four interboard communications "module switch" registers, which are used to generate interrupts for V960E-specific events.

The local bus register addresses are shown in Table 6-1. The VMEbus mailbox structure is shown in Figure 6-1.

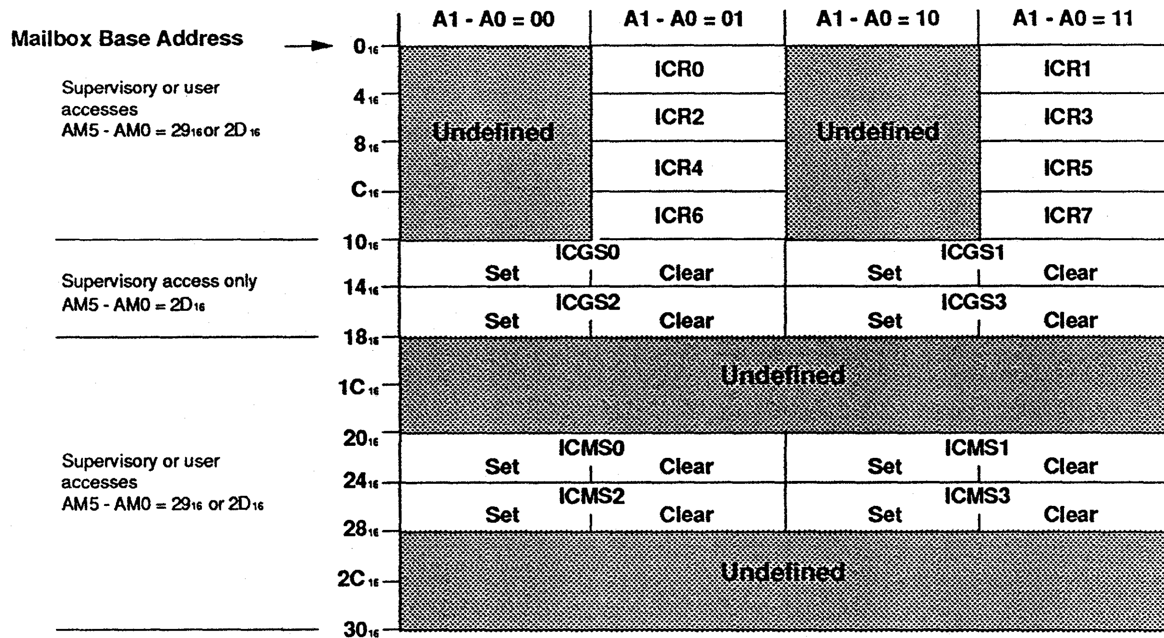


FIGURE 6-2. VME mailbox structure

All accesses are defined as 8-bit, and accesses to undefined areas may result in a bus error. An access from the VMEbus to the appropriate address in the VMEbus short space results in the VIC's responding (as a slave) to the access.

The VMEbus mailbox can be mapped to any of 256 256-byte boundaries within the VMEbus *short* addressing space. All registers are accessible from the supervisory short space (AM5 - AM0 = $2D_{16}$) and all but the global switches are accessible from the user short space (AM5 - AM0 = 29_{16}). The mailbox interface is enabled by writing a 1 to $0200,0080_{16}$ and disabled by writing a 0 (see Table 6-4).

TABLE 6-4
Mailbox enable

Port address: $0200,0080_{16}$. Size: Long. Type: Write.	
D0	**Function**
0	Mailbox disabled (default).
1	Mailbox enabled.

The mailbox base address is an 8-bit value stored in a latch that is compared to address lines A15-A8 on a VMEbus short space access (Table 6-5). The mailbox base address is stored in port B of the CIO. When the CIO has been initialized properly, the

mailbox base is modified by writing to $02E0,0008_{16}$ (port B of CIO). Appendix A contains an example of initializing the CIO for the HK80/V960E.

TABLE 6-5
HK80/V960E "short" space slave mapping on VMEbus (mailbox)

CIO Port B Compare Address	Mailbox Base Address for 2- and 8-Mbyte HK80/V960E
00_{16}	$XXXX,0000_{16}$
01_{16}	$XXXX,0100_{16}$
02_{16}	$XXXX,0200_{16}$
03_{16}	$XXXX,0300_{16}$
. . .	. . .
$0F_{16}$	$XXXX,0F00_{16}$
10_{16}	$XXXX,1000_{16}$
11_{16}	$XXXX,1100_{16}$
12_{16}	$XXXX,1200_{16}$
13_{16}	$XXXX,1300_{16}$
. . .	. . .
FF_{16}	$XXXX,FF00_{16}$

Also see Figure 6-2, "VME Mailbox Structure."

6.6 VMEbus SYSTEM CONTROLLER

Nearly all VMEbus operations of the HK80/V960E are handled by the VMEbus Interface Controller processor (VIC068). The VIC processor can be jumpered to provide the VMEbus system controller functions via jumper J10 (see Table 6-6).

As the system controller, the VIC drives Sysclk (SYSCLK), Bus Clear (BCLR), and System Reset (SYSRESET). The system controller also provides the system bus arbitration in one of three modes: "prioritized," "round robin," and "single level" arbitration. See the VIC arbiter and requester configuration register (ARCR). If configured as the system controller, the VIC also monitors the VMEbus interface as a watchdog timer (with a programmable time-out; see section 6.10).

The VIC processor is configured as the system controller by installing jumper J10 (Table 6-6). When the HK80/V960E is

configured as the system controller, it *must* be installed in slot 1 with a programmable time-out.

TABLE 6-6
Bus control jumpers

Jumper	Function
J10	System Controller Enable — When this jumper is installed, the HK80/V960E acts as a VMEbus system controller as described in the VIC User's Manual.

NOTE: Only one board in a VME system should be system controller.

6.7 VMEbus MASTER INTERFACE

The HK80/V960E can access the VMEbus with any of the three address modes "short," "standard," and "extended" on any of the four bus request levels. Refer to the VIC registers — arbiter and requester configuration register (ARCR) and the address modifier source register (AMSR). Short addresses use 16 address lines to specify a target address. Standard addresses use 24 address lines, and extended addresses use all 32 address lines. Table 6-7 shows the relationship between the on-card physical address and the corresponding VMEbus and VSB regions.

TABLE 6-7
Relationship of physical addresses to VMEbus and VSB memory regions

On-card Addresses	Bus Address	Memory Region
$0100{,}0000_{16}$ – $0100{,}FFFF_{16}$	0000_{16} – $FFFF_{16}$	VMEbus short address space
$0300{,}0000_{16}$ – $03FF{,}FFFF_{16}$	$00{,}0000_{16}$ – $FF{,}FFFF_{16}$	VMEbus standard address space
$0400{,}0000_{16}$ – $3FFF{,}FFFF_{16}$	$0400{,}0000_{16}$ – $3FFF{,}FFFF_{16}$	VSB address space
$4000{,}0000_{16}$ – $FFFF{,}FFFF_{16}$	$4000{,}0000_{16}$ – $FFFF{,}FFFF_{16}$	VMEbus extended address space

NOTE: $F000{,}0000_{16}$ – $FFFF{,}FFFF_{16}$ is not available to VME if ROMINH = 0. See Table 5-1.

Extended VME addresses from $0000{,}0000_{16}$ to $4000{,}0000_{16}$ are not accessible. The region from $0400{,}0000_{16}$ to $4000{,}0000_{16}$ is the only accessible VSB region.

The VMEbus master release modes are programmed by writing to the RCR (release control register) of the VIC chip. If the HK80/V960E is the bus master when the requested bus operation is completed, the bus will be released according to the state contained in the RCR register. The release mode is either:

ROR — Release-on-request will release the VMEbus (BBSY*) when a request is detected and there are no HK80/V960E bus requests.

RWD — Release-when-done will release the bus when there are no further HK80/V960E bus requests.

ROC — Release-on-clear will retain the bus until BCLR* has been asserted by the system controller.

6.8 VMEbus SLAVE INTERFACE

The HK80/V960E can be accessed from the VMEbus in both "extended" and "standard" space. "Short" space is used for the mailbox only. The slave logic for each space is enabled or disabled by writing to the appropriate address.

6.8.1 Extended Space

For the HK80/V960E to respond to a VMEbus extended address, the following steps *must* be taken:

1. The VIC register SS1CR0 (slave select one, control register 0) must be configured to respond to A32/D32 types of cycles (bits 2, 3, and 4 must be set to 100_2).

2. The extended space compare address must be written to port A of the CIO.

3. The extended space enable at $0200,0100_{16}$ must be set. (See Table 6-8.)

TABLE 6-8
Slave "extended" space enable

Port address: $0200,0100_{16}$. Size: Long. Type: Write.

DO	Function
0	Extended space disabled (default).
1	Extended space enabled.

The slave extended space compare address can map the HK80/V960E's RAM to one of 256 16-Mbyte boundaries. The compare address is stored in port A of the CIO and is compared to the VMEbus address lines A31-A24 (Table 6-9). When the CIO is initialized correctly, the slave compare address is modified by writing to $02E0,0010_{16}$ (CIO port A). For detailed instructions for initializing the CIO, refer to section 9 and Appendix A. When the HK80/V960E is selected as a slave in the extended space, all on-card RAM is mapped to the bus, starting at the base of the 16-Mbyte region that corresponds to the slave compare address.

TABLE 6-9
HK80/V960E "extended" space slave mapping on VMEbus

	VMEbus Address		**HK80/V960E Memory Mapped to Bus**	
CIO Port A Compare Address	**2-Mbyte HK80/V960E**	**8-Mbyte HK80/V960E**	**2-Mbyte HK80/V960E**	**8-Mbyte HK80/V960E**
00_{16}	$0000,0000_{16}$ – $001F,FFFF_{16}$	$0000,0000_{16}$ – $007F,FFFF_{16}$	2 Mbytes	8 Mbytes
01_{16}	$0100,0000_{16}$ – $011F,FFFF_{16}$	$0100,0000_{16}$ – $017F,FFFF_{16}$	2 Mbytes	8 Mbytes
02_{16}	$0200,0000_{16}$ – $021F,FFFF_{16}$	$0200,0000_{16}$ – $027F,FFFF_{16}$	2 Mbytes	8 Mbytes
03_{16}	$0300,0000_{16}$ – $031F,FFFF_{16}$	$0000,0000_{16}$ – $037F,FFFF_{16}$	2 Mbytes	8 Mbytes
. . .	. . .	. . .	. . .	. . .
$0F_{16}$	$0F00,0000_{16}$ – $0F1F,FFFF_{16}$	$0F00,0000_{16}$ – $0F7F,FFFF_{16}$	2 Mbytes	8 Mbytes
10_{16}	$1000,0000_{16}$ – $101F,FFFF_{16}$	$1000,0000_{16}$ – $107F,FFFF_{16}$	2 Mbytes	8 Mbytes
11_{16}	$1100,0000_{16}$ – $111F,FFFF_{16}$	$1100,0000_{16}$ – $117F,FFFF_{16}$	2 Mbytes	8 Mbytes
12_{16}	$1200,0000_{16}$ – $121F,FFFF_{16}$	$1200,0000_{16}$ – $127F,FFFF_{16}$	2 Mbytes	8 Mbytes
13_{16}	$1300,0000_{16}$ – $131F,FFFF_{16}$	$1300,0000_{16}$ – $137F,FFFF_{16}$	2 Mbytes	8 Mbytes
. . .	. . .	. . .	. . .	. . .
FF_{16}	$FF00,0000_{16}$ – $FF1F,FFFF_{16}$	$FF00,0000_{16}$ – $FF7F,FFFF_{16}$	2 Mbytes	8 Mbytes

6.8.2 Standard Space

For the HK80/V960E to respond to a VMEbus standard address, the following steps *must be taken:*

1. VIC register SS0CR0 (slave select 0, control register 0) must be configured to respond to A24/D32 types of cycles (bits 2, 3, and 4 must be set to 101_2).

2. The standard space compare address must be written to port C of the CIO.

3. The standard space enable at $0200,00C0_{16}$ must be set. (See Table 6-10.)

TABLE 6-10
Slave "standard" space enable

Port address: $0200,00C0_{16}$. Size: Long. Type: Write.	
D0	**Function**
0	Standard space disabled (default).
1	Standard space enabled.

The slave standard space compare address can map 1 Mbyte of the internal RAM to one of 16 1-Mbyte boundaries. The compare address for the standard space is stored in port C of the CIO (4 bits only) and is compared to VMEbus address lines A23-A20. When the CIO is initialized correctly, the slave compare address is modified by writing to $02E0,0000_{16}$ (CIO port C). When the HK80/V960E is selected as a slave in the standard space, 1 Mbyte of internal RAM is mapped to the bus, as described in Table 6-11:

TABLE 6-11
HK80/V960E "standard" space slave mapping on VMEbus

		HK80/V960E Memory Mapped to Bus	
CIO Port C Compare Address	**VMEbus Address for 2- and 8-Mbyte HK80/V960E**	**2-Mbyte HK80/V960E**	**8-Mbyte HK80/V960E**
0_{16}	$XX00,0000_{16}$ – $XX0F,FFFF_{16}$	1st Mbyte	1st Mbyte
1_{16}	$XX10,0000_{16}$ – $XX1F,FFFF_{16}$	2nd Mbyte	2nd Mbyte
2_{16}	$XX20,0000_{16}$ – $XX2F,FFFF_{16}$	1st Mbyte	3rd Mbyte
3_{16}	$XX30,0000_{16}$ – $XX3F,FFFF_{16}$	2nd Mbyte	4th Mbyte
4_{16}	$XX40,0000_{16}$ – $XX4F,FFFF_{16}$	1st Mbyte	5th Mbyte
5_{16}	$XX50,0000_{16}$ – $XX5F,FFFF_{16}$	2nd Mbyte	6th Mbyte
6_{16}	$XX60,0000_{16}$ – $XX6F,FFFF_{16}$	1st Mbyte	7th Mbyte
7_{16}	$XX70,0000_{16}$ – $XX7F,FFFF_{16}$	2nd Mbyte	8th Mbyte
8_{16}	$XX80,0000_{16}$ – $XX8F,FFFF_{16}$	1st Mbyte	1st Mbyte
9_{16}	$XX90,0000_{16}$ – $XX9F,FFFF_{16}$	2nd Mbyte	2nd Mbyte
A_{16}	$XXA0,0000_{16}$ – $XXAF,FFFF_{16}$	1st Mbyte	3rd Mbyte
B_{16}	$XXB0,0000_{16}$ – $XXBF,FFFF_{16}$	2nd Mbyte	4th Mbyte
C_{16}	$XXC0,0000_{16}$ – $XXCF,FFFF_{16}$	1st Mbyte	5th Mbyte
D_{16}	$XXD0,0000_{16}$ – $XXDF,FFFF_{16}$	2nd Mbyte	6th Mbyte
E_{16}	$XXE0,0000_{16}$ – $XXEF,FFFF_{16}$	1st Mbyte	7th Mbyte
F_{16}	$XXF0,0000_{16}$ – $XXFF,FFFF_{16}$	2nd Mbyte	8th Mbyte

6.8.3 Short Space

Refer to section 6.5 ("Mailbox Interface") for information on short space.

6.9 SYSFAIL CONTROL

The SYSFAIL line is controlled by the VIC processor. It is both an input and an open-collector output.

As an *output,* the VIC asserts the SYSFAIL line after power-up, and it remains asserted until self-tests and diagnostics are complete. It can then be removed (or set) by setting control bits in Interprocessor Communication Registers 6 and 7 (that is, ICR6 and ICR7). See the VIC manual for further details. At power-up, all other boards in the system should also do the same; that is, they should assert SYSFAIL until their diagnostics are complete. Once all boards are initialized, SYSFAIL should not be asserted by any board, except to indicate a failure of some kind.

As an *input,* the SYSFAIL line is used to indicate a system failure. If the VIC detects SYSFAIL asserted and if the VIC "Error Group Interrupt" is enabled, and the Error Group Interrupt Control Register (EGICR) has the SYSFAIL interrupt enabled, then the processor will be interrupted, and the Error Group Interrupt Vector Base Register (EGIVBR) will indicate a SYSFAIL interrupt.

6.10 VMEbus AND LOCAL BUS WATCHDOG TIMERS

All local accesses and accesses to the VMEbus are monitored by the VIC chip. The VIC chip has two timers; one for the local bus (default is 32 microseconds) and one for the VMEbus (default is 64 microseconds). These values may be changed via the VIC Transfer Timeout Register (TTR). The local timer defaults to being on, while the VMEbus timer is only on if the VIC is configured as the system controller.

If the *VMEbus timer* expires, BERR is asserted (on-card), which will issue an NMI (nonmaskable interrupt) to the 80960CA and drive BERR on VMEbus.

If the *local timer* expires, BERR is asserted (on-card only), which will issue an NMI (Non-Maskable Interrupt) to the 80960CA.

Note: If the timer values are changed to "infinite," the corresponding bus will hang indefinitely if a nonexistent or unresponding location is accessed.

6.11 VMEbus INTERFACE

The VMEbus interface consists of P1 and P2. P1 is used for most of the VME address, data, and control lines. P2 is used for the *extended* VME address and data lines, and all of the VSB lines. The VSB is described in section 7.

6.11.1 VMEbus PIN ASSIGNMENTS, P1

Not all of the P1 signals are used on the HK80/V960E. See section 6.1 and 6.2 for details and signal descriptions.

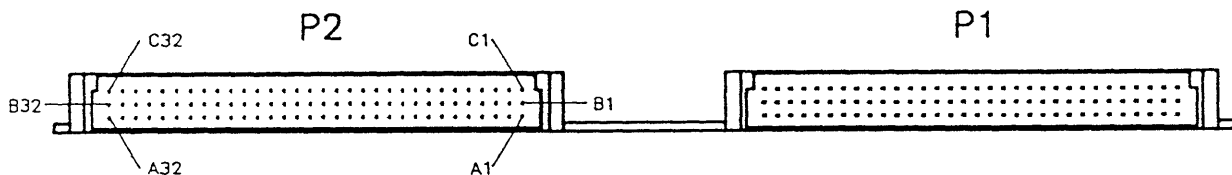


FIGURE 6-3. P1 and P2 VMEbus and VSB connectors

TABLE 6-12
VMEbus connector pin assignments, P1

P1 Pin Number	Row A Signal Mnemonic	Row B Signal Mnemonic	Row C Signal Mnemonic
1	D00	BBSY*	D08
2	D01	BCLR*	D09
3	D02	ACFAIL*	D10
4	D03	BG0IN*	D11
5	D04	BG0OUT*	D12
6	D05	BG1IN*	D13
7	D06	BG1OUT*	D14
8	D07	BG2IN*	D15
9	Gnd	BG2OUT*	Gnd
10	SYSCLK	BG3IN*	SYSFAIL*
11	Gnd	BG3OUT	BERR*
12	DS1*	BR0*	SYSRESET*
13	DS0*	BR1*	LWORD*
Continues.			

TABLE 6-12 — *Continued.*
VMEbus connector pin assignments, P1

P1 Pin Number	Row A Signal Mnemonic	Row B Signal Mnemonic	Row C Signal Mnemonic
14	WRITE*	BR2*	AM5
15	Gnd	BR3*	A23
16	DTACK*	AM0	A22
17	Gnd	AM1	A21
18	AS*	AM2	A20
19	Gnd	AM3	A19
20	IACK*	Gnd	A18
21	IACKIN*	SERCLK	A17
22	IACKOUT*	SERDAT*	A16
23	AM4	Gnd	A15
24	A07	IRQ7*	A14
25	A06	IRQ6*	A13
26	A05	IRQ5*	A12
27	A04	IRQ4*	A11
28	A03	IRQ3*	A10
29	A02	IRQ2*	A09
30	A01	IRQ1*	A08
31	-12V	+5V STDBY	+12V
32	+5V	+5V	+5V

6.11.2 VMEbus and VSB PIN ASSIGNMENTS, P2

P2 is used for both the VMEbus and the VSB. The center row of pins (row B) are the upper address and data lines of the VMEbus. The outer two rows (A and C) make up the VSB.

The use of P2 is *required* in order to meet VME power specifications.

TABLE 6-13
VMEbus and VSB connector pin assignments, P2

P2 Pin Number	Row A VSB Signal Mnemonic	Row B VMEbus Signal Mnemonic	Row C VSB Signal Mnemonic
1	AD00	+5	AD01
2	AD02	Gnd	AD03
3	AD04	(reserved)	AD05
4	AD06	A24	AD07
5	AD08	A25	AD09
6	AD10	A26	AD11
7	AD12	A27	AD13
8	AD14	A28	AD15
9	AD16	A29	AD17
10	AD18	A30	AD19
11	AD20	A31	AD21
12	AD22	Gnd	AD23
13	AD24	+5	AD25
14	AD26	D16	AD27
15	AD28	D17	AD29
16	AD30	D18	AD31
17	Gnd	D19	Gnd
18	IRQ*	D20	Gnd
19	DS*	D21	Gnd
20	WR*	D22	Gnd
21	SPACE0	D23	SIZE0
22	SPACE1	Gnd	PAS*
23	LOCK*	D24	SIZE1
24	ERR*	D25	Gnd
25	Gnd	D26	ACK*
26	Gnd	D27	AC
27	Gnd	D28	ASACK1*
28	Gnd	D29	ASACK0*
29	Gnd	D30	CACHE*
30	Gnd	D31	WAIT*
31	BGIN*	Gnd	BUSY*
32	BREQ*	+5	BGOUT*

7

VME Subsystem Bus (VSB) Control

7.1 INTRODUCTION

The VSB is a local bus extension designed for high-speed access to memory or other facilities without the need to use the VMEbus. The HK80/V960E operates on the VSB in master or secondary modes only; it cannot operate as a slave. It has the required arbitration logic to handle multiple VSB masters. The VSB is a super-set of the VMX32bus; VMX32bus slaves may be used.

7.2 VME SUBSYSTEM BUS (VSB) SIGNAL DESCRIPTIONS

The following signals on connector P2 are used for the VSB interface. For a complete listing of the P2 pin assignments, refer to section 7.5.

AD00-31 MULTIPLEXED ADDRESSED/DATA LINES. The three-state multiplexed address/data path (32 lines) that is controlled by the three-state drivers on the master and slave devices.

PAS* VSB ADDRESS STROBE. A three-state line in which the falling edge indicates that a valid address is present on AD31-AD00.

SPACE0-SPACE1 VSB ADDRESS SPACE SELECT. Three-state signals that select one of four address spaces or signify an interrupt acknowledge or parallel arbitration cycle. On the HK80/V960E, these signals are not used; they are driven high when the HK80/V960E is the VSB master, which selects the System Address Space.

DS* VSB DATA STROBE. Three-state signal whose falling edge indicates a transfer will occur over AD31-AD00. During write cycles,

write data are valid at the falling edge of DS*.

WR*	VSB WRITE. WR*. A three-state signal used to indicate a *read* or *write* operation. When the signal is low, the operation is a write. When the signal is high, the operation is a read.
SIZE0,SIZE1	VSB BUS SIZE. Three-state lines that, in conjunction with addresses AD00 and AD01, determine the data transfer size and position on the data bus.
LOCK*	VSB BUS LOCK. When asserted, this line indicates that the bus is locked and that no other master can obtain possession of the bus. This allows for noninterruptible cycles, such as read-modify-write cycles, to occur from the VSB to a dual-ported resource. LOCK* can also indicate that a block transfer cycle is in progress.
ASACK0* **ASACK1***	VSB ADDRESS/SIZE ACKNOWLEDGE. Open-collector lines that serve two functions: (1) The responding SLAVE drives its size code on these lines, and (2) The responding slave drives at least one of these lines to inform the HK80/V960E to switch the multiplexed address/data bus from address to data.
WAIT*	WAIT. An open-collector line that is gated with AC (Decode Complete) on the master device. The condition AC active and WAIT* inactive, while PAS* is asserted, means that no VSB slave module has decoded the address being driven at that time or that there are no VSB slave modules installed. This gives the VSB master the option to switch to the VMEbus when VSB slaves are not responding, which allows VSB and VMEbus to share a common address space.
AC	VSB DECODE COMPLETE. An open-collector line that is asserted by slave modules to indicate to the master that address decoding has been completed. A slave device allows AC to go high after completing decoding or other conditions (see WAIT*), regardless whether the device is selected by the current address on the bus.
CACHE*	VSB CACHEABLE. An open-collector signal that, when asserted, indicates to the master that the selected address location is cacheable. CACHE* is asserted only by the

selected VSB slave module. This signal is not used on the HK80/V960E.

ACK*
VSB DATA TRANSFER ACKNOWLEDGE. An open-collector line that is asserted by the selected slave module to complete the handshake for a transfer operation.

ERR*
VSB DATA ERROR. An open-collector line that is asserted by the selected slave device to indicate a fault condition while attempting the data transfer operation. This would typically be the result of a parity error detected on a slave device.

IRQ*
VSB INTERRUPT REQUEST. An open-collector line that, when asserted, indicates that a master or slave device is attempting to interrupt another master. On the HK80/V960E, this signal generates a local interrupt to the VIC (LIRQ3), and the interrupt level to the processor may be configured in the VIC.

BREQ*
VSB BUS REQUEST. An open-collector line that is asserted by a requester whenever bus mastership is required.

BGIN*
VSB BUS GRANT IN. A totem-pole line that, as an input to the HK80/V960E, indicates that it has been granted the bus. BGIN and BGOUT form a bus grant daisy-chain.

BGOUT*
VSB BUS GRANT OUT. A totem-pole line that, as an output from the HK80/V960E, indicates to the next board in the daisy chain that it may use the bus.

BUSY*
VSB BUS BUSY. An open-collector line that is asserted by a requester that has been granted the bus, to indicate ownership of the bus.

GA0-GA2
VSB GEOGRAPHICAL ADDRESSES. These lines are connected to ground on the HK80/V960E; the geographical addressing feature is not implemented.

7.3 VSB OPERATION

VSB is accessible from $0400,0000_{16}$ to $4000,0000_{16}$.

Physically, the bus interface uses 32 multiplexed address and data lines. Data transfers may be 8, 16, 24 or 32 bits in length. It is an asynchronous bus.

There is one interrupt line, IRQ, associated with the VSB. When asserted, this signal generates a local interrupt to the VIC (LIRQ3), and the interrupt level to the processor may be configured in the VIC.

There are two control bits that affect the operation of the VSB interface.

Release-On-Request The HK80/V960E supports two VSB release modes. The bus can be released between every access or only if another master requests the bus. A one-bit latch at address $0200,0010_{16}$ controls the VSB release mode.

TABLE 7-1
VSB release modes

Port address: $0200,0010_{16}$. Size: Long. Type: Write.	
DO	**Function**
0	Release only if another request.
1	Release after every operation.

VSB Arbiter Enable The "first" VSB master board — the primary master, should be the arbiter. Other VSB masters should not be the arbiter. The arbiter indicates the beginning of the VSB arbitration daisy chain. The VSB-arbiter-enable bit *must be set true* if the HK80/V960E is the "first" board, that is, the arbiter. A 1-bit latch at address $0200,0018_{16}$ controls the VSB arbiter enable.

TABLE 7-2
VSB arbiter enable

Port address: $0200,0018_{16}$. Size: Long. Type: Write.	
DO	**Function**
0	Not enabled: HK80/V960E is not arbiter.
1	Enabled: HK80/V960E is arbiter.

7.4 VSB Termination

The VSB signals must be properly terminated to ensure correct bus operation. Use this chart to determine if VSB resistor packs should be installed on the HK80/V960E for your system configuration. The HK80/V960E VSB resistor pack terminations are

designated RN18-RN24 and are located on the lower left of the HK80/V960E, just above the P2 connector.

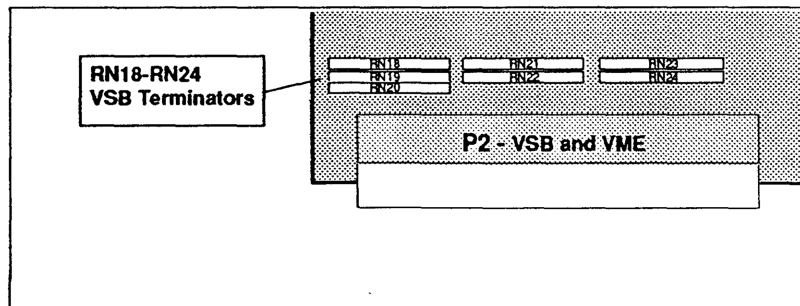


FIGURE 7-1. Location of VSB terminators

TABLE 7-3
VSB terminations

	HK80/V960E as End Board	Other VSB Boards
Install	VSB terminations	none
Remove	none	VSB terminations

Summary: Remove VSB terminations on all but *one* end board.

The VSB specification calls for the terminators to be within 2 inches of one end of the signal lines. If your VSB backplane includes the signal terminations, then the resistor packs should be removed on all of the VSB modules. Six or fewer boards may be used on the VSB.

7.5 VMEbus AND VSB PIN ASSIGNMENTS, P2

P2 is used for both the VMEbus and the VSB. The center row of pins (row B) are the upper address and data lines of the VMEbus. The outer two rows (A and C) make up the VSB.

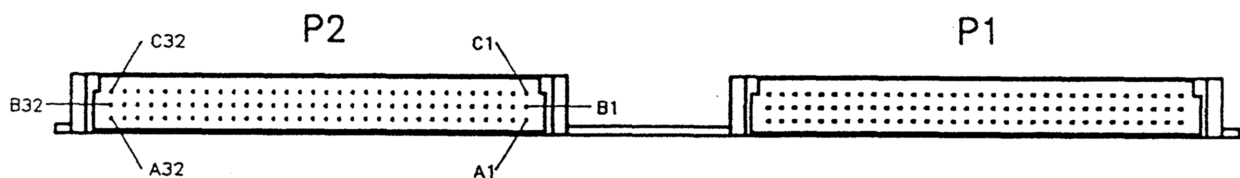


FIGURE 7-2. VSB connector, P2

TABLE 7-4
VMEbus and VSB connector pin assignments, P2

P2 Pin Number	Row A VSB Signal Mnemonic	Row B VMEbus Signal Mnemonic	Row C VSB Signal Mnemonic
1	AD00	+5	AD01
2	AD02	Gnd	AD03
3	AD04	(reserved)	AD05
4	AD06	A24	AD07
5	AD08	A25	AD09
6	AD10	A26	AD11
7	AD12	A27	AD13
8	AD14	A28	AD15
9	AD16	A29	AD17
10	AD18	A30	AD19
11	AD20	A31	AD21
12	AD22	Gnd	AD23
13	AD24	+5	AD25
14	AD26	D16	AD27
15	AD28	D17	AD29
16	AD30	D18	AD31
17	Gnd	D19	Gnd
18	IRQ*	D20	Gnd
19	DS*	D21	Gnd
20	WR*	D22	Gnd
21	SPACE0	D23	SIZE0
22	SPACE1	Gnd	PAS*
23	LOCK*	D24	SIZE1
24	ERR*	D25	Gnd
25	Gnd	D26	ACK*
26	Gnd	D27	AC
27	Gnd	D28	ASACK1*
28	Gnd	D29	ASACK0*
29	Gnd	D30	CACHE*
30	Gnd	D31	WAIT*
31	BGIN*	Gnd	BUSY*
32	BREQ*	+5	BGOUT*

The use of P2 is *required* in order to meet VME power specifications.

8

User LEDs and Front Panel Interface

8.1 USER LEDs

There are four LEDs located near the reset button (Fig. 8-1) whose meanings may be defined by software.

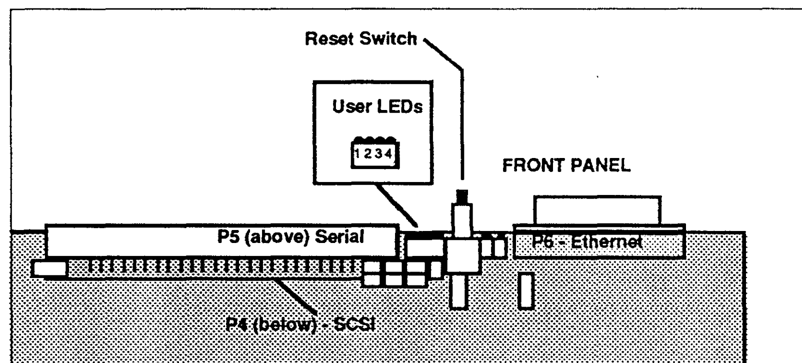


FIGURE 8-1. Location of user LEDs

TABLE 8-1
User LEDs — addresses

LED Number	Address (write-only)
1	$0200{,}0020_{16}$
2	$0200{,}0028_{16}$
3	$0200{,}0030_{16}$
4	$0200{,}0038_{16}$

Writing a 0 turns the chosen LED on; writing a 1 turns it off. At power-on or after a system reset, the LEDs will be ON.

