EV80C196KB
Evaluation Board
User’s Manual
Copyright 1989, Intel Corporation

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patents licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication, or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the U.S. government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

376, Above, ActionMedia, BITBUS, Code Builder, DeskWare, Digital Studio, DVI, EtherExpress, ETOX, ExCA, FaxBACK, Grand Challenge, i, i287, i386, i387, i486, i487, i750, i860, i960, ICE, iLBX, Inboard, Intel, Intel287, Intel386, Intel387, Intel486, Intel487, Intel inside, Intellec, iPSC, iRMX, iSBC, iSBX, iWarp, LANprint, LANSelct, LANSel, LANSight, LANSspace, LANSpool, MAPNET, Matched, MCS, Media Mail, NetPort, NetSentry, OpenNET, Paragon, PRO750, ProSolver, READY-LAN, Reference Point, RMX/80, SatisFAXtion, SnapIN 386, Storage Broker, SugarCube, The Computer Inside, TokenExpress, Visual Edge, and WYPIWYF.

and the combinations of ICE, iCS, iRMX, iSBC, iSBX, iSX, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

IBM is a registered trademark and AT is a trademark of International Business Machines, Inc.

Microsoft, MS, MS-DOS, and XENIX are registered trademarks and Multiplan is a trademark of Microsoft Corporation.

Addional copies of this manual or other Intel Literature may be obtained from:

Intel Corporation, Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641
# CONTENTS

SECTION Description .................................................. PAGE

INTRODUCTION ........................................................................ 9
GETTING STARTED WITH THE EV80C196KB .................. 9
  Powering the Board ......................................................... 9
  Connecting to your PC .................................................... 9
  Starting the Host Software .............................................. 9
HARDWARE OVERVIEW OF THE EV80C196KB BOARD  10
  Block Diagram of the EV80C196KB Board ...................... 10
  Processor .......................................................................... 10
  Memory ............................................................................. 10
  Host Interface ..................................................................... 11
  Digital I/O .......................................................................... 11
  Analog Inputs ....................................................................... 11
  Decoding ............................................................................. 12
  Configuration Jumper Locations (Figure 3a) ....................... 14
  Memory Configuration Jumper Locations (Figure 3b) ........... 15
  Expansion Ports, Connectors and LEDs Locations (Figure 4) 16
  Host Serial Connector (Figure 5) ........................................ 17
  80C196KB Serial Port Connector (Figure 6) ....................... 17
  Analog Input Connector (Figure 7) ...................................... 18
  I/O Expansion Connector (Figure 8) .................................... 18
  Memory-I/O Expansion Connector (Figure 9) ....................... 19
  Power Supply Connector (Figure 10) ................................... 19
  25-pin to 9-pin Adapter (Figure 11) .................................... 20
INTRODUCTION TO iRISM-iECM96 SOFTWARE ............... 21
  Features ............................................................................. 21
  Restrictions ........................................................................ 22
OVERVIEW ........................................................................... 23
  Embedded Controller Monitor ........................................... 23
USER INTERFACE ............................................................... 24
  Background Information ................................................... 24
  Initiating and Terminating iECM-96 ................................. 25
  Default Base Commands ................................................... 28
FILE OPERATIONS .................................................................................... 29
  Loading and Saving Object Code .................................................... 29
  Other File Operations .................................................................... 30
PROGRAM CONTROL ................................................................................ 32
  Resetting the Target ..................................................................... 32
  Breakpoints ................................................................................. 32
  Program Execution ...................................................................... 33
  Program Stepping ........................................................................ 35
DISPLAYING AND MODIFYING PROGRAM VARIABLES ....................... 37
  Supported Data Types .................................................................. 37
  BYTE Commands ......................................................................... 38
  WORD Commands ........................................................................ 39
  DWORD Commands ....................................................................... 40
  REAL Commands ......................................................................... 41
  STACK Commands ........................................................................ 42
  STRING Commands ...................................................................... 42
  Processor Variables ...................................................................... 43
ASSEMBLY AND DISASSEMBLY .............................................................. 44
  Single Line Assembly Commands ................................................. 44
  Disassembly Commands ................................................................ 45
SYMBOL OPERATIONS ............................................................................ 46
RISM .......................................................................................................... 47
  RISM Variables ........................................................................... 47
  RISM Structure ........................................................................... 48
  Receiving Data from the Host ....................................................... 48
  Sending Data to the Host ............................................................. 48
  RISM Commands ......................................................................... 49

Schematics and Parts List .................................................................. Appendix A
Specific iRISM Information ............................................................... Appendix B
Listing of iRISM-196KB ................................................................. Appendix C
Timing Analysis ............................................................................. Appendix D
Programmable Logic Equations ......................................................... Appendix E
Standard Memory-I/O Connector ......................................................... Appendix F
Sample Session ............................................................................... Appendix G
Figure 1.

EV80C196KB Evaluation Board
INTRODUCTION

The EV80C196KB is a next-generation version of the EV80C196KA. The major changes are the use of a standard memory expansion bus compatible with the EV80C51FB and EV80C186 boards, and the removal of the card edge bus. Also, the HOLD/HLDA feature of the 80C196KB is supported. The EV80C196KB is designed to be a software evaluation tool for the ROMless 80C196KB 16-bit microcontroller. As such, ports 3 and 4 are not available for use as I/O ports unless offboard latches/buffers and decoding logic are used. All unreserved functions of the 80C196KB are available to you except for the Non-Maskable Interrupt (NMI), the TRAP instruction, and 512 bytes of address space. The Chip Configuration Byte is also used by the monitor, but most of its functions are provided by external logic.

GETTING STARTED WITH THE EV80C196KB

Powering up the Board

Power (+5, +/-12 Volts) must be connected to JP4 as shown on the board's silk-screen next to JP4 and in figure 10. Included with the board is a packet containing a Molex connector and crimp terminals for your convenience.

Power supply requirements for the EV80C196KB board are as follows:

+ 5 VDC +/- 5 % @ 280 mA (150 mA if LED's are disabled by removing jumper shunt E16)
+ 12 VDC +/- 20 % @ 15 mA
- 12 VDC +/- 20 % @ 15 mA

Upon power-up (or after a reset) the board goes through initializations and a shifting-pattern is displayed on the Port 1 LEDs when initialization has completed properly.

Connecting to your PC

Once you have applied power to the board, you need to connect P1 to a PC serial port. P1 is configured to interface pin-to-pin with a standard nine-pin AT-type serial connector (see figure 5 for pinout). Make certain that you use a cable providing all nine signals, as they are all needed for proper operation of the host interface. When you have connected the cable, you may observe that the 80C196KB is held in reset, and all the LEDs turn on. This is because one of the host signals is used to reset the part, and the signal is often in a reset condition prior to invoking the host software on your PC.

Note: if you have a 25-pin serial port it will be necessary to make a 25-pin to 9-pin adaptor (see figure 11 for details).

Starting the Host Software

After the you have made both connections to the board, you can invoke the host interface. Install the disk in drive A of your system. At the DOS prompt type "A:ECM96"<CR>. Your PC should eventually display the iECM-96 monitor screen. If you have problems please refer to the sub-section "Initiating and Terminating iECM-96" in the "USER INTERFACE" section of this manual. For further details on using the monitor, refer to the "USER INTERFACE" section.
HARDWARE OVERVIEW OF THE EV80C196KB BOARD

The EV80C196KB Microcontroller Evaluation board is delivered with an 80C196KB, 8 K-words and 8 K-bytes of user code/data memory, a UART for host communications and analog-input filtering with a precision voltage reference. Also included is programmable chip-select, bus-width and wait-state-counter logic which allows you to custom tailor the board to look like your own system. The board's physical dimensions are 6 1/2" x 7 3/4" with an overall height of 3/4". There are six main sections to the EV80C196KB board: Processor, Memory, Host Interface, Digital I/O, Analog Inputs and Decoding.

![Block Diagram of the 80C196KB Board](image)

**Figure 2.**

**Processor**

The Intel® 80C196KB is a 16-bit embedded microcontroller. Being a member of the MCS®-96 family, the 80C196KB uses the same powerful instruction set and the same architecture as the existing MCS-96 products. The 80C196KB is an enhanced CMOS version of the 8097BH. Its enhancements include up/down and capture modes on Timer2, multiplying speeds almost 3 times as fast, overall execution nearly twice as fast, Hold/Hold Acknowledge logic, and power-down and idle modes to save power. For more information, please refer to the 1989 "16-Bit Embedded Controller Handbook," Intel Corporation order number 270646-001 and the 80C196KB Datasheet order number 270634-001.

**Memory**

There are five 28-pin memory sockets provided on the EV80C196KB board: U1, U6, U8, U13 and U14. The sockets are designed to support byte-wide, JEDEC-pinout, memory devices of various types and sizes, i.e. 8K x 8 SRAM or 16K x 8 EPROM. U1 and U8, U6 and U13 are connected as two 16-bit memory banks and U14 is connected as an 8-bit memory bank.
See appendix B and appendix C for details on reserved areas of memory.

**Host Interface**

The PC host interface is accomplished with the 82510 UART (U20) connected to P1 via RS-232 drivers. The UART resides in the address range 1E00H - 1EFFH. Therefore, register 0 in the UART would be at address 1E00H of the 80C196KB, reg. 1 would be at 1E01H, reg. 2 would be at 1E02H, etc. up to reg. 7 at 1E07H. The registers will repeat again with reg. 0 at 1E08H due to the limited decoding granularity of the EPLD. Pin 12 of the UART, OUT1#, is used to tell the PC host when the 80C196KB is executing user code by a true level on the Ring Indicator input of the host serial port.

**Digital I/O**

With the exception of the NMI input, which is used by the Host Interface, all Digital I/O functions of the 80C196KB are available to you. There are eight LEDs on-board along with buffer/drivers which allow you to quickly observe the state of Port 1, HSO.0 and Port 2.5/PWM (see figure 4 or the schematics in appendix A for location). The TxD and RxD pins of the 80C196KB (Port 2.0 and Port 2.1) are connected to RS-232 buffer/drivers, which are connected to P2. All of the I/O signals are available on JP2 (see figure 8 or the schematics in appendix A for pinout).

**Note:** because RxD is connected to an RS-232 receiver (U19 pin 3) any attempt to use it as a digital input will result in a contention. If you would like to use it as a digital input, remove jumper shunt E19 to disconnect the receiver.

**Analog Inputs**

The Port 0 inputs of the 80C196KB double as both digital and analog inputs. The EV80C196KB board includes circuitry to make the analog inputs easier to use. A precision voltage source for Vref is provided on board (U3 and U4) which can be carefully adjusted by trimming RP1. Also, jumper shunt E4 allows Vref to be connected to Vcc instead of the output of U3. By removing E4 entirely, an off board reference can be connected to JP1. By removing jumper shunt E2, ANGND can be isolated from Vss. Protective clamping diodes are installed on each channel. RC networks are provided in sockets (to allow you to change the input impedance to match your application) on all of the analog input channels. If Port 0 is to be used
as a digital input, it is recommended that the capacitors be removed, and the resistors replaced with wires. For additional connection information refer to figure 7 or the schematics in appendix A. The ground and power planes beneath the analog circuitry (D1, D2, R3, C2, U3, U4, JP1 and the analog connections on the 80C196KB) are isolated from the digital power and ground planes of the board to keep noise from the analog inputs.

Decoding

The decoding logic on the EV80C196KB board serves three purposes: to provide Chip-Enable signals to memory and peripheral devices, to select the buswidth for the device(s) being accessed and to provide wait-states for slow devices. This section is provided in case you need to modify the memory configuration of the EV80C196KB board. It is not necessary to understand this section for normal usage of the board.

The heart of the decoding logic is U12, a 24-pin 5AC312 Intel EPLD or a C22V10 programmable logic array which is socketed to allow easy changes. For the sake of convenience it will be referred to as "the EPLD" throughout this text. The EPLD uses latched addresses A8-A15 along with CLKOUT, HLDA#, RESET# and STALE (STretched ALE) from the 80C196KB as decode inputs.

There are 4 enable outputs from the EPLD, all of which are low-level true, however only one should be true at a time to avoid bus contention. They are decoded from the address lines, and an internally-latched signal called MAP. MAP is cleared when the RESET# input is true, and set when the Monitor EPROMs are accessed in the address range 1D00H-1DFFH. MAP will always be set when the board is in the USER mode.

<table>
<thead>
<tr>
<th>Pin 21 = CE0</th>
<th>Enables memory in U1 and U8 (monitor EPROM as shipped)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE0</td>
<td>(ADDRESS RANGE 2000H - 27FF and NOT MAP)</td>
</tr>
<tr>
<td></td>
<td>or ADDRESS RANGE 0H - FFH</td>
</tr>
<tr>
<td></td>
<td>or ADDRESS RANGE 1D00H - 1DFFH</td>
</tr>
<tr>
<td>Pin 22 = CE1</td>
<td>Enables memory in U6 and U13 (user 16-bit ROMsim/RAM as shipped)</td>
</tr>
<tr>
<td>CE1</td>
<td>(ADDRESS RANGE 2000H - 27FFH and MAP)</td>
</tr>
<tr>
<td></td>
<td>or ADDRESS RANGE 2800H - 5FFFH</td>
</tr>
<tr>
<td>Pin 15 - CE2</td>
<td>Enables memory in U14 (user 8-bit ROMsim/RAM as shipped)</td>
</tr>
<tr>
<td>CE2</td>
<td>ADDRESS RANGE 6000H - 7FFFH</td>
</tr>
<tr>
<td>Pin 14 - CS510</td>
<td>Enables U20, the 82510 UART, which is used for host communications</td>
</tr>
<tr>
<td>CS510</td>
<td>ADDRESS RANGE 1E00H - 1EFFH</td>
</tr>
</tbody>
</table>
The BUSWIDTH output of the EPLD, pin 16, is fed into the buswidth pin of the 80C196KB. Therefore, it is driven low for accesses to 8-bit memory and high for accesses to 16-bit memory. As shipped, it goes low simultaneously with CE2 or CS510 as these are the only areas of memory mapped as 8-bit.

Programmed into the EPLD is a 3-bit wait-state machine clocked by the rising edge of CLKOUT from the 80C196KB. The transition sequence of the wait-state machine is controlled by the current state of the machine and the inputs to the EPLD (for further details see appendix E). While the bus of the 80C196KB is idle the wait-state machine is locked in state 0, which is called async_start. The conditions for leaving async_start are 1) ALE being asserted, 2) HLDA# not being asserted and 3) a value on A8-A15 requiring wait-states. Because the falling edge of ALE can occur before the next rising edge of CLKOUT can clock the wait-state machine, a signal called STALE (for Stretched ALE) is used. STALE does not go low until after the rising edge of CLKOUT.

During async_start, the output WAIT# from the EPLD is asserted asynchronously based upon a value on A8-A15 requiring wait-states. If no wait-states are required, WAIT# will not be asserted and the wait-state machine will remain in async_start. However, if one or more wait-states are needed WAIT# will be asserted and the wait_state machine will transition out of async_start on the next rising edge of CLKOUT. The next state entered depends on how many wait-states are needed. If only one is required the next state is remove_hold, where WAIT# is deasserted regardless of the inputs to the EPLD. If two wait-states are needed the next state is hold_2, where WAIT# is always asserted, then the state after that is remove_hold. The additional states, hold_3 - hold_7, work just like hold_2 with WAIT# always asserted. The wait_state machine will count through from hold_2 to hold_n to generate n wait-states before jumping to remove_hold to deassert WAIT#. The maximum number of wait-states is seven.

The previous paragraph described how the signal WAIT# is generated based on the rising edge of CLKOUT. However, the 80C196KB needs to have a valid signal on its READY input pin until the falling edge of CLKOUT. Therefore, it was necessary to clock WAIT# through a negative-edge-triggered-JK flip-flop (U15A) by the falling edge of CLKOUT to generate a signal called WAITN#. As in the EPLD, WAITN# is asserted asynchronously while ALE is high and WAIT# is asserted. After ALE goes low WAITN# will remain asserted until WAIT# is deasserted and the flip-flop is clocked. Besides the WAIT# signal, the WAITN# signal can be asserted by the USEREADY signal from the expansion bus. As shipped, the EPLD has the following configuration:

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Wait States</th>
<th>Enable Signal</th>
<th>Memory Region in User Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROMsim/RAM</td>
<td>0</td>
<td>CE1</td>
<td>2000H-5FFFH</td>
</tr>
<tr>
<td>ROMsim/RAM</td>
<td>0</td>
<td>CE2</td>
<td>6000H-7FFFH</td>
</tr>
<tr>
<td>Monitor EPROM</td>
<td>1</td>
<td>CE0</td>
<td>0-FFH, 1D00H-1DFFH</td>
</tr>
<tr>
<td>82510 UART</td>
<td>2</td>
<td>CS510</td>
<td>1E00H-1EFFH</td>
</tr>
<tr>
<td>Unimplemented</td>
<td>0</td>
<td>N/A</td>
<td>100H-1CFFH, C000H-FFFFH</td>
</tr>
<tr>
<td>Unimplemented</td>
<td>1</td>
<td>N/A</td>
<td>8000H - RFFFH</td>
</tr>
</tbody>
</table>
E4 Analog Voltage Reference Source
A-B AVref = VCC
B-C AVref = U3/U4
--- Avref from JP1

E2 Analog Ground Reference
A-B AVss = Vss
--- Avss from JP1

E3 2000H-3FFFH Memory Location
A-B External
B-C Internal

E7 82510 UART Interrupt Signal to 80C196KB
A-B UART Interrupt = EXINT/P2.2
B-C UART Interrupt = NMI

E16 LED Driver Enable
A-B Enabled
--- Disabled

E6 80C196KB CDE U5 pin 14
A-B CDE = Vss
B-C CDE = Vcc

E19 80C196KB RXD signal from P2
A-B RXD driven by U19 pin 3
--- RXD can be used by JP2

Figure 3a.
Configuration Jumper Locations
Figure 3b.
Memory Configuration Jumper Locations
Figure 4. Expansion Ports, Connectors and LEDs
### P1 Host Serial Connector
**DB-9S RS232**

<table>
<thead>
<tr>
<th>Pin Nos.</th>
<th>Host RS-232 Signal Name</th>
<th>Connection on Evaluation Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (AB)</td>
<td>SG Signal Ground</td>
<td>Digital Ground</td>
</tr>
<tr>
<td>4 (CD)</td>
<td>DTR Data Terminal Ready</td>
<td>INIT thru E20-C</td>
</tr>
<tr>
<td>3 (BA)</td>
<td>TxD Transmit Data</td>
<td>RxD of 82510</td>
</tr>
<tr>
<td>2 (BB)</td>
<td>RxD Receive Data</td>
<td>TxD of 82510</td>
</tr>
<tr>
<td>1 (CF)</td>
<td>DCD Data Carrier Detect</td>
<td>DTR P1-pin 4</td>
</tr>
</tbody>
</table>

---

**Figure 5.**

### P2 Serial Port Connector
**DB-9S RS232**

<table>
<thead>
<tr>
<th>Pin Nos.</th>
<th>Host RS-232 Signal Name</th>
<th>Connection on Evaluation Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (AB)</td>
<td>SG Signal Ground</td>
<td>Digital Ground</td>
</tr>
<tr>
<td>4 (CD)</td>
<td>DTR Data Terminal Ready</td>
<td>INIT thru E20-A</td>
</tr>
<tr>
<td>3 (BA)</td>
<td>TxD Transmit Data</td>
<td>RxD of 82516KB</td>
</tr>
<tr>
<td>2 (BB)</td>
<td>RxD Receive Data</td>
<td>TxD of 82516KB</td>
</tr>
<tr>
<td>1 (CF)</td>
<td>DCD Data Carrier Detect</td>
<td>DTR P2-pin 4</td>
</tr>
</tbody>
</table>

---

**Figure 6.**
JP1 Analog Input Connector
2x13 Pin MOLEX 39-51-2604 or Equiv.

JP2 I/O Expansion Connector
2x25 Pin MOLEX 39-51-5004 or Equiv.

Figure 7.

Figure 8.
JP3 Memory-I/O Expansion Connector
2x30 Pin MOLEX 39-51-6004 or Equiv.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vcc</td>
<td>1 - Vcc</td>
</tr>
<tr>
<td>A0</td>
<td>2 - D0 Bi-directional</td>
</tr>
<tr>
<td>A1</td>
<td>3 - D1 Bi-directional</td>
</tr>
<tr>
<td>A2</td>
<td>4 - D2 Bi-directional</td>
</tr>
<tr>
<td>A3</td>
<td>5 - D3 Bi-directional</td>
</tr>
<tr>
<td>A4</td>
<td>6 - D4 Bi-directional</td>
</tr>
<tr>
<td>A5</td>
<td>7 - D5 Bi-directional</td>
</tr>
<tr>
<td>A6</td>
<td>8 - D6 Bi-directional</td>
</tr>
<tr>
<td>A7</td>
<td>9 - D7 Bi-directional</td>
</tr>
<tr>
<td>Vss</td>
<td>10 - Vss</td>
</tr>
<tr>
<td>A8</td>
<td>11 - D8 Bi-directional</td>
</tr>
<tr>
<td>A9</td>
<td>12 - D9 Bi-directional</td>
</tr>
<tr>
<td>A10</td>
<td>13 - D10 Bi-directional</td>
</tr>
<tr>
<td>A11</td>
<td>14 - D11 Bi-directional</td>
</tr>
<tr>
<td>A12</td>
<td>15 - D12 Bi-directional</td>
</tr>
<tr>
<td>A13</td>
<td>16 - D13 Bi-directional</td>
</tr>
<tr>
<td>A14</td>
<td>17 - D14 Bi-directional</td>
</tr>
<tr>
<td>A15</td>
<td>18 - D15 Bi-directional</td>
</tr>
<tr>
<td>Vss</td>
<td>19 - Vss</td>
</tr>
<tr>
<td>CLKOUT</td>
<td>20 - Vss</td>
</tr>
<tr>
<td>RD#</td>
<td>21 - WR# Output</td>
</tr>
<tr>
<td>BREQ#</td>
<td>22 - BHE# Output</td>
</tr>
<tr>
<td>ALE</td>
<td>23 - UserReady Input</td>
</tr>
<tr>
<td>NMI</td>
<td>24 - INST Output</td>
</tr>
<tr>
<td>RESET#</td>
<td>25 - P2.2/EXINT Bi-directional</td>
</tr>
<tr>
<td>No Connection</td>
<td>26 - No Connection</td>
</tr>
<tr>
<td>HLD#</td>
<td>27 - HOLD# Input</td>
</tr>
<tr>
<td>-12VDC</td>
<td>28 - +12VDC</td>
</tr>
<tr>
<td>Vss</td>
<td>29 - Vss</td>
</tr>
<tr>
<td>Vcc</td>
<td>30 - Vcc</td>
</tr>
</tbody>
</table>

Figure 9.

JP4 Power Supply Connector
4 Pin MOLEX 26-03-3041 or Equiv.

Figure 10.
Figure 11.
25-pin-to-9-pin Adapter
INTRODUCTION TO iRISM-IECM SOFTWARE

The EV80C196KB board uses an Embedded Controller Monitor (ECM) written for the MCS-96 family of 16-bit microcontrollers. This monitor supports basic debug facilities (LOAD, GO, STEP etc.) in the user's target system. The ECM is broken into two independent programs, one of these executes in the EV80C196KB (iRISM-96KB) and the other executes in a IBM PC or BIOS compatible clone(iECM-96). These two programs communicate via an asynchronous serial channel using a binary protocol defined specifically for this application.

The partitioning of the ECM into two separate programs supports a number of goals in the development of this system:

- The system is easy to adapt to a new target because the code which runs in the target is very simple and small.

- The feature set of the user interface is not limited by the resources of the target since the user interface is implemented in the host PC.

- Concurrent operation of the ECM and the target system was easily achieved. This allows you to interrogate and (carefully) modify the state of the target system while it is running.

This manual section describes the user interface provided by the iECM-96, the interface between this PC resident software and the target resident software, and the structure of the software in the target. Appendix B lists the resources of the 80C196KB that are reserved for this RISM implementation. Appendix C is the listing for the iRISM software which runs in the 80C196KB on this board. It uses an Intel 82510 UART for host communications.

The iECM-96 was designed and implemented by Intel to support users of the MCS-96 architecture, and is placed in the public domain with no restrictions or warranties of any kind.

Features

- Host system is an IBM PC AT, PC XT, or BIOS-compatible clone. (Interfaces via COM1 or COM2 at 9600 baud.)

- Sixteen software execution breakpoints

- Concurrent interrogation of target memory and registers

- Supports BYTE, CHARACTER, WORD, STRING, DOUBLE-WORD and FPAL-96 REAL variable types.

- Single-Line Assembler/Disassembler

- Symbolics compatible with Intel's OMF debug records

- Supports LOAD, SAVE, LIST, LOG, and command INCLUDE files.
Restrictions

Two words of user stack are reserved for use by the iRISM-96 software. Other memory and/or registers in the target memory will be used by the iRISM-96 software. The exact number and location of this memory is implementation dependent. See appendix B or C for further information.

An asynchronous serial port capable of operation at 9600 baud must be available in the target system. The RISM described in this document uses an Intel 82510 UART. This version also uses the NMI (Non-Maskable Interrupt) to signal that a received data character is available.

The TRAP instruction is reserved.

Breakpoints and program stepping will not operate if the user's code is in EPROM or other nonchangeable memory.
OVERVIEW

Embedded Controller Monitor (ECM)

An ECM (Embedded Controller Monitor) provides basic debug capability and is installed in your target system. Capabilities include loading object files into system RAM, examining and modifying variables, executing code, and stepping through code. In the past, most of these monitors have been configured to run with a standard "dumb" CRT with some form of auxiliary port for loading and saving object code from a host system. It is now common for a personal computer to act as the host for program translation and also emulate a dumb CRT during user interaction with the ECM. The ECM developed for the MCS-96 family makes the assumption that the user interface will always be a personal computer; no provision is made for interface to a dumb CRT. By making this assumption it is possible to reduce the size and complexity of the code that must be installed in the target system. A term has been coined for this code resident in the target -- RISM. The term RISM stands for Reduced Instruction Set Monitor and is an obvious takeoff of the term RISC (Reduced Instruction Set Computer) used to describe a class of computer architectures. The RISM consists of about 300 bytes of MCS-96 code which provide primitive operations. Software running in the host uses the RISM commands to provide a complete user interface to the target system. The advantage of this approach is that the ECM can be readily adapted to different target systems and requires only a small part of the available target memory space. The disadvantage is that the user interface must be provided by a personal computer.

The structure of the RISM is a short section of initialization code and an interrupt service routine (ISR) that processes interrupts from the host system. The RISM ISR consists of a short prologue and then a case-jump to one of 20 to 25 command executors. These executors are simple and short; the flow though the entire ISR (including the prologue) is 15-20 instructions. The serial communication occurs at 9600 baud, which limits the frequency of these interrupts to 1 Khz. In the worst case the EV80C196KB board will be slowed by the execution of a fairly short RISM ISR every millisecond while executing user code. It is possible to operate the EV80C196KB board so that no real-time is lost to the iECM-96 unless the user is actively interrogating the target. (See the section "Initiating and Terminating the iECM-96" and the description of the RISM REPORT_STATUS command for details on this).
USER INTERFACE

The user interface to the iECM-96 supports commands to initiate and configure the ECM-96, perform I/O operations involving DOS files, execute user programs, and interrogate variables in the target system. Interrogation can be done in a number of formats and in most cases can be done concurrently with user code execution. A single line assembler and disassembler are also provided.