8.2 FRONT PANEL INTERFACE (FPI), J2

There are five status outputs that allow remote monitoring of the HK80/V960E. Connections are made through a 14-pin header J2 located behind the Centronics connector (Fig. 8-2). Because there is no front panel connector associated with J2, a ribbon cable must be brought out the side of the card cage, or an empty slot must be left above the HK80/V960E. The connector pin assignments are outlined in Table 8-2:

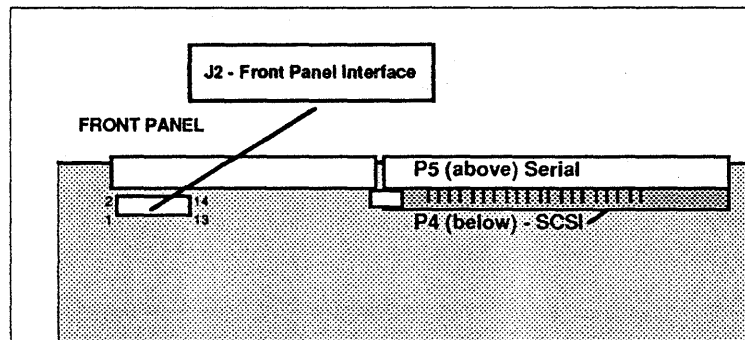


FIGURE 8-2. Location of front panel interface, J2

TABLE 8-2
Front panel interface connector pin assignments, J2

J2 Pin	Name	Meaning
2	Super	The 80960CA is in supervisory state.
4	User	The 80960CA is in user state.
6	DMA	The 80960CA is bus master in DMA mode.
8	HALT	The 80960CA is halted.
10	BUS	VIC owns the local bus (that is, a VME slave access is in progress.)
1,3,5,7,9	VCC	Vcc (+5) volts
12,14	GND	Signal ground

The output signals are low when true. Each is suitable for connection to an LED cathode. An external resistor must be provided for each output to limit current to 15 milliamperes.

Two input signals are also provided on J2 for interrupt and reset:

TABLE 8-3
J2 interrupt and reset signals

J2 Pin	Name	Function
J2-11	INTR*	External interrupt input: Connected to the VIC local interrupt LIRQ4. (Refer to section 6.)
J2-13	Reset*	When low, causes the HK80/V960E to reset.

9

CIO Usage

9.1 INTRODUCTION

The on-card CIO device performs a variety of functions. In addition to the three 16-bit timers that may be used to generate interrupts or count events, the CIO has three parallel ports.

While the three 16-bit timers may be used for user-defined applications, ports A, B, and C are used for comparing slave addresses in the VMEbus interface. All three ports should be programmed as outputs.

The CIO interrupt is tied to XINT7 of the 80960CA.

Refer to the CIO manual for further details.

9.2 PORT C BIT DEFINITION

Port C of the CIO is used for the VMEbus *standard* space accesses. This 4-bit port is compared to VMEbus address lines A23-A20 on *standard* space accesses only. There are 16 possible values that can be written to port C. Each allows 1 Mbyte of on-card memory to appear on the bus. Table 9-1 lists the effect of each value on the slave decode:

TABLE 9-1
HK80/V960E "standard" space slave mapping on VMEbus

		HK80/V960E Memory Mapped to Bus	
CIO Port C Compare Address	**VMEbus Address for 2- and 8-Mbyte HK80/V960E**	**2-Mbyte HK80/V960E**	**8-Mbyte HK80/V960E**
0_{16}	$XX00,0000_{16}$ – $XX0F,FFFF_{16}$	1st Mbyte	1st Mbyte
1_{16}	$XX10,0000_{16}$ – $XX1F,FFFF_{16}$	2nd Mbyte	2nd Mbyte
2_{16}	$XX20,0000_{16}$ – $XX2F,FFFF_{16}$	1st Mbyte	3rd Mbyte
3_{16}	$XX30,0000_{16}$ – $XX3F,FFFF_{16}$	2nd Mbyte	4th Mbyte
4_{16}	$XX40,0000_{16}$ – $XX4F,FFFF_{16}$	1st Mbyte	5th Mbyte
5_{16}	$XX50,0000_{16}$ – $XX5F,FFFF_{16}$	2nd Mbyte	6th Mbyte
6_{16}	$XX60,0000_{16}$ – $XX6F,FFFF_{16}$	1st Mbyte	7th Mbyte
7_{16}	$XX70,0000_{16}$ – $XX7F,FFFF_{16}$	2nd Mbyte	8th Mbyte
8_{16}	$XX80,0000_{16}$ – $XX8F,FFFF_{16}$	1st Mbyte	1st Mbyte
9_{16}	$XX90,0000_{16}$ – $XX9F,FFFF_{16}$	2nd Mbyte	2nd Mbyte
A_{16}	$XXA0,0000_{16}$ – $XXAF,FFFF_{16}$	1st Mbyte	3rd Mbyte
B_{16}	$XXB0,0000_{16}$ – $XXBF,FFFF_{16}$	2nd Mbyte	4th Mbyte
C_{16}	$XXC0,0000_{16}$ – $XXCF,FFFF_{16}$	1st Mbyte	5th Mbyte
D_{16}	$XXD0,0000_{16}$ – $XXDF,FFFF_{16}$	2nd Mbyte	6th Mbyte
E_{16}	$XXE0,0000_{16}$ – $XXEF,FFFF_{16}$	1st Mbyte	7th Mbyte
F_{16}	$XXF0,0000_{16}$ – $XXFF,FFFF_{16}$	2nd Mbyte	8th Mbyte

9.3 PORT B BIT DEFINITION

Port B of the CIO is used for comparison of VMEbus *short* address space accesses. The 8-bit value written to port B is used to match the upper 8 address lines of a VME *short* space address (VME A8-A15) (Table 9-2). This allows the mailbox to be mapped to 256 different locations (on 256-byte boundaries) in short address space.

TABLE 9-2
HK80/V960E "short" space slave mapping on VMEbus

CIO Port C Compare Address	VMEbus Address for 2- and 8-Mbyte HK80/V960E
0_{16}	$XX00,0000_{16}$ – $XX0F,FFFF_{16}$
1_{16}	$XX10,0000_{16}$ – $XX1F,FFFF_{16}$
2_{16}	$XX20,0000_{16}$ – $XX2F,FFFF_{16}$
3_{16}	$XX30,0000_{16}$ – $XX3F,FFFF_{16}$
4_{16}	$XX40,0000_{16}$ – $XX4F,FFFF_{16}$
5_{16}	$XX50,0000_{16}$ – $XX5F,FFFF_{16}$
6_{16}	$XX60,0000_{16}$ – $XX6F,FFFF_{16}$
7_{16}	$XX70,0000_{16}$ – $XX7F,FFFF_{16}$
8_{16}	$XX80,0000_{16}$ – $XX8F,FFFF_{16}$
9_{16}	$XX90,0000_{16}$ – $XX9F,FFFF_{16}$
A_{16}	$XXA0,0000_{16}$ – $XXAF,FFFF_{16}$
B_{16}	$XXB0,0000_{16}$ – $XXBF,FFFF_{16}$
C_{16}	$XXC0,0000_{16}$ – $XXCF,FFFF_{16}$
D_{16}	$XXD0,0000_{16}$ – $XXDF,FFFF_{16}$
E_{16}	$XXE0,0000_{16}$ – $XXEF,FFFF_{16}$
F_{16}	$XXF0,0000_{16}$ – $XXFF,FFFF_{16}$

9.4 PORT A BIT DEFINITION

Port A on the CIO chip is used to compare VMEbus *extended* space accesses. This 8-bit value is compared directly to VMEbus address lines 24-31 (Table 9-3). This allows local RAM to be mapped to one of 256 16-Mbyte locations in the extended address space on the bus.

TABLE 9-3
HK80/V960E "extended" space slave mapping on VMEbus

	VMEbus Address		HK80/V960E Memory Mapped to Bus	
CIO Port A Compare Address	2-Mbyte HK80/V960E	8-Mbyte HK80/V960E	2-Mbyte HK80/V960E	8-Mbyte HK80/V960E
00_{16}	$0000{,}0000_{16}$ – $001F{,}FFFF_{16}$	$0000{,}0000_{16}$ – $007F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
01_{16}	$0100{,}0000_{16}$ – $011F{,}FFFF_{16}$	$0100{,}0000_{16}$ – $017F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
02_{16}	$0200{,}0000_{16}$ – $021F{,}FFFF_{16}$	$0200{,}0000_{16}$ – $027F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
03_{16}	$0300{,}0000_{16}$ – $031F{,}FFFF_{16}$	$0000{,}0000_{16}$ – $037F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
. . .	. . .	. . .	. . .	. . .
$0F_{16}$	$0F00{,}0000_{16}$ – $0F1F{,}FFFF_{16}$	$0F00{,}0000_{16}$ – $0F7F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
10_{16}	$1000{,}0000_{16}$ – $101F{,}FFFF_{16}$	$1000{,}0000_{16}$ – $107F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
11_{16}	$1100{,}0000_{16}$ – $111F{,}FFFF_{16}$	$1100{,}0000_{16}$ – $117F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
12_{16}	$1200{,}0000_{16}$ – $121F{,}FFFF_{16}$	$1200{,}0000_{16}$ – $127F{,}FFFF_{16}$	2 Mbytes	8 Mbytes
. . .	. . .	. . .	. . .	. . .
FF_{16}	$FF00{,}0000_{16}$ – $FF1F{,}FFFF_{16}$	$FF00{,}0000_{16}$ – $FF7F{,}FFFF_{16}$	2 Mbytes	8 Mbytes

9.5 COUNTER/TIMERS

There are three independent, 16-bit counter/timers in the CIO. For long delays, timers 1 and 2 may be internally linked together to form a 32-bit counter chain. When programmed as timers, the following equation may be used to determine the time constant value for a particular interrupt rate.

$$TC = 2{,}000{,}000 \,/\, \text{interrupt rate (in Hz)}$$

The CIO is externally clocked at 4-MHz (±0.01%), which is internally divided by two to make an internal count rate of 2 MHz. The maximum cumulative timing error will be about 9 seconds per day, although the typical error is less than 1 second per day. Better long-term accuracy may be achieved via a power line (60 Hz) interrupt (using a bus interrupt) or the real-time clock (RTC) (refer to section 14).

9.6 REGISTER ADDRESS SUMMARY (CIO)

TABLE 9-4
CIO register addresses

Register	Address	Function
Port C, Data	$02E0,0000_{16}$	VME standard space compare address
Port B, Data	$02E0,0008_{16}$	VME short space compare address
Port A, Data	$02E0,0010_{16}$	VME extended space compare address
Control Registers	$02E0,0018_{16}$	CIO configuration and control

All registers are 8 bits wide.

Refer to the CIO manual for further details about register usage and descriptions.

9.7 CIO INITIALIZATION

Appendix A shows a typical initialization sequence for the CIO. The first byte of each data pair in "ciotable" specifies an internal CIO register; the second byte is the control data. Read section 3 for information concerning CIO interrupt vectors.

Read the Z8536 technical manual for more details on programming the CIO. Some people find the CIO technical manual difficult to understand. We encourage you to read all of it twice, before you pass judgment.

10

Serial I/O

10.1 INTRODUCTION

There are four RS-232C serial I/O ports on the HK80/V960E board. Each port may optionally be configured for RS-422 operation with a special interface cable, as detailed in section 10.11. Each port has a separate baud rate generator and can operate in asynchronous or synchronous modes.

Refer to the AMD Z85C30 SCC manual for programming details.

The SCC interrupts are tied to the XINT5 (ports A and B) and XINT4 (ports C and D) signals on the 80960CA.

10.2 RS-232 PIN ASSIGNMENTS, P5

Data transmission conventions are with respect to the external serial device. The HK80/V960E board is wired as a "Data Set." The connector is shown in Figure 10-1 and pin assignments are listed in Table 1.

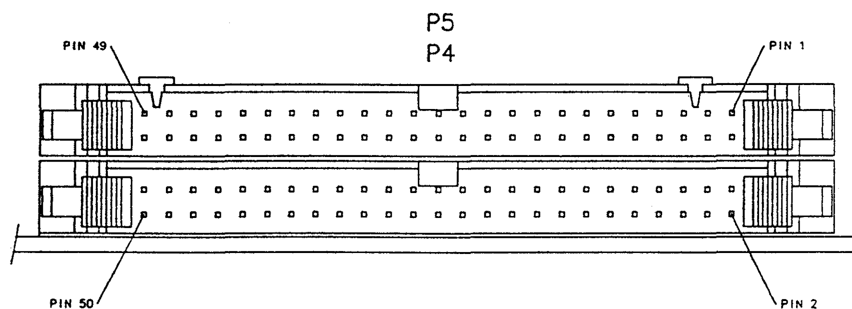


FIGURE 10-1. RS-232 connector, P5

TABLE 10-1a
Serial port pin assignments, P5 — Port A

Pin	"D" Pin	RS-232 Function	Direction	SCC Pin Function
1	2	Port A Tx Data	In	Rcv Data
2	15	Tx Clock	In	Rcv Clock
3	3	Rcv Data	Out	Tx Data
4	16	(not used)		
5	4	Request to Send[a]	In	DCD
6	17	Rcv Clock	In	Tx Clock
7	5	Clear to Send	Out	DTR
8	18	Ring Detect	In	Ring Ind
9	6	Data Set Ready	Out	RTS
10	19	(not used)		
11	7	Gnd		Sig Gnd
12	20	Data Terminal Ready[a]	In	CTS

[a]This signal uses default pull-up resistors that are controlled by J1.

TABLE 10-1b
Serial port pin assignments, P5 — Port B

Pin	"D" Pin	RS-232 Function	Direction	SCC Pin Function
13	2	Port B Tx Data	In	Rcv Data
14	15	Tx Clock	In	Rcv Clock
15	3	Rcv Data	Out	Tx Data
16	16	+12v (via J3)		
17	4	Request to Send[a]	In	DCD
18	17	Rcv Clock	Out	Tx Clock
19	5	Clear to Send	Out	DTR
20	18	+5v (via J4)		
21	6	Data Set Ready	Out	RTS
22	19	-12v (via J5)		
23	7	Gnd		Sig Gnd
24	20	Data Terminal Ready[a]	In	CTS

[a]This signal uses default pull-up resistors that are controlled by J1.

TABLE 10-1c
Serial port pin assignments (P5) — Port C

Pin	"D" Pin	RS-232 Function	Direction	SCC Pin Function
25	2	Port C Tx Data	In	Rcv Data
26	15	Tx Clock	In	Rcv Clock
27	3	Rcv Data	Out	Tx Data
28	16	(not used)		
29	4	Request to Send[a]	In	DCD
30	17	Rcv Clock	In	Tx Clock
31	5	Clear to Send	Out	DTR
32	18	Ring Detect	In	Ring Ind
33	6	Data Set Ready	Out	RTS
34	19	(not used)		
35	7	Gnd		Sig Gnd
36	20	Data Terminal Ready[a]	In	CTS

[a]This signal uses default pull-up resistors that are controlled by J1.

TABLE 10-1d
Serial port pin assignments (P5) — Port D

Pin	"D" Pin	RS-232 Function	Direction	SCC Pin Function
37	2	Port D Tx Data	In	Rcv Data
38	15	Tx Clock	In	Rcv Clock
39	3	Rcv Data	Out	Tx Data
40	16	+12v (via J6)		
41	4	Request to Send[a]	In	DCD
42	17	Rcv Clock	Out	Tx Clock
43	5	Clear to Send	Out	DTR
44	18	+5v (via J7)		
45	6	Data Set Ready	Out	RTS
46	19	-12v (via J8)		
47	7	Gnd		Sig Gnd
48	20	Data Terminal Ready[a]	In	CTS
49		(not used)		
50		(not used)		

[a]This signal uses default pull-up resistors that are controlled by J1.

Ports B and D are wired somewhat differently from ports A and C. In particular, the RS-232 signals *Rcv Clock* ("D" Pin 17) is an output on ports B and D, and *Ring Detects* are provided only on ports A and C. Note that the interconnect cable from P5 is arranged in such a manner that the "D" connector pin assignments are correct for RS-232C conventions. Not all pins on the "D" connectors are used. Recommended mating connectors are Ansley P/N 609-5001CE and Molex P/N 15-29-8508.

Signals marked with a superscript **a** in Table 10 have default pull-up resistors that are controlled by J1.

Note: The serial ports may *appear* to be inoperative if J1 is set to default "FALSE" and if the device connected to the port does not drive the DTR and RTS pins TRUE. The monitor software, for example, initializes the SCC channels to respect the state of DTR and RTS. The RI signals for ports A and C are routed to the VIC chip (refer to section 6.4.2.2).

10.3 SIGNAL NAMING CONVENTIONS (RS-232)

Since the RS-232 ports are configured as "data sets," the naming convention for the interface signals may be confusing. The interface signal names are with respect to the terminal device attached to the port while the SCC pins are with respect to the SCC as if it, too, were a terminal device. Thus all signal pairs, for example, "RTS" and "CTS," are switched between the interface connector and the SCC chip. For example, "Transmit Data," Px-1, is the data transmitted from the device to the HK80/V960E board; the data appears at the SCC receiver as "Received Data." For the same reason, the "DTR" and "RTS" interface signals appear as the "CTS" and "DSR" bits in the SCC, respectively. If you weren't confused before, any normal person should be by now. Study the chart below and see if that helps.

TABLE 10-2
RS-232 signal naming conventions

SCC Signal	Interface Signal	Direction
Tx Data	Rcv Data	to device
Rcv Data	Tx Data	from device
Tx Clock	Rcv Clock	from device (ports A & C)
Tx Clock	Rcv Clock	to device (ports B & D)
Rcv Clock	Tx Clock	See Table 10-1.
RTS	DSR	to device
CTS	DTR	from device
DTR	CTS	to device
DCD	RTS	from device
—	Ring Ind.	from device

The SCC was designed to look like a "data terminal" device. Using it as a "data set" creates this nomenclature problem. Of course, if you connect the HK80/V960E board to a modem ("data set"), then the SCC signal names are correct; however, a cable adapter is needed to properly connect to the modem. Three pairs of signals must be reversed.

TABLE 10-3
RS-232 reversal cable

SCC Signal	Px Pin Numbers	"D" Pin Number at HK80/ V960E	"D" Pin Number at Modem	RS-232 Signal
x	x	1	1	Prot Gnd
Rcv Data	1	2	3	Rcv Data
Tx Data	3	3	2	Tx Data
DCD	5	4	6	DSR
RTS	9	6	4	RTS
DTR	7	5	20	DTR
CTS	12	20	5	CTS
(Ring Ind)	8	18	22	Ring Ind
(Sig Gnd)	11	7	7	Sig Gnd

Summary: The HK80/V960E may be directly connected to a "data terminal" device. However, a *cable reversal* is required for a connection to a "data set" device (for example, a modem).

10.4 CONNECTOR CONVENTIONS

Paragraph 3.1 of the EIA RS-232-C standard says the following concerning the mechanical interface between data communications equipment:

> *"The female connector shall be associated with...the data communications equipment.... An extension cable with a male connector shall be provided with the data terminal equipment....When additional functions are provided in a separate unit inserted between the data terminal equipment and the data communications equipment, the female connector...shall be associated with the side of this unit which interfaces with the data terminal equipment while the extension cable with the male connector shall be provided on the side which interfaces with the data communications equipment."*

Substituting "modem" for "data communications equipment" and "terminal" for "data terminal equipment" leaves us with the impression that the modem should have a *female* connector and the terminal should have a *male.*

The Heurikon HK80/V960E microcomputer interface cables are designed with female "D" connectors, because the serial I/O ports are configured as data sets (modems). Terminal manufacturers typically use a female connector also, despite the fact that they produce terminals, not modems. Thus, the extension cable used to run between a terminal and the HK80/V960E (or a modem) will have male connectors at both ends.

If you do any work with RS-232 communications, you will end up with many types of cable adapters — double males, double females, double males and females with reversal, cables with males and females at both ends, you name it! We will be happy to help make special cables to fit your needs.

10.5 SCC INITIALIZATION SEQUENCE

The following table shows a typical initialization sequence for the SCC. This example is for port A. Other ports are programmed in the same manner, substituting the correct control port address.

TABLE 10-4
SCC initialization sequence

Data (hex)	Register Address	Function
00	0220,0008$_{16}$ (write)	Reset SCC register counter
09,C0	" "	Force reset (do for ports A & C only)
04,4C	" "	Async mode, x16 clock, 2 stop bits tx
05,EA	" "	Tx: RTS, Enable, 8 data bits
03,E1	" "	Rcv: Enable, 8 data bits
01,00	" "	No Interrupt, Update status
0B,56	" "	No Xtal, Tx & Rcv clk internal, BR out
0C,baudL	" "	Set low half of baud rate constant
0D,baudH	" "	Set high half of baud rate constant
0E,03	" "	Null, BR enable

Note: the notation "09,C0" (etc.) means the values 09$_{16}$ and C0$_{16}$ should be sent to the specified SCC port. The first byte selects the internal SCC register; the second byte is the control data. The above sequence only initializes the ports for standard asynchronous I/O without interrupts. The *baudL* and *baudH* values refer to the low and high halves of the baud rate constant, which may be determined from Table 10-6 ("Baud Rate Constants") below.

For information concerning SCC interrupt vectors, refer to section 3. Read the Z8530 technical manual for more details on SCC programming.

10.6 PORT ADDRESS SUMMARY

TABLE 10-5
SCC register addresses

Register	Port A	Port B	Port C	Port D
Control	0220,0008$_{16}$	0220,0000$_{16}$	0230,0008$_{16}$	0230,0000$_{16}$
Data	0220,0018$_{16}$	0220,0010$_{16}$	0230,0018$_{16}$	0230,0010$_{16}$

All ports are eight bits.

10.7 SERIAL DMA

Serial ports A, C, and D are hardwired to the 80960CA's DMA as follows:

Port A — DMA channel 2
Port C — DMA channel 1
Port D — DMA channel 0

The assertion of the W/REQx pin of the SCC is used as the DMA request for the associated DMA channel. The DMA channel is required to read the Port Data register to service the request. Refer to the 80960CA user's manual for details on the DMA.

10.8 BAUD RATE CONSTANTS

If the internal SCC baud rate generator logic has been selected, the actual baud rate must be specified during the SCC initialization sequence by loading a 16-bit time constant value into each generator. The following table gives the values to use for some common baud rates. Other rates may be generated by applying a formula.

TABLE 10-6
Baud rate constants

Baud Rate	x1 clock rate	x16 clock rate
110	72,725	4541
300	26,665	1665
1200	6665	415
2400	3331	206
4800	1665	102
9600	831	50
19,200	415	24
38,400	206	11

The time constant values listed above are computed as follows:

$$TC_{(x16)} = \frac{500{,}000}{\text{baud}} - 2$$

$$TC_{(x1)} = \frac{8{,}000{,}000}{\text{baud}} - 2$$

The x16 mode will obtain better results with asynchronous protocols because the receiver can search for the middle of the start bit. (In fact, the x1 mode will probably produce frequent receiver errors.)

The maximum SCC data speed is 1 megabit per second, using the x1 clock and synchronous mode. For asynchronous trans-

mission, the maximum practical rate using the x16 clock is 62,500 baud.

10.9 RS-422 OPERATION

As an option, one or more of the serial ports on the HK80/V960E may be configured for RS-422 operation. The RS-422 option may either be installed when the board is ordered, or an existing HK80/V960E board may be factory-upgraded to add the option. Please contact Heurikon's Customer Service department for more information.

10.10 RELEVANT JUMPERS (SERIAL I/O)

TABLE 10-7
Relevant jumpers — serial I/O

Jumper	Function	Position
1	RS-232 A,B,C,D Status Default	J1-A (True) J1-B (False)
3	+12 power port B	Installed
4	+5 power port B	Installed
5	-12 power port B	Installed
6	+12 power port D	Installed
7	+5 power port D	Installed
8	-12 power port D	Installed

10.11 SERIAL I/O CABLE DRAWING

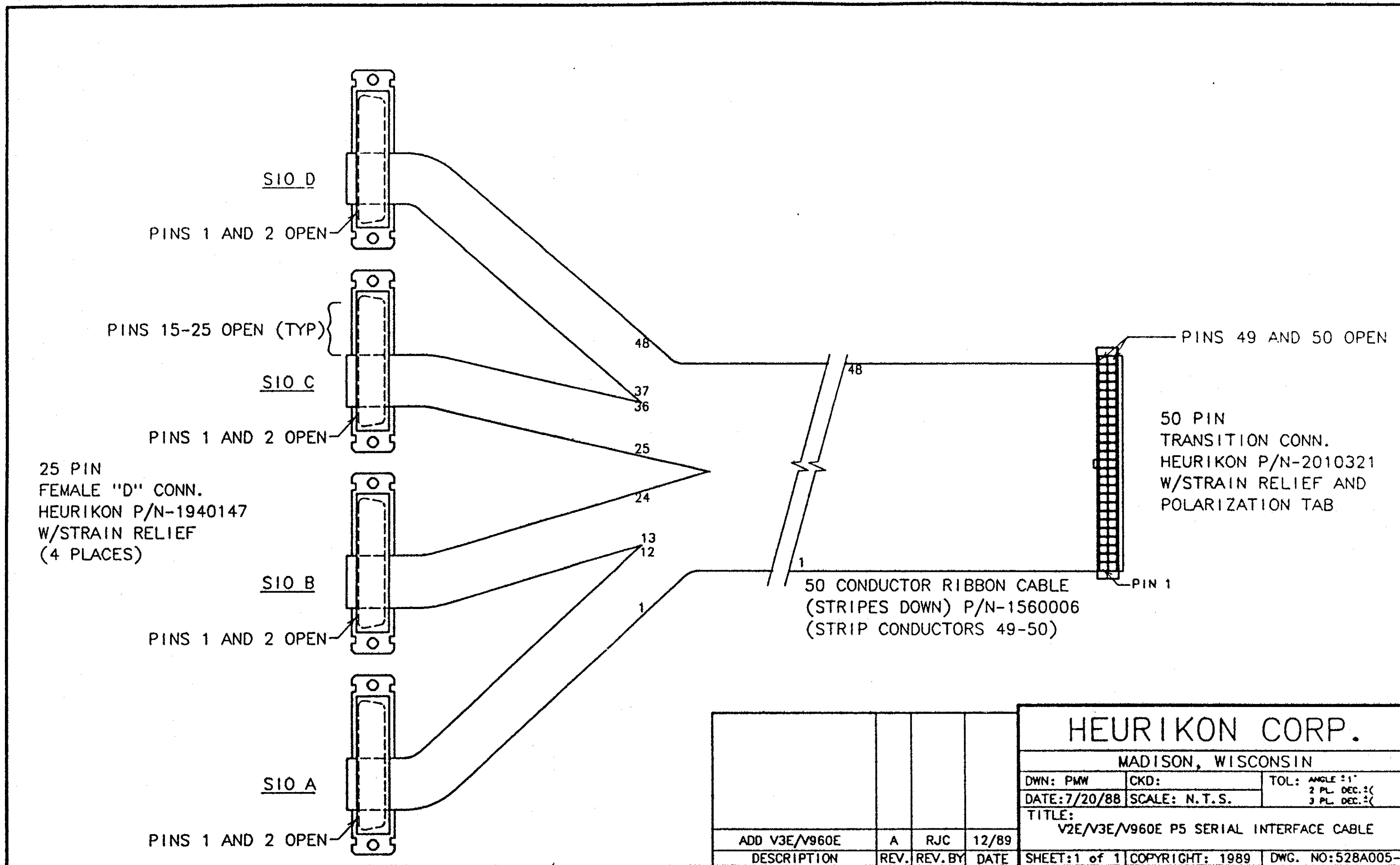


FIGURE 10-2. Serial I/O cable

11

Ethernet Interface

11.1 INTRODUCTION

The HK80/V960E is equipped with an Ethernet interface, which uses the Intel 82596CA 32-bit coprocessor to implement a standard IEEE-802.3 CSMA/CD 10BASE5 (10 megabits per second) Ethernet interface.

The Ethernet interrupt is tied to the XINT3 interrupt pin of the 80960CA.

Refer to the 82595CA user's manual (Intel publication number 296443-001) for more detail on the operation and programming of this device.

11.2 COMPONENTS

The Ethernet interface consists of two functional units — the Network Interface Controller and the Serial Network Interface.

11.2.1 Network Interface Controller

The network interface controller on the HK80/V960E is the Intel 82596CA high-performance 32-bit local area network coprocessor. It performs complete CSMA/CD Medium Access Control (MAC) functions according to IEEE-802.3 independently of the CPU. Features include:

- On-chip memory management
- Bus master with on-chip DMA using a 32-bit RAM interface
- Statistics management
- Transmit and receive FIFOs
- Network monitor mode
- Self-test diagnostics and loopback mode
- 82586 software compatibility mode

- Little-/big-endian (that is, Intel/Motorola) byte ordering
- Burst bus transfers

The 82596CA runs at the CPU (80960CA) speed and has up to a 105 Mbytes per second bus bandwidth.

11.2.2 Serial Network Interface

The Manchester encoder/decoder serial network interface for the Ethernet interface on the HK80/V960E is the Intel 82C501AD. Conforming to IEEE-802.3, it interfaces the network interface controller (82596CA) to the Ethernet network, performing the required Manchester encoding/decoding of network packets. Features include:

- 10 megabits per second Manchester encoding and decoding with receive clock recovery
- Loopback capability for diagnostics
- Selectable for use with Ethernet 1.0 or IEEE-802.3 transceivers via jumper J11 (see Table 11.6).

11.3 ETHERNET ACCESS

There are four ways for the CPU (80960CA) to communicate with the Ethernet portion of the HK80/V960E: ARB, PORT, CA, and LE/BE (Table 11-1).

TABLE 11-1
Ethernet accesses

Access	R/W	Address	D31-D4	D3-D0	Function
ARB	W	$0200,01C0_{16}$	$XXXX,XXXX_{16}$	0	Ethernet disabled.
				1	Ethernet enabled.
PORT	W	$0280,00xx_{16}$	$XXXX,XXXX_{16}$	0	Reset the 82596CA.
			Results address	1	Perform a SELF TEST on 82596CA.
			SCP address	2	Write a new SCP address.
			Dump address	3	Dump the 82596CA registers.
CA	R/W	$0290,00xx_{16}$	$XXXX,XXXX_{16}$	X	Channel Attention
LE/BE	W	$0200,0008_{16}$	$0000,000x_{16}$	0	82596CA is big-endian (Motorola).
				1	82596CA is little-endian (Intel).

11.3.1 Arbiter Enable

To use the Ethernet facilities of the HK80/V960E, the Ethernet Arbiter (ARB) must be enabled. Without this bit set, 82596CA requests to the CPU (80960CA) will be ignored. A 1-bit latch at address $0200{,}01C0_{16}$ controls the Ethernet Arbiter.

11.3.2 Port Access

The 82596CA has a CPU port process state that allows the CPU (80960CA) to cause the 82596CA to execute certain functions when address $0280{,}0000_{16}$ is accessed. These functions are:

- Reset: Performs a reset of the 82596CA without disturbing the rest of the system.
- Self-test: Performs a self-test on the 82596CA and writes the results to a specified location in memory.
- New SCP: Write an alternate system configuration pointer address (SCP). This function is useful when the default SCP ($00FF{,}FFF6_{16}$) conflicts with system memory (on the HK80/V960E, this function must be used, because the default conflicts).
- Dump: Performs a dump of the internal state (registers and memory) of the 82596CA, and writes it to a specified location in memory.

The format of the PORT commands are summarized in the table below.

TABLE 11-2
82596CA port accesses

Address is $0280{,}00xx_{16}$

Function	D31............................D4	D3	D2	D1	D0
Reset	...............Don't care................	0	0	0	0
Self-test	A31........Self-test results address.......A4	0	0	0	1
New SCP	A31.........Alternate SCP address.......A4	0	0	1	0
Dump	A31...........Dump area pointer..........A4	0	0	1	1

For every PORT command, there must be two accesses (Table 11-3).

TABLE 11-3
82596CA port access definition

	First Access	Second Access
Big endian	D15-D0 > Lower Command Word	D31-D16 > Upper Command Word
Little endian	D31-D16 > Lower Command Word	D15-D0 > Upper Command Word

Therefore, two back-to-back 32-bit accesses each using the same 32-bit data may be used to complete the PORT command (because the "other half" of the 32-bit value is ignored). A delay of at least one clock cycle is required between successive PORT commands. Software should account for this. Refer to the 82596CA user's manual and the 82596CA initialization code in Appendix A for more details.

11.3.3 Channel Attention (CA)

Accessing address $0290{,}00xx_{16}$ issues Channel Attention (CA) to the 82596CA and causes it to begin executing memory-resident command blocks. The first CA after a reset forces the 82596CA into the initialization sequence beginning at location $00FF{,}FFF6_{16}$ or an alternate SCP address written to the 82596CA using the PORT access mechanism. All subsequent CAs cause the 82596CA to begin executing new command sequences (memory-resident command blocks) from the system control block (SCB).

Since the default SCP address ($00FF{,}FFF6_{16}$) is not accessible memory on the HK80/V960E, the NewSCP PORT Access command must be issued prior to the first CA after a reset.

Refer to the 82596CA user's manual for more details.

11.3.4 Ethernet Byte Ordering

The 82596CA supports both little-endian (Intel) and big-endian (Motorola) byte ordering. Byte ordering determines which memory location stores the least significant byte of the operand. For little-endian systems, the least significant byte is stored at the lowest byte address. For big-endian systems, the most significant byte is stored at the lowest address. The number of bytes per operand depends on the data type. The byte ordering can be selected by accessing address $0200{,}0008_{16}$ (Table 11-4).

TABLE 11-4
Ethernet byte ordering

Port address $0200,0008_{16}$. Size: Long. Type: Write.	
Byte Ordering (LE/BE)	**D0**
Big endian	0 (default after reset)
Little endian	1

11.4 ETHERNET PORT PIN ASSIGNMENTS, P6

Connector P6 is an Ethernet 15-pin D connector (Fig. 11-1).

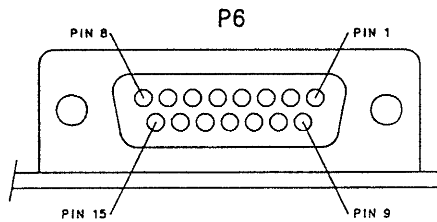


FIGURE 11-1. Ethernet connector, P6

TABLE 11-5
Ethernet connector pin assignments, P6

Pin Number	Name	Function
1	CIS	Control In Shield
2	CI+	Control In +
3	DO+	Data Out +
4	DIS	Data In Shield
5	DI+	Data In +
6	VC	Voltage Common
7	CO+	Control Out +
8	COS	Control Out Shield
9	CI-	Control In -
10	DO-	Data Out -
11	DOS	Data Out Shield
12	DI-	Data In -
13	VP	Voltage Plus
14	VS	Voltage Shield
15	CO-	Control Out -

11.5 TRANSCEIVER CONFIGURATION

The transmit differential line for the Ethernet interface may be configured for either half- or full-step modes to facilitate its use with different types of transceivers, via configuration jumper J11.

Ethernet configuration is briefly summarized in the following table:

TABLE 11-6
Transmit differential line configuration, J11

Position	Configuration
J11 installed	0 idle differential voltage on TX lines: half-step mode (for use with IEEE-802.3-type transceivers)
J11 not installed	+ (positive) idle differential voltage on TX lines: full-step mode (for use with Ethernet 1.0-type transceivers)

Currently the 82596CA driver code is proprietary and confidential. Please contact the factory for programming examples. Future revisions of the manual will include 82596CA programming examples.

12

SCSI Port

12.1 INTRODUCTION

The HK80/V960E uses the Western Digital WD33C93 chip to implement a Small Computer System Interface (SCSI) port (commonly pronounced "scuzzy").

The SCSI port may be used to connect to a variety of peripheral devices. Most common are Winchester disks, floppy diskettes, and streamer tape drives.

Supported features and modes include:

- Initiator role
- Target role
- Arbitration
- Disconnect
- Reconnect
- 80960CA DMA interface

Data transfer functions can be handled in a polled I/O mode or by using the DMA functions provided by the MPU. The SCSI interrupt is tied to XINT6 of the 80960CA.

Refer to the WD33C93A technical specification for programming details.

12.2 SCSI DMA

The DMA for the SCSI port is provided through the 80960CA's DMA port 3. The SCSI handshakes both request and acknowledge with the DMA port. The DMA should be programmed to perform dual address transfers using the byte assembly feature of the DMA in order to reduce the number of RAM accesses. The DMA Control Word of the 80960CA (using the **sdma** instruction) should be set as follows:

SCSI Read: $0000,00A3_{16}$

That is:
- 8 to 32 bit
- Source address hold
- Demand mode (synchronize)
- Source synchronized

SCSI Write: $0000,00DC_{16}$

That is:
- 32 to 8 bit
- Destination address hold
- Demand mode (synchronize)
- Destination synchronized

Refer to the 80960CA user's manual for further details on DMA.

Using these modes, the SCSI/DMA interface can support a 4-Mbyte per second transfer rate with the WD33C93A chip.

12.3 REGISTER ADDRESS SUMMARY (SCSI)

TABLE 12-1
SCSI register address summary

Address	R/W	Bits	Function
$0240,0000_{16}$	W	8	Set Controller Address Register
$0240,0000_{16}$	R	8	Read Auxiliary Register
$0240,0008_{16}$	R/W	8	SCSI Controller Registers
$02D0,0000_{16}$	R/W	8	SCSI Data Register (DMA) address
$0200,0140_{16}$	W	1	SCSI Bus Reset (1=reset, 0=release)

12.4 SCSI RESET

The SCSI reset pin (RST — pin 40) is connected to the VIC interrupt pin LIRQ7, thus enabling the VIC to interrupt the processor if a SCSI reset occurs. See section 6.4 and Table 6-3.

12.5 SCSI PORT PIN ASSIGNMENTS, P4

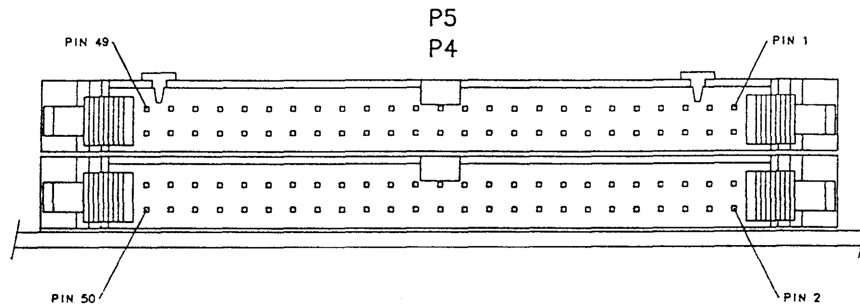


FIGURE 12-1. SCSI connector, P4

TABLE 12-2
SCSI pin assignments, P4

Pin number	Name	Function
Odd pins	Ground	
2	DB0/	Data bit 0
4	DB1/	Data bit 1
6	DB2/	Data bit 2
8	DB3/	Data bit 3
10	DB4/	Data bit 4
12	DB5/	Data bit 5
14	DB6/	Data bit 6
16	DB7/	Data bit 7
18	DBP/	Data parity bit
26	TERMPWR	Termination Power (+5)
32	ATN/	Attention
34	Spare	
36	BSY/	SCSI Bus busy
38	ACK/	Transfer acknowledge
40	RST/	Reset
42	MSG/	Message
44	SEL/	Select
46	C/D	Control/Data
48	REQ/	Transfer request
50	I/O/	Data movement direction

Recommended mating connectors are Ansley P/N 609-5001CE and Molex P/N 15-29-8508.

12.6 SCSI TERMINATION

If necessary, use RN4-RN6 at the top of the HK80/V960E near the SCSI connector on the HK80/V960E for SCSI termination (Fig. 12-2). Jumper J9 should be installed if termination is present.

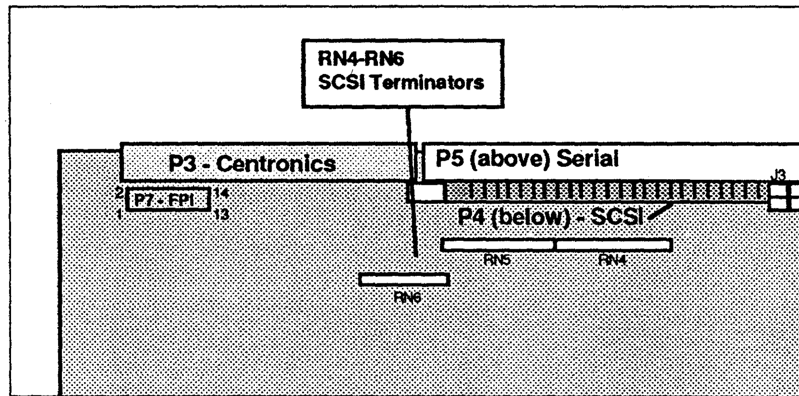


FIGURE 12-2. Location of SCSI terminators

13

Centronics Port

13.1 INTRODUCTION

This 8-bit parallel port is designed for direct connection to a Centronics compatible printer (or other) device. Since the handshake lines (STROBE and INIT) are under software control, this interface can be used as a general-purpose output port.

13.2 CENTRONICS PORT PIN ASSIGNMENTS, P3

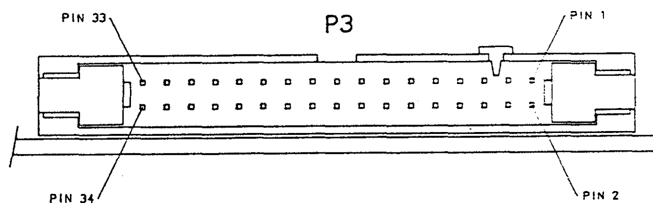


FIGURE 13-1. Centronics connector, P3

TABLE 13-1
Centronics pin assignments, P3

P3 Pin	Centronics Pin	Direction	Signal
2-24 (even)	(19-30)		Gnd
1	1	Output	STROBE/
3	2	Output	DATA1 (D0)
5	3	Output	DATA2
7	4	Output	DATA3
9	5	Output	DATA4
11	6	Output	DATA5
13	7	Output	DATA6
15	8	Output	DATA7
17	9	Output	DATA8 (D7)
19	10	Input	ACK/
21	11	Input	BUSY
23	12	Input	PE
25	13	Input	SELECT
26	31	Output	INIT/
27	14		Gnd
28	32	Input	ERROR/
29	15		n/c
30	33	Input	spare 1
31	16		Gnd
32	34	Input	spare 2
33	17		n/c
34	35	Input	spare 3
—	18		n/c
—	36		n/c

Recommended mating connectors are Ansley P/N 609-3401CE and Molex P/N 15-29-8348.

The falling edge of ACK/ is used to turn on the Centronics interrupt signal going to the VIC local interrupt line L1RQ1. To clear the interrupt signal, *read* from the interrupt reset location, $02C0{,}0018_{16}$.

13.3 CENTRONICS CONTROL PORT ADDRESS

The Centronics interface logic uses the following physical memory addresses for data and control functions:

TABLE 13-2
Centronics control addresses

Address	Direction	Function
$02C0,0000_{16}$	W	Data Latch (see below)
$02C0,0000_{16}$	R	Status Port (see below)
$02C0,0008_{16}$	W	Turn STROBE on
$02C0,0008_{16}$	R	Turn STROBE off
$02C0,0010_{16}$	W	Turn INIT on
$02C0,0010_{16}$	R	Turn INIT off
$02C0,0018_{16}$	R	Reset ACK Interrupt

TABLE 13-3
Centronics data/status addresses

Bit	$02C0,0000_{16}$ (Write) Data Latch	$02C0,0000_{16}$ (Read) Status Port
D7	DATA8	(spare 1)
D6	DATA7	(spare 2)
D5	DATA6	(spare 3)
D4	DATA5	ERROR/
D3	DATA4	SELECT
D2	DATA3	PE
D1	DATA2	BUSY
D0	DATA1	ACK/ (Negative true pulse)

After power-on, the state of the Data Latch is indeterminate; STROBE and INIT will be false. The Data Latch is not changed by a board reset; however, STROBE and INIT will go false.

Follow this procedure when using this port for a Centronics printer:

1. Wait for the printer BUSY signal to go false.
2. Write the character to port $02C0,0000_{16}$.
3. Assert STROBE (write to $02C0,0008_{16}$).

4. Delay at least one microsecond.

5. De-assert STROBE (read from $02C0,0008_{16}$).

6. Wait for ACK (wait for an interrupt via the VIC). The ACK signal at the Centronics status port (bit D0 of $02C0,0000_{16}$) will be just a fleeting pulse.

7. Reset the ACK interrupt signal by reading from $02C0,0018_{16}$. (See Table 13.2.)

8. Repeat for the next character.

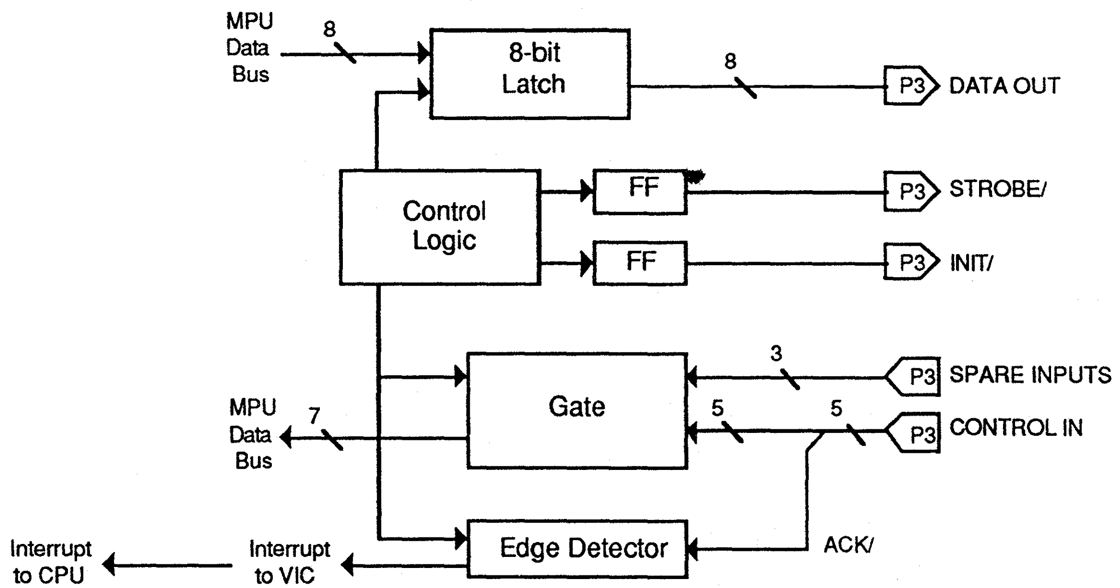


FIGURE 13-2. Centronics interface — block diagram

13.4 CENTRONICS PRINTER INTERFACE CABLE

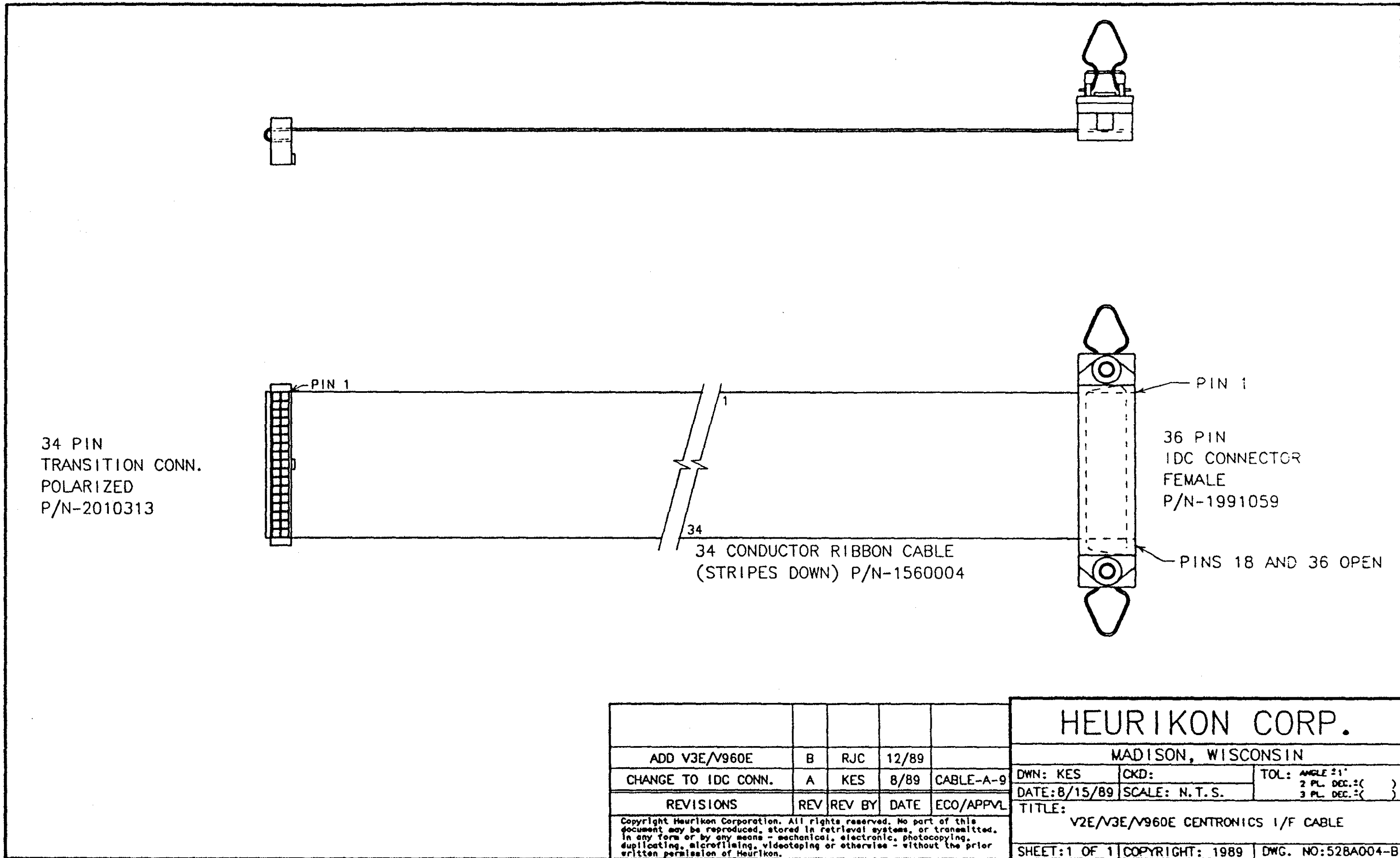


FIGURE 13-3. Centronics printer interface cable

14

Real-Time Clock (RTC)

14.1 INTRODUCTION

The HK80/V960E has a real-time clock module (Dallas Semiconductor, part number DS1216F or equivalent), which includes a built-in CMOS watch circuit and a lithium battery. The module is located underneath the HK80/V960E EPROM (that is, the module also functions as a socket for the EPROM).

14.2 RTC IMPLEMENTATION

The RTC logic does not generate interrupts; a CIO timer channel is used for that purpose. The RTC contents, however, may be used to check for long-term drift of the system clock, and as an absolute time and date reference after a power failure. Leap year accounting is included. Heurikon can provide complete operating system software support for the RTC module.

The RTC module time resolution is 10 milliseconds. The RTC internal oscillator is accurate to 1 minute per month, at 25 degrees C.

The clock contents are set or read using a special sequence of commands, as detailed in the program example, in Appendix A.

To access the RTC, a specific sequence of 64 accesses must occur to "unlock" the device for use. Then, a series of serial read commands may be initiated at the addresses shown in Table 14-1 to perform the actual reading and writing of the clock.

TABLE 14-1
RTC accesses

Byte Read at Address	Function
$02F0,0000_{16}$	Write a 0 to RTC.
$02F0,0003_{16}$	Write a 1 to RTC.
$02F0,0004_{16}$	Read RTC.

Note: Do not execute the module access instructions out of ROM. The instruction fetch cycles will interfere with the module access sequence. Also, be certain the reset disable bit (rtc_data.day bit D4) is always written as a 1.

Example code for the real-time clock is provided in Appendix A.

15

Summary Information

15.1 SOFTWARE INITIALIZATION SUMMARY

Refer to the example code in Appendix B for guidance on software initialization.

15.2 ON-CARD I/O ADDRESSES

This section is a summary of the on-card port addresses. It is intended as a general reference for finding additional information about a particular device. Refer to section 5.5 for a pictorial description of the system memory map.

TABLE 15-1
Address summary

Address (Hexadecimal)	Type	Device	Reference Section
4xxx,xxxx	R/W	VMEbus (Extended Space)	6.7
0400,0000 – 3FFF,FFFF	R/W	VSB bus	7.3
03xx,xxxx	R/W	VMEbus (Standard Space)	6.7
02F0,00xx	R/W	RTC	14
02E0,00xx	R/W	CIO	9.6
02D0,00xx	R/W	SCSI DMA	12.3
02C0,00xx	R/W	Centronics	13.3
02B0,0000	R	VIC (Interrupt Acknowledge)	6.4.2.2
02A0,0xxx	R/W	VIC (Registers)	6.3
0290,xxxx	R/W	Ethernet Channel Attention	11.3
0280,00xx	W	Ethernet Port Access	11.3
Continues.			

TABLE 15-1 — *Continued.*
Address summary

Address (Hexadecimal)	Type	Device	Reference Section
0270,xxxx	R/W	NV-RAM (Read/Write)	5.7
0240,00xx	R/W	SCSI	12.3
0230,00xx	R/W	SCC2 (Ports C & D)	10.6
0220,00xx	R/W	SCC1 (Ports A & B)	10.6
0210,0000	R	Error Status Latch	3.4.2
0200,01C0	-	Ethernet Arbiter Enable	11.3
0200,0180	-	Reserved	
0200,0140	W	SCSI Reset	12.3
0200,0100	W	VME Extended Space Enable	6.8.1
0200,00C0	W	VME Standard Space Enable	6.8.2
0200,0080	W	VME Short Space Enable	6.5
0200,0040	W	ROMINH	5.2
0200,0038	W	User LED 4	8.1
0200,0030	W	User LED 3	8.1
0200,0028	W	User LED 2	8.1
0200,0020	W	User LED 1	8.1
0200,0018	W	VSB Arbiter Enable	7.3
0200,0010	W	VSB Release on Request	7.3
0200,0008	W	Ethernet LE/BE Select	11.3
0100,xxxx	R/W	VMEbus (Short Space)	6.7
00xx,xxxx	R/W	On-card RAM	5.3

15.3 HARDWARE CONFIGURATION JUMPERS

Jumper settings and terminator configurations are detailed in the manual section pertaining to the associated device. This section can be used as a cross reference for finding additional information.

TABLE 15-2
Jumper and terminator configurations

Jumper or Terminator	Function	Reference Section	Standard Configuration
1	Ports A,B,C,D defaults	10.10	Installed
3	RS-232 power, +12V port B	10.10	Removed
4	RS-232 power, +5V port B	10.10	Removed
5	RS-232 power, -12V port B	10.10	Removed
6	RS-232 power, +12V port D	10.10	Removed
7	RS-232 power, +5V port D	10.10	Removed
8	RS-232 power, -12V port D	10.10	Removed
9	SCSI bus power	12.6	Removed
10	System controller	6.6	Installed
11	Ethernet transceiver type	11.5	Not installed
17	ROM size	5.2	1 Mbit
RN4-RN6	SCSI terminators	12.6	Installed
RN18-RN24	VSB terminators	7.4	Installed

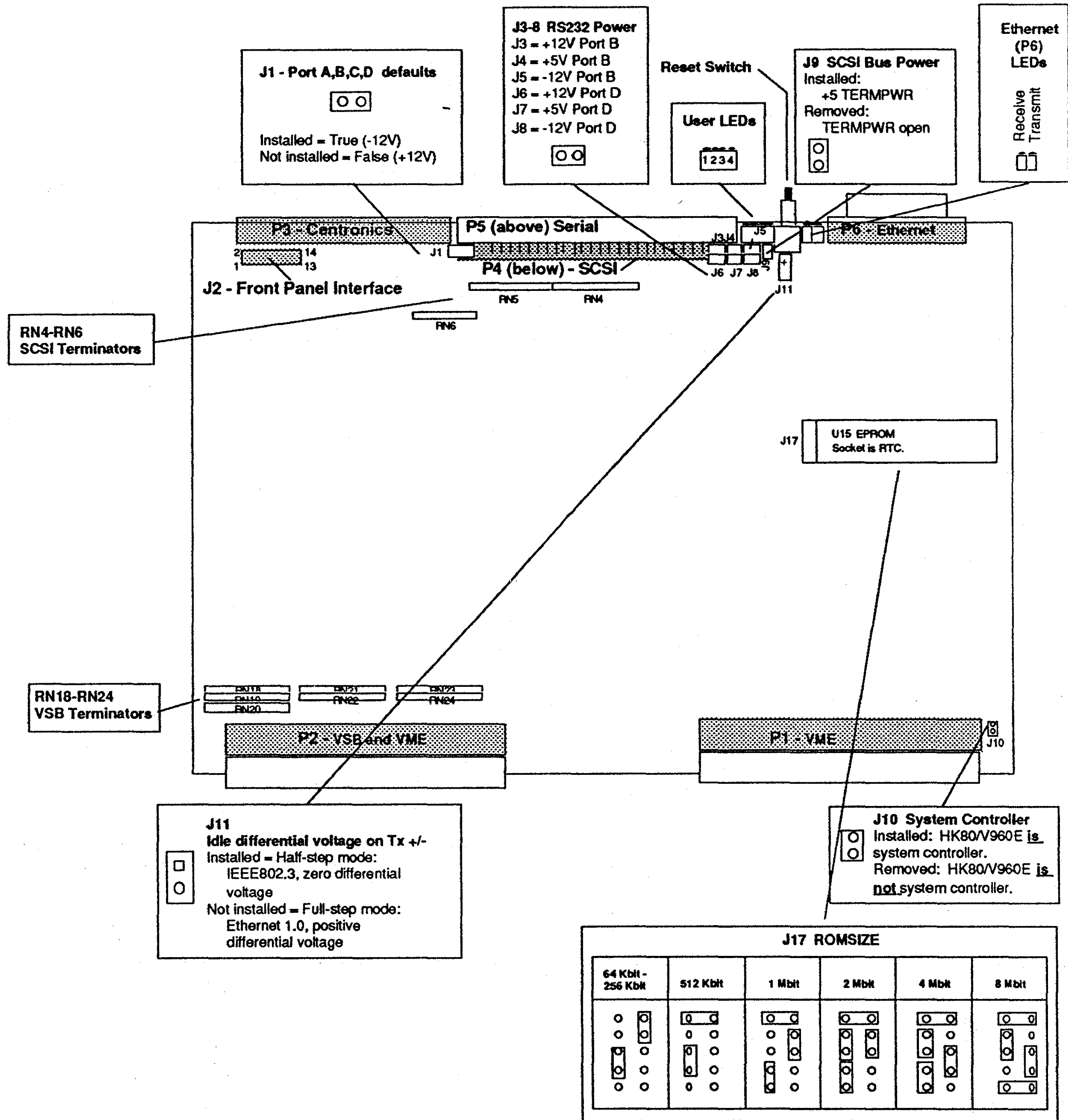


FIGURE 15-1. HK80/V960E jumper locations

15.4 POWER REQUIREMENTS

TABLE 15-3
HK80/V960E power requirements

Voltage	Current	Usage
+5	9.0 A	All logic
+12	1.0 A	RS-232 interface and Ethernet
-12	1.0 A	RS-232 interface

All "+5" and "Gnd" pins on P1 **and** P2 *must* be connected to ensure proper operation.

15.5 ENVIRONMENTAL REQUIREMENTS

Operating temperature: 0 to +55 degrees Centigrade, ambient, at board.
Humidity: 0% to 85%.
Storage temperature: -40 to +70 degrees C.
Typical power dissipation: About 45 W

FAN COOLING IS REQUIRED for the HK80/V960E board whenever power is applied, even when the board is on an extender card. Recommended air flow rate is 2-3 cubic feet per minute, depending on card cage constraints and other factors.

15.6 MECHANICAL SPECIFICATIONS

TABLE 15-4
Mechanical specifications

Width	Depth	Height (above board)
9.187 in.	6.299 in.	0.6 in.
233.35 mm	160 mm	15.25 mm

Standard board spacing is 0.8 inches. The HK80/V960E is a 10-layer board.

Appendix A — Code Examples

This appendix contains the example code listed below:

README — A brief description of the example files.

Board.c — This file is the catchall for the miscellaneous board-related functions.

Board.h — This file describes the HK80/V960E hardware addresses and data structures.

Bug.h — This file is intended to provide standard constants and data structures common to all files independent of processor, compiler, and board model.

BoardAsm.s — This file contains much of the 80960CA-specific data structures and functions necessary to configure the HK80/V960E properly. Many of the processor-specific functions must be configured as shown in this file for the HK80/V960E to function reliably.

CIO.c — This file contains the functions necessary to read, write, and configure the Z85C36 counter/timer parallel port chip.

Proc.c — The functions contained in this file provide the monitor with the commands to handle interrupts and faults as well as providing program tracing.

Proc.h — The interrupt wrapper is a relocatable assembly language module that is allocated on the stack. The interrupt table vector location is initialized to point to the wrapper and the wrapper is initialized to point to the interrupt handler. This level of indirection will reduce the necessity for assembly code.

ProcAsm.s — This file contains routines for interrupt functions.

RTC.c — This file contains functions for operating the real-time clock.

SCC.c — This file contains the functions necessary to read, write, and configure the Z85C30-16 serial controller.

SCSI.c — This file contains the functions necessary to read, write, and configure the WD33C93A SCSI controller.