Note: on the disk included with the EV80C196KB is a file called DEMO.LOG. DEMO.LOG is a sample iECM-96 session for you to invoke and become more familiar with the features of iECM-96. Appendix G is a printout of DEMO.LST which was created by turning on the list feature and invoking DEMO.LOG by typing "Include demo.log"<CR> at the iECM-96 "*" prompt.

Background Information

Numeric and Symbolic Input
The command parser used by the iECM-96 software requires that numeric inputs always start with the digits 0-9. If hexadecimal numbers are entered which start with A-F they must be preceded by a "0". For example, enter "0AA55" instead of "AA55". This requirement is similar to ASM-96. If symbolic information has been downloaded as part of an object file (see "Loading and Saving Object Code") then you can enter a valid symbol name whenever a number is expected. The symbol name must be preceded by a period (".") so that the parser knows to try searching the symbol table. If the symbol is ambiguous then it will not be accepted by the parser. The probability of ambiguous references can be reduced by specifying the module name along with the symbol name. The module name must be preceded with a colon (" :"). If a variable TEMP is declared both in MODULE1 and in MODULE2, then a reference to the TEMP declared by MODULE1 would be ":MODULE1.TEMP". PLM-96 or C-96 line numbers can be called out by a pound sign ("#") followed by the line number.

Symbolic Output
The symbolic output routines, in general, deal only with address information. They will not try to convert data values into symbolic form. When the symbol table is searched for a symbol name to associate with a given value the routines also perform type checking. If one, and only one, symbol matches both the type and value of the address being displayed then the output routines will display the symbol name along with the numeric value of the address. If more than one label has been assigned to a given address then the symbolic output routines will ignore all of them. The exception to this rule occurs when the disassembler finds multiple labels assigned to a given code address. The disassembler will display all the known symbolic labels attached to a code address.

If the symbols table gets very large the symbolic output routines will become painfully slow, particularly on an 8088 based PC. This problem can be avoided by using modular programming and translating a subset of the modules in the debug mode. Another alternative is to use the "SYMBOLS OFF" command to suppress symbolic output. Symbolic input is not affected by this command.
Controlling Lengthy Commands
Most of the commands supported by iECM-96 appear to complete without delay. Some commands (e.g. displaying or filling a large area of memory) take an appreciable length of time to complete. In general these commands can be aborted by entering a CARRIAGE-RETURN. Those commands which display a large amount of information can be paused by hitting the SPACE bar. After you have checked the data currently on the screen you can depress the SPACE bar again to resume the output.

Aborting from iECM-96
Entering a control-C will cause the iECM-96 to close any open files and return to DOS.

Initiating and Terminating iECM-96
This section describes the commands for invoking iECM-96 from DOS and exiting back to DOS.

ECM96
This command, entered at the DOS prompt, loads the iECM-96 software and executes it. Several options are available with this command. Option strings always start with a hyphen ("-"") and can be entered in upper or lower case. The operation of these options is described below. Any or all of these options can be entered in any order, if the options are contradictory then the actual option accepted is the last one entered.

-COM2, -COM1
These options tell the iECM-96 software which serial communication port is to be used. If neither of these options is entered then COM1 will be used as a default. If iECM-96 detects valid CTS (Clear To Send) and DSR (Data Set Ready) signals from the appropriate COM port it will sign on and display a command prompt. If the target is stopped the command prompt will be an asterisk ("*"). If the target is already running the prompt will be a greater-than sign (">").

-DIAG
If CTS or DSR are not present, iECM-96 will complain about it and ask if you want to proceed or exit. It is possible, but not likely, that iECM-96 will operate properly even after complaining. It is more likely that there is a problem with the serial port or the cabling which will prevent proper operation. If the problem is not obvious (e.g. disconnected cable or no power to the target hardware) then the -DIAG invocation option can be used to help isolate the problem. The -DIAG option puts the iECM-96 system in a special mode which allows many tests to be used to find interfacing problems, or target bugs.

The diagnostic mode is intended to support debugging of boards which use the iECM-96. It can be particularly useful in systems which have multiple address decoding modes, such as the EV80C196KB. Upon reset this board has EPROM at location 2080H, the address where the 80C196KB starts execution. After executing some initialization code, the board can change the address decoding so that ROMsim/RAM is available in the partition which contains 2080H and the RISM is relocated to another area. This allows you to download code which is designed to operate in the on-chip ROM of MCS-96 family parts (2000H - 3FFFH). The diagnostic mode allows the use of diagnostic routines which disappear from memory space
when the RAM is mapped into the system. It also provides a simple routine to check the communications interface between the host and the target.

In the EV80C196KB board, there is a serial port loop-back mode which allows debugging the host/board interface. Upon reset the board is in the echo mode. Until it receives an ASCII slash ("/") or reverse-slash ("\") it will increment every character it receives from the host and send the incremented value back to the host. It will also display the binary code of the character the board received on the Port 1 LED's. If a reverse slash is received by the RISM it will leave the echo mode (set USER_MAP flag true), remap memory and start normal operation. If a slash is received it will stop echoing incremented received data and start responding to RISM commands with the diagnostic flag set. In this mode there are diagnostic routines resident in EPROM which are useful for debugging the board. Initially after invoking the diagnostic mode, the Program Counter points to the beginning of a RAM test at 2200H. See the source code listing in appendix C for further details.

Note: The target hardware will have to be reset before using the DIAG command option.

Note: When executing diagnostic routines from EPROM, certain commands such as Breakpoints and Stepping will not work as they need to modify the code to work properly.

When the host software is invoked in the diagnostic mode it will tell you to enter characters on the keyboard. These characters will be sent to the target and the response from the target will be displayed on screen. This is a simple confidence check on the serial communication channel. You are told to enter a slash or reverse-slash to terminate this mode and proceed in either the diagnostic mode or the normal user's mode. If the user interface is invoked without the -DIAG option it will immediately transmit a reverse-slash which should put the target in the normal mode. Systems which do not implement the diagnostic mode will load the reverse-slash into the RISM_DATA register where it will languish till more useful data is sent by the host.

-8096, -8096BH, -C196KB
These three options control the single line assembler and the disassembler in the iECM-96. If the 8096 (8x9x-90) or 8096BH (8x9xBH) options are selected then the additional instructions in the 80C196KB will be considered invalid for both the single line assembler and the disassembler. If none of these options are selected then the iECM-96 will default to C196KB mode.

-NOTYPES
This option will cause the object file loader to ignore type definition records in the object module. If this is invoked then the symbolic I/O routines will only recognize basic data types such as BYTES, WORDS, and LONGS. More complex data types such as PLM arrays and structures will not be recognized. This option is included because early versions of the host software got confused while loading certain type definition records generated by C-96. These problems have been fixed but the option was left in case similar problems remain.
These two options control how the host software detects whether or not the user's code is running. If poll mode is selected then the host will periodically poll the target with a REPORT_STATUS command. This takes no additional hardware but forces the target to waste instruction cycles responding to the poll. The signaling mode avoids this overhead but requires that the target set the Ring Indicator modem control line whenever it is running user code. The user interface will then check this line before it issues a REPORT_STATUS command. If neither of these options is selected then the signal mode is selected as a default. On the EV80C196KB the OUT1# pin of the 82510 is used to generate this running signal. Therefore, the signal mode is recommend.

RESET SYSTEM
RES SYSTEM
RESET
RES

This command and its abbreviations will reset the entire target hardware system if the target system is implemented to support this operation. On the EV80C196KB jumper shunt E20 must be installed from B to C for this command to work properly. This command operates by dropping the DTR modem control line. This comes into the target as DSR. After dropping DTR the iECM-96 software will wait about 1 second to allow the target to complete its initialization routines. The iECM-96 will politely warn of this time delay and then ignore the user until it expires. Unless special precautions are taken in the design of the target system, any data in RAM (including downloaded object code) may be corrupted by the reset. On the EV80C196KB, the RAM contents should not be affected by a RESET.

DOS
This command enables you to temporarily leave iECM-96 and return to DOS. Once you have suspended iECM, you may perform other functions in DOS, including using other software programs, such as ASM-96, as long as there is sufficient memory to do so.

To reenter iECM, type exit at the DOS prompt. iECM will return with all conditions in effect at the time it was suspended.

QUIT
This command will close any files that iECM-96 has opened and exit to DOS. Note that this command can be used even if the target is running. iECM-96 sets the selected COM port to 9600 baud, 8 bits, no parity, and one STOP bit. The port will be left in this state by iECM-96 when control is returned to DOS.
Default Base Commands

These commands are used to set the default base for numeric input and output. The valid bases are: 16 (hexadecimal), 10 (decimal), and 8 (octal). The default base is used to display variables. It is not used to display addresses (which are displayed in hexadecimal) or breakpoint numbers (which are displayed in decimal). The default base is also used to enter numbers into the command parser, but it is possible to override the default base during input by adding a character at the end of the number which forces the appropriate base to be used. The override characters are H (or h) for hexadecimal, T (or t) for decimal, and O (or o) for octal. The override character must appear immediately following the last digit of the number with no intervening space.

BASE
This command will display the current default base.

BASE=<valid_base>
This command will set the current default base to <valid_base>. When entering this command it is advisable to use an override character to select the new default base:

BASE=10O ; selects octal
BASE=10T ; selects decimal
BASE=10H ; selects hexadecimal

This avoids confusion when changing bases. As an example of the confusion which is avoided, consider the following commands entered while the base is hexadecimal. The command:

BASE=10

will leave the default base as hexadecimal and the command:

BASE=16

will result in an error because 16H (22T) is not a valid base. The command:

BASE=0A

will select decimal as the default base but it is cleaner and simpler to use the override character:

BASE=10T

This works independently of the current default base and leaves a useful record in log or list files which may be open.
FILE OPERATIONS

iECM-96 uses files in the host system to load and save object code, enter predefined strings of commands, to keep a log of commands that are entered by the user, and to keep a record of an entire debug session which includes both the characters entered by the user and the response generated by iECM-96 on the host screen. The commands which operate with files are described in the following sections.

Loading and Saving Object Code

iECM-96 accepts object files which are generated by Intel's development tools. iECM-96 will not accept files which contain unresolved externals or files which contain relocatable records. These files must be passed through RL-96 in order to resolve the externals and/or absolutely locate the relocatable segments. iECM-96 will also not accept HEX format files. There is a utility on the disk (HEXOBJ.EXE) for converting HEX format files to Intel object format files loadable by iECM-96. While still in DOS type "HEXOBJ <filename>.hex <filename>.obj"<CR> to convert <filename>.hex to a usable format for iECM-96. HEXOBJ does not attempt to convert any symbolic information contained in the HEX file. The iECM-96 commands which operate on object files are:

LOAD <filename>
LOADSYM <filename>
SAVE <addr> TO <addr> IN <filename>

The metasymbol <filename> means that a valid MS-DOS file name must be entered in that position of the command string.

LOAD <filename>
This command loads the content records of the object file <filename> into the target memory and loads any associated symbolic information into a symbol table maintained in the host system’s memory.

LOADSYM <filename>
This command loads the symbolic information from <filename> into the symbol table maintained in the host system but does not load the content records into the target’s memory. This command is useful when you have left a debug session with the target still running a program that has been loaded. At a later time you can re-invoke iECM-96 and interrogate the running program without stopping it. The LOADSYM command allows the use of the symbolic information contained in the object file without reloading the content records. (Content records cannot be loaded while the target is running).

SAVE <addr> TO <addr> IN <filename>
This command saves a region of memory as an object file which can be reloaded into the target memory at some latter time. No attempt is made to include any symbolic information which may have been in the symbol table maintained in the host system.
Other File Operations

In addition to object files, the iECM-96 makes use of include files, log files, and list files. Include files contain commands to be executed by iECM-96, they must contain the exact sequence of ASCII characters that you would enter from the keyboard to execute the command. Include files can be tedious to generate with a text editor so iECM-96 can generate log files in which are stored characters entered by the user. The intent is that log files be used later as include files to recreate command sequences. List files keep a running record of both commands entered by the user and of the response generated by iECM-96. Comments can be included in list and log files to make them easier to understand. A comment starts with a semicolon (';') and ends with a carriage return or ESC. The semicolon is considered to be part of the comment but not the CR or ESC. The command parser will ignore comments but will put them in the list and log files.

Note: on the software disk included with the EV80C196KB is a file called DEMO.LOG. DEMO.LOG is a sample iECM-96 session for you to invoke and become more familiar with the features of iECM-96. Appendix G is a printout of DEMO.LST which was created by turning on the list feature and invoking DEMO.LOG by typing "include demo.log"<CR> at the iECM-96 "**" prompt.

The list and log files commands allow for default filenames and allow either overwriting existing data in the file or appending data at the end of the file. This allows you to gather list and log data in the default files which avoids the creation and management of a large number of separate files. Log and list files are stamped with the date and time whenever they are opened to make it easier to use this capability and then go back and sort out the data from several debug sessions with a text editor.

The commands involved in include, log, and list operations are:

```
INCLUDE <filename>
PAUSE LIST
LIST <filename>
LOG
LOG <filename>
LISTOFF
LISTON
LOGOFF
LOGON
```

Three of these commands require you to supply a valid file name, the rest use the appropriate file name that has already been entered.

INCLUDE <filename>
This command will attempt to open <filename> as a read only file. If the file can be opened then the command parser will take commands from that file until the end of the file is reached. The include file will then be closed. Only one include file will be opened at a time.
PAUSE
This command is documented in this section because it is intended to be used as part of INCLUDE files. It is not really a file oriented command itself. When this command is entered the iECM-96 will stop parsing commands until a SPACE character is entered from the keyboard (it can't come from an INCLUDE file). This provides a method of pausing in the middle of an INCLUDE file operation until you have a chance to see what's going on and acknowledge the pause condition by depressing the SPACE bar.

LIST
This command behaves like the LIST <filename> command described below except that it uses the last <filename> that was entered as part of a LIST <filename> command. If no such command has been entered then the default filename "LIST.ECM" will be used.

LIST <filename>
This command will attempt to open <filename> as a writable file. If a file with <filename> already exists then iECM-96 will ask if the file is to be overwritten or if the new data should be appended to the end of the existing file. It will then open the file and stamp it with the current date and time from the system clock. After this, commands entered by the user and the responses generated by iECM-96 will be recorded in the file.

LOG
This command behaves like the LOG <filename> command described below except that it uses the last <filename> that was entered as part of a LOG <filename> command. If no such command has been entered then the default filename "LOG.ECM" will be used.

LOG <filename>
This command will attempt to open <filename> as a writable file. If a file with <filename> already exists then iECM-96 will ask if the file is to be overwritten or if the new data should be appended to the end of the file. It will then open the file and stamp it with the current date and time. After this, commands entered by the user will be recorded in the file. Note that this file may contain nonprintable characters (e.g. ESC).

LISTOFF and LISTON
The LISTOFF closes a LIST file that has been specified by the LIST command. This stops new list information from being recorded. The LISTON re-opens the list file in the append mode so that recording can start again. LISTON also stamps the list file with the current date and time from the system clock.

LOGOFF and LOGON
The LOGOFF closes a log file that has been specified by the LOG command. This stops new list information from being recorded. The LOGON re-opens the log file in the append mode so that recording can start again. LOGON also stamps the list file with the current date and time from the system clock.
PROGRAM CONTROL

Commands which control program execution allow you to reset the processor, set execution breakpoints, start execution, stop execution, step, and super step. The commands will be grouped by their major function for the sake of discussion.

Resetting the Target

The processor can be reset by executing the iECM-96 command:

RESET CHIP

This command physically resets the processor by setting the RISM_DATA register to 0XXXX0001 and issuing a MONITOR_ESC RISM command which will cause the target to perform a RST instruction.

Breakpoints

iECM-96 provides sixteen program execution breakpoints. If a given breakpoint is inactive it is set to zero, if it is active then it is set to the address of the first byte of an instruction. Breakpoints set to addresses which are not the first byte of an instruction will cause unpredictable errors in the execution of the user's code. When execution is started iECM-96 saves the user code byte at any active breakpoint and substitutes a TRAP instruction for that byte. Executing a TRAP instruction will cause the iECM-96 to restore the user code bytes where the TRAP instructions were substituted and then decrement the user's program counter so that it points at the original instruction. The user's program will appear to stop execution immediately before executing the instruction with a breakpoint set on it. All the TRAPs will be removed from the user's code and the original code restored.

Note: Most monitor programs similar to iECM-96 display a message on the console when a break occurs (e.g. "Program break at 1234H"). This is not done in iECM-96 because the system supports concurrent interrogation of the target which the user's code is running; it is possible (perhaps probable) that the break will occur while you are in the middle of displaying or modifying the state of the target. Any special break message would have to interrupt the execution of the command. Because of this the iECM-96 does not output a special break message. You have two ways to find out that a break occurred:

1). The prompt will change from a greater-than ">" to an asterisk ("*").

2). The status of the processor shown in the "control panel" at the top of the console screen will change from "running" to "stopped".

Commands which set the breakpoint array are:

BR
BR [ <bp_number> ]
BR [ <bp_number> ] = <code_addr>

The square brackets in the latter two commands are part of the command syntax and must be entered by the user, the angle brackets are part of the "meta" language used to describe the syntax. Breakpoints can be displayed while your code is running but they cannot be modified.
NOTE: BR[0] and BR[1] can also be set by the GO command by using the TILL clause; all of the breakpoints will be cleared by the GO command if the FOREVER clause is used.

BR
This command will display all of the active breakpoints (i.e. those not set to zero). You will also be informed if no breakpoints are active.

BR [ <bp_number> ]
This command will display the setting of the selected breakpoint and wait for input from you. If you enter a carriage-return the command will terminate. If you enter an ESC the next sequential breakpoint will be displayed. If you enter a numeric value then the selected breakpoint will be loaded with the value and the iECM-96 will again wait for input. At this point you can enter either a CARRIAGE-RETURN or an ESC. As before, the ESC will cause the iECM-96 to display the next breakpoint and the CARRIAGE-RETURN will terminate the command. This command will wrap around from the last breakpoint (15t) to the first breakpoint (0).

BR [ <bp_number> ] = <code_addr>
This command sets the specific breakpoint specified by <bp_number> to the value <code_addr>.

Program Execution

These commands start and stop execution of user code. The commands provided are:

GO
GO FOREVER
GO FROM <code_addr>
GO FROM <code_addr> FOREVER
GO FROM <code_addr> TILL <code_addr>
GO FROM <code_addr> TILL <code_addr> OR <code_addr>
GO TILL <code_addr>
GO TILL <code_addr> OR <code_addr>
HALT

If a GO with breakpoint command is entered, the user code bytes at the breakpoints will be saved and TRAPs will be installed. When a breakpoint is reached the user's software will stop before the instruction which caused the breakpoint and the iECM-96 software will restore the original user code. Note that this is different from the operation of iSBE-96 (and most ICE modules) which stop just after the instruction executes. A problem associated with stopping before the break instruction executes is that subsequent GO commands may run into the breakpoint before any user code is executed. The iECM-96 avoids this problem by skipping the setting of any breakpoints set on the instruction that the current PC points to. If this happens to remove the last breakpoint set then you will be warned but the GO will still execute with no breakpoints enabled. IF this happens you can use the HALT command to stop the program.
None of the GO commands can be executed while the user's code is already running; the HALT command cannot be executed if the user's code is not running. The GO commands which set breakpoints use BP[0] and possibly BP[1]. Any break value already in one of these breakpoints will be overwritten and destroyed by these GO commands. If possible the user should reserve the first two breakpoints for use by the GO commands and set the remaining breakpoints (if required) explicitly with the BR commands.

GO
This command starts execution of the user's code using the current value of user's PC and the current breakpoint array.

GO FOREVER
This command clears the breakpoint array and starts execution at the current value of the user's PC.

GO FROM <code_addr>
This command loads the user's PC with <code_addr> and starts execution of the user's code using the current breakpoint array.

GO FROM <code_addr> FOREVER
This command loads the user's PC with <code_addr>, clears the breakpoint array, and starts execution of the user's code.

GO FROM <code_addr> TILL <code_addr>
This command loads the user's PC with the <code_addr> which follows the FROM keyword, sets the first breakpoint (BP[0]) to the <code_addr> which follows the TILL keyword, and then starts execution of the user's code.

GO FROM <code_addr> TILL <code_addr> OR <code_addr>
This command acts like the previous command except that it also sets the second breakpoint (BP[1]) to the <code_addr> which follows the OR keyword.

GO TILL <code_addr>
This command sets the first breakpoint (BP[0]) to <code_addr> and then starts the execution of user code using the current setting of the user's PC and the breakpoint array.

GO TILL <code_addr> OR <code_addr>
This command acts like the previous command except that it also sets the second breakpoint (BP[1]) to the <code_addr> which follows the OR keyword.

HALT
This command stops execution of user code by forcing the processor to execute a jump to self instruction in a reserved location.
Program Stepping

These commands allow stepping through programs one instruction at a time. Between instructions the iECM-96 commands can be used to check the state of the variables changed by the instruction to ensure that the program is operating properly. Stepping through code allows a far more detailed look at what is going on in the program. The price that is paid for this detail is that stepping does not occur in real time; this makes it difficult or perhaps impossible to use on code that is tied to real time events.

Stepping while interrupts are enabled would be confusing since interrupt service routines will be stepped through as well as sequential code. iECM-96 avoids this problem by artificially locking out interrupts while stepping, ignoring the state of the interrupt enable (El) or interrupt mask.

Super-Stepping is similar to stepping except that interrupts are not artificially suppressed. Also, an interrupt service routine or a subroutine call (and the body of the subroutine that is called) is treated as one indivisible instruction by the super-step command. This allows the user to ignore the details of subroutines and interrupt service routines while checking out code. Every time an instruction is "super-stepped" all the service routines associated with enabled pending interrupts will be executed. This may allow limited stepping through code while operating in a concurrent environment but the system will not operate in real time. A better approach is to use the GO command to execute to a specified breakpoint and then step through the code being tested looking for proper operation.

iECM-96 implements the step operation by using the TRAP instruction. To step over a given instruction iECM-96 determines all the possible subsequent instructions and places TRAPS at these locations. After doing this it allows the user's program to execute until it runs into one of these TRAPS and then restores all of the user code bytes which were overwritten with TRAPS. If iECM-96 is to step over a conditional branch, two possible subsequent instructions exist in the sequential code of the program. Any other instruction can only have one "next" instruction. A TRAP is also set at location 2080H in case the target is reset during the step.

Super-stepping is accomplished by setting TRAPS like the STEP except for CALL instructions which are treated as a special case. During a STEP the iECM-96 will put the TRAP at the target address of a call; during a super-step the TRAP will be placed at the instruction following the CALL. Interrupts are suppressed during STEP (not SS) operations by saving the user's El bit, clearing it before the STEP occurs, and then restoring it. In order to make sure the instruction which is executed does not modify the El bit, several instructions (PUSHF, POPF, PUSHA, POPA, DI, EI) are simulated by the iECM-96 software rather than being executed by the target processor. The 80C196KB instruction IDLPD is also simulated during STEP to prevent the target from locking up. The simulation treats the IDLPD as a two byte NO-OP. Note that the simulation of instructions only occurs during STEP operations. During a GO or SS command all instructions are executed by the target.
The iECM-96 commands which implement step operations are:

- **STEP**
  - **STEP** <count>
  - **STEP FROM** <code_addr>
  - **STEP FROM** <code_addr> <count>
- **SS**
  - **SS** <count>
  - **SS FROM** <code_addr>
  - **SS FROM** <code_addr> <count>

Aside from the style of the actual step operation, the SS and STEP commands behave the same. They will be described together and will be called single-stepping.

- **{STEP | SS}**
  - This command single-steps one time.

- **{STEP | SS}** <count>
  - This command single-steps <count> times.

- **{STEP | SS} FROM** <code_addr>
  - This command loads the user's pc (PC) with <code_addr> and then single-steps one time.

- **{STEP | SS} FROM** <code_addr> <count>
  - This command loads the user's pc (PC) with <code_addr> and then single-steps <count> times.
DISPLAYING AND MODIFYING PROGRAM VARIABLES

iECM-96 provides commands to display and modify program variables in several formats. In addition to simple variables such as bytes and words, more complicated variables such as reals and character strings are supported. iECM-96 commands allow variables to be displayed or initialized either individually or as regions of memory which contain variables of the given type.

Supported Data Types

BYTE
A BYTE is an eight-bit variable. No alignment rules are enforced for BYTE variables.

CHAR
A CHAR is a special case of a BYTE. CHAR variables are displayed as ASCII characters.

WORD
A WORD is a 16-bit variable. The address of a WORD is the address of its least significant byte. A WORD must start at an even byte address.

DWORD
A DWORD is a 32-bit variable. The address of a DWORD is the address of its least significant byte. A DWORD must always start at an even byte address. If a DWORD variable is to be accessed as a register by an 8096 instruction then a more restrictive alignment rule is enforced: it must start at an address which is evenly divisible by 4. This more restrictive alignment rule will only apply to iECM-96 commands when using the single line assembler.

REAL
A REAL is a 32-bit binary floating point number which conforms to the FPAL96 definition. The 32 bits contain a sign bit, an 8-bit exponent field, and a 23-bit fraction field. iECM-96 commands use standard scientific notation to deal with REAL numbers. Note that the FPAL96 has special representations for ±infinity and for NaN's (Not a Number—used to signal error conditions) if iECM-96 detects one of these special values it will output an appropriate text string instead of trying to display the value in scientific notation.

STACK
A STACK variable is a 16-bit variable which resides in the system stack. The addresses of stack variables (<stack_addr>) are taken to be relative to the current stack pointer and must be word aligned.

STRING
A STRING is a sequence of ASCII characters which are terminated by the NUL character. The ASCII character NUL has the binary value of zero.

In addition to supporting access to variables of the above types, iECM-96 also provides commands to access the special program variables PC (program counter), PSW (program status word) and SP (stack pointer). These commands are discussed at the end of this section under the heading "Processor Variables".
BYTE Commands

There are four forms for the BYTE commands:

- **BYTE <byte_address>**
- **BYTE <byte_address> = <byte_value>**
- **BYTE <byte_address> TO <byte_address>**
- **BYTE <byte_address> TO <byte_address> = <byte_value>**

All of these commands can be used whether or not the user's program is running.

**BYTE <byte_address>**

This form is used to examine and then possibly change one or more sequential BYTE variables. When this command is invoked iECM-96 will display the <byte_address> symbolically if a valid symbol exists for that <byte_address>. Whether or not the symbolic display occurs, iECM-96 will display the <byte_address> in hexadecimal notation, the value of the BYTE in the default base and wait for an input from you. You can respond with a CARRIAGE-RETURN character, an ESC character, or by entering a numeric value. A CARRIAGE-RETURN will terminate the command. An ESC will result in the display of the next sequential BYTE variable. If a numeric value is entered then the BYTE variable will be set to this value and the iECM-96 will again wait for input. At this point you can respond only with an ESC or CARRIAGE-RETURN. As before, the ESC will display the next sequential BYTE and the CARRIAGE-RETURN will terminate the command.