VME.c — This file contains the functions necessary to initialize the VMEbus as well as examples for performing several basic VME functions.

```
The code examples in this directory/manual are provided to
give you an example of how to interface to the 80960CA and
the devices on the V960E board. The code examples consist
of the device- and processor-specific sections of the V960E
debug monitor.

Note that the complete programming environment has not been
provided. These files will compile and function properly
but are nothing more than a collection of files and perform
no useful function without the upper level programs. For more
detailed programming examples and the programming
environment contact your sales representative and ask about
the debug monitor or functional test software. These software
packages provide more complete device and environment
examples.
```

```
/************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"
#include "NvMonDefs.h"

/************************************************************************
 * Board.c: This file is the catchall for the miscellaneous board-related
 *          functions. Defined in this module are:
 *
 ***/

extern NV_HkDefined  HKFields;
extern NV_MonDefs    NvMonDefs;
char BoardModel[] = "V960E";

/************************************************************************
 * BoardInit(): Initialize the minimum hardware to the default state
 *              defined by the NV device structures.
 ***/

BoardInit()
{
    SetSerDevs();         /* Initialize serial to default state.       */
    ConfigVmeBus();       /* Initialize VMEBus to default state.       */
    InitCIOState();
    ResetSCSI();
    BPInit();
}

/************************************************************************
 * BoardConfig(): Initialize the board hardware completely to the state
 *                defined by the NV device structures.
 ***/
```



```
BoardConfig()
{
    SetSerDevs();          /* Initialize serial to NV specified state.  */
    ConfigVmeBus();        /* Initialize VMEbus to NV specified state.  */
    InitSCSIState();
    ConfigEthernet();      /* Initialize 596CA  to NV specified state.  */
    ConfigVsbBus();        /* Initialize VSBbus to NV specified state.  */
}

/************************************************************************
 * ConfigEthernet(): Initialize the Ethernet byte ordering and arbiter.
 ***/

ConfigEthernet()
{
    NV_MonDefPtr Conf = &NvMonDefs;

    if (EthByteEndian(Conf)) {              /* Set up byte ordering.     */
        *ETHERNET_LEBE = 0xFF;
    } else {
        *ETHERNET_LEBE = 0x00;
    }
    if (EthArbiterEnbl(Conf)) {             /* Enable bus arbitration.   */
        *ETHERNET_ARB_EN  = 0xFF;
    } else {
        *ETHERNET_ARB_EN  = 0x00;
    }
}

/************************************************************************
 * ConfigVsbBus(): Initialize the VSB release modes and arbiter.
 ***/

ConfigVsbBus()
{
    NV_MonDefPtr Conf = &NvMonDefs;

    if (VsbReleaseMode(Conf)) {           /* Enable VSB Master interface.*/
        *VSB_RLSE_REQ = 0x00;
    } else {
        *VSB_RLSE_REQ = 0xFF;
    }
    if (VsbMasterEnbl(Conf)) {            /* Set up Release Mode for VSB.*/
        *VSB_ENBL_ARB  = 0xFF;
    } else {
        *VSB_ENBL_ARB  = 0x00;
    }
}


/************************************************************************
 * PrStatus(): This monitor function should print useful information
 *             about the board configuration.
 ***/

PrStatus()
{
    unsigned long Temp;

    xprintf("\n     VME System controller -> ");
    if (IsSystemController()) {
        xprintf("On\n");
    } else {
        xprintf("Off\n");
    }
    Temp = HKFields.Manuf.SerialNumber;
```

```
    xprintf("    Ethernet physical ID  -> 00:80:F9:89:%2.2x:%2.2x\n",
           (Temp >> 8) & 0xFF, Temp & 0xFF);
}


/*************************************************************************
 * SetLedDisplay(): This function presents the lower four bits of the
 *                  'Value' on the user LEDs.
 ***/

SetLedDisplay(Value)
unsigned long Value;
{
    *LED1 = (~Value);
    *LED2 = (~Value >> 1);
    *LED3 = (~Value >> 2);
    *LED4 = (~Value >> 3);
}


/*************************************************************************
 * MemTop(): This function determines the address of the last long
 *           word in DRAM. The size of the DRAM is determined by the
 *           NV memory configuration.
 ***/

unsigned char *MemTop()
{
    return((unsigned char *) (0x400 + HKFields.Hardware.DRAMSize - 4));
}


/*************************************************************************
 * MemBase(): This function determines the base address of the available
 *            DRAM. The base of RAM is determined by the compiler created
 *            variable 'end' which indicates the end of the 'bss' section.
 ***/

extern unsigned long end[];
unsigned char *MemBase()
{
    return((unsigned char *) end);
}


/*************************************************************************
 * Delay(): This function is intended to provide a fixed delay for
 *          timing. It isn't very accurate ! (very compiler dependent).
 ***/

#define HUND_SEC_DELAY 25000

Delay(HundSec)
int HundSec;
{
    volatile int i;

    for(i=HundSec * HUND_SEC_DELAY; i; i--);
}


/*************************************************************************
 * IntErr(): When an unexpected interrupt is received it is necessary to
 *           remove the error condition before returning to the monitor.
 *           This function is called from the function UnExpIntr() which
 *           parses the interrupt record for the address and the vector
```



```
 *          associated with the interrupt. The device is dealt with
 *          accordingly and the monitor is resumed.
 *
 *          Because the interrupt condition may be a program which
 *          is determined to beat its head into a wall it is
 *          necessary to abort the program and return directly to
 *          the monitor level. This is done in a function IntRecov()
 *          which causes the processor to return into the line editor.
 ***/

/* Generic resonse messages */
static char NMIEStr[]   = "\n\n^GUnexpected NMI Exception at 0x%.8X - %s(%s)\n";
static char DevIntStr[] = "\n\n^GUnexpected %s Interrupt at 0x%.8X\n";
static char UnkIntStr[] = "\n\n^GUnexpected Interrupt at 0x%.8X Vector 0x%x\n";

/* Error type for NMI */
char *BusMasterErrTable[] = { "??? unknown", "Ethernet", "VMEBus", "80960CA" };
char *NmiErrTable[]       = { "Bus Error",   "Parity Error" };

IntrErr(Addr, Vector)
long Vector;
char *Addr;
{
    unsigned char Status;

    switch(Vector) {
        case NMI_VECTOR: {
            Status = *STATUS_LATCH;
            xprintf(NMIEStr,Addr,NmiErrTable[Status & 0x01],
                BusMasterErrTable[((Status & 0x06) >> 1)]);
            break;
        }
        case CIO_VECTOR: {
            InitCIOState();
            xprintf(DevIntStr,"CIO",Addr);
            break;
        }
        case SCSI_VECTOR: {
            ResetSCSI();
            xprintf(DevIntStr,"SCSI",Addr);
            break;
        }
        case SCCAB_VECTOR:
        case SCCCD_VECTOR: {
            SetSerDevs();
            xprintf(DevIntStr,"SCC",Addr);
            break;
        }
        case ETH_VECTOR: {
            xprintf(DevIntStr,"ETHERNET",Addr);
            break;
        }
        case IPL2_VECTOR:
        case IPL1_VECTOR:
        case IPL0_VECTOR: {
            xprintf(DevIntStr,"VIC-IPL(%x)",Addr, Vector);
            UnMaskVMEInt(0);
            break;
        }
        default: {
            xprintf(UnkIntStr, Addr, Vector);
            break;
        }
    }
    DumpRegs();
    FlushCache();
```

```
    IntRecovery();          /* Restart Monitor.*/
}
```

```
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

/***************************************************************************
 * Board.h: This file describes the v960e hardware addresses and data
 *          structures. Included in this file are the definitions for:
 *
 *           80960CA Interrupt Vector Assignments
 *           Z85C36 CIO Counter Timer
 *           Z85C30 SCC Serial Controller, Ports A-D
 *           WD33C93 SCSI Controller
 *           DS1216F Real-time Clock
 *           VSB Interface
 *           User LEDs
 *           NMI Status Latch
 *           CENTRONICS Interface
 *           82596CA Ethernet Controller
 *           VIC068 VMEBus Controller and Configuration Registers
 *           28C64 EEPROM
 *
 ***/

/***************************************************************************
 * This section defines the interrupt vectors and mask bit associated
 * with each v960 interrupt source.
 */

#define   NMI_VECTOR          0xF8   /* Vector Definitions for the V960 */
#define   DMA_CHAN3_VECTOR    0xC2   /* are fixed according to how the  */
#define   DMA_CHAN2_VECTOR    0xB2   /* Interrupt Map registers are set */
#define   DMA_CHAN1_VECTOR    0xA2   /* This definition is accurate for */
#define   DMA_CHAN0_VECTOR    0x92   /* the Map defined in the file     */
#define   CIO_VECTOR          0x82   /* 80960CAs.s                      */
#define   SCSI_VECTOR         0x72
#define   SCCAB_VECTOR        0x62
#define   SCCCD_VECTOR        0x52
```



```
#define   ETH_VECTOR          0x42
#define   IPL2_VECTOR         0x32
#define   IPL1_VECTOR         0x22
#define   IPL0_VECTOR         0x12

#define   DMA_CHAN3_MASK      0x800   /* The Interrupt Mask Bits are     */
#define   DMA_CHAN2_MASK      0x400   /* fixed and should never be       */
#define   DMA_CHAN1_MASK      0x200   /* modified. The mask bits are     */
#define   DMA_CHAN0_MASK      0x100   /* determined by the v960          */
#define   CIO_INT_MASK        0x080   /* hardware.                       */
#define   SCSI_INT_MASK       0x040
#define   SCCAB_INT_MASK      0x020
#define   SCCCD_INT_MASK      0x010
#define   ETH_INT_MASK        0x008
#define   IPL2_INT_MASK       0x004
#define   IPL1_INT_MASK       0x002
#define   IPL0_INT_MASK       0x001

/***************************************************************************
 * CIO:  Definitions for the Z85C36 CIO Counter Timer and parallel ports
 ***/

#define    CIOPORT     0x02E00000

#define CIO_AData       ((volatile unsigned char *) (CIOPORT + 0x10))
#define CIO_BData       ((volatile unsigned char *) (CIOPORT + 0x08))
#define CIO_CData       ((volatile unsigned char *) (CIOPORT + 0x00))
#define CIO_CTRL        ((volatile unsigned char *) (CIOPORT + 0x18))

/***************************************************************************
 * SCC:  Definition for the Z85C30 serial ports A-D.
 ***/

#define SCC_REG_SPREAD  0x0F       /* Distance between registers      */
#define SCC_PORT_SPREAD 0x08       /* Distance between ports          */

#define BaudToTimeConst(baud)    ((500000 / baud) - 2)

struct SCCPort {                   /* Serial device structure         */
    unsigned char Control;
    unsigned char Dummy[SCC_REG_SPREAD];
    unsigned char Data;
};                                 /* Define port addresses           */

#define SCC_PORTB    ((struct SCCPort *) 0x02200000)
#define SCC_PORTA    ((struct SCCPort *) ((int) SCC_PORTB + SCC_PORT_SPREAD))
#define SCC_PORTD    ((struct SCCPort *) 0x02300000)
#define SCC_PORTC    ((struct SCCPort *) ((int) SCC_PORTD + SCC_PORT_SPREAD))

/***************************************************************************
 * SCSI:  Definition for the WD33C93 SCSI interface.
 ***/

#define SCSI_ADDR      0x02400000      /* Base Address of SCSI schip   */

#define SCSI_RESET     ((unsigned char *) 0x02000140)   /* Bus reset   */

struct SCSIChip {                      /* Define SCSI structure        */
    unsigned char SC_AddrPtr;
    unsigned char SC_Dummy[7];
    unsigned char SC_Register;
};                                   /* Define macros to read and write */
```

```
#define SCSI ((struct SCSIChip *) SCSI_ADDR)

#define SCWriteReg(Reg, Val)      SCSI->SC_AddrPtr = Reg;\
                                  SCSI->SC_Register = Val

#define SCReadReg(Reg, Val)       SCSI->SC_AddrPtr = Reg;\
                                  Val = SCSI->SC_Register


/*************************************************************************
 * SCSI bus interface controller registers
 ***/

#define SREG_OWNID       0x00
#define SREG_CTRL        0x01
#define SREG_TIMEOUT     0x02
#define SREG_TSECT       0x03
#define SREG_THEAD       0x04
#define SREG_TCYLH       0x05
#define SREG_TCYLL       0x06
#define SREG_HH_LADR     0x07
#define SREG_HM_LADR     0x08
#define SREG_LM_LADR     0x09
#define SREG_LL_LADR     0x0A
#define SREG_SECT        0x0B
#define SREG_HEAD        0x0C
#define SREG_CYLH        0x0D
#define SREG_CYLL        0x0E
#define SREG_TLUN        0x0F
#define SREG_CPHASE      0x10
#define SREG_SYNT        0x11
#define SREG_HTCNT       0x12
#define SREG_MTCNT       0x13
#define SREG_LTCNT       0x14
#define SREG_DEST_ID     0x15
#define SREG_SRC_ID      0x16
#define SREG_SCSI_STAT   0x17
#define SREG_CMD         0x18
#define SREG_DATA        0x19

#define SREG_CDB1        0x03
#define SREG_CDB2        0x04
#define SREG_CDB3        0x05
#define SREG_CDB4        0x06
#define SREG_CDB5        0x07
#define SREG_CDB6        0x08
#define SREG_CDB7        0x09
#define SREG_CDB8        0x0A
#define SREG_CDB9        0x0B
#define SREG_CDB10       0x0C
#define SREG_CDB11       0x0D
#define SREG_CDB12       0x0E


/*************************************************************************
 *  DMA Control Words for SCSI Read and Write transfers.
 ***/

#define SCDMA_RCTRL_WORD  0x000000A3  /* Control word for SCSI DMA read  */
#define SCDMA_WCTRL_WORD  0x000000DC  /* Control word for SCSI DMA write */

#define SCDMA_CHANNEL     0x3         /* Channel associated with SCSI    */
#define SCDMA_ADDRESS     0x02D00000  /* DMA Acknowledge address         */


/*************************************************************************
```



```
 * RTC: Data structures and addresses for the real-time clock
 ***/

#define WATCHBASE       ((volatile unsigned char *)  0x02F00000)
#define WR0_WATCH       ((volatile unsigned char *)  (WATCHBASE    ))
#define WR1_WATCH       ((volatile unsigned char *)  (WATCHBASE + 3))
#define RD_WATCH        ((volatile unsigned char *)  (WATCHBASE + 4))

struct rtc_data {          /*  D7  D6  D5  D4   :  D3  D2  D1  D0  */
  unsigned char dotsec;    /* -- 0.1 sec -----  : -- 0.01 sec ----- */
  unsigned char sec;       /* -- 10 sec ------  : -- seconds ------ */
  unsigned char min;       /* ---10 min ------  : -- minutes ------ */
  unsigned char hour;      /* -- A 0 B Hr ----  : -- hours -------- */
  unsigned char weekday;   /* -- 0 0 0 1 -----  : -- day ---------- */
  unsigned char date;      /* -- 10 date------  : -- date --------- */
  unsigned char month;     /* -- 10 Month ----  : -- month -------- */
  unsigned char year;      /* -- 10 year -----  : -- year --------- */
};


/*************************************************************************
 * VSB: Control bits associated with the VSB interface
 ***/

#define VSB_ENBL_ARB    ((unsigned char *) 0x02000018)
#define VSB_RLSE_REQ    ((unsigned char *) 0x02000010)


/*************************************************************************
 * LED: This is the definitions for the four user LEDS.
 */

#define LED1            ((unsigned char *) 0x02000020)
#define LED2            ((unsigned char *) 0x02000028)
#define LED3            ((unsigned char *) 0x02000030)
#define LED4            ((unsigned char *) 0x02000038)


/*************************************************************************
 * The status latch returns a 3 bit error code indicating the state of
 * the system when an NMI exception has occurred. The format of the latch is:
 *
 *          Bits
 *      2     1     0                        Indication
 *   -------------------------------------------------------------
 *      x     x     0        NMI caused by a bus error.
 *      x     x     1        NMI caused by a parity error.
 *
 *      0     1     x        NMI occurred while Ethernet owned the bus.
 *      1     0     x        NMI occurred while slave VMEBus owned the bus.
 *      1     1     x        NMI occurred while the processor owned the bus.
 *
 *      0     0     x        Bus ownership unknown (shouldn't occur).
 *
 */

#define STATUS_LATCH       ((unsigned char *) 0x02100000)

#define STATUS_PERR(x)     ((x & 0x1) == 1)
#define STATUS_BERR(x)     ((x & 0x1) == 0)

#define STATUS_80960CA(x)  ((x & 0x6) == 6)
#define STATUS_82596CA(x)  ((x & 0x6) == 2)
#define STATUS_VMESLAVE(x) ((x & 0x6) == 4)
```

```
/***************************************************************************
 * CENTRONICS:  Definition for the Centronics Interface
 */

#define CENT_BASE               ((unsigned char *) 0x02C00000)

#define CENT_DATA               ((unsigned char *) (CENT_BASE + 0x00))
#define CENT_STATUS             ((unsigned char *) (CENT_BASE + 0x00))
#define CENT_SET_STROBE         ((unsigned char *) (CENT_BASE + 0x08))
#define CENT_CLEAR_STROBE       ((unsigned char *) (CENT_BASE + 0x08))
#define CENT_SET_INIT           ((unsigned char *) (CENT_BASE + 0x10))
#define CENT_CLEAR_INIT         ((unsigned char *) (CENT_BASE + 0x10))
#define CENT_INT_ENBL           ((unsigned char *) (CENT_BASE + 0x18))
#define CENT_INT_CLR            ((unsigned char *) (CENT_BASE + 0x18))


/***************************************************************************
 * 82596CA: Definition for the Ethernet data structures and addresses
 ***/

#define ETHERNET_PORT    ((unsigned char *) 0x02800000)
#define ETHERNET_CA      ((unsigned char *) 0x02900000)
#define ETHERNET_LEBE    ((unsigned char *) 0x02000008)
#define ETHERNET_ARB_EN  ((unsigned char *) 0x020001C0)


/***************************************************************************
 * VIC068: Definition for the VIC Chip Registers
 ***/

#define VIC             ((struct VicChip *) 0x02A00000)

#define VIC_IACK_BASE   ((unsigned char *)  0x02B00000)

#define VIC_IACK_IPL0   ((unsigned char *)  (VIC_IACK_BASE + 0x10))
#define VIC_IACK_IPL1   ((unsigned char *)  (VIC_IACK_BASE + 0x04))
#define VIC_IACK_IPL2   ((unsigned char *)  (VIC_IACK_BASE + 0x08))

typedef struct VicReg {         /* Structure to define register spacing */
        unsigned char Reg;
        unsigned char Dummy[3];
} VicReg;

struct VicChip {                /* VIC068 Register description         */
    VicReg VMEIntIntCntrl;
    VicReg VMEIntCntrl[7];
    VicReg DMAIntCntrl;
    VicReg LocIntCntrl[7];
    VicReg ICGSIntCntrl;
    VicReg ICMSIntCntrl;
    VicReg ErrIntCntrl;
    VicReg ICGSVecBase;
    VicReg ICMSVecBase;
    VicReg LocVecBase;
    VicReg ErrVecBase;
    VicReg ICSwitch;
    VicReg ICR[8];
    VicReg VMEIntReqStat;
    VicReg VMEIntVec[7];
    VicReg TranTimeOut;
    VicReg LocBusTiming;
    VicReg BlkTranDef;
    VicReg VMEConfig;
    VicReg ArbReqConfig;
    VicReg AddModSrc;
    VicReg BerrStat;
```



```
    VicReg DMAStat;
    VicReg SlvSel[2][2];
    VicReg RelCntrl;
    VicReg BlkTranCntrl;
    VicReg BlkTranLen[2];
    VicReg SysReset;
};


/***************************************************************************
 * Mailbox structure definitions as they would appear on the
 * VMEBus.
 ***/

typedef struct ICReg {          /* This how the Interconnect Registers */
    unsigned char Reg;          /* appear on the VMEbus                */
    unsigned char Dummy;
} ICReg;

typedef struct Switch {         /* The Module and Global Switches      */
    unsigned char Set;
    unsigned char Clear;
} Switch;

typedef struct MailBox {        /* The Mailbox data structure consists */
    ICReg  ICR[8];              /* of 8 Interconnect registers 4       */
    Switch ICGS[4];             /* module and 4 global switches. The   */
    Switch Dummy[4];            /* base address of this structure is   */
    Switch ICMS[4];             /* determined by the CIO port B value. */
} MailBox;


/***************************************************************************
 * VMEBus configuration registers for slave mapping
 ***/

#define SLAVE_EXT_ENBL  ((unsigned char *) 0x02000100)
#define SLAVE_STD_ENBL  ((unsigned char *) 0x020000C0)
#define SLAVE_SHT_ENBL  ((unsigned char *) 0x02000080)


/***************************************************************************
 * 28C64 EEPROM:  Definition for the NV Memory Interface
 ***/

#define NV_BASE         0x02700000      /* Base address of NV memory         */
#define NV_SIZE         0x00002000      /* Size in bytes of NV memory        */
#define NV_PROTECTED    0x00001800      /* Beginning of protected NV memory */
#define NV_MON_DEFS     0x00001600      /* Beginning of monitor NV defs.     */

#define NV_MAX_NBR_WRITES     10000     /* Limit on the number of writes     */
#define NV_PAGE_SIZE          1         /* Page size of 32 for fast program */
#define NV_SPACING            8         /* Number of bytes between bytes     */
```

```
/**************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

/***************************************************************************
 *      Bug.h:  This file is intended to provide standard constants and
 *              data structures common to all files independent of
 *              processor compiler and board model.
 ***/

/***************************************************************************
 *      Define the constants for TRUE, FALSE, NULL and ERROR.
 ***/

#define NULL    0
#define TRUE    1
#define FALSE   0
#define ERROR   -1

#define FAILED  0
#define PASSED  1

#define   READ   0
#define   WRITE  1

/***************************************************************************
 *      Character definitions
 ***/

#define   EOF  0
#define   DEL  0x7F
#define   ESC  0x1B
#define   SP   ' '
#define   BS   '\b'
#define   CR   '\r'
#define   LF   '\n'
#define   TAB  '\t'
```



```
/***************************************************************************
 *      UNIX style time structure
 ***/

struct tm {
    unsigned long tm_fsec;      /* fractions of seconds (0 - 99) */
    unsigned long tm_sec;       /* seconds (0 - 59) */
    unsigned long tm_min;       /* minutes (0 - 59) */
    unsigned long tm_hour;      /* hours   (0 - 23) */
    unsigned long tm_mday;      /* day of month (1 - 31) */
    unsigned long tm_mon;       /* month of year (0 - 11) */
    unsigned long tm_year;      /* year - 1900 */
    unsigned long tm_wday;      /* day of week (sunday = 0) */
};

typedef struct tm tm;
```

```
# ---------------------------------------------------------------------------
# This file contains much of the 80960CA-specific data structures and functions
# necessary to configure the v960 properly. Many of the processor-specific
# functions must be configured as seen in this file for the v960 to function
# reliably.
#
# ---------------------------------------------------------------------------

                .file   "BugAsm.s"
                .text
                .align 4

# ---------------------------------------------------------------------------
# Initialization:  The initialization of the v960 includes reading a new prcb
#                  and control table, which must be located in RAM. Also, the
#                  Vector and Fault tables are initialized, and the board is
#                  initialized to a known state.
# ---------------------------------------------------------------------------

                .set    REQUEST_INTR,       0x000
                .set    INVALID_CACHE,      0x100
                .set    CONFIG_CACHE,       0x200
                .set    RE_INITIALIZE,      0x300
                .set    LD_CTRL_REG0,       0x400
                .set    LD_CTRL_REG1,       0x401
                .set    LD_CTRL_REG2,       0x402
                .set    LD_CTRL_REG3,       0x403
                .set    LD_CTRL_REG4,       0x404
                .set    LD_CTRL_REG5,       0x405
                .set    LD_CTRL_REG6,       0x406

                .globl _start_ip
                .globl _ColdStart
                .globl _MonEntryPt
                .globl _end
                .globl __RcvTrace

# Pause 500 mSec for RAM and then do 8 RAS/CAS cycles to initialize
# memory.

_MonEntryPt:
_ColdStart:
                ldconst 0x02000020, r4
                ldconst 0xFFFFFFFF, r5
                st      r5, (r4)                    # Clear LED's
                st      r5, 0x08(r4)
                st      r5, 0x10(r4)
                st      r5, 0x18(r4)
_start_ip:      ldconst 0x000, r4                   # Counter
                ldconst 0x400, r6                   # RAM Address
                ldconst 0x010, r5                   # Loop Count
RamInit:        st      r5, (r6)
                addo    1, r4, r4
                cmpobne r4, r5, RamInit             # Write to RAM

                ldconst 0xeeeeeeee, r3
                lda     _int_table, r4
                lda     _end, r5

ClearSysMem:    st      r3, (r4)
                addo    4, r4, r4
                cmpobne r4, r5, ClearSysMem

                ldconst RE_INITIALIZE,r3    # Re-initialize op Code
                lda     getpcb,r4           # Address to start execution
                lda     newprcb,r5          # Address of new prcb block
```



```
                sysctl  r3, r4, r5          # Reinitialize 80960CA.

getpcb:         ldconst LD_CTRL_REG2,r3     # Read Register Group.
                sysctl  r3, r3, r3          # Number 2 from ctrl table.

                ldconst LD_CTRL_REG3,r3     # Read Register Group.
                sysctl  r3, r3, r3          # Number 3 from ctrl table.

                ldconst LD_CTRL_REG4,r3     # Read Register Group.
                sysctl  r3, r3, r3          # Number 4 from ctrl table.

                ldconst LD_CTRL_REG5,r3     # Read Register Group.
                sysctl  r3, r3, r3          # Number 5 from ctrl table.

SetState:       ldconst 0x1F1002,r3         # Modify state to priority 0,
                ldconst 0x000002,r4         # supervisory / executing state
                modpc   0,r3,r4
                ldconst 0x001000,r3         # Modify arithmetic controls so
                ldconst 0x001000,r4         # no imp flts and iof masked.
                modac   0,r3,r4

                lda     _sup_stack,r4       # New supervisory stack
                mov     r4,sp               # address.
                mov     0,g14               # Fix compiler bug.

StartMon:       callx   _VectInit           # Initialize Vector Table.
                callx   _FaultInit          # Initialize Fault Table.

                callx   _StartMonitor               # Start program.
                ret

                .globl __warm

__warm:         callx   _VectInit           # Initialize Vector Table.
                callx   _FaultInit          # Initialize Fault Table.
                b       SetState

                .globl _IntRecovery
#
# Note that this section is necessary for the B step parts to work.
#

_IntRecovery:   ldconst RE_INITIALIZE,r3    # Re-initialize op Code
                lda     Recov0,r4           # Address to start execution
                lda     newprcb,r5          # Addesss of new prcb block
                sysctl  r3, r4, r5          # Reinitialize 80960CA.

Recov0:         ldconst 0x1F1002,r3         # Modify state to priority 0,
                ldconst 0x000002,r4         # supervisory / executing state.
                modpc   0,r3,r4
                ldconst 0x001000,r3         # Modify arithmetic controls so
                ldconst 0x001000,r4         # no imp flts and iof masked.
                modac   0,r3,r4

                lda     _sup_stack,r4       # New supervisory stack
                mov     r4,sp               # address.
                mov     0,g14               # Fix compiler bug.
                callx   _LineEdit

# Note that this section is necessary for the A step parts to work.
#
#               lda     Recov1,rip       # returns to ExcInt
#               ret
#Recov1:        lda     Recov2,rip       # returns to UnExpInt
#               ret
#Recov2:        lda     _LineEdit,rip    # returns to before interrupt
```

```
#               ret

                .globl _AtomicModify

_AtomicModify:  atmod g0, g1, g2
                ret

# ---------------------------------------------------------------------------
#
# Region Table: There are three different ways that the v960 memory can be
#               configured.
#
#                1) 32-bit, burst enabled is the configuration for region 0
#                   and should never be configured otherwise. This allows the
#                   on card DRAM to use burst.
#
#                2) 32-bit, burst disabled is the configuration for regions
#                   1 through 14 and should never be configured otherwise.
#                   Region 15 should be configured this way when ROM has been
#                   inhibited.
#
#                3) 8-bit, burst disabled is the configuration for region 15
#                   when ROM is not inhibited. This allows the ROM to be
#                   accessed as an 8-bit-wide memory.
#
# ---------------------------------------------------------------------------

                .set    BURST_32BIT,      0x00100003
                .set    NONBURST_32BIT,   0x00100002
                .set    NONBURST_8BIT,    0x00000002

# ---------------------------------------------------------------------------
# PRCB:  The processor control block indicates the interrupt and fault tables
#        to be used, sets up a pointer to the new control table and initailizes
#        the stacks, caches and control registers.
# ---------------------------------------------------------------------------

                .align 4

newprcb:        .word   _flt_table          # Fault table base address (ram)
                .word   _ctl_table          # Control table base address (rom)
                .word   0x00001000          # AC register initial image
                                            # Mask integer overflow faults
                .word   0x40000001          # Fault Configuration Word
                                            # (Mask unaligned bus req. faults)
                .word   _int_table          # Interrupt table base address
                .word   _sys_table          # System procedure table base
                .word   0x00000000          # Reserved
                .word   _int_stack          # Interrupt stack pointer
                .word   0x00010000          # Instruction cache config
                .word   0x00000005          # Register cache config
                                            # Num cached register sets = 5
                .space  2*4                 # Make an even quad word

# ---------------------------------------------------------------------------
# Control Table: The control table is organized as 7 groups of 4 words each.
#                Groups 2-5 indicate the memory region configurations. Group 0
#                is the breakpoint registers, Group 1 the interrupt Map and
#                control registers, and Group 6 is the misc. registers.
# ---------------------------------------------------------------------------

                .align 4
                .globl _ctl_table
                .globl _Region15
```



```
_ctl_table:
                # ----- Breakpoint Registers ---
CtlGroup0:      .word   0x00000000          # IPB0 IP Breakpoint register 0
                .word   0x00000000          # IPB1 IP Breakpoint register 1
                .word   0x00000000          # DAB0 Data Addr Breakpoint reg
                .word   0x00000000          # DAB1 Data Addr Breakpoint reg

                # ----- Interrupt map and control registers ---
CtlGroup1:      .word   0x00004321          # IMAP0 Interrupt Map register 0
                .word   0x00008765          # IMAP1 Interrupt Map register 1
                .word   0x0000CBA9          # IMAP2 Interrupt Map register 2
                .word   0x00008000          # Interrupt controller

                # ----- Memory Region Configuration Registers ---
CtlGroup2:      .word      BURST_32BIT          # Region 0
                .word   NONBURST_32BIT          # Region 1
                .word   NONBURST_32BIT          # Region 2
                .word   NONBURST_32BIT          # Region 3
CtlGroup3:      .word   NONBURST_32BIT          # Region 4
                .word   NONBURST_32BIT          # Region 5
                .word   NONBURST_32BIT          # Region 6
                .word   NONBURST_32BIT          # Region 7
CtlGroup4:      .word   NONBURST_32BIT          # Region 8
                .word   NONBURST_32BIT          # Region 9
                .word   NONBURST_32BIT          # Region 10
                .word   NONBURST_32BIT          # Region 11
CtlGroup5:      .word   NONBURST_32BIT          # Region 12
                .word   NONBURST_32BIT          # Region 13
                .word   NONBURST_32BIT          # Region 14
_Region15:      .word   NONBURST_8BIT           # Region 15

                # ----- Breakpoint, Trace, and Bus Control registers ----