**BYTE <byte_address> = <byte_value>**

This form is used to set an individual BYTE variable without first checking its current value. When invoked, this command sets the BYTE variable at <byte_address> to <byte_value>.

**BYTE <byte_address> TO <byte_address>**

This form is used to display a region of memory as a sequence of BYTE variables. When this command is invoked, iECM-96 will start by displaying the current default base and then a series of lines showing the contents of the selected memory region. If a symbol exists in iECM-96's symbol table for the next <byte_address> then this symbol will be displayed. Whether or not the symbolic display happens, the next line will start with a hexadecimal display of the address of the next BYTE variable to be displayed followed by the display of up to 16 bytes of memory as BYTE variables in the default base. A new line will be started whenever 16 bytes of memory have been displayed on the line or a valid symbol exists in iECM-96's symbol table for the next <byte_address> to be displayed. The command terminates when all of the BYTE variables in the selected range have been displayed. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

**BYTE <byte_address> TO <byte_address> = <byte_value>**

This form is used to initialize a region of memory to the given <byte_value>. Note that this command will take a little over a millisecond (at 9600 baud) for each BYTE loaded. This command can be terminated by entering a carriage return but this leaves only part of the memory region initialized.
WORD Commands

There are four basic forms for the WORD commands:

WORD <word_address>
WORD <word_address> = <word_value>
WORD <word_address> TO <word_address>
WORD <word_address> TO cwordIaddress> = <word-value>

All of these commands can be used whether or not the user's program is running.

WORD <word_address>
This form is used to examine and then possibly change one or more sequential WORD variables. When this command is invoked iECM-96 will display the <word_address> symbolically if a valid symbol exists for that <word_address>. Whether or not the symbolic display occurs, iECM-96 will display the <word_address> in hexadecimal notation, the value of the WORD in the default base and wait for an input from you. You can respond with a CARRIAGE-RETURN character, an ESC character, or by entering a numeric value. A CARRIAGE-RETURN will terminate the command. An ESC will result in the display of the next sequential WORD variable. If a numeric value is entered then the WORD variable will be set to this value and the iECM-96 will again wait for input. At this point you can respond only with an ESC or CARRIAGE-RETURN. As before, the ESC will display the next sequential WORD and the CARRIAGE-RETURN will terminate the command.

WORD <word_address> = <word_value>
This form is used to set an individual WORD variable without first checking its current value. When invoked, this command sets the WORD variable at <word_address> to <word_value>.

WORD <word_address> TO <word_address>
This form is used to display a region of memory as a sequence of WORD variables. When this command is invoked iECM-96 will start by displaying the current default base and then a series of lines showing the contents of the selected memory region. If a symbol exists in iECM-96's symbol table for the next <word_address> then this symbol will be displayed. Whether or not the symbolic display happens, the next line will start with a hexadecimal display of the address of the next WORD variable to be displayed followed by the display of up to 16 bytes of memory as WORD variables in the default base. A new line will be started whenever 16 bytes of memory have been displayed on the line or a valid symbol exists in iECM-96's symbol table for the next <word_address> to be displayed. The command terminates when all of the WORD variables in the selected range have been displayed. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

WORD <word_address> TO <word_address> = <word_value>
This form is used to initialize a region of memory to the given <word_value>. Note that this command will take a little over a millisecond (at 9600 baud) for each WORD loaded. This command can be terminated by entering a carriage return but this leaves only part of the memory region initialized.
DWORD Commands

There are four basic forms for the DWORD commands:

- DWORD <dword_address>
- DWORD <dword_address> = <dword_value>
- DWORD <dword_address> TO <dword_address>
- DWORD <dword_address> TO <dword_address> = <dword_value>

All of these commands can be used whether or not the user's program is running.

DWORD <dword_address>
This form is used to examine and then possibly change one or more sequential DWORD variables. When this command is invoked iECM-96 will display the <dword_address> symbolically if a valid symbol exists for that <dword_address>. Whether or not the symbolic display occurs, iECM-96 will display the <dword_address> in hexadecimal notation, the value of the DWORD in the default base and wait for an input from you. You can respond with a CARRIAGE-RETURN character, an ESC character, or by entering a numeric value. A CARRIAGE-RETURN will terminate the command. An ESC will result in the display of the next sequential DWORD variable. If a numeric value is entered then the DWORD variable will be set to this value and the iECM-96 will again wait for input. At this point you can respond only with an ESC or CARRIAGE-RETURN. As before, the ESC will display the next sequential DWORD and the CARRIAGE-RETURN will terminate the command.

DWORD <dword_address> = <dword_value>
This form is used to set an individual DWORD variable without first checking its current value. When invoked, this command sets the DWORD variable at <dword_address> to <dword_value>.

DWORD <dword_address> TO <dword_address>
This form is used to display a region of memory as a sequence of DWORD variables. When this command is invoked, iECM-96 will start by displaying the current default base and then a series of lines showing the contents of the selected memory region. If a symbol exists in iECM-96's symbol table for the next <dword_address> then this symbol will be displayed. Whether or not the symbolic display happens, the next line will start with a hexadecimal display of the address of the next DWORD variable to be displayed followed by the display of up to 16 bytes of memory as DWORD variables in the default base. A new line will be started whenever 16 bytes of memory have been displayed on the line or a valid symbol exists in iECM-96's symbol table for the next <dword_address> to be displayed. The command terminates when all of the DWORD variables in the selected range have been displayed. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

DWORD <dword_address> TO <dword_address> = <dword_value>
This form is used to initialize a region of memory to the given <dword_value>. Note that this command will take a little over a millisecond (at 9600 baud) for each DWORD loaded. This command can be terminated by entering a carriage return but this leaves only part of the memory region initialized.
REAL Commands

There are four basic forms for the REAL commands:

REAL <real_address>
REAL <real_address> = <real_value>
REAL <real_address> TO <real_address>
REAL <real_address> TO <real_address> = <real_value>

All of these commands can be used whether or not the user's program is running.

REAL <real_address>

This form is used to examine and then possibly change one or more sequential REAL variables. When this command is invoked, iECM-96 will display the <real_address> symbolically if a valid symbol exists for that <real_address>. Whether or not the symbolic display occurs, iECM-96 will display the <real_address> in hexadecimal notation, the value of the REAL in the default base and wait for an input from you. You can respond with a CARRIAGE-RETURN character, an ESC character, or by entering a numeric value. A CARRIAGE-RETURN will terminate the command. An ESC will result in the display of the next sequential REAL variable. If a numeric value is entered then the REAL variable will be set to this value and the iECM-96 will again wait for input. At this point you can respond only with an ESC or CARRIAGE-RETURN. As before, the ESC will display the next sequential REAL and the CARRIAGE-RETURN will terminate the command.

REAL <real_address> = <real_value>

This form is used to set an individual REAL variable without first checking its current value. When invoked, this command sets the REAL variable at <real_address> to <real_value>.

REAL <real_address> TO <real_address>

This form is used to display a region of memory as a sequence of REAL variables. When this command is invoked, iECM-96 will display a series of lines showing the contents of the selected memory region. If a symbol exists in iECM-96's symbol table for the next <real_address> then this symbol will be displayed. Whether or not the symbolic display happens, the next line will start with a hexadecimal display of the address of the next REAL variable to be displayed followed by the display of up to 16 bytes of memory as REAL variables in the default base. A new line will be started whenever 16 bytes of memory have been displayed on the line or a valid symbol exists in iECM-96's symbol table for the next <real_address> to be displayed. The command terminates when all of the REAL variables in the selected range have been displayed. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

REAL <real_address> TO <real_address> = <real_value>

This form is used to initialize a region of memory to the given <real_value>. Note that this command will take a little over a millisecond (at 9600 baud) for each REAL loaded. This command can be terminated by entering a carriage return but this leaves only part of the memory region initialized.
STACK Commands

There are two basic forms for the STACK commands:

STACK <stack_address>
STACK <stack_address> TO <stack_address>

Both of these commands can be used whether or not the user's program is running.

STACK <stack_address>
This command is useful for accessing a 16-bit variable which is known to be a fixed offset in the system stack. When this command is invoked, iECM-96 executes a "WORD <word_address> command where the <word_addr> is formed by adding <stack_address> to the current value of the system stack pointer.

STACK <stack_address> TO <stack_address>
This command is useful for accessing a sequence of 16-bit variables which are known to start at a fixed offset in the system stack. When this command is invoked, iECM-96 executes a "WORD <word_address> TO <word_address> command where both <word_address> fields are formed by adding the corresponding <stack_address> to the current value of the system stack pointer. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

STRING commands

There is only one form of the STRING command:

STRING <byte_address>
If a symbol exists for <byte_address> in the iECM-96's symbol table then this symbol will be displayed. Whether or not the symbolic display happens, the next line will start with a hexadecimal display of <byte_address> followed by the NUL terminated ASCII string starting at that address. For long strings only the first 60 characters are displayed. When trailing characters are stripped, decimal points ("."") are substituted for the first three characters stripped.
Processor Variables

Several commands are provided to access variables which are associated with the processor rather than with the program:

- **PC**
  
  \[ PC = \text{byte\_address} \]

- **PSW**
  
  \[ PSW = \text{word\_value} \]

- **SP**
  
  \[ SP = \text{word\_address} \]

The processor variables can be modified only while the target is stopped, they can be read at any time. These commands allow the display and loading of the program counter (PC), program status word (PSW) and stack pointer (SP). Display is in the default base.

**NOTE:** The examination of the SP will be confusing if you don't understand the following paragraphs.

The iECM-96 software uses two words in the user's stack to store the PC and PSW during a host interface interrupt. When the user displays the SP (or uses the STACK command) the value shown for SP is adjusted by 4 bytes to compensate for this overhead so that it becomes more or less invisible to the user (the user must still allow for the extra stack space used). This is convenient but creates confusion if you display using the SP command and then use the WORD command to look at location 18H which is the register address of the stack pointer. Location 18H will be 4 less than "SP".

An additional consideration is what happens when you attempt to write into the stack pointer using the SP command. Before returning from the RISM interrupt service routine (ISR) which actually updates the stackpointer, the RISM places in the stack a return address and associated PSW for the idle loop it executes while the target is "stopped". This prevents the target from getting lost upon return from the ISR. You should not attempt to modify the stack pointer from the console through the use of its register address (18H); it should only be modified by the SP commands or by execution of user code in the target. This decreases the possibility of the target getting confused.

Specific implementations of the RISM may actually prevent the user from writing into "WORD 18" and thereby force the user to use the "SP" command.
ASSEMBLY AND DISASSEMBLY

iECM-96 supports the examination and modification of code memory using the standard mnemonics for the MCS-96 assembler (ASM-96). Although standard mnemonics are used, the iECM-96 does not build a symbol table of user symbols as assembly mnemonics are entered. This makes it a single-line assembler (SLA) because references are never made to information entered on other lines. No labels are generated by the SLA, although it can use labels which are loaded as symbolic information along with object code when a file translated in the debug mode has been loaded. The iECM-96 SLA will accept mnemonics for all instructions which can actually be executed by the target processor. It will not accept "generic" instructions such as BE or CALL which are processed by ASM-96 into standard MCS-96 instructions. It will accept JE and SCALL or LCALL which are the specific instructions the MCS-96 processors understand.

SLA (Single Line Assembly) Commands

The commands which invoke the SLA are:

ASM <code_address>
ASM

The SLA is useful for writing short code pieces on-line for testing or patching programs but is not intended as a replacement for a true assembler such as ASM-96. The SLA can be invoked whether or not user code is running, but there is an obvious danger in modifying code that is being executed.

ASM <code_addr>
This command causes the iECM-96 software to enter the SLA mode. The assembly program counter (APC) will be set to <code_addr> and lines of "assembly language" entered by the user will be converted to object code and loaded into the target’s memory. iECM-96 will complain if erroneous inputs are made but will remain in the SLA mode. This mode is terminated by entering the only "directive" understood by the SLA: END.

ASM
This command operates identically to the ASM <code_addr> command except that the APC is not initialized. If this is the first time that the SLA has been used then APC will be set to 2080H. If it is not then APC will point at the byte following the last instruction generated by the SLA.
Disassembly Commands

The disassembler converts binary object code in the target memory to ASM-96 mnemonics. There are several commands which invoke the disassembler:

- DASM
- DASM <count>
- DASM <code_addr>
- DASM <code_addr>,<count>
- DASM <code_addr> TO <code_addr>

These commands are useful for examining a portion of the program for which listings are not available or for checking program patches, and can be used whether or not user code is running.

DASM
This command disassembles the instruction currently pointed to by the user's program counter (PC).

DASM <count>
This command reads the current value of the user's program counter (PC) and disassembles <count> instructions starting at that location. The parameter <count> must be less than 256T (100H) so that the command parser can distinguish this command from the command "DASM <code_addr>. This restriction does not apply to the DASM <code_addr>,<count> instruction. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

DASM <code_addr>
This command disassembles the instruction at <code_addr>. The parameter <code_addr> must be greater or equal to 256T (100H) so that the command parser can distinguish it from the DASM <count> instruction.

DASM <code_addr>,<count>
This command disassembles <count> instructions starting with the one at <code_addr>. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.

DASM <code_addr> TO <code_addr>
This command disassembles the region of memory specified. If an instruction crosses the ending address of the region it will be completely disassembled before the command terminates. During lengthy displays you can stop the output to the console by hitting the SPACE bar. Display can be resumed by hitting the SPACE bar a second time. The command can be terminated by entering a carriage return.
SYMBOL OPERATIONS

iECM-96 supports several commands dealing with symbolic information that can be loaded along with object code. The commands are:

SYMBOLS
SYMBOLS OFF
SYMBOLS ON
FLUSH

An additional command, "LOADSYM <filename>" can be used to load iECM-96's symbol table without affecting the target's memory. This command is described in the section "File Operations".

SYMBOLS
This command displays the symbols that are currently in iECM-96's symbol table.

SYMBOLS OFF
This command suppresses searching the symbol table during output. It does not prevent the use of the symbol table during input. This command is provided because symbolic output with large symbol tables can be very slow.

SYMBOLS ON
This command reenables symbolic output.

FLUSH
This command deletes all the symbols currently in the symbol table.
RISM

This section will describe the elements of the RISM which will be common to all implementations. Additional documentation of this implementation is in appendices B and C.

RISM Variables

RISM_DATA
RISM_DATA is a 32-bit register which acts as the primary data interface between software running in the host and the RISM running in the target.

RISM_ADDR
RISM_ADDR is a 16-bit register which contains the address to be used for reading and writing target memory.

RISM_STAT
RISM_STAT is an 8-bit register used to store RISM status and state information. This register contains the following Boolean flags:

- DLE_FLAG
  This flag indicates the next character received by the RISM should be treated as a data byte even if its value corresponds to an implemented command.

- RUN_FLAG
  This flag indicates that the target is running user code.

- TRAP_FLAG
  This flag indicates that the target was running user code but that a software TRAP occurred which suspended its execution.

- DIAGNOSTIC_FLAG
  This is an optional flag that indicates that the target is operating in a diagnostic mode. The details of this are implementation dependent.

USER_PC
USER_PC is used to save the user's program counter while the user's code is not executing.

USER_PSW
USER_PSW is used to save the user's program status word while the user's code is not executing.

Other Variables
Specific implementations of RISMs will require other variables to be used for temporary storage.
RISM Structure

The RISM resides in the target system and provides the interface between the target system and the user interface which resides in the host system. A design goal of the RISM was to keep it compact and simple. This serves two purposes:

1. The RISM can reside in a user's system with minimal impact on available memory
2. The RISM is easy to port into the target's environment.

The goals were met by keeping the internal state structure of the RISM as simple as possible. There are only three internal flags which can change the way that the RISM deals with a character sent by the host.

- **DLE_FLAG**: If this flag is set then the next received character is assumed to be a data byte as opposed to a command byte.
- **RUN_FLAG**: This flag is set if the target is running user code. It can modify the operation of some of the RISM commands.
- **TRAP_FLAG**: This flag is set if the user code has been halted because it executed a TRAP instruction. The TRAP_FLAG is cleared whenever the RISM starts the execution of user code.

Receiving Data from the Host

When the RISM receives a character from the host its first task is to determine if it represents a command or data. If the character is less than 32 (decimal) then it is assumed to be a command, if not then it is taken to be data. If the host needs to send a data byte which has a value less than 32 then it first must issue a SET_DLE command. If the DLE_FLAG is set then the next character received by the RISM will be interpreted as data (even if it is less than 32) and then the DLE_FLAG will be cleared. Once the RISM has determined that the received character is a data byte it processes it by shifting the 32-bit RISM_DATA register left eight places and then placing the data byte in the lower byte of the RISM_DATA register. The data shifted out of the upper byte of the RISM_DATA register is discarded.

Sending Data to the Host

When the host expects data to be returned from the RISM it sends a TRANSMIT command byte and waits for a response. The RISM transmits the lower byte of the 32-bit RISM_DATA register and right shifts the RISM_DATA register right by eight bits. As part of this command the RISM increments its RISM_ADDR register. The RISM only transmits data in response to an TRANSMIT command, never on its own initiative or even in response to other commands from the host.
RISM Commands

This section will detail the operation of each of the commands sent to the RISM.

SET_DLE_FLAG (Code 00H)
This command sets the DLE_FLAG. This will force the next character received by the RISM to be treated as data even if its value corresponds to a RISM command. The code which overrides the normal selection of command or data also clears the DLE_FLAG so that it applies only to the first character received after the SET_DLE_FLAG command.

TRANSMIT (Code 02H)
This command will transmit the lower eight bits of the RISM_DATA register to the host, right shift the data register eight places, and increment the RISM_ADDR register. Sequential TRANSMIT commands are used to read the RISM_DATA register and the RISM_ADDR register indicates the address that corresponds to the least significant byte in the RISM_DATA register.

READ_BYTE (Code 04H)
This command will read the byte of memory pointed to by the RISM_ADDR register and place the result in the least significant byte of the RISM_DATA register.

READ_WORD (Code 05H)
This command will read the word of memory pointed to by the RISM_ADDR register and place the result in the least significant word of the RISM_DATA register.

READ_DOUBLE (Code 06H)
This command will read the double-word of memory pointed to by the address register and place the result in the RISM_DATA register.

WRITE_BYTE (Code 07H)
This command stores the least significant byte of the RISM_DATA register in the byte of memory pointed to by the RISM_ADDR register and increments the RISM_ADDR register (by one) to point at the next memory byte.

WRITE_WORD (Code 08H)
This command stores the least significant word of the RISM_DATA register in the word of memory pointed to by the RISM_ADDR register and increments the RISM_ADDR register (by two) to point at the next memory word.

WRITE_DOUBLE (Code 09H)
This command stores the RISM_DATA register in the double-word of memory pointed to by the RISM_ADDR register and increments the RISM_ADDR register (by four) to point at the next memory double-word.

LOAD_ADDRESS (Code 0AH)
This command loads the RISM_ADDR register with the least significant word in the RISM_DATA register.

INDIRECT_ADDRESS (Code 0BH)
This command reads the memory word pointed to by the RISM_ADDR and stores it into the RISM_ADDR register. The RISM_DATA register is not modified by this command.
READ_PSW (Code 0CH)
This command loads the RISM_DATA register with the PSW (Program Status Word) associated with the user's code. Most RISM implementations will have to check RUN_FLAG to determine how to access the user's PSW.

WRITE_PSW (Code 0x0D)
This command loads the PSW (Program Status Word) associated with the user's code from the RISM_DATA register. The host software will only invoke this command while user code is not running.

READ_SP (Code 0x0E)
This command loads the RISM_DATA register with the SP (Stack Pointer) associated with the user's code.

WRITE_SP (Code 0x0F)
This command loads the SP (Stack Pointer) from the RISM_DATA register. This command must also push two values into the newly created stack area. These values are the PC (first) and PSW (second) associated with the idle loop which executes while user code is not running. The host software will only invoke this command while user code is not running.

READ_PC (Code 0x10)
This command loads the RISM_DATA register with the PC (Program Counter) associated with the user's code. Most RISM implementations will have to check RUN_FLAG to determine how to access the user's PC.

WRITE_PC (Code 0x11)
This command loads the PC (Program Counter) associated with the user's code from the RISM_DATA register. The host software will only invoke this command while user code is not running.

START_USER (Code 0x12)
This command is responsible for starting the execution of user code, clearing the TRAP_FLAG, and setting RUN_FLAG. The action of this command relies on it being executed as part of an ISR (interrupt service routine). At the start of the ISR the current PC and PSW are pushed into the stack. If the user code is not running the PC and PSW which are pushed into the stack will be associated with an idle loop which the RISM runs while it waits for an interrupt. The START_USER command deletes the PC and PSW from the stack and replaces them with USER_PC and USER_PSW. When control returns from the ISR the user's code will execute rather than the idle loop. The host software will not issue a GO command if the user code is already running.

STOP_USER (Code 0x13)
This command is responsible for stopping the execution of user code and clearing the RUN_FLAG. The action of the HALT command mirrors that of the GO command. In the case of the HALT command the user's PC and PSW are pushed into the stack upon entry to the ISR. The STOP_USER command saves this user information in USER_PC and USER_PSW and replaces it with PC and PSW values which are associated with the idle loop. When control returns from the ISR the idle loop will execute rather than the user's code. The host software will not issue a HALT command unless the user code is running.
TRAP_ISR
This is a pseudo-command. It can not be issued directly by the host software but is executed when a TRAP instruction is executed. The TRAP instruction is used by iECM-96 to implement software breakpoints and single stepping. A separate entry point into the STOP_USER is provided for the TRAP vector. Code at this entry point sets the TRAP_FLAG and then drops into the code which implements the STOP_USER command.

REPORT_STATUS (Code 0x14)
This command loads the least significant word of the RISM_DATA register with status information. Valid status values are:

- 0--Indicates that user code is stopped (RUN_FLAG and TRAP_FLAG are both FALSE).
- 1--Indicates that user code is running (RUN_FLAG is TRUE)
- 2--Indicates that user code executed a TRAP instruction (TRAP_FLAG is TRUE)

The host software will periodically poll the target system to check on its status and this polling can rob execution time from the user's program. This loss of target processor cycles can be avoided by setting the Ring Indicator modem status line signal whenever the RUN_FLAG is set. The host software will assume that the target is running user code whenever it detects the ring indicator and will only issue REPORT_STATUS commands if the ring indicator is off.

MONITOR_ESCAPE (Code 0x15)
This command provides for the addition of RISM commands for special purposes; it uses the RISM_DATA register to extend the command set of the RISM. The basic RISM requires only one of these "extended" commands; if the lower 16-bits of the RISM_DATA register is one (RISM_DATA = 0XXXX001H) then the target processor should execute either a RST (ReSeT) instruction or a software initialization routine.

Start Up Commands ("/" or "\")
Upon reset the board is in the echo mode. Until it receives an ASCII slash ("/") or reverse-slash ("\") it should increment every character it receives from the host and send the incremented value back to the host. It will also display the binary code of the character received on the Port 1 LED's. If a reverse-slash is received by the RISM it will leave the echo mode (set USER_MAP flag true), remap memory and start normal operation. If a slash is received it will stop echoing incremented received data and start responding to RISM commands with the diagnostic flag set. In this mode there are diagnostic routine resident in EPROM which are useful for debugging the board. See the -DIAG option under Initiating and Terminating iECM-96 in the USER INTERFACE section of this manual for additional information on the Diagnostics Mode.
Appendix A.

Schematics and Parts List
## Bill Of Materials

**December 27, 1988 15:53:23**

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Reference</th>
<th>Part</th>
<th>Vendor</th>
<th>Manuf.</th>
<th>Part#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>U5</td>
<td>80C196-PLCC</td>
<td>INTEL</td>
<td>INTEL</td>
<td>N80C196KB12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>U20</td>
<td>82510</td>
<td>INTEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>U1, U8,</td>
<td>JEDEC 28PIN</td>
<td>INTEL</td>
<td></td>
<td>D27C64</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>U6, U13, U14</td>
<td>JEDEC 28PIN</td>
<td>Sterling Hitachi</td>
<td></td>
<td>HM6264P-10</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>U17</td>
<td>74AC00</td>
<td>Hamilton Fairchild</td>
<td>74AC00PC</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>U21</td>
<td>74AC08</td>
<td>Hamilton Fairchild</td>
<td>74AC08PC</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>U2</td>
<td>74AC14</td>
<td>Hamilton Fairchild</td>
<td>74AC14PC</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>U16</td>
<td>74AC32</td>
<td>Hamilton Fairchild</td>
<td>74AC32PC</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>U7</td>
<td>74AC74</td>
<td>Hamilton Fairchild</td>
<td>74AC74PC</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>U15</td>
<td>74AC112</td>
<td>Hamilton GE/RCA</td>
<td></td>
<td>CD74AC112E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>U9</td>
<td>74AC240</td>
<td>Hamilton Fairchild</td>
<td>74AC240PC</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>U10, U11</td>
<td>74AC373</td>
<td>Hamilton Fairchild</td>
<td>74AC373PC</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>U18</td>
<td>14C86</td>
<td>Hamilton National</td>
<td>DS14C86N</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>U19</td>
<td>14C89</td>
<td>Hamilton National</td>
<td>DS14C89N</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>U12</td>
<td>22V10</td>
<td>Luscombe Cypress</td>
<td>PALC22V10-35PC</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>U4</td>
<td>PMIREF02</td>
<td>Hamilton PMI</td>
<td></td>
<td>REF02HP</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>U3</td>
<td>LM358N</td>
<td>Hamilton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>D2, D1</td>
<td>diode</td>
<td>Hamilton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>R3</td>
<td>resistor</td>
<td>Sterling Dale</td>
<td></td>
<td>MDF-1603-271G</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>C2</td>
<td>cap</td>
<td>Hamilton Sprague</td>
<td></td>
<td>926CXR7562X050B</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>X1</td>
<td>12MHz</td>
<td>Sterling M-TRON</td>
<td></td>
<td>MP-1 12.0000</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>X2</td>
<td>18.432MHz</td>
<td>Sterling M-TRON</td>
<td></td>
<td>MP-1 18.4320</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>S1</td>
<td>RESET</td>
<td>Digi-key Panasonic</td>
<td></td>
<td>P9950</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>D5, D3, D4, D6, D7</td>
<td></td>
<td>Hamilton</td>
<td></td>
<td>1N4305</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>DP1</td>
<td>HDSP-48XX</td>
<td>Sterling Lite-On</td>
<td></td>
<td>LTA1000G</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>R10</td>
<td>180</td>
<td>Hamilton Mepco</td>
<td></td>
<td>CR25-180</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>R5</td>
<td>4.7K</td>
<td>Hamilton Mepco</td>
<td></td>
<td>CR25-4.7K</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>R2</td>
<td>10K</td>
<td>Hamilton Mepco</td>
<td></td>
<td>CR25-10K</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
<td>R1, R7, R18</td>
<td>100K</td>
<td>Hamilton Mepco</td>
<td></td>
<td>CR25-100K</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>R6, R4, R6, R9, R11, R12, R13, R14, R15, R16, R17</td>
<td>1M</td>
<td>Hamilton Mepco</td>
<td></td>
<td>CR25-1M</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>RP2</td>
<td>180 ohm SIP</td>
<td>Hamilton Bourns</td>
<td></td>
<td>4610X-101-181</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>RP1</td>
<td>10K POT</td>
<td>Hamilton Bourns</td>
<td></td>
<td>3009P-1-103</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
<td>C4, C5, C8, C18</td>
<td>40uF</td>
<td>Hamilton Sprague</td>
<td></td>
<td>1C10C0G3307050B</td>
</tr>
<tr>
<td>34</td>
<td>3</td>
<td>C12, C6, C18</td>
<td>0.1uF</td>
<td>Hamilton Sprague</td>
<td></td>
<td>1C10Z5UI03M050B</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>C11, C14, C16, C17, C19, C20, C21, C22, C23, C24, C25, C26, C27, C28, C30, C32, C33, C34, C35</td>
<td>0.1uF</td>
<td>Hamilton Sprague</td>
<td></td>
<td>1C10Z5UI04M050B</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>C1, C7, C14, C15</td>
<td>1.0uF</td>
<td>Hamilton Sprague</td>
<td></td>
<td>150D105X9015A2</td>
</tr>
<tr>
<td>37</td>
<td>2</td>
<td>C31, C37</td>
<td>6.8uF</td>
<td>Hamilton Sprague</td>
<td></td>
<td>199D685X9035DA1</td>
</tr>
</tbody>
</table>
### Bill Of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Reference</th>
<th>Part</th>
<th>Vendor</th>
<th>Manuf.</th>
<th>Part#</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>1</td>
<td>C29</td>
<td>22uF</td>
<td>Hamilton Sprague</td>
<td>150D286X9015B2</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>2</td>
<td>P1,P2</td>
<td>DB9 Female</td>
<td>Sterling AMP</td>
<td>207084-1</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>16</td>
<td>E7,E1,E3,E4,E5,E6,E8,E9, E10,E11,E12,E13,E14,E17, E18,E20</td>
<td>3PIN JUMPER</td>
<td>Marshall A P Prod.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>E15</td>
<td>4PIN JUMPER</td>
<td>Marshall A P Prod.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>JP4</td>
<td>POWER CONNECTOR</td>
<td>Hamilton Molex</td>
<td>09-74 1041</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>JP1</td>
<td>CON26</td>
<td>Marshall A P Prod.</td>
<td>929665-01-36</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>JP2</td>
<td>CON50</td>
<td>Marshall A P Prod.</td>
<td>929665-01-36</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>JP3</td>
<td>CON60</td>
<td>Marshall A P Prod.</td>
<td>929665-01-36</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B.