CtlGroup6:
                .word   0x00000000          # N/U     Not Used.
                .word   0x00000000          # BPCON   Breakpoint Control Reg
                .word   0x00000001          # TC      Trace Controls
                .word   0x00000001          # BCON    Bus Configuration Ctrl


# ---------------------------------------------------------------------------
# STACK DEFINITIONS: The following data definitions define the stacks for the
#                    80960CA. The interrupt, supervisory and user stacks are
#                    defined. Depending on the application, the size of these
#                    definitions may be increased or decreased.
#
# DATA STRUCTURES:   Space for the interrupt, fault and system procedure
#                    tables are defined here. The size of these tables is a
#                    fixed quantity. Details of how these structures are used
#                    can be found in the 80960CA manual. The initialization of
#                    these structures is performed by other functions.
#
# ---------------------------------------------------------------------------

                .align 4
                .data

                .globl _int_table
                .bss    _int_table,  0x0420, 8

                .globl _flt_table
                .bss    _flt_table,  0x0200, 8

                .globl _sys_table
                .bss    _sys_table,  0x0200, 8

                .globl _usr_stack
```

```
                .bss    _usr_stack,  0x0800, 8

                .globl _int_stack
                .bss    _int_stack,  0x0800, 8

                .globl _sup_stack
                .bss    _sup_stack,  0x2000, 8


 # ----------------------------------------------------------------------------
 # Powerup detection: The following routines determine powerup conditions and
 #                    allow the user to set the powerup magic number
 # ----------------------------------------------------------------------------
                .set    POWER_UP_MAGIC_NUMBER, 0x52364767
                .set    POWER_UP_LOCATION,      0x00000004

                .text
                .align 4

                .globl   _IsPowerUp

_IsPowerUp:     ldconst POWER_UP_LOCATION, r5
                ldconst POWER_UP_MAGIC_NUMBER, r6
                ld      (r5),r4
                cmpobne r4, r6, IsPowerUP
                mov     0x0,g0
                ret

IsPowerUP:      mov     0x1,g0
                ret


 # ----------------------------------------------------------------------------
 # Powerup detection: The following routines determine powerup conditions and
 #                    allow the user to set the powerup magic number
 # ----------------------------------------------------------------------------

                .text
                .align 4

                .globl   _SetNotPowerUp

_SetNotPowerUp: ldconst POWER_UP_LOCATION, r5
                ldconst POWER_UP_MAGIC_NUMBER, r6
                st          r6, (r5)
                ret
```

```
/***********************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"

/***********************************************************************
 * CIO.c:  This file contains the functions necessary to read, write and
 *         configure the Z85C36 Counter Timer / parallel port chip.
 *         The functions defined in this module are listed below:
 *
 *                ResetCIO()        InitCIOState()    StartTimer()
 *                WriteCIOPortA()   WriteCIOPortB()   WriteCIOPortC()
 *                ciointr()
 ***/

/***********************************************************************
 * This file contains all the CIO specific subroutines necessary to reset,
 * initialize, read and write to the CIO ports and counter timers.
 *
 * ResetCIO():        Sets the CIO to the hardware reset state.
 *
 * InitCIOState():    This is the default state of the CIO and it should be set
 *                    to this state at reset.
 *
 * WriteCIOPortA()
 * WriteCIOPortB()
 * WriteCIOPortC():   These are the routines used to write to ports A-C of
 *                    the CIO.
 *
 */


/***********************************************************************
 * ResetCIO(): This function resets the counter timer regardless
 *             what state the chip might be in.
```



```
 ***/

ResetCIO()
{
    volatile unsigned char *p, c;

    p = CIO_CTRL;
    c = *p;             /* make sure we're waiting for a reg ptr  */
    *p = 0x00;          /* master int ctl reg ptr          */
    c = *p;             /* (must be a good reason to do it again) */
    *p = 0x00;
    *p = 0x01;          /* reset bit on, off               */
    *p = 0x00;
}


/***********************************************************************
 * InitCIOState(): This function initializes the counter timer to the
 *                 state expected by the monitor. The configuration sets
 *                 the parallel ports as bit output ports so that the
 *                 VME slave comparison addresses can be written to ports
 *                 A, B and C.
 ***/

InitCIOState()
{
    static unsigned char ciotable[] = {
        0x00,  0x00,     /* Clear register interrupts VIS     */
        0x28,  0x00,     /* Port B as bit port                */
        0x20,  0x00,     /* Port A as bit port                */
        0x2B,  0x00,     /* Port B all outputs                */
        0x23,  0x00,     /* Port A all outputs                */
        0x06,  0x00,     /* Port C all outputs                */
        0x2C,  0x00,     /* Port B normal i/o                 */
        0x24,  0x00,     /* Port A normal i/o                 */
        0x07,  0x00,     /* Port C normal i/o                 */
        0x2D,  0x00,     /* All pattern registers cleared     */
        0x09,  0x20,     /* Clear interrupts                  */
        0x01,  0x94      /* enable port A and port B          */

    };
    register int cnt;
    volatile unsigned char *p;

    ResetCIO();
    p = CIO_CTRL;
    for(cnt = 0; cnt < sizeof(ciotable); cnt++)
        *p = ciotable[cnt];

}


/***********************************************************************
 * WriteCIOPortA():
 * WriteCIOPortB():
 * WriteCIOPortC():  These functions provide the ability to write to the
 *                   CIO output ports. Ports A, B and C are used for the
 *                   VMEbus slave maps for the Extended, Short and Standard
 *                   spaces, respectively.
 ***/

WriteCIOPortA(Data)
unsigned char Data;
{
    *CIO_AData = Data;
}
```

```
WriteCIOPortB(Data)
unsigned char Data;
{
    *CIO_BData = Data;
}

WriteCIOPortC(Data)
unsigned char Data;
{
    *CIO_CData = Data;
}


/**************************************************************************
 * StartTimer():  This function is intended to provide an example of how
 *                to initialize the CIO counter timers. Here the CIO is
 *                initialized, the interrupt handler is attached, and then
 *                the counter is started. In this example the location
 *                'NumTicks' is incremented for every interrupt received
 *                and a dot is printed every second. This function is
 *                turned off by calling InitCIOState() and disconnecting
 *                the interrupt handler.
 ***/

volatile int NumTicks;

StartTimer()
{
    int cnt;
    int ciointr();
    static unsigned char ctitable[] = {
        0x00,    0x86,                      /* Enable master interrupt VIS     */
        0x1E,    0x80,                      /* Channel 3 Continuous           */
        0x1A,    0x82,    0x1B,    0x35,    /* Channel 3 Count (1/60th sec)    */
        0x0C,    0x20,                      /* Clear IP and IUS for channel 3  */
        0x1D,    0x80,                      /* Channel 2 Continuous           */
        0x18,    0x50,    0x19,    0x8A,    /* Channel 2 Count (1/97th sec)    */
        0x0B,    0x20,                      /* Clear IP and IUS for channel 2  */

        0x1C,    0x80,                      /* Channel 1 Continuous           */
        0x16,    0x31,    0x17,    0xC3,    /* Channel 1 Count (1/157th sec)   */
        0x0A,    0x20,                      /* Clear IP and IUS for channel 1  */
        0x05,    0x00,                      /* Set up port 3   */
        0x06,    0xFF,
        0x07,    0x00,
        0x01,    0x40,                      /* Enable counters 1, 2, and 3     */
        0x0C,    0xC6,                      /* Enable Interrupts, start count  */
        0x0B,    0xC6,
        0x0A,    0xC6
    };

    xprintf("NumTicks loaded at 0x%x\n", &NumTicks);
    ConnectHandler(CIO_VECTOR, ciointr);
    NumTicks = 0;
    ResetCIO();
    *CIO_CTRL = 0x04;
    *CIO_CTRL = 0x80;
    for(cnt = 0; cnt < sizeof(ctitable); cnt++)
        *CIO_CTRL = ctitable[cnt];
    UnMaskInts(0x80);
}

/**************************************************************************
 * ciointr():  This is the interrupt handler for the counter timer.
```



```
 *              This function removes the interrupt in the device and
 *              then clears the interrupt in the processor.
 ***/

static ciointr()
{
    unsigned char Vector, Status;
    int i;

    for( i = 0 ; i < 0x1000 ; i++);
    Vector = *CIO_CTRL;
    *CIO_CTRL = 0x04;
    Vector = *CIO_CTRL;

    *CIO_CTRL = 0x0A;
    Status = *CIO_CTRL;
    *CIO_CTRL = 0x0A;
    if ((NumTicks++ % 157) == 0) {
        PutC('.');
    }
    *CIO_CTRL = 0x24;
    for( i = 0 ; i < 0x1000 ; i++);    /* This delay is necessary to allow    */
                                       /* the CIO to drive the interrupt high.*/
    ClrIntPend();                      /* The interrupt mask in the processor */
                                       /* must be cleared in the processor.   */
}
```

```c
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Proc.h"

/***************************************************************************
 *  Proc.c: The functions contained in this file provide the monitor
 *          with commands to handle interrupts and faults as well as
 *          providing program tracing. The functions contained in
 *          this module are listed below:
 *
 *               VectToVectAddr()      VectInit()
 *               ConnectHandler()      DisConnectHandler()
 *               FaultInit()           FaultErr()
 *
 *          And trace files:
 *
 *               Trace()        ExecTrace()     BPInit()
 *               Step()         BPoint()        DispTrace()
 *               RcvTrace()     SaveState()     OneWordInstr()
 *               ModeToMask()
 *
 ***/

extern unsigned long int_table[];   /* Address of interrupt table */
extern unsigned long flt_table[];   /* Address of fault table     */

unsigned long TraceEnabled;  /* Trace controls register */
unsigned long TraceMask;  /* Trace controls register */
unsigned char TraceFlag;

/***************************************************************************
 * VectToVectAddr():  Converts 'Vector" to a vector address contained in
 *                    the interrupt table.
 ***/
```



```c
unsigned long *VecToVecAddr(Vector)
unsigned long Vector;
{
   return((unsigned long *) (int_table + 1 + Vector));
}

/***************************************************************************
 * VectInit():  The Vector table consists of 36 Bytes of Pending interrupt
 *              bits followed by 992 Bytes of Vector Table. A total of
 *              1028 bytes of data.
 *
 *              Note that the NMI interrupt always resides in the
 *              80960CA Data RAM at location 0.
 ***/

VectInit()
{
  int i, UnExpIntr();
  unsigned long *VectPtr;

  VectPtr = int_table;
  for(i = 0; i < 9; i++) {
    *VectPtr++ = 0;
  }
  for(i = 9; i < 257; i++) {
    *VectPtr++ = (unsigned long) UnExpIntr;
  }
  *((unsigned long *) 0) = (unsigned long) UnExpIntr;
}

/***************************************************************************
 * ConnectHandler():  The function allocates a interrupt wrapper, links
 *                    the wrapper into the interrupt table and then
 *                    initializes the wrapper to call the Handler address.
 ***/

struct IntWrapper IntCode = {
  0xb2805000, 0xb2a06010, 0xb2c06020, 0xa2e06030, 0x8c200040, 0x59084004,
  0x9027f400, 0xfffffff8, 0x8c2800ff, 0x58894084, 0x5c801602, 0x86003000,
  0xeeeeeeee,
  0x8c200040, 0x59084104, 0xb0805000, 0xb0a06010,
  0xb0c06020, 0xa0e06030, 0x0a000000,
  0xeeeeeeee, 0xeeeeeeee, 0xeeeeeeee
};

ConnectHandler(Vector, Handler)
unsigned long Vector;
int Handler();
{
    unsigned long *CodePtr, *MemPtr;
    struct IntWrapper *Wrapper;
    int i, UnExpIntr();
    unsigned long *VectPtr, *VecToVecAddr();
    char *Malloc();

    VectPtr = VecToVecAddr(Vector);
    FlushCache();

    if (*VectPtr != (unsigned long) UnExpIntr) {
       Wrapper = (struct IntWrapper *) *VectPtr;
       Wrapper->CallAddr = (unsigned long) Handler;
       return;
    }

    MemPtr = (unsigned long *) Malloc(sizeof(struct IntWrapper));
```

```
    CodePtr = (unsigned long *) &IntCode;
    Wrapper = (struct IntWrapper *) MemPtr;

    for (i = 0; i < (sizeof(struct IntWrapper) / sizeof(unsigned long)); i++) {
      *MemPtr++ = *CodePtr++;
    }
    Wrapper->CallAddr = (unsigned long) Handler;

    *VectPtr = (unsigned long) Wrapper;
    if (Vector == NMI_VECTOR) {
      *((unsigned long *) 0) = (unsigned long) Wrapper;
    }
    FlushCache();
}

/**************************************************************************
 * DisConnectHandler(): Modifies vector table back to unexpected
 *                      interrupt handler.
 ***/

DisConnectHandler(Vector)
unsigned long Vector;
{
    unsigned long OldWrapper, *VecToVecAddr();
    int UnExpIntr();

    OldWrapper = *VecToVecAddr(Vector);
    Free(OldWrapper);
    *VecToVecAddr(Vector) = (unsigned long) UnExpIntr;
}

/**************************************************************************
 * FaultErr(): This function is called when a processor fault is called
 *             if the fault was unexpected then an error message is
 *             printed indicating the cause of the fault.
 ***/

static char FltStr[] = "\n\n^GUnexpected fault at 0x%x, Type '%s' %s\n";
static char Unknown[]  = "Unknown";
static char Reserved[] = "Reserved";

char *FaultTypes[] = {
"Parallel",          /* 0x00 */
"Trace",             /* 0x01 */
"Operation",         /* 0x02 */
"Arithmetic",        /* 0x03 */
 Reserved,           /* 0x04 */
"Constraint",        /* 0x05 */
 Reserved,           /* 0x06 */
"Protection",        /* 0x07 */
 Reserved,           /* 0x08 */
 Reserved,           /* 0x09 */
"Type Mismatch",     /* 0x0A */
 Unknown,            /* 0x0B */
 Unknown,            /* 0x0C */
 Unknown,            /* 0x0D */
 Unknown,            /* 0x0E */
 Unknown,            /* 0x0F */
};

char *TraceFaultTypes[] = {
 Unknown,            /* 0x01 */
"Instruction",       /* 0x02 */
"Branch",            /* 0x04 */
```



```
"Call",              /* 0x08 */
"Return",            /* 0x10 */
"PreReturn",         /* 0x20 */
"Supervisory",       /* 0x40 */
"BreakPt"            /* 0x80 */
};

char *OperFaultTypes[] = {
"Inv. OpCode",       /* 0x01 */
"Unimplemented",     /* 0x02 */
"Unaligned",         /* 0x03 */
"Inv. Operand"       /* 0x04 */
};


FaultErr(Addr, Type)
long Type;
char *Addr;
{
    unsigned long SubTypeBit, TypeBits;
    char *TypeStr, *SubTypeStr;

    TypeBits = ((Type >> 16) & 0x0F);
    TypeStr = FaultTypes[TypeBits];
    if (TypeBits == 1) {
        SubTypeBit = FindBitSet(Type & 0xFF);
        SubTypeStr = TraceFaultTypes[SubTypeBit];
    } else if (TypeBits == 2) {
        SubTypeBit = ((Type - 1) & 0x03);
        SubTypeStr = OperFaultTypes[SubTypeBit];
    } else {
        SubTypeStr = "";
    }
    xprintf(FltStr, Addr, TypeStr, SubTypeStr);
    DumpRegs();
    LineEdit();
}

/**************************************************************************
 * FaultInit():  The fault table consists of 32 - 8 bytes fault entries.
 *               All faults are initialized to the unexpected fault
 *               handler.
 ***/

struct FltWrapper FltCode = {
  0xb2805000, 0xb2a06010, 0xb2c06020, 0xa2e06030, 0x8c200040, 0x59084004,
  0x9027f400, 0xfffffff8, 0x8c283000, 0x00ff00ff, 0x58894084, 0x9087f400,
  0xfffffffc, 0x86003000,
  0xeeeeeeee,
  0x8c200040, 0x59084104, 0xb0805000, 0xb0a06010,
  0xb0c06020, 0xa0e06030, 0x0a000000,
  0xeeeeeeee, 0xeeeeeeee, 0xeeeeeeee
};


FaultInit()
{
  int i, UnExpFault();
  unsigned long *FaultPtr;

  TraceFlag = 0;
  FaultPtr = flt_table;
  for(i = 0; i < 32; i ++) {
    *FaultPtr++ = (unsigned long) UnExpFault;
    *FaultPtr++ = (unsigned long) 0;
```

```
    }
}


/*************************************************************************
 *  Trace events that happen on the next V960 bug 'call' instruction.
 *  The 'call' routine must be modified to set the pc and tc.
 ***/

#define     PCTRACE_ENABLE          0x01
#define     MAX_BREAK_POINTS        20
#define     FMARK_OPCODE            0x66003e00

#define     STEP        0x02        /* Single step trace */
#define     BRANCH      0x04        /* branch trace */
#define     CALL        0x08        /* call trace */
#define     RETURN      0x10        /* return trace */
#define     PRERETURN   0x20        /* Pre-return trace */
#define     SUPERVISOR  0x40        /* supervisor trace */
#define     BREAKPOINT  0x80        /* supervisor trace */
#define     ALL         0xFE        /* All traces */

struct TraceTable {
    char *Name;
    unsigned long Mask;
};

static struct TraceTable TTable[] = {
    "Step",         STEP,
    "Branch",       BRANCH,
    "Call",         CALL,
    "Return",       RETURN,
    "PreReturn",    PRERETURN,
    "Supervisor",   SUPERVISOR,
    "BreakPoint",   BREAKPOINT
};

static int BreakPointFlag;
static unsigned long *BreakPointAddr;
static unsigned long OldTraceMask;

extern unsigned long LocalTraceRegFile[], GlobalTraceRegFile[],
                     CntrlTraceRegFile[];
extern unsigned long LocalRegFile[], GlobalRegFile[], CntrlRegFile[];


Trace(Flag, ModeStr)
char Flag, *ModeStr;
{
    unsigned long Mode;

    if (ModeStr != NULL) {
        if ((Mode = ModeToMask(ModeStr)) == 0) {
            xprintf("\nIllegal Mode request: %s", ModeStr);
            return;
        }
    }
    if (Flag == 'a') {
        TraceMask = TraceMask | Mode;
    }
    if (Flag == 'r') {
        TraceMask = TraceMask & ~Mode;
    }
    DispTrace();
    if (TraceMask)
        TraceFlag = TRUE;
```



```
    else
        TraceFlag = FALSE;
}


/*************************************************************************
 * ModeToMask(): This function converts the 'Mode' indicating a tracing
 *               mode into a bit mask corresponding to the trace
 *               mask register. This is useful for turning on and off
 *               the trace mechanisms.
 ***/

static ModeToMask(Mode)
char *Mode;
{
    unsigned long i;

    for( i = 0; i < (sizeof(TTable) / sizeof(struct TraceTable)) ; i++) {
        if (CmpStr(Mode,TTable[i].Name))
            return(TTable[i].Mask);
    }
    return(0x00);
}

/*************************************************************************
 * DispTrace(): This function displays which trace mechanisms have been
 *              enabled.
 ***/

static DispTrace()
{
    unsigned long i;

    PrNewLine();
    for( i = 0; i < (sizeof(TTable) / sizeof(struct TraceTable)) ; i++) {
        if (TraceMask & TTable[i].Mask)
            xprintf("%s trace on\n", TTable[i].Name);
    }
}

struct BPts {
    unsigned long Address;
    unsigned long OpCode;
};

static struct BPts BreakPoints[MAX_BREAK_POINTS];
static int NumBreakPoints;


/*************************************************************************
 * BPInit(): This function initializes the breakpoint data structures
 *           as containing no breakpoints.
 ***/

BPInit()
{
    int i;

    for(i = 0; i < MAX_BREAK_POINTS ; i++) {
        BreakPoints[i].Address = NULL;
        BreakPoints[i].OpCode  = NULL;
    }
    TraceEnabled = TraceFlag = TraceMask = NumBreakPoints = 0;
    BreakPointFlag = FALSE;
}
```

```
/*************************************************************************
 * BPoint(): This monitor function provide the abilty to add, remove and
 *           display breakpoints. The 'Flag' indicates the operation of
 *           add (-a), remove (-r) and display (-d). The 'Address' is used
 *           only for the add and remove functions that add or remove
 *           a breakpoint.
 ***/

BPoint(Flag, Address)
unsigned char Flag;
unsigned long Address;
{
    int i;
    unsigned long OpCode;
    unsigned long *Memory;

    Memory = (unsigned long *) (Address & 0xFFFFFFFC);

    if (Flag == 'a') {
        if (!OneWordInstr(*(Memory - 1))) {
            xprintf("\nIllegal breakpoint address");
            return;
        }
        for (i = 0; i < MAX_BREAK_POINTS ; i++) {
            if (BreakPoints[i].Address == NULL) {
                BreakPoints[i].Address = (unsigned long) Memory;
                BreakPoints[i].OpCode  = *Memory;
                *Memory = (unsigned long) FMARK_OPCODE;
                NumBreakPoints++;
                break;
            }
        }
        if (i == MAX_BREAK_POINTS) {
            xprintf("\nMaxBreak points exceeded\n");
        }
    } else if (Flag == 'r') {
        for (i = 0; i < MAX_BREAK_POINTS ; i++) {
            if (BreakPoints[i].Address == (unsigned long) Memory) {
                *Memory = BreakPoints[i].OpCode;
                BreakPoints[i].Address = 0;
                NumBreakPoints--;
                break;
            }
        }
        if (i == MAX_BREAK_POINTS) {
            xprintf("\nBreakPoint at 0x%x not set\n", Address);
        }
    } else if (Flag == 'd') {
        for (i = 0; i < MAX_BREAK_POINTS ; i++) {
            if (BreakPoints[i].Address != NULL) {
                xprintf("\nBreakPoint at 0x%x",BreakPoints[i].Address);
            }
        }
        PrNewLine();
    }
    if (NumBreakPoints) {
        TraceMask |= BREAKPOINT;
        TraceFlag = TRUE;
    } else {
        TraceMask &= ~BREAKPOINT;
        TraceFlag = FALSE;
    }
}

/*************************************************************************
```



```
 * SaveState(): This function copies the state of the register files
 *              of the faulting program to the trace save area. This
 *              is necessary because unexpected faults at the monitor
 *              would destroy the original registers.
 ***/

static SaveState()
{
    int i;

    for(i = 0; i < 16; i++) {
        LocalTraceRegFile[i] = LocalRegFile[i];
        GlobalTraceRegFile[i] = GlobalRegFile[i];
    }
    for(i = 0; i < 8; i++) {
        CntrlTraceRegFile[i] = CntrlRegFile[i];
    }
}

/*************************************************************************
 * RcvTrace(): This function is called when a fault has occurred that
 *             was set up to occur. The state of the program is saved
 *             useful information is printed and the monitor is called
 *             again.
 ***/

RcvTrace(Addr, Record)
unsigned long Record;
unsigned long *Addr;
{
    unsigned long Type, SubType, i;

    Type    = ((Record >> 16) & 0xFF);
    SubType = (Record & 0xFE);

    if (BreakPointFlag) {
        SaveState();
        BreakPointFlag = FALSE;
        *(unsigned long *) BreakPointAddr = FMARK_OPCODE;
        TraceMask = OldTraceMask;
        ResumeTrace();
    }
    if ((Type == 1) && (SubType & TraceMask)) {
        SaveState();
        for( i = 0; i < (sizeof(TTable) / sizeof(struct TraceTable)) ; i++) {
            if (SubType & TTable[i].Mask)
                break;
        }
        if (SubType & BREAKPOINT) {
            BreakPointFlag = TRUE;
            BreakPointAddr = Addr;
        } else {
            BreakPointFlag = FALSE;
        }
        DumpRegs();
        xprintf("\nRecieved fault type '%s' at 0x%x",TTable[i].Name, Addr);
        DisAssemble(Addr, 2);
    } else {
        FaultErr(Addr, Record);
    }
}

/*************************************************************************
 * Step(): This monitor function provides the ability to step through
```

```
 *        programs being debugged.
 ***/

Step()
{
    int i;

    if(!TraceEnabled) {
        xprintf("\nTrace not initiated by call");
        return;
    }

    if (BreakPointFlag) {
        for (i = 0; i < MAX_BREAK_POINTS ; i++) {
            if (BreakPoints[i].Address == (unsigned long) BreakPointAddr) {
                *(unsigned long *) BreakPointAddr = BreakPoints[i].OpCode;
                break;
            }
        }
        LocalTraceRegFile[2] -= 4;
        OldTraceMask = TraceMask;
        TraceMask |= STEP;
        ResumeTrace();
    } else {
        ResumeTrace();
    }
}

/***************************************************************************
 * OneWordInstr(): This function examines the 'OpCode' of an instruction
 *                 and determines if the instruction is a one word
 *                 instruction. This is necessary to determine if a
 *                 breakpoint can be asserted at a specific address.
 ***/

static OneWordInstr(OpCode)
unsigned long OpCode;
{
    unsigned long Mode;

    if (((OpCode >> 24) & 0xFF) < 0x80)
        return TRUE;
    if ((OpCode & 0x00001000) == 0)
        return TRUE;
    Mode = ((OpCode >> 10) & 0x0F);
    if ((Mode == 0x4) || (Mode == 0x7))
        return TRUE;
    return FALSE;
}

/***************************************************************************
 * ExecTrace(): This monitor function initiates the trace mechanism
 *              for the function 'Funct' and calls the function
 *              with arguments 'Arg0' to 'Arg7'.
 ***/

ExecTrace(Funct, Arg0, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6, Arg7)
int (*Funct)();
unsigned long Arg0, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6, Arg7;
{
    StartTrace(Arg0, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6, Arg7, Funct);
}
```

```
#define   NMI_VECTOR          0xF8    /* Vector Definitions for the V960 */

/***************************************************************************
 * The Interrupt Wrapper is a relocatable assembly language module which   *
 * is allocated on the stack. The Interrupt table vector location is       *
 * initialized to point to the wrapper and the wrapper is initialized to   *
 * point to the interrupt handler. This level of indirection will reduce   *
 * the dependency of the software on the type of processor and remove      *
 * the necessity for assembly code.                                        *
 *                                                                         *
 * The assembly language module is included below:                         *
 *                                                                         *
 *                                                                         *
 *      stq     g0, (sp)                # Save registers g0-g14 and
 *      stq     g4, 16(sp)              # bump stack pointer.
 *      stq     g8, 32(sp)
 *      stt     g12,48(sp)
 *      lda     0x40, r4
 *      addo    r4, sp, sp
 *
 *      ld      -8(fp),r4               # get vector
 *      ldconst 0xFF,r5                 # mask
 *      and     r4,r5,g1                # Vector Level
 *      mov     rip,g0                  # Address of exception
 *      callx   _IntHdl                 # IntHdl(Addr,Vector)
 *
 *      lda     0x40, r4
 *      subo    r4, sp, sp              # Restore processor state.
 *      ldq     (sp), g0                # Registers g0-g14, sp
 *      ldq     16(sp), g4
 *      ldq     32(sp), g8
 *      ldt     48(sp), g12
 *      ret                             # Return to program.
 *
 *     .space   4                       # For debug or storage
 *     .space   4                       #
 *     .space   4                       #
 *                                                                         *
 *************** disassembly for Interrupt Wrapper **********************
 *
  0:  b2805000               stq     g0,  (sp)
  4:  b2a06010               stq     g4,  0x10 (sp)
  8:  b2c06020               stq     g8,  0x20 (sp)
  c:  a2e06030               stt     g12,  0x30 (sp)
 10:  8c200040               lda     0x40, r4
 14:  59084004               addo    r4, sp, sp
 18:  9027f400 fffffff8      ld      0xfffffff8 (fp), r4
 20:  8c2800ff               lda     0xff, r5
 24:  58894084               and     r4, r5, g1
 28:  5c801602               mov     rip, g0
 2c:  86003000 xxxxxxxx      callx   0x0
 34:  8c200040               lda     0x40, r4
 38:  59084104               subo    r4, sp, sp
 3c:  b0805000               ldq     (sp), g0
 40:  b0a06010               ldq     0x10 (sp), g4
 44:  b0c06020               ldq     0x20 (sp), g8
 48:  a0e06030               ldt     0x30 (sp), g12
 4c:  0a000000               ret
 50:  xxxxxxxx              .word    0x0
 54:  xxxxxxxx              .word    0x0
 58:  xxxxxxxx              .word    0x0
 *
 ***************************************************************************/

struct IntWrapper {
  unsigned long CodeSeg0[12];
```



```
  unsigned long CallAddr;
  unsigned long CodeSeg1[7];
  unsigned long DatSeg0[3];
};

/**************** disassembly for fault Wrapper **************************
 *
 *      stq     g0, (sp)                # Save registers g0-g14 and
 *      stq     g4, 16(sp)              # bump stack pointer.
 *      stq     g8, 32(sp)
 *      stt     g12,48(sp)
 *      lda     0x40, r4
 *      addo    r4, sp, sp
 *
 *      ld     0xfffffff8 (fp), r4      # Read fault type off stack.
 *      lda     0xff00ff, r5            # Mask off good bits.
 *      and     r4, r5, g1
 *      ld     0xfffffffc (fp), g0      # Read fault address of stack.
 *      callx      0x00                 # Call fault handler.
 *
 *      lda     0x40, r4
 *      subo    r4, sp, sp              # Restore processor state.
 *      ldq     (sp), g0                # Registers g0-g14, sp
 *      ldq     16(sp), g4
 *      ldq     32(sp), g8
 *      ldt     48(sp), g12
 *      ret                             # Return to program.
 *
 *     .space    4                      # For debug or storage
 *     .space    4                      #
 *     .space    4                      #
 *
 *
 ***********************************************************************
 *
 *      0:  b2805000               stq     g0,  (sp)
 *      4:  b2a06010               stq     g4,  0x10 (sp)
 *      8:  b2c06020               stq     g8,  0x20 (sp)
 *      c:  a2e06030               stt     g12,  0x30 (sp)
 *     10:  8c200040               lda     0x40, r4
 *     14:  59084004               addo     r4, sp, sp
 *     18:  9027f400 fffffff8      ld     0xfffffff8 (fp), r4
 *     20:  8c283000 00ff00ff      lda     0xff00ff, r5
 *     28:  58894084               and     r4, r5, g1
 *     2c:  9087f400 fffffffc      ld     0xfffffffc (fp), g0
 *     34:  86003000 00000000      callx     0x0
 *     3c:  8c200040               lda     0x40, r4
 *     40:  59084104               subo     r4, sp, sp
 *     44:  b0805000               ldq     (sp), g0
 *     48:  b0a06010               ldq     0x10 (sp), g4
 *     4c:  b0c06020               ldq     0x20 (sp), g8
 *     50:  a0e06030               ldt     0x30 (sp), g12
 *     54:  0a000000               ret
 *     58:  00000000              .word    0x0
 *     5c:  00000000              .word    0x0
 *     60:  00000000              .word    0x0
 ***********************************************************************/

struct FltWrapper {
  unsigned long CodeSeg0[14];
  unsigned long CallAddr;
  unsigned long CodeSeg1[7];
  unsigned long DatSeg0[3];
};
```

```
#*********************************************************************
#
# Copyright (c) 1990 Heurikon Corporation
# All Rights Reserved
#
# THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
#         The copyright notice above does not evidence any
#         actual or intended publication of such source code.
#
# Heurikon hereby grants you permission to copy and modify
# this software and its documentation.  Heurikon grants
# this permission provided that the above copyright notice
# appears in all copies and that both the copyright notice and
# this permission notice appear in supporting documentation.  In
# addition, Heurikon grants this permission provided that you
# prominently mark as not part of the original any modifications
# made to this software or documentation, and that the name of
# Heurikon Corporation not be used in advertising or publicity
# pertaining to distribution of the software or the documentation
# without specific, written prior permission.
#
# Heurikon Corporation does not warrant, guarantee or make any
# representations regarding the use of, or the results of the use
# of, the software and documentation in terms of correctness,
# accuracy, reliability, currentness, or otherwise; and you rely
# on the software, documentation and results solely at your own
# risk.
#
#
#
#
# MODIFICATIONS:
#
#****

#*********************************************************************