Specific iRISM Information
APPENDIX B

Specific iRISM Information

The EV80C196KB is designed to be a software evaluation tool for the ROMless 80C196KB 16-bit microcontroller. As such, ports 3 and 4 are not available for use as I/O ports unless offboard latches/buffers and decoding logic are used. All unreserved functions of the 80C196KB are available to you except for the Non-Maskable Interrupt (NMI), the TRAP instruction, and 512 bytes of address space. The Chip Configuration Byte is also used by the monitor, but most of its functions are provided by external logic.

Reserved Functions

The NMI pin is reserved for use by the Host Interface. In order for the Host Interface to function properly, jumper-shunt E7 must be installed from B-C. However, if your application demands the use of NMI (available on JP3), you can alter the RISM source file (96KBRISM.A96, included on your disk) to use EXTINT instead of NMI, and change jumper-shunt E7 to A-B.

The TRAP instruction is reserved.

On the EV80C196KB jumper shunt E20 must be installed from B to C for the RESET SYSTEM command to work properly. If you wish to run code in the board while it is not connected to a host, you should remove jumper shunt E20 prior to disconnecting the board from the host. If E20 is left installed, the board may reset as the connection is broken.

Reserved Memory

User ROMsim as shipped is 24K bytes from address 2000H to 7FFFH. The board is reconfigurable to accept various memory devices. However, breakpoints and program stepping will not operate when your code is in EPROM or other nonchangeable memory. Normally you should write your code to begin at address 2080H and download it to ROMsim using iECM-96.

Two words of user stack space must be reserved for use by the iRISM-96 software while the board is processing a host interrupt. Register locations 30H-38H are reserved for use by the iRISM monitor code. You must ensure that no registers in this partition are used by code which is to operate with the RISM. The easiest way of doing this is to generate an ASM-96 module which declares an RSEG at 30H which is nine bytes long. This module can then be linked into the final program to prevent the linker from assigning these registers to some other module.

You must not alter the TRAP vector at 2010H or the NMI vector at 203EH.

Memory from 2014H-202FH is reserved for use by the iRISM monitor.
Appendix C.

Listing of iRISM-196KB
ERR LOC  OBJECT  LINE  SOURCE STATEMENT

1    EV96 module main
2    ; ================
3    ; This file contains a RISM designed to operate the EV80C196KB evaluation
4    ; board. It includes the required RISM features and the optional diagnostic
5    ; mode. The board also supports remapping the memory space after reset.
6    ; This allows the RISM code to gain control on reset and, after the
7    ; initialization routines are complete, remap memory so that user code
8    ; can be loaded into RAM at the reset location (2080H).
9    ;
10   ; The serial link is provided by an external UART (82510) with the received
11   ; data interrupt tied to the NMI (Non Maskable Interrupt) of the processor.
12   ; The use of the NMI for this purpose allows the user to maintain control
13   ; of the system even if the running program locks out the interrupts or
14   ; modifies the mask register.
15   ;
16   ; In addition to the NMI and its vector, this RISM uses the following
17   ; resources:
18   ;
19   ; Two words in the system stack
20   ;
21   ; The TRAP instruction and its vector
22   ;
23   ; External memory partitions (0000H-00FFH),
24   ; (1000H-1EFFH), and
25   ; (2014H-202FH)
26   ;
27   ; ( Note that all of these partitions, (except 1D00H-1EFFH and
28   ; 2018H), are reserved by the MCS-96 architecture. )
29   ;
30   ; Nine bytes of registers in the partition (30H-38H). The
31   ; user must ensure that no registers in this partition are used
32   ; by code which is to operate with the RISM. The easiest way of
33   ; doing this is to generate an ASM-96(tm) module which declares an
34   ; RSEG at 30H which is nine bytes long. This module can then be
35   ; linked into the final program to prevent the linker from assigning
36   ; these registers to some other module.
37   ;
38   ;
39   ;
40   ; select
; define symbols for the register mapped I/O locations
; -------------------------------------------------------------
0000 zero equ 00H:word ; R/W Zero Register
0002 ad_command equ 02H:byte ; R A/D command register
0002 ad_result_lo equ 02H:byte ; R Low byte of result and channel
0004 ad_result_hi equ 03H:byte ; R High byte of result
0006 hsi_mode equ 03H:byte ; W Controls HSI transition detector
0008 hsi_time equ 04H:word ; R HSI time tag
0008 hso_time equ 04H:word ; W HSO time tag
0006 hsi_status equ 06H:byte ; R HSI status register (reads f1ro)
0006 hso_command equ 06H:byte ; W HSO command tag
0007 sbuf equ 07H:byte ; R/W Serial port buffer
0008 int_mask equ 08H:byte ; R/W Interrupt mask register
0009 int_pending equ 09H:byte ; R/W Interrupt pending register
0011 spcon equ 11H:byte ; W Serial port control register
0011 spstat equ 11H:byte ; R Serial port status register
001A watchdog equ 0AH:byte ; W Watchdog timer
001C timer1 equ 0AH:word ; R Timer1 register
001C timer2 equ 0CH:word ; R Timer2 register
001E port0 equ 0EH:byte ; R I/O port 0
001F baud_reg equ 0FH:byte ; W Baud rate register
001F iop0 equ 0FH:byte ; R/W I/O port 1
001F iop2 equ 10H:byte ; R/W I/O port 2
001F ioc0 equ 15H:byte ; W I/O control register 0 (HSI/O)
001F ioc1 equ 15H:byte ; R I/O status register 0
001F ioc1 equ 16H:byte ; W I/O control register 1 (Port2)
001F ioc1 equ 16H:byte ; R I/O status register 1
001F pwm_control equ 17H:byte ; W PWM control register
001F sp equ 18H:word ; R/W System stack pointer

; This section defines utility macros non-specific to this program
; -------------------------------------------------------------

DEFINE_BIT macro name,bitnum
    name equ bitnum
endm

SET_BIT macro regnum,bitnum
    orb regnum,#( 1 SHL (bitnum mod 8) )
endm

CLR_BIT macro regnum,bitnum
    andb regnum,#not( 1 SHL (bitnum mod 8) )
endm

BL macro label
bnc label
endm
ERR LOC OBJECT LINE
8000 offset equ 8000H ; Code offset before REMAP
0000 rism_psw equ 0000H ; No Interrupts enabled

; This section contains EQUates which may change with different versions
; ---------------------------------------------------------------
8000 offset equ 8000H ; Code offset before REMAP
0000 rism_psw equ 0000H ; No Interrupts enabled

; Tell the commands what to use for psw while monitor is running
; ---------------------------------------------------------------

; This section contains several macros generate specifically for this program
; ---------------------------------------------------------------------------

; ENTER RISM
; A macro which generates the prologue for the RISM ISR
;
; EXIT_RISM
; A macro which generates the epilogue for the RISM ISR

; SEND_DATA_BYTE
; A macro which passes the lower eight bits of RISM_DATA to
; the serial port, it assumes the port is ready for data

; BYTE_PROTECT
; A macro which terminates the RISM ISR if the RISM is about
; to write into a byte it should not modify.

; WORD_PROTECT
; A macro which terminates the RISM ISR if the RISM is about
; to write into a word it should not modify.

; DWORD_PROTECT
; A macro which terminates the RISM ISR if the RISM is about
; to write into a double-word it should not modify.

eject
<table>
<thead>
<tr>
<th>LINE</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>ENTER_RISM</td>
<td>macro</td>
</tr>
<tr>
<td>129</td>
<td>pushf</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>EXIT_RISM</td>
<td>macro</td>
</tr>
<tr>
<td>132</td>
<td>popf</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>SEND_DATA_BYTE</td>
<td>macro</td>
</tr>
<tr>
<td>136</td>
<td>stb RISM_DATA, txc_rxd[0]</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>BYTE_PROTECT</td>
<td>macro</td>
</tr>
<tr>
<td>139</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>; No special protection</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>WORD_PROTECT</td>
<td>macro</td>
</tr>
<tr>
<td>142</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>; No special protection</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>DWORD_PROTECT</td>
<td>macro</td>
</tr>
<tr>
<td>145</td>
<td>endn</td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>; No special protection</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>object</td>
<td></td>
</tr>
</tbody>
</table>
These registers are used only by the diagnostic routines. They are not required for normal execution.

```assembly
rseg at lch
```

```assembly
ax:    dsw  l
al     equ ax:byte
ah     equ (ax+1):byte
dx:    dsw  l
bx:    dsw  l
cx:    dsw  l
```

`$eject`
These registers MUST be reserved for the RISM
---------------------------
0030 170  rseg at 30H
171  
172
0030 173  RISM_DATA:  dsl 1 ; The RISM data register
0034 174  RISM_ADDR:  dsw 1 ; The RISM address register
175  
0036 176  tempw:  dsw 1 ; Temp for use by monitor
0036 177  tempb equ tempw:byte
0036 178  char equ tempw:byte
179  
0038 180  RISM_STAT:  doh 1 ; Contains rism state flags
181  DEFINE_BIT  DLE_FLAG,0
183  DEFINE_BIT  RUN_FLAG,2
185  DEFINE_BIT  TRAP_FLAG,1
187  DEFINE_BIT  USER_MAR,3
189  DEFINE_BIT  DIAGNOSTIC_FLAG,7
191  
192 ; These variables are used by monitor when in diagnostic mode only.
194 ; -----------------------------------------------
195  
003A 196  dUSER_PC:  dsw 1 ; Saves user's pc during halt
003C 197  dUSER_PSW:  dsw 1 ; Saves user's psw during halt
198  
200  dseg at 200H
201 ; These variables are used in the normal (non-diagnostic) mode
203 ; -----------------------------------------------
204  
202  
205  USER_PC:  dsw 1 ; Saves user's pc during halt
206  USER_PSW:  dsw 1 ; Saves user's psw during halt
207  
The serial channel is provided by an external 82510 UART which uses the NMI as an interrupt to the processor. The addresses associated with this device are defined below.

```plaintext
EVENT
213
dseg at 1E00H
214
215
uart: dsb 100H

1EO0
217
txd_rxd equ uart byte ; bank0 (if dlab=0) or bank1
1EO0
218
baud_a_lo equ uart byte ; bank0 (if dlab=1)
1EO1
219
baud_a_hi equ uart+1 byte ; bank0 (if dlab=1)
1EO1
220
gener_enabl equ uart+1 byte ; bank0 (if dlab=0)
1EO2
221
general_int equ uart+2 byte ; bank0
1EO3
222
line_config equ uart+3 byte ; bank0
1EO4
223
modem_constr equ uart+4 byte ; bank0
1EO5
224
line_status equ uart+5 byte ; bank0
1EO6
225
modem_stats equ uart+6 byte ; bank0
1EO7
226
addr_contr0 equ uart+7 byte ; bank0
1E00
227
clock_config equ uart byte ; bank3
1E04
228
lo_mode equ uart+4 byte ; bank3

230 ;
The memory map of the board is changed by reading or writing to an
231 ; address between 1000H and 1DFFH. In this code, this is accomplished by
232 ; branching to address 1000H to continue RISM execution. The memory map
233 ; of this board, both before and after RESET, are as follows:
234 ;
235 ; Address After RESET After REMAP
236 ;
237 ; 0000-00FFH as data Internal Reg. file Internal Reg. file
238 ; 0000-00FFH as code RISM Monitor EPROM RISM Monitor EPROM
239 ; 0100-1FFFFH Unused Unused--User expansion possible
240 ; 1D00-1DFFH RISM Monitor EPROM RISM Monitor EPROM
241 ; 1F00-1FFFFH External UART (1170) External UART (1170)
242 ; 1F00-1FFFFH Unused (Port 3 & 4) Unused (Port 3 & 4)
243 ; 2000-20FFH RISIM Int. Vect. EPROM User Int. Vect. RAM (NOT TRAP!)
244 ; 2140-21FFH RISIM EPROM RISIM Data RAM
245 ; 2030-207FH RISIM Int. Vect. EPROM User Int. Vect. RAM (NOT NMI!)
246 ; 2040-207FH Unused RISIM EPROM User Data RAM
247 ; 2080-27FFH RISIM Monitor EPROM User 16-Bit Code/Data RAM
248 ; 2800-5FFFFH 16-Bit Code/Data RAM User 16-Bit Code/Data RAM
249 ; 6000-7FFFFH 8-Bit Code/Data RAM User 8-Bit Code/Data RAM
250 ; 8000-FFFFH Unused Unused--User expansion possible
251 ;
252 ; EXIT
```
Interrupt service routine addresses to be used in RISM EPROM.