# ProcAsm.s: This file contains the assembly language functions used by
#            the board, monitor, and processor functions to perform
#            processor-specific functions. Below is a list of functions
#            defined in this module that can be used by other functions.
#***

               .text
               .align 4

               .globl _ReadIntMask      # Exported Functions.
               .globl _ReadIntPend
               .globl _ClrIntPend
               .globl _MaskInts
               .globl _UnMaskInts
               .globl _UnExpFault
               .globl _UnExpIntr
               .globl _FlushCache
               .globl _ResumeTrace
               .globl _StartTrace
               .globl _ReadTCW
               .globl _ModifyTCW

#*********************************************************************
# Below is a list of data structures that can be referenced by other
# functions.
#***

               .globl _LocalTraceRegFile    # Exported Data.
               .globl _GlobalTraceRegFile
```



```
               .globl _CntrlTraceRegFile


#*********************************************************************
# Below is a list of functions referenced from this module that must
# be defined in another module.
#***

               .globl _LocalRegFile     # Imported Functions.
               .globl _GlobalRegFile
               .globl _CntrlRegFile
               .globl _LocalTraceRegFile
               .globl _GlobalTraceRegFile
               .globl _CntrlTraceRegFile
               .globl _FaultErr
               .globl _ReadPCW
               .globl _ModifyPCW
               .globl _RcvTrace
               .globl _TraceEnabled
               .globl _TraceMask
               .globl _BPInit

#*********************************************************************
# Constants associated with the 'sysctl' instruction.
#***

               .set    REQUEST_INTR,     0x000
               .set    INVALID_CACHE,    0x100
               .set    CONFIG_CACHE,     0x200
               .set    RE_INITIALIZE,    0x300
               .set    LD_CTRL_REG0,     0x400
               .set    LD_CTRL_REG1,     0x401
               .set    LD_CTRL_REG2,     0x402
               .set    LD_CTRL_REG3,     0x403
               .set    LD_CTRL_REG4,     0x404
               .set    LD_CTRL_REG5,     0x405
               .set    LD_CTRL_REG6,     0x406


#*********************************************************************
# Below are the basic procesor functions and an example of the calling
# sequence from a C program.
#***

_ReadIntPend:  mov     sf0,g0             # Data = ReadIntPend();
               ret                        # Returns interrupt pending reg.

_ClrIntPend:   mov     0x0,sf0            # ClearIntPend();
               ret                        # Clears interrupt pending reg.

_MaskInts:     notand  sf1,g0,sf1         # MaskInts(IntMask);
               ret                        # Turn off bits in mask.

_ReadIntMask:  mov     sf1,g0             # Mask = ReadIntMask();
               ret                        # returns interrupt mask register.

_UnMaskInts:   mov     0,sf0              # UnMaskInts(IntMask);
               or      g0,sf1,sf1         # Clear interrupt pending register
               ret                        # and turn on bits in mask.

_FlushCache:   ldconst INVALID_CACHE,r3   # FlushCache();
               sysctl  r3, r3, r3         # Invalidate cache opcode
               ret
```

```
_ModifyPCW:     modpc   g0,g0,g1          # ModifyPCW(PCWMask);
                ret                       #

_ReadPCW:       modpc   0,0,g0            # ReadPCW();
                ret                       #

_ModifyTCW:     modtc   g0,g1,g2          # ModifyTCW(TCWMask);
                ret                       #

_ReadTCW:       modtc   0,0,g0            # ReadTCW();
                ret                       #

#**********************************************************************
# ERROR RECOVERY: The following routines are intended to provide error
#                 recovery from unexpected faults and interrupts. The
#                 functions UnExpIntr and UnExpFault should be written
#                 to all unused vector locations in both the interrupt
#                 and the fault table.
#***

#**********************************************************************
# UnExpFault: This is the fault recovery mechanism, which notifies the
#             user of the fault and then restarts the system. If the
#             trace flag indicates tracing is enabled and a masked fault
#             has occurred then the monitor is returned to gracefully after
#             the program state has been saved.
#***

_UnExpFault:    flushreg                  # Flush all registers.

                lda     _GlobalRegFile,r4 # Save Global Registers.
                stq     g0,   (r4)
                stq     g4,   16(r4)
                stq     g8,   32(r4)
                stq     g12,  48(r4)

                ldconst  0, g14           # Required to reset context.

                mov      pfp,r5           # Get Previous Frame ptr.
                ldconst 0xFFFFFFF0,r6
                and     r6,r5,r5
                st      r5, 60(r4)        # Save as FP of faulting proc.

                lda     _LocalRegFile,r4  # Save Local Registers.

                ldq     (r5), g0
                stq     g0,   (r4)
                ldq     16(r5), g0
                stq     g0,   16(r4)
                ldq     32(r5), g0
                stq     g0,   32(r4)
                ldq     48(r5), g0
                stq     g0,   48(r4)

                lda     _CntrlRegFile,r4  # Save Local Registers.
                modpc   0,0,r5
                st      r5,   (r4)
                modac   0,0,r5
                st      r5,   4(r4)
                mov     sf0,r5
                st      r5,   8(r4)
                mov     sf1,r5
                st      r5,   12(r4)
                modtc   0,0,r5
                st      r5, 16(r4)
```



```
                ld      -8(fp),r4         # Get fault type.
                ldconst 0xFF00FF,r5       # Mask.
                and     r4,r5,g1          # Fault Type and Subtype.
                ld    -4(fp),g0           # Fault Address.
                lda     _TraceFlag, r4
                ld      (r4), r5
                cmpobe  0, r5, NotTrace

                callx   _RcvTrace         # Print error message.
                callx   _LineEdit

NotTrace:       callx   _FaultErr         # Print error message.
                callx   _start_ip         # Start over.

#**********************************************************************
# UnExpIntr:  This is the interrupt recovery mechanism, which notifies the
#             user of the interrupt, removes the interrupt, and then restarts
#             the system.
#***

_UnExpIntr:     flushreg                  # Flush all registers.

                lda     _GlobalRegFile,r4 # Save Global Registers.
                stq     g0,   (r4)
                stq     g4,   16(r4)
                stq     g8,   32(r4)
                stq     g12,  48(r4)

                lda     _LocalRegFile,r4  # Save Local Registers.
                mov      pfp,r5
                ldconst 0xFFFFFFF0,r6
                and     r6,r5,r5
                ldq     (r5), g0
                stq     g0,   (r4)
                ldq     16(r5), g0
                stq     g0,   16(r4)
                ldq     32(r5), g0
                stq     g0,   32(r4)
                ldq     48(r5), g0
                stq     g0,   48(r4)

                lda     _CntrlRegFile,r4  # Save Local Registers.
                modpc   0,0,r5
                st      r5,   (r4)
                modac   0,0,r5
                st      r5,   4(r4)
                mov     sf0,r5
                st      r5,   8(r4)
                mov     sf1,r5
                st      r5,   12(r4)
                modtc   0,0,r5
                st      r5, 16(r4)

                ld      -8(fp),r4         # Get vector.
                ldconst 0xFF,r5           # Mask.
                and     r4,r5,g1          # Vector Level.
                mov     rip,g0            # Interrupt address.

                callx   _IntrErr          # Print error message.
                callx   _start_ip         # Start over.

#**********************************************************************
```

```
 # ResumeTrace():  This function is called when tracing is enabled and
 #                 an expected trace is recognized.
 #***

_ResumeTrace:   flushreg

                lda     _GlobalTraceRegFile,r5      # Restore Global Registers.
                lda     _LocalTraceRegFile,r3       # Restore Global Registers.
                ld      _60(r5), pfp
                andnot  0x0000000f, pfp, r14

                ldq       (r3), r4
                stq     r4, (r14)
                ldq     16(r3), r4
                stq     r4, 16(r14)
                ldq     32(r3), r4
                stq     r4, 32(r14)
                ldq     48(r3), r4
                stq     r4, 48(r14)

                lda     _GlobalTraceRegFile,r3      # Restore Global Registers.
                ldq     _(r3),  g0
                ldq     16(r3),  g4
                ldq     32(r3),  g8
                ldt     48(r3),  g12

                lda      _LocalTraceRegFile,r3      # Restore RIP.
                ld      8(r3), r4
                ldconst 0xfffffffc, r6
                and     r6, r4, r4
                and     0x03, rip, r5
                or      r4, r5, rip

                lda      _CntrlTraceRegFile,r3       # Restore PCW.
                ld      (r3), r4
                ldconst 0xffe0eefc, r5
                modpc   0x00,0x00, r6
                and     r5, r6, r7
                not     r5, r5
                and     r4, r5, r4
                or      r4, r7, r8
                or 0x01, r8, r8
                st      r8, -0x10(fp)

                ld      4(r3), r4              # Restore ACW.
                ldconst 0xffff6ef8, r5
                modac   0x00, 0x00, r6
                and     r5, r6, r7
                not     r5, r5
                and     r4, r5, r4
                or      r4, r7, r8
                st      r8, -0x0c(fp)

                flushreg                       # Return into Context.

                lda    _TraceMask, r3          # Update trace settings.
                 ld    _(r3), r4
                 ldconst    0xFE, r5
                 modtc      r5,  r4, r4

                mov     0x02, r5
                modpc   0x00, r5, r5           # Set to Supervisory State.
                or      0x01, pfp, pfp         # Set Return from fault.
                ret
```



```
#*************************************************************************
# StartTrace():  This function is called when tracing is enabled and
#                starts an instruction trace of the function defined
#                by register g8 using parameters g0-g7. An example call
#                is:
#
#   StartTrace(Arg0, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6, Arg7, Function);
#
#***

_StartTrace:     lda    _TraceEnabled, r3         # Flag tracing started.
                 ldconst_  1, r4
                 st    r4, (r3)

                 lda    _TraceMask, r3            # Set trace bits.
                 ld    (r3), r4
                 ldconst    0xFE, r5
                 modtc      r5,  r4, r4

                 ldconst    0x01,  r5             # Enable tracing.
                 modpc      r4, r5, r5

                 callx      (g8)                  # Call function.

                 ldconst    0, r3
                 ldconst    0xFE, r5
                 modtc      r5,  r3, r4          # Disable trace bits.

                 ldconst    0x01,  r5             # Disable tracing.
                 modpc      r4, r5, r3

                 callx      _BPInit
                 ret

#*************************************************************************
# Below are data definitions that are used in storing trace data.
#***
                .align 4
                .data

                .bss   _LocalTraceRegFile,  0x0040, 8
                .bss   _GlobalTraceRegFile, 0x0040, 8
                .bss   _CntrlTraceRegFile,  0x0020, 8
```

```
/************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"

unsigned char Key[8];                   /* bss and data versions of RTC Key  */
static unsigned char InitKey[] = {
    0xC5, 0x3A, 0xA3, 0x5C, 0xC5, 0x3A, 0xA3, 0x5C
};

/************************************************************************
 * rtc_acc: This function reads or writes the real-time clock, depending
 *          on 'Type'. The 'data' is received and returned in the format
 *          of the real-time clock (Board.h). This function cannot be
 *          loaded into ROM; because of the way the RTC operates, the
 *          clock would be reset by ROM execution.
 ***/

static rtc_acc(data, Type)
unsigned char *data;
int Type;
{
    int i, bit;
    unsigned char temp;

    i = *RD_WATCH;
    for(i = 0; i < 8; i++){
        for(bit = 1; bit & 0xFF; bit <<= 1){
            temp = (Key[i] & bit) ? *WR1_WATCH : *WR0_WATCH;
        }
    }

    if (Type) {
        for(i = 0; i < 8; i++){
            for(bit = 1; bit & 0xFF; bit <<= 1){
```



```
                temp = (data[i] & bit) ? *WR1_WATCH : *WR0_WATCH;
            }
        }
    } else {
        for(i = 0; i < 8; i++){
            data[i] = 0;
            for(bit = 1; bit & 0xFF; bit <<= 1){
                data[i] |= (*RD_WATCH & 1)? bit : 0;
            }
        }
    }
}

/************************************************************************
 * RtcAcc: This function accepts the structure 'Time' and either reads
 *         the time into or writes the new time from this structure.
 *         'Flag' indicates whether the function is reading or writing
 *         the time. There are several very strange things that should be
 *         described about this function:
 *
 *         Because the RTC stores the time as packed nibbles internally
 *         it is necessary to convert to packed nibbles when writing
 *         and to binary when reading the RTC.
 *
 *         Because the ROM cannot be accessed when the RTC is being read
 *         it is necessary to copy the function rtc_acc into RAM and then
 *         execute the function. This is also why the 'Key' is located in
 *         the 'bss' section. Great care was taken to assure that the
 *         function rtc_acc was relocatable so be careful !!!.
 ***/

RtcAcc(Time, Flag)
tm *Time;
int Flag;
{
    int (*Funct)();
    int Size, nibble(), rtc_acc();
    char *Malloc();
    unsigned long tmp;
        struct rtc_data RtcData;

    CopyMem(InitKey, Key, sizeof(InitKey));

    if (Flag == WRITE) {
        RtcData.hour    = BinToHex(Time->tm_hour);      /* Write */
        RtcData.min     = BinToHex(Time->tm_min);
        RtcData.month   = BinToHex(Time->tm_mon);
        RtcData.weekday = Time->tm_wday | 0x10;
            if (Time->tm_wday == 0)                         /* Converts sunday to 7
 */
            RtcData.weekday = 0x17;
        RtcData.date    = BinToHex(Time->tm_mday);
        RtcData.year    = BinToHex(Time->tm_year);
        RtcData.sec     = 0;
        RtcData.dotsec  = 0;
        }

#ifdef RAM_MON                                  /* If RAM based monitor         */
        rtc_acc(&RtcData, Flag);
#else                                           /* If EPROM based monitor       */
    Size = (int) RtcAcc - (int) rtc_acc;        /* Size of function to copy     */
    Funct = (int (*)()) Malloc(Size);           /* Allocate memory for function.*/
    FlushCache();
    CopyMem(rtc_acc, Funct, Size);              /* Copy function to memory.     */
    Funct(&RtcData, Flag);                      /* Call function.               */
    Free(Funct);
```

```
#endif

    if (Flag == READ) {
        Time->tm_fsec  = HexToBin(RtcData.dotsec);      /* Read  */
        Time->tm_sec   = HexToBin(RtcData.sec);
        Time->tm_min   = HexToBin(RtcData.min);
        Time->tm_hour  = HexToBin(RtcData.hour);
        Time->tm_mday  = HexToBin(RtcData.date);
        Time->tm_mon   = HexToBin(RtcData.month);
        Time->tm_year  = HexToBin(RtcData.year);
        Time->tm_wday  = (RtcData.weekday & 0x7);
                If (Time->tm_wday == 7)                  /* Converts sunday to 0 */
            Time->tm_wday = 0;
    }
};
```

```
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/
```


```
#include "Bug.h"
#include "Board.h"
#include "NvMonDefs.h"

/***************************************************************************
 * SCC.c:  This file contains the functions necessary to read, write and
 *         configure the Z85C30-16 Serial Controller.
 *         The functions defined in this module are listed below:
 *
 *              GetChar()       PutChar()       KeyHit()
 *              TxEmpty()       ChangeBaud()    SetSerDevs()
 *              SCCReset()      FoundBreak()    ConfigPort()
 ***/

extern NV_MonDefs    NvMonDefs;       /* Monitor defined configuration    */

volatile unsigned long ConDev;        /* Console Device                   */
volatile unsigned long ModDev;        /* Modem/Download Device            */

static unsigned long SerDevList[] = {    /* List of port assignments      */
  (unsigned long) SCC_PORTA,             /* Corresponds to NV definitions.*/
  (unsigned long) SCC_PORTB,
  (unsigned long) SCC_PORTC,
  (unsigned long) SCC_PORTD,
};

/***************************************************************************
 * GetChar():  Get a character from specified device 'Port'. This function
 *             is also set up to check for a 'break' and allows the monitor
 *             to perform functions on break, like reset or baud changes.
 *******/

GetChar(Port)
```



```
volatile struct SCCPort *Port;
{
    unsigned char Data;

    Port->Control = 0;
    while (1) {
        if (Port->Control & 0x01) {
                        Data = Port->Data;
            if (Port->Control & 0x80) {
                Port->Control = 0x10;        /* Reset Ext/Status Ints     */
                Port->Control = 0x10;        /* Only works if done twice  */
                                FoundBreak(Port);
                        } else {
                return(Data);
                        }
        }
        }
}

/***************************************************************************
 * PutChar():  Put a character 'c' to specified device 'Port'
 *******/

PutChar(Port, c)
volatile struct SCCPort *Port;
char c;
{
    Port->Control = 0;
    while (!(Port->Control &  0x04));
    Port->Data = c;
}

/***************************************************************************
 * KeyHit():  Check for character on specified device 'Port'. This is
 *            useful during powerup and transparent mode.
 *******/

KeyHit(Port)
volatile struct SCCPort *Port;
{
    Port->Control = 0;
    return(Port->Control & 0x01);
}

/***************************************************************************
 * TxEmpty():  Check transmitter if empty on specified device 'Port'. This
 *             function is useful for transparent mode.
 *******/

TxEmpty(Port)
volatile struct SCCPort *Port;
{
    return((Port->Control & 0x04) ? TRUE : FALSE);
}

/***************************************************************************
 * ChangeBaud():  Change baud rate for specified port 'Port' to rate 'Baud'.
 *******/

ChangeBaud(Baud, Port)
volatile struct SCCPort *Port;
int Baud;
{
    int tc;
    unsigned short dummy;
```

```
        for (tc = 0; tc < 0x1000; tc++);
    tc = BaudToTimeConst(Baud);

    dummy = Port->Control;
    Port->Control = 0x0C;
    Port->Control = tc;
    Port->Control = 0x0D;
    Port->Control = tc >> 8;
        for (tc = 0; tc < 0x1000; tc++);
}

/*****************************************************************************
 * SCCReset():  This function hard resets both ports associated with 'Port'
 *              because it's too clumsy to reset individual ports.
 *******/

static SCCReset(Port)
volatile struct SCCPort *Port;
{
    Port->Control = 0;
    Port->Control = 0x09;
    Port->Control = 0xC0;
}

/*************************************************************************
 * SetSerDevs():  This function uses the current definitions in the
 *                NV structure 'NvMonDefs' to configure the serial ports.
 *                This function is called once when NvMonDefs contains
 *                the default system configuration and once after the
 *                NV memory has been read with the user's configuration.
 *
 * NOTICE:        It is important that the NvMonDefs be valid when this
 *                function is called!
 **/

SetSerDevs()
{
    SCCReset(SCC_PORTB);       /* Reset all serial devices.              */
    SCCReset(SCC_PORTD);

    ConDev = SerDevList[NvMonDefs.Console.PortNum];  /* Set up Console. */
    ConfigPort(ConDev, &NvMonDefs.Console);
    ChangeBaud(NvMonDefs.Console.Baud,  ConDev);

    ModDev = SerDevList[NvMonDefs.DownLoad.PortNum]; /* Set up Download.*/
    ConfigPort(ModDev, &NvMonDefs.DownLoad);
    ChangeBaud(NvMonDefs.DownLoad.Baud, ModDev);
}

/*************************************************************************
 * ConfigPort():    Initialize specified port 'Port' to the configuration
 *                  specified by 'Conf'. The configurable portion of this
 *                  function includes:
 *
 *                    Data Bits ... 5,6,7 or 8.
 *                    Stop Bits ... 1,or 2.
 *                    Parity    ... None, Even or Odd.
 *                    XOnXOff   ... On/Off
 **/

static ConfigPort(Port, Conf)
volatile struct SCCPort *Port;
NVU_Port *Conf;
{
```



```
    static unsigned char SCCTabl[] = {
        0x09, 0x00,    /* No Reset                       */
        0x0A, 0x00,    /* NRZ                            */
        0x0B, 0x56,    /* TxClk = RxClk = Baud Rate Gen  */
        0x0E, 0x02,    /* Baud Rate Generator Source     */
        0x0E, 0x03,    /* Start Baud Rate Generator      */
        0x0F, 0x80,    /* Enable interrupt on break      */
        0x01, 0x00,
    };

    register int Cnt;
        register unsigned char Mask;

    for (Cnt = 0; Cnt < 0x1000; Cnt++);

    Port->Control = 0;
    for(Cnt = 0; Cnt < sizeof(SCCTabl); Cnt++)
        Port->Control = SCCTabl[Cnt];

    Mask = 0x0;
    if (Parity(Conf) == SP_PARITY_EVEN)      /* Determine parity.          */
        Mask = 0x3;
    if (Parity(Conf) == SP_PARITY_ODD)
        Mask = 0x1;

    if (StopBits(Conf))                      /* Determine stop bits.       */
        Mask = Mask | 0x08;

    Port->Control = 0x04;                    /* Write register 4           */
    Port->Control = 0x44 | Mask;             /* 16x clock, parity, stop bits */

        Mask = DataBits(Conf);                   /* Determine data bits.       */
        Mask = ((Mask & 0x1) << 1)
                 + ((Mask & 0x2) >> 1);
    Port->Control = 0x05;
    Port->Control = (0x8A | (Mask << 5));    /* Set Tx bit size, enable Tx.  */
        Mask = Mask << 6;
    if (XOnXOff(Conf))                       /* Turn on auto enables.      */
              Mask = Mask | 0x20;
    Port->Control = 0x03;
    Port->Control = (0x01 | Mask);           /* Set Rx Bit Size, Enable Rx   */

    Port->Control = 0x38;                    /* Reset highest IUS.          */
    Port->Control = 0x30;                    /* Reset errors.              */
    Port->Control = 0x10;                    /* Reset Ext/Status Ints.     */
    for (Cnt = 0; Cnt < 0x1000; Cnt++);
}

/*****************************************************************************
 * FoundBreak():  This function performs functions defined by the NV memory
 *                configuration when a break is received.  Either the monitor
 *                is reset or the baud rate is changed.
 *******/

static FoundBreak(Port)
volatile struct SCCPort *Port;
{
    NVU_Port *Conf;

        if ((unsigned long) Port == ConDev) {
        Conf = &NvMonDefs.Console;
    } else if ((unsigned long) Port == ModDev) {
        Conf = &NvMonDefs.DownLoad;
    } else {
```

```
                return;
        }

    if (ResetOnBreak(Conf))             /* If reset on break allowed    */
                MonEntryPt();                   /* Reset monitor                */
    if (ChBaudOnBreak(Conf)) {          /* If baud changes on break     */
                Conf->Baud = GetNextBaud(Conf->Baud);
        ChangeBaud(Conf->Baud,  Port);
                xprintf("\nChanged baud rate to %d\n", Conf->Baud);
    }
}
```

```
/************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *        The copyright notice above does not evidence any
 *        actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"
#include "NvMonDefs.h"

extern NV_MonDefs    NvMonDefs;

/************************************************************************
 * SCSI.c:  This file contains the functions necessary to read, write and
 *          configure the WD33C93A SCSI Controller.
 *          The functions defined in this module are listed below:
 *
 * ResetSCSI():  Sets the SCSI to the hardware reset state and removes
 *               the reset interrupt.
 *
 * InitSCSI():   This sets the state of the SCSI according to the NV
 *               definitions.
 ***/

extern NV_MonDefs    NvMonDefs;      /* Monitor-defined configuration    */

#define SC_RESET    0x00    /* Issues an RESET Command to WD33C93 */
#define FREQ_SEL    0x80    /* Select Frequency for Divisor of 4  */

ResetSCSI()
{
    unsigned char Stat;

    MaskInts(SCSI_INT_MASK);             /* Disable Interrupts.            */
    SCWriteReg(SREG_OWNID, FREQ_SEL);    /* Initailize for 16MHZ operation.*/
    SCReadReg(SREG_SCSI_STAT, Stat);     /* Read Status register.          */
    SCWriteReg(SREG_CMD, SC_RESET);      /* Generate SCSI Reset.           */
    SCReadReg(SREG_SCSI_STAT, Stat);     /* Remove SCSI Interrupt.         */
}
```



```
InitSCSIState()
{
    unsigned char Stat;
    NV_MonDefPtr Conf = &NvMonDefs;

    ResetSCSI();
    if (ScsiResetEnbl(Conf)) {          /* Reset SCSI on reset ?  */
        *SCSI_RESET = 1;                /* Toggle the reset line. */
        Delay(100);                     /* Leave on ~ 1 second.   */
        *SCSI_RESET = 0;                /* Remove SCSI reset.     */
        }
}
```

```
/**************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"
#include "NvMonDefs.h"

/**************************************************************************
 * VME.c:  This file contains the functions necessary to initialize the
 *         VMEbus as well as examples of how to perform several basic
 *         VME functions.
 *
 *                ConfigBus()     SlaveEnable()    SlaveDis()
 *                UnMaskVMEInt()
 ***/

extern NV_MonDefs    NvMonDefs;     /* NV Monitor definitions           */

/**************************************************************************
 * ConfigVmeBus():  This function uses the current definitions in the
 *                  NV structure 'NvMonDefs' to configure the VME bus.
 *                  This function is called once when NvMonDefs contains
 *                  the defualt system configuration and once after the
 *                  NV memory has been read with the users configuration.
 *
 *                  Configured in the function are the following:
 *
 *                    Extended Space  ... Address and Enable
 *                    Standard Space  ... Address and Enable
 *                    Short    Space  ... Address and Enable
 *                    Bus Req Level   ... BR3, BR2, BR1, BR0
 *                    Bus Rel Modes   ... WhenDone, OnReq, OnClear, Never
 *                    Local Bus Timer ... 4us to Infinite
 *                    VME Bus Timer   ... 4us to Infinite
 *                    Arbiter Mode    ... RoundRobin, Priority
 *                    Write Post Slv  ... On/Off
```



```
 *                    Write Post Mst  ... On/Off
 *                    Turbo mode      ... On/Off
 *                    Sys Fail State  ... On/Off
 *                    Indiv R-Mod-Wr  ... On/Off
 *
 * NOTICE:          It is important that the NvMonDefs be valid when this
 *                  function is called!
 ***/

ConfigVmeBus()
{
    NVU_BusConfig *Conf = &NvMonDefs.VmeBus;
        unsigned char RegVal, Mask;
        int i;

    VIC->ArbReqConfig.Reg = (BusReqLev(Conf) << 5)
                                              | (ArbiterMode(Conf) ? 0x80 : 0);
    VIC->TranTimeOut.Reg  = (VmeBusTimer(Conf) << 5)
                                              | (LocBusTimer(Conf) << 2);
    VIC->RelCntrl.Reg     = (MastRelMode(Conf) << 6);
    VIC->VMEConfig.Reg    = (TurboMode(Conf) ? 2 : 0)
                                              | (IndivRMC(Conf) ? 0x40 : 0);

    RegVal = VIC->SlvSel[1][0].Reg & 0x3F;
        Mask = (SlaveWrPost(Conf) ? 0x80 : 0) | (MasterWrPost(Conf) ? 0x40 : 0);
    VIC->SlvSel[1][0].Reg = RegVal | Mask;

    if (ExtSlaveEnbl(Conf)) {
        SlaveEnable('e',ExtSlaveMap(Conf));
    } else {
        SlaveDis('e');
        }
    if (StdSlaveEnbl(Conf)) {
        SlaveEnable('s',StdSlaveMap(Conf));
    } else {
        SlaveDis('s');
        }
    if (ShtSlaveEnbl(Conf)) {
        SlaveEnable('c',ShtSlaveMap(Conf));
    } else {
        SlaveDis('c');
        }

    VIC->ICR[7].Reg  = 0x00;            /* Allow masking of SYSFAIL.    */
    if (Sysfail(Conf)) {
        VIC->ICR[6].Reg  = 0x40;        /* Remove SYSFAIL.              */
        } else {
        VIC->ICR[6].Reg  = 0x00;        /* Assert SYSFAIL.              */
        }
        for (i = 0; i < 5 ; i++ )
        VIC->ICR[i].Reg  = Conf->IComReg[i];
}


/**************************************************************************
 * IsSystemController(): This function returns true if the board is the
 *                       VMEbus system controller.
 ***/

IsSystemController()
{
    if (VIC->VMEConfig.Reg & 0x01) {
            return FALSE;
        } else {
            return TRUE;
        }
```

```
}

/***************************************************************************
 * SlaveEnable(): This monitor function allows for enabling and disabling
 *                of the 3 slave VMEBus address spaces. The 'Flag'
 *                indicates either 'e' (extended space), 's' (standard
 *                space) or 'c' (communications or short space). The
 *                'Address' should contain the base address to be
 *                mapped to. The significant portion of the address field
 *                is defined as:
 *
 *                         FFxxxxxx  for the extended space
 *                         xxFxxxxx  for the standard space
 *                         xxxxFFxx  for the short space
 ****/

SlaveEnable(Flag, Address)
char Flag;
unsigned long Address;
{
        unsigned char RegVal;

    switch (Flag) {
        case 'E':
        case 'e': {
            RegVal = VIC->SlvSel[1][0].Reg & 0xC0;
            VIC->SlvSel[1][0].Reg = RegVal | 0x10;
            VIC->SlvSel[1][1].Reg = 0x00;
            WriteCIOPortA((unsigned char) (Address >> 24));
            *SLAVE_EXT_ENBL = 1;
            break;
        }
        case 'S':
        case 's': {
            VIC->SlvSel[0][0].Reg = 0x14;
            VIC->SlvSel[0][1].Reg = 0x00;
            WriteCIOPortC((unsigned char) (Address >> 20));
            *SLAVE_STD_ENBL = 1;
            break;
        }
        case 'C':
        case 'c': {
            WriteCIOPortB((unsigned char) (Address >> 8));
            *SLAVE_SHT_ENBL = 1;
            break;
        }
                default: {
            xprintf("\nIllegal flag expected -e, -s or -c");
                }
    }
}

/***************************************************************************
 * SlaveDis(): Disables the VMEbus Address Space specified by 'Flag'
 *             which indicates either 'e' (extended space), 's'
 *             (standard space) or 'c' (communications or short space).
 ***/

SlaveDis(Flag)
char Flag;
{
    switch (Flag) {
        case 'E':
        case 'e': {
            *SLAVE_EXT_ENBL = 0;
            break;
```



```
        }
        case 'S':
        case 's': {
            *SLAVE_STD_ENBL = 0;
            break;
        }
        case 'C':
        case 'c': {
            *SLAVE_SHT_ENBL = 0;
            break;
        }
                default: {
            xprintf("\nIllegal flag expected -e, -s or -c");
                }
    }
}

/***************************************************************************
 * UnMaskVMEInt(): Unmasks VME interrupt number IRQNum. The interrupt
 *                 is presented to the 80960CA on IPL0 (INT0), IPL1
 *                 (INT1) or IPL2 (INT2) according to IPLNum. If the
 *                 IRQNum is 0 then all interrupts are masked off.
 ***/

UnMaskVMEInt(IRQNum, IPLNum)
unsigned char IRQNum, IPLNum;
{
    if(IRQNum == 0){
        for(IRQNum = 0; IRQNum < 8; IRQNum++){
            VIC->VMEIntCntrl[IRQNum].Reg = 0x80;
        }
    } else{
        VIC->VMEIntCntrl[IRQNum - 1].Reg = (1 << IPLNum);
    }
}
```

Appendix B — NV-RAM Information

The NV-RAM memory is an 8,192-byte EEPROM that contains manufacturing, service, and hardware configuration information; monitor and board initialization information; and user-defined information. The start address, size, and description of the device are given in Table B-1:

TABLE B-1
EEPROM addresses

Device Address	Byte Offsets	Data
$0270{,}0000_{16}$	$0 - 15FF_{16}$	User-defined data area
$0270{,}B000_{16}$	$1600_{16} - 17FF_{16}$	Monitor/board initialization
$0270{,}C000_{16}$	$1800_{16} - 1FFF_{16}$	Manufacturing/service hardware information

This appendix contains the following files:

NvMonDefs.h This header file defines the bit field assignments for the NVRAM/EEPROM, as they are defined by the board.

NVAssign.h This header file defines the bit field assignments for the NVRAM/EEPROM, as they are defined by Heurikon.

NVDefs.h This header file includes the basic error codes and the codes passed to NVOp to indicate the type of operations to perform on nonvolatile memory.

NVLib.c This file contains the nonvolatile library functions used to manage NVRAM or EEPROM.

NV.c This file contains the functions necessary to read, write, and configure the 28C64 EEPROM.

```
/*************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "NVAssign.h"        /* Pull in the Internal data definitions       */

#define MPU_80960CA     4    /* Fixed Hardware devices                      */
#define MMU_NONE        0
#define CACHE_NONE      0
#define FPU_NONE        0
#define DMA_960CA       5
#define MEMEXP_NONE     0
#define STREAM_NONE     0

#define ETH_82596CA     1    /* Ethernet may be optional                    */
#define ETH_NONE        0

#define HDISK_NONE      0    /* SCSI may be optional                        */
#define HDISK_WD33C93A  4

/*************************************************************************
 *  NvMonDefs.h: This header file defines the bit field assignments
 *               for the NVRAM/EEPROM, as they are defined by the board.
 *               It can be used where a program needs to know which bit fields
 *               are assigned to what.
 *               This section describes the board specifics and includes the
 *               Heurikon-specific structures and internal data structures
 *               necessary to maintain NV memory (NVAssign.h).
 *
 * NOTICE:      Because different compilers may generate different spacing
 *              between structures and structure elements based on the
 *              alignment it is important to define structures carefully.
 *              Problems can be avoided by forcing shorts and longs onto
 *              long and short boundaries and padding structures to be
 *              a multiple of long words in size.
 *
 *              An early version of the ic960 compiler generated the wrong
```



```
 *              structure addresses when the structures were organized as
 *              (long, short, byte) quantities in that order. If the smaller
 *              fields are first in the structure it works much better, so
 *              be careful !!!!.
 ***/

/******* SERIAL DEFINITIONS *********************************************
 * This structure provides the definitions for a serial port. This
 * includes the port number, baud rate and configuration.
 * This structure should be loaded in the user-configurable portion of
 * the nonvolatile memory array.
 ***/

typedef struct NVU_Port {              /* Port struct = 8/4 bytes        */
    unsigned char  Reserved;
    unsigned char  PortNum;            /* Port number (A,B,C or D)       */
    unsigned short PortFlags;          /* Flags for port                 */
    unsigned long  Baud;               /* Port baud rate                 */
} NVU_Port;

/* Warning: These macros only work with pointers                         */

#define Parity(x)           (x->PortFlags & 0x0003)
#define DataBits(x)        ((x->PortFlags & 0x000C) >> 2)
#define XOnXOff(x)          (x->PortFlags & 0x0010)
#define ChBaudOnBreak(x)    (x->PortFlags & 0x0040)
#define ResetOnBreak(x)     (x->PortFlags & 0x0080)
#define StopBits(x)         (x->PortFlags & 0x0100)

#define SP_APORT            0      /* Serial Port Assignments            */
#define SP_BPORT            1
#define SP_CPORT            2
#define SP_DPORT            3

#define SP_PARITY_EVEN      0      /* Parity Type Assignments            */
#define SP_PARITY_ODD       1
#define SP_PARITY_NONE      2
#define SP_PARITY_FORCE     3

#define SP_DATA_5BITS       0      /* Data Bits Assignments              */
#define SP_DATA_6BITS       1
#define SP_DATA_7BITS       2
#define SP_DATA_8BITS       3

#define SP_STOP_1BITS       0
#define SP_STOP_2BITS       1

/******* BOOT DEFINITIONS ***********************************************
 * This sections defines the boot parameters for loading an application
 * from a device and executing the application. This section should be
 * located in the user section of the nonvolatile memory device.
 ***/

typedef struct NVU_Boot {              /* Boot struct = 32/20 bytes      */
    unsigned char AutoBootDev;         /* Auto Boot Device               */
    unsigned char Device;              /* Boot Device                    */
    unsigned char Number;              /* Boot Device Number             */
    unsigned char BootFlags;           /* Boot Flags                     */
    unsigned long LoadAddress;         /* Load Address                   */
    unsigned long RomSize;             /* Boot ROM Size                  */
    unsigned long RomBase;             /* Boot ROM Base address          */
#ifdef NV_SMALL
    char Reserved[4];
#else
    char Reserved[16];
#endif
```

```
} NVU_Boot;

#define  ClrMemOnBoot(x)     (x->BootFlags & 0x01)    /* Clear on boot    */

#define AB_DONT            0          /* Auto Boot Definitions           */
#define AB_WINCH           1
#define AB_FLOPPY          2
#define AB_TAPE            3
#define AB_SERIAL          4
#define AB_ROM             6
#define AB_ETHERNET        7


/******* VME BUS DEFINITIONS *********************************************
 * This structure defines the VMEbus configuration of the slave interface
 * and Vic configuration registers. This structure should be loaded in
 * the user-defined section of the NV memory.
 ***/

typedef struct NVU_BusConfig {         /* BusConfig struct = 16/4 bytes  */
    unsigned char   IComReg[5];        /* Communications reg 0-4 values  */
    unsigned char   Padding;           /* Reserved                       */
    unsigned short  MiscBusFlags;      /* Misc bus configuration bits    */
    unsigned long   SlaveBusMap;       /* Slave bus map configuration    */
    unsigned char   Reserved[4];       /* Reserved                       */
} NVU_BusConfig;

#define ExtSlaveMap(x)      (x->SlaveBusMap  & 0xFF000000)
#define StdSlaveMap(x)      (x->SlaveBusMap  & 0x00F00000)
#define ShtSlaveMap(x)      (x->SlaveBusMap  & 0x0000FF00)

#define ExtSlaveEnbl(x)     (x->SlaveBusMap  & 0x00080000)
#define StdSlaveEnbl(x)     (x->SlaveBusMap  & 0x00040000)
#define ShtSlaveEnbl(x)     (x->SlaveBusMap  & 0x00020000)

#define BusReqLev(x)        (x->MiscBusFlags & 0x0003)
#define MastRelMode(x)     ((x->MiscBusFlags & 0x000C) >> 2)
#define LocBusTimer(x)     ((x->MiscBusFlags & 0x0070) >> 4)
#define VmeBusTimer(x)     ((x->MiscBusFlags & 0x0380) >> 7)
#define ArbiterMode(x)      (x->MiscBusFlags & 0x0400)
#define SlaveWrPost(x)      (x->MiscBusFlags & 0x0800)
#define MasterWrPost(x)     (x->MiscBusFlags & 0x1000)
#define Sysfail(x)          (x->MiscBusFlags & 0x2000)
#define TurboMode(x)        (x->MiscBusFlags & 0x4000)
#define IndivRMC(x)         (x->MiscBusFlags & 0x8000)


/******* MONITOR DEFINED DEFINITIONS *************************************
 * This section binds the Monitor-defined data structures into one
 * common structure, which should be loaded into NV memory in the user
 * read/write section.
 ***/

typedef struct NV_MonDefs {           /* Mon Defs struct = 76/48          */
    NV_Internal    Internal;          /* Internal definitions             */
    unsigned long  MiscFlags;         /* Misc monitor flags               */
    unsigned long  ProcFlags;         /* Proc monitor flags               */
    NVU_Port       Console;           /* Console Port Configuration       */
    NVU_Port       DownLoad;          /* Download Port Configuration      */
    NVU_Boot       Boot;              /* Boot Definitions                 */
    NVU_BusConfig  VmeBus;            /* Bus Configuration Definitions    */
} NV_MonDefs, *NV_MonDefPtr;

#define  ClrMemOnPowerUp(x) (x->MiscFlags & 0x01)  /* Clear on powerup        */
#define  ClrMemOnReset(x)   (x->MiscFlags & 0x02)  /* Clear on reset          */
#define  DoPowerDiag(x)     (x->MiscFlags & 0x04)  /* Do powerup diagnostics */
```



```
#define  VsbMasterEnbl(x)    (x->MiscFlags & 0x08)  /* VSB Master enable      */
#define  VsbReleaseMode(x)   (x->MiscFlags & 0x10)  /* VSB Release modes      */
#define  EthArbiterEnbl(x)   (x->MiscFlags & 0x20)  /* Ethernet Arbiter Enbl  */
#define  EthByteEndian(x)    (x->MiscFlags & 0x40)  /* Ethernet Arbiter Enbl  */
#define  ScsiResetEnbl(x)    (x->MiscFlags & 0x80)  /* Scsi reset enable      */

#define  InstCacheEnble(x)   (x->ProcFlags & 0x10)  /* Instruction cache Enbl */
#define  RegCacheSize(x)     (x->ProcFlags & 0x0F)  /* 960 Regs cached        */
```

```
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

/***************************************************************************
 *  NVAssign.h: This header file defines the bit field assignments
 *              for the NVRAM/EEPROM, as they are defined by Heurikon.
 *              It can be used where a program needs to know which bit fields
 *              are assigned to what.
 *              Note that the memory is divided into two separate sections:
 *              the Heurikon-defined, or write-protected, region and the
 *              user-defined region that can be modified interactively
 *              from the monitor or external programs.
 *
 * NOTICE:      Because different compilers may generate different spacing
 *              between structures and structure elements based on the
 *              alignment it is important to be careful defining structures.
 *              Problems can be avoided by forcing shorts and longs onto
 *              long and short boundaries and padding structures to be
 *              a multiple of long words in size.
 *
 *    NOTE:     The definition 'NV_SMALL' is intended to conserve space
 *              for smaller NV devices, which can be as small as 128 bytes.
 *
 ***/

/******* INTERNAL BIT DEFINITIONS ************************************
 *  This structure provides the internal structures necessary to
 *  maintain a nonvolatile section of memory. The magic number is
 *  used to quickly determine if the structure has been initialized.
 *  The checksum is used to verify the validity of the data. The
 *  write count indicates the number of times the section has been
 *  written and provides an indicator of the lifetime of the component.
 *
 *  This structure must be the first entry in a nonvolatile section.
 *  Many of the functions that manipulate nonvolative sections assume that
 *  this is the first structure in the section and will not function
```



```
 *  if it is omitted.
 ***/

typedef struct NV_Internal {        /* Internal structure = 8 bytes    */
    unsigned short Magic;           /* Magic number                    */
    unsigned short WriteCnt;        /* Write Count                     */
    unsigned long  ChkSum;          /* CheckSum                        */
} NV_Internal, *NV_InternalPtr;

#define NV_MAGIC        0x57CE      /* Magic number for nv memory      */

/******* BOARD BIT DEFINITIONS ****************************************
 * The Manufacturing structure provides information necessary to
 * track the board's manufacturing history, revision, ship date, etc.
 * This structure is located in the write-protected region of the
 * nonvolatile memory device. Modification should only be done
 * by Heurikon's manufacturing departement.
 ***/

typedef struct NVH_Manufacturing { /* Manuf struct = 88/8 bytes       */
    unsigned char  Revision;        /* Board Revision                  */
    unsigned char  ECOLevel;        /* Board ECO Level                 */
    unsigned short SerialNumber;    /* Board Serial Number             */
#ifdef NV_SMALL
    char Reserved[4];
#else
    char Model[8];                  /* Board Model                     */
    char ManDate[12];               /* Manufacturing Date              */
    char ManPartNum[12];            /* Manufacturing Part Number       */
    char WorkOrderNum[12];          /* Work Order Number               */
    char Reserved[40];
#endif
} NVH_Manufacturing;

/******* SERVICE DEFINITIONS ******************************************
 * This structure provides the service record of the board. This
 * structure consists of the RMA number, Ship Date, Technician name
 * and a short description of the problem. The last 3 records are
 * allowed to be stored in nonvolatile memory.
 ***/

typedef struct NVH_ServRec {        /* ServRec Struct = 72 bytes       */
    char RecNum[12];                /* Service Record Number           */
    char Date[12];                  /* Service Record Date             */
    char Tech[8];                   /* Service Record Technician       */
    char Problem[40];               /* Service Record Technician       */
} NVH_ServRec;

typedef struct NVH_Service {        /* Service Struct = 232 bytes      */
    NVH_ServRec Rec[3];             /* Storage for the last three      */
    char Reserved[16];              /* service records                 */
} NVH_Service;

/******* HARDWARE DEFINITIONS *********************************************
 * Board Hardware definitions are provided by this structure, which
 * describes memory sizes and peripheral configuration.
 ***/

typedef struct NVH_Hardware {       /* Hardware Struct = 36/24 bytes   */
    unsigned char MPUType;          /* Processor Type                  */
    unsigned char MMUType;          /* MMU Type                        */
    unsigned char CacheType;        /* Cache Type                      */
    unsigned char FPUType;          /* Floating Point Type             */
    unsigned char DMAType;          /* DMA Type                        */
    unsigned char MemExpType;       /* Memory Expansion Type           */
    unsigned char DiskType;         /* Hard Disk Controller Type       */
```

```
    unsigned char TapeType;         /* Streaming Tape Type              */
    unsigned char EthernetType;     /* Ethernet Controller Type         */
    unsigned char Padding[3];
    unsigned long DRAMSize;         /* Dynamic RAM Size                 */
    unsigned long SRAMSize;         /* Static RAM Size                  */
    unsigned long NVMemSize;        /* Nonvolatile memory size          */
#ifndef NV_SMALL
    char Reserved[12];
#endif
} NVH_Hardware;

#define RAMSIZ_0        0x00000000
#define RAMSIZ_128      0x00000080
#define RAMSIZ_8K       0x00002000
#define RAMSIZ_16K      0x00004000
#define RAMSIZ_32K      0x00008000
#define RAMSIZ_64K      0x00010000
#define RAMSIZ_128K     0x00020000
#define RAMSIZ_256K     0x00040000
#define RAMSIZ_512K     0x00080000
#define RAMSIZ_1M       0x00100000
#define RAMSIZ_2M       0x00200000
#define RAMSIZ_3M       0x00300000
#define RAMSIZ_4M       0x00400000
#define RAMSIZ_8M       0x00800000
#define RAMSIZ_12M      0x00C00000
#define RAMSIZ_16M      0x01000000
#define RAMSIZ_32M      0x02000000
#define RAMSIZ_64M      0x04000000

/******* COMBINED HEURIKON DEFINED VALUES ********************************
 *  The combination of the hardware, manufacturing record and the service
 *  record are bound together in this structure, which is stored in the
 *  protected region of the nonvolatile memory.
 ***/

typedef struct NV_HkDefined {           /* Hk struct = 40/444 bytes       */
    NV_Internal         Internal;       /* Internal definitions           */
    NVH_Hardware        Hardware;       /* Hardware definitions           */
    NVH_Manufacturing   Manuf;          /* Manuf definitions              */
#ifndef NV_SMALL
    NVH_Service         Service;        /* Service record                 */
#endif
} NV_HkDefined;
```

```c
/****************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

/****************************************************************************
 *      NvDefs.h:  This header file includes the basic error codes and the
 *             codes passed to NVOp to indicate the type of operations
 *             to perform on nonvolatile memory.
 ***/

/****************************************************************************
 *      The Error flags are defined below. Note that these error codes have
 *  been used to construct error tables and must not be modified for
 *  any reason.
 */

#define NVE_NONE          0          /* No error                          */
#define NVE_OVERFLOW    1            /* Warning: To many writes done      */
#define NVE_MAGIC         2          /* Bad magic number in NVRAM image   */
#define NVE_CKSUM         3          /* Bad checksum in NVRAM image       */
#define NVE_STORE         4          /* Could not write NVRAM to memory   */
#define NVE_CMD         5            /* Unknown command requested         */

#define NV_OP_FIX         0     /* NVOp Command to fix checksum       */
#define NV_OP_CLEAR       1     /* NVOp Command to clear NV section   */
#define NV_OP_CK          2     /* NVOp to checksum NV sections       */
#define NV_OP_OPEN        3     /* NvOp to Open NV Section            */
#define NV_OP_SAVE        4     /* NvOp to Save NV Section            */
```

```
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION.
 *         The copyright notice above does not evidence any
 *         actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

/***************************************************************************
 * NVLib.c:  This file contains the nonvolatile library functions used to
 *           manage NVRAM or EEPROM. The functions defined in this module
 *           are listed below:
 *
 *               NVSet()          NVDisplay()      NVInit()
 *               FindGroup()      FindField()      DispFieldName()
 *               FieldRead()      FieldWrite()     Continue()
 *               NVUpdate()       NVOpen()         NVOp()
 */

#include        "Bug.h"
#include        "NVDefs.h"
#include        "NVAssign.h"

extern NVGroup       NVGroups[];    /* NV memory groupings structure     */
extern NV_HkDefined  HKFields;      /* Heurikon defined structure        */

/***************************************************************************
 * NV Error Strings():
 *
 * NOTE:  The Error table strings are defined according to the
 *        definitions in 'NVDefs.h'. Neither of these files should be
 *        modified without fear of complete disaster.
 ****/

static char NVErr0Str[] = "No error";
static char NVErr1Str[] = "Maximum write count exceeded";
static char NVErr2Str[] = "Bad magic number";
static char NVErr3Str[] = "Illegal checksum";
static char NVErr4Str[] = "Write to NV memory does not verify";
static char NVErr5Str[] = "Unknown command";
```



```
static char *NVErrTable[] = {
   NVErr0Str,  NVErr1Str,  NVErr2Str,  NVErr3Str,  NVErr4Str,
   NVErr5Str
};

/***************************************************************************
 *  String definitions for error reporting.
 ***/

static char NVSupStr1[] = "\nError while %s NV memory. %s.";
static char NVSupStr2[] = "\nWarning protected region of NV memory %s.";

/****************************************************************************
 * NVOp(): This function provides the operations on the NV memory
 *         structures to read, write, clear, checksum and update
 *         NV memory.
 ****/

NVOp(Operation, Base, Size, Offset)
unsigned long Operation, Size, Offset;
unsigned char *Base;
{
    int ByteNum, DataSize;
    unsigned char *DataSect;
        unsigned char NVRamAcc();
    unsigned long Sum;
    NV_Internal *Internals = (NV_Internal *) Base;

    DataSect = (unsigned char *) &Internals[1];
    DataSize = Size - sizeof(NV_Internal);
    Sum = CheckSumMem(DataSect, DataSize);

    switch (Operation) {
        case NV_OP_FIX: {
            Internals->Magic  = NV_MAGIC;
            Internals->ChkSum = Sum;
            return(NVE_NONE);
        }
        case NV_OP_CLEAR: {
            ClearMem(DataSect, DataSize);
            Internals->Magic    = NV_MAGIC;
            Internals->ChkSum   = 0;
            return(NVE_NONE);
        }
        case NV_OP_CK: {
            if (Internals->Magic != NV_MAGIC)
                return(NVE_MAGIC);
            if (Internals->ChkSum != Sum)
                return(NVE_CKSUM);
            if (Internals->WriteCnt > NVRMaxNbrWrites())
                return(NVE_OVERFLOW);
            return(NVE_NONE);
        }
        case NV_OP_OPEN: {
            nv_recall();
            for (ByteNum = 0; ByteNum < Size; ByteNum++) {
                Base[ByteNum] = NVRamAcc(READ, Offset + ByteNum);
            }
            return(NVE_NONE);
        }
        case NV_OP_SAVE: {
            nv_store();
            for (ByteNum = 0; ByteNum < Size; ByteNum++) {
                NVRamAcc(WRITE, Offset + ByteNum, &Base[ByteNum]);
            }
            nv_recall();
```

```
            for (ByteNum = 0; ByteNum < Size; ByteNum++) {
                if (Base[ByteNum] != NVRamAcc(READ, Offset + ByteNum)) {
                    return(NVE_STORE);
                }
            }
            return(NVE_NONE);
        }
        default: {
            return(NVE_CMD);
        }
    }
}

/***************************************************************************
 * NVUpdate(): This monitor command updates the NVRAM from the image
 *             maintained in memory. Error messages are displayed if an
 *             error occurs.
 ****/

NVUpdate()
{
    register int Err;
    NV_Internal   *NvMon      = (NV_Internal *) NvMonAddr();
    unsigned long NvMonSiz   = NvMonSize();
    unsigned long NvMonOff   = NvMonOffset();
    unsigned long NvHkOff    = NvHkOffset();

    NvMon->WriteCnt++;
    Err = NVOp(NV_OP_CK, NvMon, NvMonSiz);
    if (Err != NVE_NONE) {
        xprintf(NVSupStr1, "reading", NVErrTable[Err]);
        return;
    }
    Err = NVOp(NV_OP_SAVE, NvMon, NvMonSiz, NvMonOff);
    if (Err != NVE_NONE) {
        xprintf(NVSupStr1, "storing", NVErrTable[Err]);
        return;
    }


    HKFields.Internal.WriteCnt++;
    NVOp(NV_OP_CK, &HKFields, sizeof(NV_HkDefined));
    Err = NVOp(NV_OP_SAVE, &HKFields, sizeof(NV_HkDefined)   , NvHkOff);
    if (Err != NVE_NONE) {
            xprintf(NVSupStr2, "was not modified");
        return;
    }
}


/***************************************************************************
 * NVOpen(): This monitor command opens the NVRAM and loads the memory
 *           image from the device. Errors are detected and displayed
 *           if they occur.
 ****/

NVOpen()
{
    register int Err;
    NV_Internal   *NvMon      = (NV_Internal *) NvMonAddr();
    unsigned long NvMonSiz   = NvMonSize();
    unsigned long NvMonOff   = NvMonOffset();
    unsigned long NvHkOff    = NvHkOffset();


    NVOp(NV_OP_OPEN , &HKFields,   sizeof(NV_HkDefined)   , NvHkOff);
```



```
    NVOp(NV_OP_OPEN , NvMon, NvMonSiz, NvMonOff);
    Err = NVOp(NV_OP_CK, &HKFields, sizeof(NV_HkDefined));
    if (Err != NVE_NONE) {
            xprintf(NVSupStr2, "is corrupt");
    }
    Err = NVOp(NV_OP_CK, NvMon, NvMonSiz);
    if (Err != NVE_NONE) {
            xprintf(NVSupStr1, "reading", NVErrTable[Err]);
    }
    return Err;
}
```

```
/***************************************************************************
 *
 * Copyright (c) 1990 Heurikon Corporation
 * All Rights Reserved
 *
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF HEURIKON CORPORATION
 *        The copyright notice above does not evidence any
 *        actual or intended publication of such source code.
 *
 * Heurikon hereby grants you permission to copy and modify
 * this software and its documentation.  Heurikon grants
 * this permission provided that the above copyright notice
 * appears in all copies and that both the copyright notice and
 * this permission notice appear in supporting documentation.  In
 * addition, Heurikon grants this permission provided that you
 * prominently mark as not part of the original any modifications
 * made to this software or documentation, and that the name of
 * Heurikon Corporation not be used in advertising or publicity
 * pertaining to distribution of the software or the documentation
 * without specific, written prior permission.
 *
 * Heurikon Corporation does not warrant, guarantee or make any
 * representations regarding the use of, or the results of the use
 * of, the software and documentation in terms of correctness,
 * accuracy, reliability, currentness, or otherwise; and you rely
 * on the software, documentation and results solely at your own
 * risk.
 *
 *
 * MODIFICATIONS:
 *
 *****/

#include "Bug.h"
#include "Board.h"
#include "NvMonDefs.h"

/***************************************************************************
 * NV.c:   This file contains the functions necessary to read, write and
 *         configure the 28C64 EEPROM.
 *         The functions defined in this module are listed below:
 *
 *               nv_recall()         nv_store()
 *               NVRMaxNbrWrites()   NvHkOffset()     NvMonOffset()
 *               NvMonSize()         NvMonAddr()      NVRamAcc()
 *
 * NOTE:  This file contains several functions that allow compatability
 *        with NVRAM that requires store and recall operations.
 *
 ***/

extern NV_HkDefined  HKFields;
extern NV_MonDefs    NvMonDefs;

/***************************************************************************
 * nv_recall():
 * nv_store():  included to provide compatability with NVRAM.
 ***/

nv_recall() { }                /* No Recall for EEPROM */
nv_store()  { }                /* No Store  for EEPROM */

/***************************************************************************
 *  NVRMaxNbrWrites(): This function allows the NV libraries to determine
 *                     the lifetime of a component without including
```



```
 *                     the board header file.
 ***/

NVRMaxNbrWrites()  {                /* Returns limit of write count */
    return(NV_MAX_NBR_WRITES);
}

/***************************************************************************
 * NvHkOffset():
 * NvMonOffset():
 * NvMonSize():
 * NvMonAddr(): These functions may seem a bit overkill but they allow the
 *              NV library functions to operate on the NV memory sections
 *              without actually compiling the board config files into the
 *              library. This is desirable because they will change from
 *              board to board.
 ***/

NvHkOffset()  {
    return(NV_PROTECTED);
}

NvMonOffset()  {
    return(NV_MON_DEFS);
}

NvMonSize()  {
    return(sizeof(NV_MonDefs));
}

NvMonAddr()  {
    return( (int) &NvMonDefs);
}


/***************************************************************************
 * NvRamAcc(): This function provides the reading and writing of the
 *             device itself. The 'Mode' indicates if a read or write
 *             is requested, the 'Cnt' indicates the byte location to
 *             modify (if NV were seen as a linear array), and 'Val'
 *             is the value when doing a write.
 *
 *             Returned from this function is the number of bytes
 *             written to the device or the value read from the device
 *             depending on 'Mode'.  This function supports bursts on
 *             writes to speed the storing of data around 32 times.
 *             The burst size is determined by NV_PAGE_SIZE. Another
 *             optimization is that only bytes that differ are written.
 ***/

unsigned char NVRamAcc(Mode, Cnt, Val)
unsigned long Mode, Cnt;
unsigned char *Val;
{
    int i, j;
    unsigned char *NVRead, *NVWrite;
    unsigned char RamVal;
    int IsSame = TRUE;

    NVRead = NVWrite = (unsigned char *) (NV_BASE + (NV_SPACING * Cnt));

    if (Mode == READ) {     /* Read:  return value */
        return(*NVRead);
    } else {                /* Write: scan page for changes. If none return */
        for (i = 0; i < NV_PAGE_SIZE; i++) {
            if (*NVRead != Val[i]) {
```

```
                IsSame = FALSE;
                break;
            }
            NVRead  += NV_SPACING;
        }
        if (IsSame) {
            return(NV_PAGE_SIZE);      /* If no changes then return */
        }
        /* write page to EEPROM */

        for (i = 0; i < NV_PAGE_SIZE; i++) {
            AtomicModify(NVWrite, 0xFF, Val[i]);
            NVWrite += NV_SPACING;
        }                               /* Repeatedly attempt to verify */

        for (j = 0; j < 0x1000; j++) {
            NVRead  = (unsigned char *) (NV_BASE + (NV_SPACING * Cnt));
            IsSame = TRUE;
            for (i = 0; i < NV_PAGE_SIZE; i++) {
                if (*NVRead != Val[I]) {
                    IsSame = FALSE;
                }
                NVRead  += NV_SPACING;
            }
            if (IsSame) {
                return(NV_PAGE_SIZE);
            }
        }
        return(NV_PAGE_SIZE);
    }
}
```

Appendix C

80960CA and 82596CA Implementation Notes and Errata

C.1 80960CA

There are currently two versions of the 80960CA CPU (location U100): the A4 and the newer B1.

The package marking on the CPU includes designation of the 80960CA version, as shown in Table C-1.

TABLE C-1
Guide to 80960CA versions

Part Description	Version	Package Markings
80960CA-33MHz	A4	Q 8231 or S V743
80960CA-25MHz	A4	S V595
80960CA-33MHz	B1	Q 8264 or S V735
80960CA-25MHz	B1	S V734

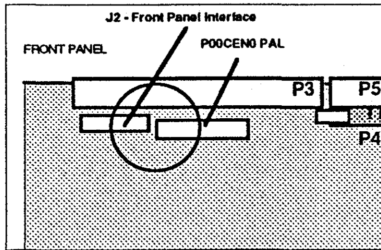
This document contains a list of functions that are either not fully implemented or do not function as originally documented for both 80960CA versions. In some instances, workaround solutions are suggested.

The details of 80960CA versions have been provided by Intel Corporation for Heurikon customers only and must remain confidential.

Main Differences between A4 and B1:

One of the main differences between the A4 step and the B1 step of the 80960CA is the addition of a "backoff" mechanism (similar to the one used on the 82596CA) that allows the HK80/V960E to recover correctly from VMEbus collisions. A collision could result when an HK80/V960E request for the VMEbus, via either the 80960CA or the 82596CA, occurs simultaneously with a VMEbus slave access to the

HK80/V960E. An HK80/V960E equipped with the A4 step of the 80960CA does not manage VMEbus collisions because the A4 step has no backoff mechanism. If a collision occurs, a deadlock usually results and the HK80/V960E hangs. This is not a problem in the B1 step of the 80960CA.



Using a 14-pin transition connector on J2:

If a 14-pin transition connector is used on J2 (the front panel interface), it might need to be trimmed slightly on the right side because the P00CEN0 PAL is so close. We can provide cable-connector assemblies that work properly with J2. The problem will be fixed in the next board revision.

C.1.1 80960CA Step A4

C.1.1.1 Type A Errata — Features That Are Not Implemented

Erratum A-1 — One-X Clock Mode Not Implemented

The one-x input mode for CLKIN is not implemented. The CLKMODE pin must be left unconnected, or must be grounded at all times for proper operation of A-0 material (in the two-x mode). When the one-x mode is implemented, it will be enabled by pulling CLKMODE high at all times.

Erratum A-2 — DMA Modes and Alignment Restrictions

The DMA modes supporting 8 to 16 (mode 1) and 16 to 8 (mode 4) were not implemented. These two modes cannot be used. The absence of these modes also places alignment constraints on other modes. These constraints are as follows:

TRANSFER TYPE	CONSTRAINT
1H 8 to 16	Not Implemented
4H 16 to 8	Not Implemented
5H 16 to 16	For synchronized transfers, the source and destination addresses must be 16-bit aligned, and the byte count must be an even number of bytes. Block transfers are not affected.

7H	**16 to 32**	Source-synchronized transfers must have 16-bit aligned source and destination addresses, and the byte count must specify an even number of bytes. Destination-synchronized transfers must have 32-bit aligned destination addresses, and the byte count must specify a multiple of 4 bytes. For block transfers, the DMA controller will perform the transfers as documented, with the exception that 8- to 8- bit transfers will be used to handle unaligned cases where the DMA should use 16- to 8-, or 8- to 16-bit transfers.
0DH	**32 to 16**	Destination-synchronized transfers must have 16-bit aligned source and destination addresses, and the byte count must specify an even number of bytes. Source-synchronized transfers must have 32-bit aligned source and 16-bit aligned destination addresses, and the byte count must specify a multiple of 4 bytes. For block transfers, the DMA controller will perform the transfers as documented, with the exception that 8- to 8-bit transfers will be used to handle unaligned cases where the DMA should use 16- to 8-, or 8- to 16-bit transfers.

Erratum A-3 — Locked Cache Not Implemented

The feature which allows interrupt routines to be locked into the instruction cache was not implemented. On the A-0 material, only the interrupt entry type 00_2 is supported.

Erratum A-4 — Self-Test Not Implemented

The internal self-test feature is not implemented. The processor does perform the external bus confidence test, but no specific internal self-testing is performed, regardless of the STEST pin setting. Even without a special self-test routine, approximately 50% of the processor's internal nodes toggle during the execution of the bus confidence test and the other initialization activity.

C.1.1.2 Type B Errata — Implemented Features That Do Not Function As Desired

Erratum B-1 — Instructions

a. **balx and bx are One Clock Slow.** The **balx** and **bx** instructions are one clock slower than documented in the first edition of the user manual. The user's manual will be changed to reflect this.

b. **calls Followed by Conditional Instruction.** A conditional instruction that immediately follows a **calls** will execute based upon the condition code state prior to the call. During proper operation the conditional instruction would execute based upon the state of the condition codes as

CONFIDENTIAL. For Heurikon customers only.

modified by the called procedure. Place two nops after **calls** instructions as a workaround if needed. Since this situation is pathological, the assembler does not emit an automatic workaround.

c. **callx Frame Spills.** If a **callx** using the *(reg1)[reg2*scale]* or *disp (reg1)[reg2*scale]* addressing modes causes a frame spill, the processor will malfunction. Insert a **call** to a dummy procedure, or insert a **flushreg** instruction prior to such **callx** instructions. The linker which supports the 80960CA warns of these **callx** occurrences.

d. **shrdi with fixup.** When execution of the **shrdi** instruction requires dividend fixup, and the **shrdi** is followed immediately by a micro-flow instruction, the processor will hang up. The ASM960 which supports the 80960CA will insert two nops after all **shrdi** instructions.

e. **Zero-Divide on Remainder and Modulo Instructions.** A divide-by-zero fault on **remi**, **remo** and **modi** instructions could cause the processor to hang up. The ASM960 which supports the 80960CA contains a switch which inserts a workaround for this condition. When these instructions are encountered, a **cmp***, **faulte** sequence is inserted to detect for zero division.

f. **Micro-Flow Branch Target after Unaligned Access.** The processor will malfunction when executing the following sequence of instructions:
   - unaligned memory reference
   - RISC branch (Machine Type C)
   - to a Micro-flow MEM-format instruction (Machine Type •).

   The ASM960 which supports the 80960CA detects branch targets of machine type • and inserts a nop between the branch target label and the branch target.

g. **Overflow Flag.** When an overflow occurs on a **divi** instruction but the integer overflow fault is disabled, the integer overflow flag in the Arithmetic Controls Register is not set. When an overflow occurs on an **addi, shli, stib, stis** or **muli** instruction, but the integer overflow fault is disabled, the integer overflow flag may not be set. The flag will not be set if a microcoded instruction accessing the arithmetic controls register (AC) is executed prior to the completion of the overflowing instruction (6 clocks for **muli,** 2 clocks for others). During proper operation, the overflow flag should be set on integer overflows when the integer overflow fault is disabled.

h. **Replaced the next description.**

i. **Current RIP is r2, not on stack.** During proper operation, the return instruction pointer (RIP) for the currently active procedure is located in the register save area of the previous stack frame. To speed return operations, the

CONFIDENTIAL. For Heurikon customers only.

processor caches the current RIP in r2 of the active register set. However, the processor uses this cached RIP as the return pointer for every return operation. Hence, modifying the RIP in the previous stack frame, even after a **flushreg,** does not change the location to which the next **ret** will return. Programs which modify the current RIP must (1) execute a **flushreg**, (2) perform a **call**, a procedure which modifies the desired RIP in memory, and (3) execute a **ret**. Since the procedure called in step 2 will not be modifying its current RIP, the modified memory-based RIP will be that cached by the processor during the return operation in step 3. During proper operation, modification of any previous stack frame contents, following a **flushreg** would properly take effect on the subsequent return operations.

This workaround is architecturally compatible and will be portable across 80960 implementations. There is, however, a faster workaround which will not be portable. **THE FOLLOWING WORKAROUND MAY NOT WORK ON FUTURE STEPPINGS OF THE CHIP, AND WILL NOT EXECUTE CORRECTLY ON ANY OTHER 80960 IMPLEMENTATIONS.** When a program desires to modify the current RIP, the program may modify the upper 30 bits of register r2 to the desired new RIP value, while preserving the values of the least significant two bits of the register. A flushreg is not required.

j. **Link Pointers and RIPs**. The processor may erroneously set one or both of the two least significant bits of a branch_and_link link pointer, and the two least significant bits of any return pointer (RIP). During proper operation these bits are always zero. Since the processor ignores these bits when the values are used by control instructions, link pointers and RIPs may be used directly by branches and calls. However, using a link pointer or RIP as a component of an effective address for memory operations will not work properly. For this version of the chip, copy the desired link pointer and clear the last two bits of the copy for use as a memory address of a load, store or atomic memory reference.

k. **Returning to user mode.** When a supervisor procedure returns to the user mode procedure which called it, the processor does not switch back to user mode. The stack switch back to the procedure stack from the supervisor stack occurs correctly, but the processor is left in supervisor mode. To work around this problem, create a dummy user-mode system procedure which calls the desired supervisor-mode system procedure with a **calls**. Upon return from the supervisor procedure, the dummy procedure should then set the processor back to user mode before returning. The workaround flows as follows.

CONFIDENTIAL. For Heurikon customers only.

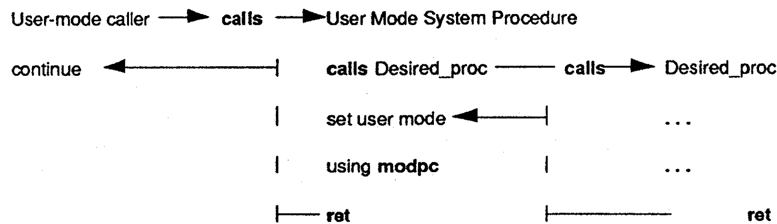


FIGURE C-1. Erratum B-2-k workaround

l. **ld, ldl, ldt, ldq Followed by a frame spill.** If load multiple instruction (ldl, ldt, ldq) is immediately followed by a call or an interrupt which forces a frame spill, the load will properly take place, but the registers which were the destination of the load will remain "scoreboarded". When this happens, the processor will hang at the first instruction that attempts to access these registers. If an **ld** instruction is executed and a frame spill occurs, the data in the destination register MAY be corrupted. There is no simple workaround relative to this particular problem other than to suggest that the user frequently flush the internally cached registers to insure that frame spills won't occur.

Erratum B-2 — Bus Controller

a. **LOCK# Deasserts Early.** The LOCK# pin will deassert one clock prior to the last ADS# of a sequence of atomic access. During proper operation, the pin should deassert after the last ASD# of a sequence of atomic accesses.

b. **Pipelined Region Limitation.** Each pipelined region which has burst enabled must have Ready Control disabled in that region. During proper operation, the ready pins would be ignored during reads in a pipelined region, but could be used in a write to a pipelined region.

c. **Region 0 Initialization.** The control table specified in the PRCB must contain the memory region configuration for region 15 in both the MCON0 and MCON15 entries. If the user desires a memory configuration for region 0 which is different from that for region 15, the user's initialization routine must reprogram the bus controller before accessing region 0. During proper operation, the control table pointed to by the PRCB would contain the accurate region 0 configuration in MCON0.

d. **Fixed on A-4 stepping.**

e. **Back-to-back accesses.** When two unaligned memory accesses of different types (e.g., unaligned LDQ followed by unaligned LDL) are executed back-to-back out of the instruction cache, an error may occur.

CONFIDENTIAL. For Heurikon customers only.

f. **Pipelined Fetches.** A two clock delay is encountered in a region programmed for pipelined accesses. This delay is only encountered for instruction fetches, not for loads or stores. The expected operation for a pipelined fetch is as follows (A = Address; D = Data):

```
A D D D D
        A D D D D
                A D D D D
```

However, the A4-stepping of the 80960CA performs a pipelined access as follows:

```
A D D D D
        X X A D D D D
                    X X A D D D
```

This, of course, reduces the bandwidth of the pipelined bus.

g. **Burst Fetches.** A one clock delay is encountered in a region programmed for burst access. This delay is only encountered for instruction fetches, not for loads or stores. The expected operation for a burst fetch is as follows (A = Address; D = Data):

```
A D D D D A D D D D A D D D D
```

However, the A4-stepping of the 80960CA performs a burst fetch as follows:

```
A D D D D X A D D D D X A D D D D
```

This, of course, also reduces the bandwidth of the pipelined bus.

Erratum B-3 — Instruction Cache

a. **Cache Disable Mode.** The cache disable configuration of the instruction cache turns off most, but not all, of the cache. When disabled, the cache will still cache two lines (16 words) of instructions. During proper operation, the entire cache would be disabled.

b. **Cache Flushing** The sysctl command to invalidate the instruction cache does not work. Furthermore, a sysctl reinitialize command does not flush the instruction cache as part of the software reset sequence. To invalidate the instruction cache, call a permanently located procedure longer than 256 words.

c. **Cache Control Initialization.** Although the instruction cache is successfully invalidated by a hardware chip reset, the cache replacement logic does not reset in a deterministic fashion. This makes it impossible to synchronously reset two processors to run in lock-step using only the RESET# pin. A possible workaround would be to have initialization routines in each processor perform a

cache flush as described above, then issue an external read to a fixed address, allowing external hardware to synchronize the processors.

Erratum B-4 — Register Cache

a. **Local Register Cache Size.** Programming a register cache size of 0 causes 15 sets to be allocated. During proper operation, the register cache should be disabled by programming 0 frames.

Erratum B-5 — DMA

a. **Temporary Lockout Condition.** The DMA controller could be frozen out by a user program which continually issues one-clock MEM-side instructions every clock. For example, a program which swamps the MEM-side of the processor with lda instructions could delay the DMA controller by as long as the longest uninterrupted sequence of ldas. This situation is rare for general code. For highly optimized code, the DMA delay could range from just a few clocks to the length of the optimized loop. There is no generally acceptable workaround. During proper operation, the user's program would not lock out the DMA.

b. **On-chip Data RAM as a Source.** The DMA will lock up if the source for a channel is in the internal data RAM. The DMA operates properly if only the destination is in internal data RAM. During proper operation, either the source or destination (or both) of a DMA transfer could be in data RAM.

c. **DACK3:0#.** The DACK3:0# pin associated with an access will stay active during Nxda cycles. During proper operation, the DACKx# pin which is asserted during a cycle would go high (inactive) during Nxda clocks. In addition, when a DMA-related bus transfer is pipelined with the following bus transfer, the DACKx# associated with the access will go high when the pipelined address for the next transfer appears. During proper operation, the DACKx# pin should go high after the last data cycle of the DMA transfer.

d. **Data Chaining Fly-by.** Fly-by transfers do not operate properly when data-chained. As a result, data chaining fly-by DMA transfers is not supported on this version of the chip.

e. **Data Chaining Alignment.** When chaining with the 16-32 and 32-16 bit modes, the source address and destination addresses must be aligned to the source and destination widths, respectively. During proper operation, there would be no such requirement.

f. **Byte Count and EOP#.** When an 8-32 or 16-32 bit destination synchronized demand transfer is terminated by an EOP# input, the final byte count will be less than the actual byte count by exactly 4 bytes. Source synchronized transfers

CONFIDENTIAL. For Heurikon customers only.

provide a correct byte count when terminated with an EOP# input. During proper operation, the byte count would reflect the actual number of bytes actually written to the destination when destination synchronized.

g. **EOP during 8-32 bit transfers.** If an EOP occurs during the last cycle of an 8-32 DMA transfer, the final byte(s) may be stored in consecutive destination addresses. This will only occur if the last word cannot be stored to a word aligned address.

h. **Corrupted 16 bit transfers.** 16-32, 32-16 and 16-16 bit transfers may corrupt data when buffers are not aligned to the "natural" boundaries while in the chaining mode.

i. **EOP/TC too early.** When programmed as an output the EOP/TC pin can potentially be asserted before the last DACK. This can only occur when the DMA channel is programmed in the DSDEM or destination synchronized demand transfer mode.

Erratum B-6 — Interrupts

a. **Temporary Lockout Condition.** No interrupt will be serviced between two back-to-back instructions which are micro-flows (Machine type = •). RISC nop instructions could be inserted between long micro-flow instructions (e.g., flushreg, sysctl) to reduce the spot interrupt latency; however, there is no generally acceptable workaround.

b. **One Software Interrupt Per Priority.** When the processor services an interrupt which was posted with the **sysctl** instruction, it clears the correct pending priority bit in the interrupt table without checking the associated pending priorities field for other interrupts. As a result, any other software posted interrupts which are also pending at the same priority level are lost. External interrupt requests are not affected. One workaround is to post no more than one interrupt per priority level with the **sysctl** instruction. If more than one software-interrupt per priority level is required, a software interrupt handler could check the pending interrupt bits associated with its priority, and re-post remaining interrupts using the **sysctl** instruction.

c. **"Posting" Priority Zero.** When the **sysctl** instruction is used to post a priority zero interrupt, the processor does not check all memory-based pending interrupt bits. Under normal operation, posting a priority zero interrupt should cause the processor to check the entire pending interrupt portion of the interrupt table for interrupts. This deviation is bothersome for multiprocessor applications. There is no general workaround. Workarounds will be application specific.

d. **Inoperative Mask Saving.** The interrupt controller option which governs the handling of the interrupt mask must be set to 00_2, disabling special mask handling. If any other

CONFIDENTIAL. For Heurikon customers only.

option is specified the interrupt mask will be cleared during an interrupt service and its proper value will be lost. If a workaround is required, interrupt handlers may save and clear the interrupt mask upon entry. However, the absence of the mask saving options precludes the use of a level sensitive priority 31 interrupt. All priority 31 interrupts must be edge sensitive in this version of the chip. During proper operation, the interrupt mask handling options should allow the user to specify what is done to the interrupt mask as an automatic part of processor interrupt handling.

e. **Pending Field Consistency Required.** If, when reading the memory-based pending interrupts field of the interrupt table, the processor detects a mismatch between the pending priorities field and the pending interrupt field, processor state will be corrupted. This mismatch can occur if an external agent only half-posts an interrupt by setting a bit in the pending priorities field without setting a bit in the associated pending interrupts field. Proper enforcement of locked atomic accesses among multiple agents should prevent this situation.

f. **DMA and interrupts.** The DMA suspension option for interrupt handling must be selected. If the DMA is not suspended during interrupt context switches, the processor could cease to operate properly.

g. **NMI Lockout.** If the NMI interrupt handler does a software reset, all future interrupts will be locked out. The only workaround is not to do a software reset while handling an NMI interrupt.

Erratum B-7 — Faults

a. **Unaligned.** Although the processor correctly performs unaligned memory accesses when the unaligned fault is disabled, the processor will hang up if an unaligned access occurs when the fault is enabled. During proper operation, a fault should be generated when an unaligned memory access is issued and the unaligned fault is enabled.

b. **Invalid Opcode in the User Mode.** An instruction which faults due to an invalid opcode may also cause an sfr protection fault. During proper operation, only the invalid opcode fault would be generated.

c. **Trace Controls on a return.** On a return from a non-trace fault handler to a user mode procedure, the Trace Controls Register (TC) is restored to its value prior to the fault. As a result, any modification of TC by a non-trace fault handler will be lost when the handler is returning to a user mode program. Furthermore, the TC event bits are automatically cleared during every return from a trace-fault handler. During normal operation the TC would not be altered by a return from a fault handler.

CONFIDENTIAL. For Heurikon customers only.

d. **Trace Faults on Faults and Interrupts.** A Trace Fault will not be taken after an implicit call even if Call Trace is enabled. To regain implicit call trace, place **fmark** instructions at the beginning of each fault handler. (Interrupts are not expected to generate implicit call faults. Neither K-series nor C-series processors do so.)

e. **Data Address Breakpoint Fault.** When a Data Address breakpoint fault occurs on a **callx,** or any call with a frame flush, the return IP (RIP) reported will be that of the call. The RIP should point to the first instruction of the called procedure. The trace fault handler must detect this condition and adjust the RIP before returning.

f. **sysctl and Faults.** No fault is generated when an unimplemented message is specified as an operand of the **sysctl** instruction. During proper operation, an operation.operand fault should be generated.

g. **Fault which shouldn't.** An instruction fetch from the on-chip data RAM will cause an operation-unimplemented fault even if the fetched data was not executed. During proper operation, this fault should only be generated if the processor attempts to execute data which was fetched from the data RAM.

h. **Data Address Breakpoints on stacks and tables.** If a data address breakpoint occurs on a memory access associated with the processor's interrupt or fault context switches, or execution of a **calls** instruction, the fault may not be signaled. If it is signaled, the associated fault record may be incorrect and the Trace Controls Register (TC) may be trashed. During proper operation, the data address breakpoint fault would be signaled after completion of all operations associated with these microcoded sequences. For this version of the silicon, it is recommended that data address breakpoints not be set on the system procedure table, fault table, interrupt table, or stack locations which will contain interrupt records or fault records.

i. **Pre-return Trace.** When a pre-return trace event occurs, a return trace is also signaled even if return trace was not enabled. The return trace event bit in the trace controls is erroneously set and the sub-type field of the fault record correctly reports a pre-return trace and incorrectly reports a return trace event. During proper operation, the TC event flags are only set if the associated TC mode is enabled.

j. **Call trace on calls.** A **calls** instruction currently signals both a supervisor trace event and a call trace event, even if the call is a local (non-supervisor) call. During proper operation, a system call (**calls**) should always signal a call event. The instruction should also signal a supervisor trace event if the system call is a supervisor call from user mode.

CONFIDENTIAL. For Heurikon customers only.

k. **Trace Control Event Bits.** The **modtc** instruction does not allow modification of the breakpoint event bits in the Trace Controls Register (TC). During proper operation, all event bits are open to modification by a **modtc** instruction.

l. **No Unimplemented sfr fault.** The processor does not signal a fault when the program attempts to access an unimplemented special function register. During proper operation a fault should be signaled.

m. **Trace Fault stack problem.** When a trace fault handler is serviced without a stack change, and the stack pointer (SP) prior to the fault is greater than FP+64 and is quad-word aligned, the processor clobbers the last word on the stack during the fault context switch. This condition occurs when a user mode trace fault handler is invoked when the processor is executing in user mode, and when a supervisor mode fault handler is invoked when the processor is executing in supervisor mode. The condition cannot occur when a supervisor mode fault handler is invoked when the processor is executing user mode code.

n. **Protection faults and Data RAM.** During proper operation, a user-mode attempt to write to protected data RAM should fault while attempts to read the data RAM are allowed even if protected. Supervisor mode programs may freely read and write any data RAM, protected or not. The first 256 bytes of data RAM are always protected while the remainder of the data RAM is optionally protected. The current version of the device operates as follows:

TABLE C-2
Erratum B-7-n — Current device operation

Address	Protected?	User-Mode Read	User-Mode Write
0H-0FFH	Always	Faults, but shouldn't	Faults
		Read completes	Write blocked
100H-3FFH	No	No fault	No fault
		Read completes	Write wrongly blocked
	Yes	Faults, but shouldn't	Faults
		Read Completes	Write blocked

Supervisor reads and writes of any data RAM complete correctly with no faults.

o. **Fault Type and Subtype errata.** Table C-3 lists the errant fault types and subtypes generated by this version of the device and the types and subtypes which should be generated during proper operation. Fault types or subtypes not listed are properly generated by this version of the device.

CONFIDENTIAL. For Heurikon customers only.

TABLE C-3
Errant and correct fault types and subtypes

Condition	Fault Generated	Correct Fault
Unaligned memory access	Operation-implemented	Operation — unaligned
Attempt to execute from on-chip RAM	Type-mismatch	Operation — unimplemented
Reference to unimplemented sfr	None	Operation — unimplemented
Invalid sysctl message	None	Operation — invalid operand
Protected RAM write in user mode	Type-mismatch	Protection — page rights
Protection-length	Subtype bit 0	Subtype bit 1

Erratum B-8 — Parallel Faults

Parallel faults exhibit unexpected operation when the following conditions occur and the processor is executing with the NIF bit cleared. When the NIF bit is set, parallel fault conditions will not occur (except item [e]).

a. **Lost Branch Trace.** When branch trace is enabled, and a branch executes in parallel with another instruction that causes a non-trace fault, the branch trace fault will not be seen. During proper operation, both faults would be reported at once with a parallel fault.

b. **Two Faults, not One: Parallel Branch.** When branch trace is enabled, and a branch executes in parallel with another instruction that causes a breakpoint trace fault, the branch trace fault will be seen after the breakpoint trace fault. During proper operation, both faults would be reported at once with a parallel fault.

c. **Two Faults, not One: Parallel R-M.** If a REG-side and a MEM-side instruction are issued in parallel, and there is an IP breakpoint set on the REG instruction, any fault arising from the MEM instruction will be seen prior to the break-point fault. During proper operation, all faults for both instructions should be reported in a single parallel fault record.

d. **Invalid Fault Record.** If both a zero-divide fault and an integer overflow fault from a multiply are reported in the same parallel fault record, the IP of the faulting multiply instruction and its associated fault record will be invalid. This situation occurs if an instruction which will cause a zero-divide fault is issued before the fault from a prior multiply is signaled. During proper operation, both faults would be correctly reported in a parallel fault record.

e. **Parallel Faults in NIF Mode.** When NIF is set: if a REG-side and MEM-side instruction could have been issued in parallel, all faults arising from the REG instruction and the MEM

CONFIDENTIAL. For Heurikon customers only.

instruction are reported in a single parallel fault record. During proper operation, the faults related to the two instructions would be reported through two faults, with the REG-related fault first.

C.1.2 80960CA Step B1

C.1.2.1 Type A Errata — Anomalies That Have Serious Consequences

None identified.

C.1.2.2 Type B Errata — Anomalies That Have Performance/Specification Implications

Erratum B-1 — Bus Controller

a. **Pipelined Fetches.** A two clock delay is encountered in a region programmed for pipelined accesses. This delay is only encountered for instruction fetches, not for loads or stores. The expected operation for a pipelined fetch is as follows (A = Address; D = Data):

```
A D D D D
        A D D D D
                A D D D D
```

However, the B-stepping of the 80960CA performs a pipelined access as follows:

```
A D D D D
        X X A D D D D
                    X X A D D D
```

This, of course, reduces the bandwidth of the pipelined bus.

b. **Burst Fetches.** A one clock delay is encountered in a region programmed for burst access. This delay is only encountered for instruction fetches, not for loads or stores. The expected operation for a burst fetch is as follows (A = Address; D = Data):

```
A D D D D A D D D D A D D D D
```

However, the B-stepping of the 80960CA performs a burst fetch as follows:

```
A D D D D X A D D D D X A D D D D
```

This, of course, also reduces the bandwidth of the pipelined bus.

CONFIDENTIAL. For Heurikon customers only.

Erratum B-2 — Reset Related

a. **Self-Test Implementation Flaw.** Unlike the A-stepping, the B-stepping does implement the self-test feature. However, a microcode flaw prevents it from being properly used. In order for the processor to initialize, you will need to tie STEST low (pin B02).

C.1.2.3 Type C Errata — Anomalies That Have Definitional Implications

Erratum C-1 — Instruction Cache

a. **Cache Disable Mode.** When the instruction cache is disabled, two cache lines (16 words) of the cache remain enabled. These two lines are not part of the 1024 byte cache, but they are more or less a cache queue. Given the burst and pipelining capability of the 80960CA's bus, disabling the entire cache is a relatively difficult task. We would appreciate feedback relative to addressing this problem.

Erratum C-2 — Register Cache

a. **Local Register Cache Size.** Programming a register cache size of 0 causes 15 sets to be allocated. During proper operation, the register cache should be disabled by programming 0 frames.

Erratum C-3 — Faults

a. **Data Address Breakpoint Fault.** When a Data Address breakpoint fault occurs on a **callx**, or any call with a frame flush, the return IP (RIP) reported will be that of the call. The architecture states that the RIP should point to the first instruction of the called procedure. The trace fault handler must detect this condition and adjust the RIP before returning.

b. **Data Address Breakpoints on stacks and tables.** If a data address breakpoint occurs on a memory access associated with the processor's interrupt or fault context switches, or execution of a **calls** instruction, the fault may not be signaled. If it is signaled, the associated fault record may be incorrect and the Trace Controls Register (TC) may be corrupted. During proper operation, the data address breakpoint fault would be signaled after completion of all operations associated with these microcoded sequences. For this version of the silicon, it is recommended that data address breakpoints not be set on the system procedure table, fault table, interrupt table, or stack locations which will contain interrupt records or fault records.

CONFIDENTIAL. For Heurikon customers only.

Erratum C-4 — Bus Controller

a. **Pipelined Region Limitation.** Each pipelined region which has burst enabled must have Ready Control disabled in that region. During proper operation, the ready pins would be ignored during reads in a pipelined region, but could be used in a write to a pipelined region.

C.2 82596CA

The details of 82596CA versions have been provided by Intel Corporation for Heurikon customers only and must remain confidential.

C.2.1 Erratum 1 — FIFO Operation Failure Region

Description: Corrected in the 82596A-1 stepping.

C.2.2 Erratum 2 — Truncated Frame on Transmit

Description: If CRS# deasserts (resulting from the end of a receive) during a 29-system-clock window that occurs near the time of the 82596 Transmit Command Block structure bus accesses, then the 82596 can transmit a truncated frame. The problem only occurs when all the following conditions are true.

- When using the Flexible Data Structure for transmit.
- The byte count in the transmit command block is greater than zero.
- The receive unit of the 82596 is receiving a frame with a destination address matching the 82596 address.
- A Transmit command is currently processing.

The 82596 will append an apparently correct CRC even if the truncation occurs. The receiving station will receive the frame without detecting a CRC error.

Consequences: If the receiving station does not compare the number of bytes received with the length field within the frame, the missing bytes can cause errors that will propagate to the upper layers of software.

Solutions:

Solution 1 When using the Flexible data structures, the byte count in the Transmit Command Block should be set to less than 54

CONFIDENTIAL. For Heurikon customers only.

bytes. If truncation occurs, the 82596 will transmit a frame of less than 64 bytes. Receiving stations will reject this short frame.

> **Note:** Short frames are quite common due to collisions, especially on networks meeting the proposed 10BASE-T standard.

Solution 2

Use 82586-compatible or simplified data structures.

C.2.3 Erratum 3 — Receive Unit (RU) Start When RU Active

Description:

If the Receive Unit (RU) is in the ready state, and an RU Start command is attempted, there is a small time-window in which the Receive Frame Descriptor list will be linked incorrectly. The state of the Receive Unit can be found in the RUS field of the System Control Block (SCB) Status word. An RU Start command is performed through the RUC field of the SCB Command word. The SCB Status and Control words are shown in Figure C-2.

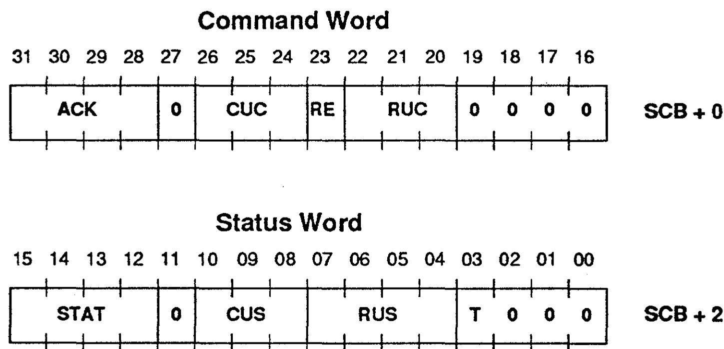


FIGURE C-2. Erratum 3 system control block status and control words

Consequences:

Incorrectly linking the Receive Frame Descriptors or Receive Buffer lists can cause lost or corrupted data in the receiving system.

Solution:

Before attempting an RU Start command through the RUC field, check the RU Status bits. If the status field indicates Ready (100 in 82586 Mode or x100 in 32-bit Segmented and Linear Modes) the 82596 receive operation should be suspended or aborted before attempting the RU Start. The 3-bit RUC field in the Command word is used to perform the Suspend (011) or Abort (100) commands.

CONFIDENTIAL. For Heurikon customers only.

C.2.4 Erratum 4 — Command Unit (CU) Abort when CU Suspended

Description:

If the Command Unit (CU) is in the Suspended state when a CU Abort command is issued, then a spurious write operation is performed after the CU is next started. This extra operation writes a 0 to the Busy bit of the prefetched Command Block (CB) of the previously aborted Command Block List (CBL). The Busy bit was already set to 0 by the CU Abort command. If no CB was prefetched before the CU Abort command, or if a CU Resume command is performed before the CU Abort command, the extra write operations will not occur.

The state of the 82596's Command Unit can be found in the CUS field of the System Control Block (SCB) Status word. A CU Start command is performed through the CUC field of the SCB Command Word. The SCB Status and Control words are shown in Figure C-3.

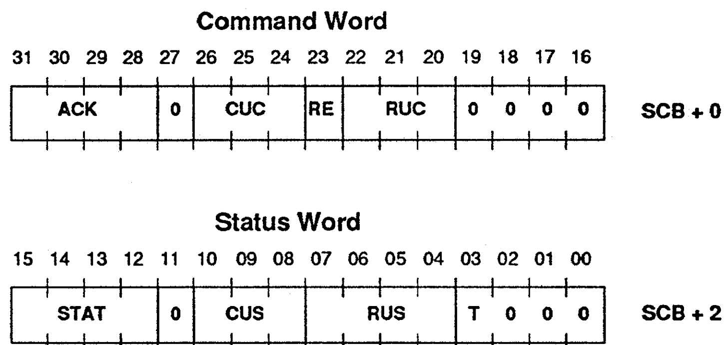


FIGURE C-3. Erratum 4 system control block status and control words

Solutions:

The second clearing of the Busy bit in the prefetched Command Block (CB) can cause a data corruption, but only if the memory space corresponding to that prefetched Command Block is allocated for some other purpose during the time after the CU Abort but before the next CU Start command. Another possible error can occur if the Command Unit is started with the previously prefetched Command Block as the start of the new Command Block List. If the software examines the status of the Busy bit, it can seem to go inactive before the 82596 has actually completed the command.

Solution 1

Before attempting a CU Suspend command or setting the S bit in a CB, a software flag should be set. Before attempting

CONFIDENTIAL. For Heurikon customers only.

a CU Abort command, the software flag should be checked. If it is set, the following procedure should be completed before an CBs are designated as available for reprocessing.

1. Generate the CU Abort command.
2. Load the CBL Offset field in the SCB with the address of a NOP CB with its EL bit set.
3. Generate a CU Start command.
4. Reprocess CB.

Solution 2

The CU status should be checked before attempting a CU Abort command. If the CU is in the Suspended state, a CU Resume command should be performed before the CU Abort. This will result in the completion of one or more CB after the suspended CB.

C.2.5 Erratum 5 — Revision of SCP Bit Values

Description:

Bits 0 through 15 (at byte ADR and ADR + 1) of the SCP should be set to zero and bit six of the Sysbus byte should be set to a one. If these bits are not set then the 82596 can fail to function properly.

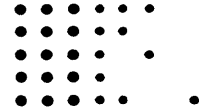
31 24	23 16	15 8	7 0	
xxxx xxxx	SYSBUS	0000 0000	0000 0000	ADR
xxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx	ADR + 4
AAAA AAAA	ISCP ADDRESS			ADR + 8

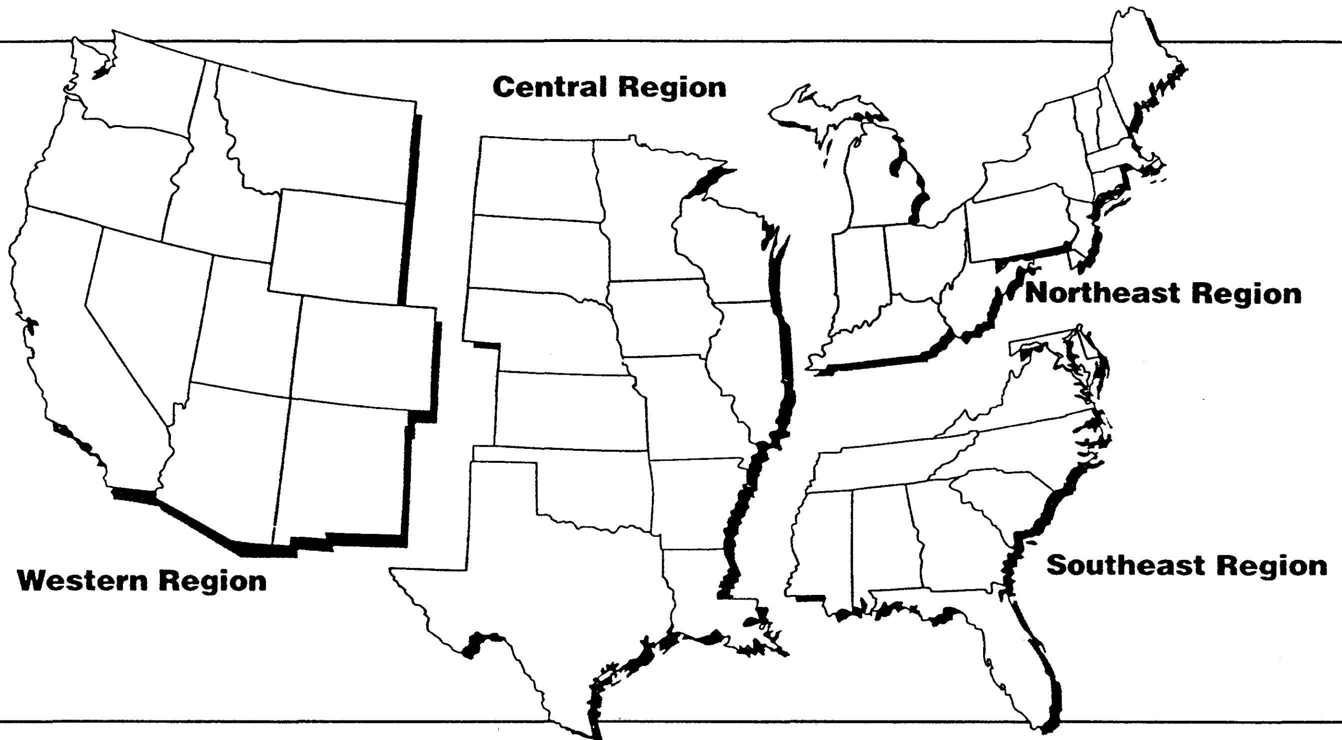
FIGURE C-4. Erratum 5 SCP values

Byte (AAAA AAAA) is defined as not checked in 82586-Compatible mode and is used as A31-A24 in the 32-bit Segmented and Linear modes.

The bits marked *x* are defined as not checked in 82586-Compatible mode and as zero in all other modes.

Sales and Customer Service Offices





Heurikon Corporate Office
8000 Excelsior Drive
Madison, WI 53717
Watts: 800-356-9602
Phone: 608-831-0900
Fax: 608-831-4249

Heurikon Customer Support and Factory Service Office
8310 Excelsior Drive
Madison, WI 53717
Watts: 800-327-1251
Phone: 608-831-5500

Heurikon Northeast Regional Office
67 South Bedford, Suite 400 W.
Burlington, MA 01803
Phone: 617-229-5831
Fax: 617-272-9115

Heurikon Southeast Regional Office
2010 Corporate Ridge, Suite 700
McLean, VA 22102
Phone: 703-749-1474
Fax: 703-556-0955

Heurikon Central Regional Office
13100 West 95th Street, Level 4D
Lenexa, KS 66215
Phone: 913-599-1860
Fax: 913-599-1918

Heurikon Western Regional Office
16496 Bernardo Center Drive, Suite 213
San Diego, CA 92128
Phone: 619-487-9771
Fax: 619-487-2562

HEURIKON® CORP.
OPEN SYSTEMS :: OPEN TOOLS

• NORTHEAST REGION

Territory	Representative
CT, DE, ME, MA, NH, NJ, NY, Eastern PA, RI and VT	**Daner-Hayes, Inc.** 62 West Plain Street Wayland, MA 01778 Tel: (508) 655-0888 Fax: (508) 655-0939
IN, KY, MI, OH, Western PA and WV	**Systems Components, Inc.** 1327 Jones, Suite 104 Ann Arbor, MI 48105 Tel: (313) 930-1800 Fax: (313) 930-1803

• CENTRAL REGION

Territory	Representative
AR, LA, OK and TX	**Acudata, Inc.** 720 Avenue F, Suite 104 Plano, TX 75074 Tel: (214) 424-3567 Fax: (214) 424-7342
MN, ND, SD and Northwest WI	**Micro Resources Corp.** 4640 W. 77th Street, Suite 109 Edina, MN 55435 Tel: (612) 830-1454 Fax: (612) 830-1380
IL, IA, KS, MO, NE and Southeast WI	**Panatek** 2500 West Higgins Road, Suite 305 Hoffman Estates, IL 60195 Tel: (708) 519-0867 Fax: (708) 519-0897

• SOUTHEAST REGION

Territory	Representative
MD, VA and Washington, DC	**Spectro Associates** 1107 Nelson Street, #203 Rockville, MD 20850 Tel: (301) 294-9770 Fax: (301) 294-9772

• WESTERN REGION

Territory	Representative
AZ, CO, NV, NM and UT	**Compware Marketing** 100 Arapahoe Ave. Suite 7 Boulder, CO 80302 Tel: (303) 786-7045 Fax: (303) 786-7047
ID, MT, OR, WA, WY and Canada (Alberta and British Columbia)	**Electronic Component Sales** 9311 S. E. 36th Street Mercer Island, WA 98040-3795 Tel: (206) 232-9301 Fax: (206) 232-1095
CA	**Qualtech** 333 West Maude Avenue, Suite 108 Sunnyvale, CA 94086 Tel: (408) 732-4800 Fax: (408) 733-7084

Index

A

address summary 15-1
arbiter enable, Ethernet 11-3
arbitration, bus 6-1
arithmetic faults 4-3

B

baud rates 10-8
branch instruction 3-20
branch trace mode 3-20
breakpoint registers 3-20
breakpoint trace mode 3-20
bus arbitration 6-1
bus collision 3-6
bus control 6-1
bus control request level 6-1
bus error 3-6
bus error (MPU) 4-1
bus grant 6-1
bus interrupts 6-4, 8
bus memory 5-3
bus priorities 6-3
bus watchdog timers 6-20
byte ordering 3-4; 11-1,2,4

C

cable, Centronics 13-5
cable, serial 10-10
cache 3-17
cache, data RAM 5-6
cache, instruction 3-21, 5-6
cache, register 3-22
caches 3-21, 5-6, 7-3
call instruction 3-20
call trace mode 3-20
call-system instruction 3-20
Centronics connector, P3 13-1
Centronics interrupt 13-2
channel attention 11-2, 4
CIO Port C function 9-1
CIO register address summary 9-5
CIO usage 9-1
clock, CIO 9-4
component locations 1-5
component map 1-4
configuration, memory 5-1
connector, Centronics, P3 13-1
connector, Ethernet 11-5
connector, SCSI 12-3
connector, VMEbus 6-21
connector, VSB 6-21
connectors 1-6,7
counter/timers (CIO) 9-4
CRT terminal, setup 2-3
customer service 2-5

D

damage in shipping 2-3
dedicated mode 3-9
DMA 3-16
DMA command register (DMAC) 3-18
DMA data alignment 3-19
DMA interrupts 3-19
DMAC 3-18

E

EEPROM 5-7
EEPROM partitions 5-8
EEPROM addresses B-1
environmental requirements 15-5
EPROM 2-4, 14-1
equipment for setup 2-1
error, timer frequency 9-4
errors, system response 4-1

ESD prevention 2-1
Ethernet arbiter enable 11-3
Ethernet byte ordering 11-2, 4
Ethernet connector 11-5
Ethernet port pin assignments, P6 11-5
expanded mode 3-9
extended space 2-2, 6-17

F
factory service 2-5
fault stack frame 3-15
fault table 3-14
features 1-1
fmark instruction 3-17, 20
FPI 8-2
front panel interface 8-2

H
hardware interrupts, MPU 3-5, 7

I
IBR 3-2
ICON register 3-9,13
IMAP registers 3-9,10,11,12
IMSK register 3-9,10,11
initialization 3-1
initialization boot record 3-2
initialization, CIO 9-5
installation 2-1
instruction cache 3-21
Instruction Cache Configuration Word 3-21
instruction trace mode 3-20
integer overflow 4-3
interrupt architecture 3-9
interrupt caching 3-9
interrupt control modes 3-9
interrupt control register 3-9, 13
interrupt handling, bus 6-9
interrupt map register format 3-12
interrupt mapping registers 3-11
interrupt mask register 3-10,11
interrupt priority 3-10
interrupt stack frame 3-6
interrupt structures 3-5
interrupt support, MPU 3-6
interrupt table 3-5
interrupt table base 3-5
interrupt, NMI 3-5
interrupt, VSB 7-3
interrupt-pending register 3-9,11
interrupter module, bus 6-8
interrupts, bus 6-4
interrupts, MPU 3-5
invalid opcode 4-2
invalid operand 4-2
IPND register 3-10,11
IRQ interrupt 7-3

J
J2 pin assignments 8-3
jumper settings 15-3
jumpers 1-6
jumpers, bus control 6-16
jumpers, watchdog 6-16

L
LEDs, user 8-1
length fault 4-3

M
mailbox 1-2; 6-1,13–15
mailbox interface 6-13–15
mark instruction 3-17, 19
mechanical specifications 15-5
memory configuration 5-1
memory map 5-4
memory space 5-3
memory timing 5-5
memory, bus 5-3
mixed mode 3-9
modpc instruction 3-10, 4-3
monitor 5-8
monitor program editor 2-6
MPU DMA support 3-16
MPU faults 3-14
MPU interrupt stack frame 3-6
MPU interrupts 3-5
MPU interrupts, hardware 3-7

N
NMI interrupt 3-6

nonmaskable interrupt 3-6
nonvolatile RAM 2-4; 5-6; B-1
NV-RAM 2-4; 5-6; B-1

O

operating temperature 2-1, 15-5
operation-unimplemented fault 3-22

P

P1 pin assignments (VMEbus) 6-21
P1 signal descriptions 6-2
P2 pin assignments (VMEbus, VSB) 6-22
P2 pin assignments (VMEbus, VSB) 7-5
P2 signal descriptions 7-1
P3 pin assignments (Centronics) 13-1
P4 pin assignments (SCSI) 12-3
P5 pin assignments (RS-232) 10-1
P6 connector 11-5
P6 pin assignments, Ethernet 11-5
parallel fault 4-3
parity error 3-7
parity, RAM 5-3
physical memory map 5-4
pin assignments, Centronics, P3 13-1
pin assignments, Ethernet, P6 11-5
pin assignments, front panel interface connector 8-2
pin assignments, RS-232, P5 10-1
pin assignments, SCSI, P4 12-3
pin assignments, VMEbus 6-21,23
pin assignments, VSB 7-5
Port A bit definition 9-3
port access 11-2
port addresses, CIO 9-5
Port B bit definition 9-2
Port C bit definition 9-1
power requirements 2-2, 15-5
PRCB 3-2, 5, 10
precautions 2-1
prereturn trace mode 3-20
privileged fault 4-3
processor control block 3-2, 5
processor reinitialization 3-5, 6
protection fault 4-3

R

RAM parity 5-3
RAM, bus 5-3
RAM, nonvolatile 2-4; 5-6; B-1
RAM, on-card 5-3
real-time clock 14-1
register address summary (CIO) 9-5
register address summary (SCSI) 12-2
register cache 3-22
Register Cache Configuration Word 3-22
register summary 3-22
reset switch 1-6,7
return instruction 3-20
return trace mode 3-20
returning boards 2-5
ROM 2-4, 14-1
RS-232 pin assignments 10-1
RTC accesses 14-1

S

SCSI connector, P4 12-3
SCSI register address summary 12-2
SCSI reset 12-2
sdma instruction 3-18, 4-3
service department 2-5
setup 2-1
setup, CRT terminal 2-3
sf0 register 3-10
sf1 register 3-10
shipping damage 2-3
short space 2-2; 6-14, 20
sizing memory 5-3
software interrupts, MPU 3-5,13
special function registers 3-9,11
standard space 2-2; 6-18
status latch 3-6, 7; 4-2
supervisor trace mode 3-20
sysctl instruction 3-12,13; 4-3
SYSFAIL bus signal 6-4
SYSFAIL control 6-20
system controller board 6-16

T

technical documents 1-7, 2-1
timers (CIO) 9-4

timing, memory 5-5
trace events 3-20
type fault 4-3
type mismatch 4-3

U

udma instruction 3-18; 4-3
unaligned words 4-3
unimplemented instruction 4-2
user LEDs 8-1

V

VMEbus interrupts 6-8
VME extended space mapping 9-3
VME short space mapping 9-2
VMEbus and local bus watchdog timers 6-20
VMEbus connector 6-21
VMEbus master interface 6-16
VMEbus slave interface 6-17
VMEbus system controller functions 6-15
VMX32bus 7-1
VSB connectors 6-21
VSB description 7-1
VSB operation 7-3
VSB termination 7-4
VSB terminators 7-5

W

wait states 5-5

Z

zero divide 4-3

Heurikon Corporation
8310 Excelsior Drive
Madison, WI 53717
Customer Support: 1-800-327-1251