Note: Of all these interrupt vectors, only the NMI and TRAP vectors are required for operation of the RISM. The other vectors are provided as fixed entry points for routines which may be loaded into RAM in the diagnostic mode.

In the diagnostic mode memory at the interrupt vectors is mapped to EPROM so it is not possible to write into the vector table.

(In the normal (i.e. non-diagnostic mode) the interrupt vector table is mapped to RAM so the vectors can be loaded as part of the normal process of loading a user's object code.

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:

CHIP_CONFIG:
### Statement Breakdown

<table>
<thead>
<tr>
<th>Location</th>
<th>Line</th>
<th>Source</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A080</td>
<td>000</td>
<td>cseg at (offset + 2080H)</td>
<td>Initialize stack pointer</td>
</tr>
<tr>
<td>A080</td>
<td>001</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A081 A1000118</td>
<td>304</td>
<td>sp, #100H</td>
<td>Initialize stack pointer</td>
</tr>
<tr>
<td>A085 3516FD</td>
<td>305</td>
<td>ios1, $, $</td>
<td>wait for timer1 overflow</td>
</tr>
<tr>
<td>A088 3516FD</td>
<td>306</td>
<td>ios1, $, $</td>
<td>...two times,</td>
</tr>
<tr>
<td>A090 3516FD</td>
<td>307</td>
<td>zero, 2000H</td>
<td>release uart reset, and wait</td>
</tr>
<tr>
<td>A093 1138</td>
<td>308</td>
<td>ios1, $, $</td>
<td>...till uart is ready</td>
</tr>
<tr>
<td>A095 B18036</td>
<td>309</td>
<td>ldb tempb, #80H</td>
<td>set flag bit in line_config reg...</td>
</tr>
<tr>
<td>A098 C701031E36</td>
<td>310</td>
<td>stb tempb, line_conf[0]</td>
<td>so that baud_a reg's are accessible</td>
</tr>
<tr>
<td>A09D B13C36</td>
<td>311</td>
<td>ldb tempb, #3CH</td>
<td>set baud rate to 9600</td>
</tr>
<tr>
<td>A0A0 C701001E36</td>
<td>312</td>
<td>stb tempb, baud_a_lo[0]</td>
<td></td>
</tr>
<tr>
<td>A0A5 C701011E00</td>
<td>313</td>
<td>stb zero, baud_a_hi[0]</td>
<td></td>
</tr>
<tr>
<td>A0AA B10336</td>
<td>314</td>
<td>ldb tempb, #03H</td>
<td>set up uart line config reg for no...</td>
</tr>
<tr>
<td>A0AD C701031E36</td>
<td>315</td>
<td>stb tempb, line_conf[0]</td>
<td>par, 1 stop, 8bit, and rxd/rxd access</td>
</tr>
<tr>
<td>A0B2 B16036</td>
<td>316</td>
<td>ldb tempb, #60H</td>
<td>switch to bank3</td>
</tr>
<tr>
<td>A0B5 C701021E36</td>
<td>317</td>
<td>stb tempb, general_int[0]</td>
<td></td>
</tr>
<tr>
<td>A0BA B15036</td>
<td>318</td>
<td>ldb tempb, #50H</td>
<td>select baud rate gen. a for both...</td>
</tr>
<tr>
<td>A0BD C701001E36</td>
<td>319</td>
<td>stb tempb, clock_conf[0]</td>
<td>rx and tx clock source</td>
</tr>
<tr>
<td>A0C2 B17F36</td>
<td>320</td>
<td>ldb tempb, #7FH</td>
<td>select OUT1 mode on pin 12</td>
</tr>
<tr>
<td>A0C5 C701041E36</td>
<td>321</td>
<td>stb tempb, io_mode[0]</td>
<td></td>
</tr>
<tr>
<td>A0CA C701021E00</td>
<td>322</td>
<td>stb zero, general_int[0]</td>
<td>switch to bank0</td>
</tr>
<tr>
<td>A0CF B10136</td>
<td>323</td>
<td>ldb tempb, #01H</td>
<td>enable receive fifo interrupt...</td>
</tr>
<tr>
<td>A0D2 C701011E36</td>
<td>324</td>
<td>stb tempb, gener_enable[0]</td>
<td>of the uart</td>
</tr>
<tr>
<td>A0DD7 A1000036</td>
<td>325</td>
<td>ldb tempb, rism_psw</td>
<td>value for rism and initial user value</td>
</tr>
<tr>
<td>A0DB C836</td>
<td>326</td>
<td>push tempb</td>
<td>set up psw for the monitor</td>
</tr>
<tr>
<td>A0DD F3</td>
<td>327</td>
<td>popf</td>
<td>load psw with rism value</td>
</tr>
<tr>
<td>A0DE 1136</td>
<td>328</td>
<td>clr char</td>
<td>show life to user</td>
</tr>
<tr>
<td>A0EE 27FE</td>
<td>329</td>
<td>call flash_load</td>
<td>show life to user</td>
</tr>
<tr>
<td>A0F1 28F1</td>
<td>330</td>
<td>br $</td>
<td>wait for interrupt</td>
</tr>
<tr>
<td>A0F2 27FE</td>
<td>331</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F3 27FE</td>
<td>332</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F4 27FE</td>
<td>333</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F5 27FE</td>
<td>334</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F6 27FE</td>
<td>335</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F7 27FE</td>
<td>336</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F8 27FE</td>
<td>337</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0F9 27FE</td>
<td>338</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FA 27FE</td>
<td>339</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FB 27FE</td>
<td>340</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FC 27FE</td>
<td>341</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FD 27FE</td>
<td>342</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FE 27FE</td>
<td>343</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>A0FF 27FE</td>
<td>344</td>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>

---

**Note:**
- The statements involve setting up various registers and handling interrupts.
- The reset_vector initialization is crucial for setting up the system registers.
- The code snippet is part of a larger program, likely for system initialization and controlling various hardware peripherals.
This code is entered from the nmi_isr if the user memory map is not turned on. This is the echo mode and diagnostic mode of the board.

; If the diagnostic flag is clear, the board is in echo mode. Any characters received from the host are incremented and sent back to the host. They are also tested for the set user command ('\') or the set diagnostics command ('/'). If either command was sent it is carried out.

; If the diagnostic flag is set, the program branches to the diag. mode code.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.

; This code places the board in user mode until the next RESET occurs, or until RISM-STAT gets altered somehow. It branches to a location which does not get remaped, and there, a remap will be performed.
This code places the board in diagnostics mode until the next RESET or RISM STAT gets altered somehow. The user's PC is loaded with the address of the memory test and a 55H/0AAH pattern flashes on the ioport LEDs while the monitor is waiting for a command.

```
set_diag:  
  set_bit RISM_STAT, DIAGNOSTIC_FLAG
  ld   sp, #100H  ; clear stack
  ld   tempw, #rism_psw  ; value for rism and initial user value
  st   tempw, dUSER_PSW  ; store rism psw as initial user psw
  ld   tempw, #(mem-tst-offset)  ; Set up user pc
  st   tempw, dUSER_PC
  diag_pause:  
    ldb  ioport1, #55h  ; wait for a timer1 overflow
    bbc  ioport1, 5, $  ; ...twice
    xorb ioport1, #0ffh  ; invert ioport1
    br   diag_pause_loop
  ;
  ;$eject
```
This code is executed to interpret a host command when this RISM is in the diagnostics mode.

; ___________________________________________________________

        diag_mode:
        bbs  RISM_STAT, DLE_FLAG, force_load_data
        cmpb char, #1FH ; check if byte is a command
        bh   load_data ; commands are <= 1FH

        diag_command:
        ldbze tempw, char ; table lookup
        add tempw, tempw
        ld   tempw, (diag_table_offset)[tempw]
        br   [tempw]

        $eject
ERR LOC OBJECT LINK SOURCE STATEMENT

A14C  3D00   dcw  (SET_DLE_FLAG  - offset) ; 00 
A14E  3D00   dcw  (exit     - offset) ; 01 
A150  4200   dcw  (TRANSMIT  - offset) ; 02 
A152  1D00   dcw  (exit     - offset) ; 03 
A154  5700   dcw  (READ_BYTE - offset) ; 04 
A156  5C00   dcw  (READ_WORD - offset) ; 05 
A158  6100   dcw  (READ_DOUBLE - offset) ; 06 
A15A  6800   dcw  (WRITE_BYTE - offset) ; 07 
A15C  6F00   dcw  (WRITE_WORD - offset) ; 08 
A15E  7400   dcw  (WRITE_DOUBLE - offset) ; 09 
A160  7C00   dcw  (LOAD_ADDRESS - offset) ; 0A 
A162  8100   dcw  (INDIRECT_ADDRESS - offset) ; 0B 
A164  B421   dcw  (dREAD_PSW  - offset) ; 0C 
A166  C121   dcw  (dWRITE_PSW - offset) ; 0D 
A168  B300   dcw  (READ_SP   - offset) ; 0E 
A16A  C621   dcw  (dWRITE_SP - offset) ; 0F 
A16C  A621   dcw  (dREAD_PC  - offset) ; 10 
A16E  A121   dcw  (dWRITE_PC - offset) ; 11 
A170  7821   dcw  (dSTART_USER - offset) ; 12 
A172  8D21   dcw  (dSTOP_USER  - offset) ; 13 
A174  C000   dcw  (REPORT_STATUS - offset) ; 14 
A176  4E00   dcw  (MONITOR_ESCAPE - offset) ; 15 

: ;

$eject
The following routines, all named beginning with a 'd' for diagnostics, are special cases of RISM commands used when the board is in diagnostics mode.

```
dSTART_USER:
   .-------------------
   ; Flush the pause routine off the stack and set up user's context.
   SET_BIT RISM_STAT, RUN_FLAG
   CLR_BIT RISM_STAT, TRAP_FLAG
   stb RISM_STAT, modem_contr[0] ; update running signal to host

A1/1 E C/01041E1B
   std RISM_STAT, modem_contr[0]

A183 65041U1B
   add sp,#4 ; reset up to overwrite RISM pc & psw,
   push dUSER_PC ; with user pc &
   push dUSER_PSW ; user psw values

A189 CC3C
   EXIT_RISM

A18D dSTOP_USER:
   .-------------------
   ; Stops "user" execution by setting up the stack to return to pause with
   ; all interrupts but serial i/o locked out.
   pop dUSER_PSW ; remove users psw & pc from stack
   pop dUSER_PC ;
   and save

A18D CC3C

A18F CC3A

A191 dset_rism_idle:
   push #{diag_pause-offset} ; the new program counter & psw

A191 C92021
   push $rism_psw

A194 C90800
   CLR_BIT RISM_STAT, RUN_FLAG

A199 C701041E38
   stb RISM_STAT, modem_contr[0] ; update running signal to host
   EXIT_RISM

$eject
```
_ERROR STATEMENT

490: WRITE_PC:
491: ;--------
492: ; user_pc:=RISM_DATA. (Assumes user code is not running)
493: ;
494: A1A1 003A30
495: st RISM_DATA, dUSER_PC
496: EXIT_RISM
497: ;
498: A1A6
500: dREAD_PC:
501: ;--------
502: RISM_DATA:=user_pc
503: ;
504: A1A6 3A3005
505: bbe RISM_STAT, RUN_FLAG, drpc_running
506: A1A9 A03A30
507: ld RISM_DATA, dUSER_PC ; If user code is not running
508: EXIT_RISM
509: ;
510: A1AE 3180230
511: ld RISM_DATA, 2[sp] ; If user code is running
512: EXIT_RISM
513: ;
516: $execute
dREAD_PSW:
; RISM_DATA:=user_psw

; user is not running
EXIT_RISM

dWRITE_PSW:
; user_sp:=RISM_DATA. (Assumes user is not running)

; user is running
EXIT_RISM

; user is running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RISM

; user is not running
EXIT_RIS
On a reset this code flashes the LEDs connected to ioport1 if they are enabled. This is useful to see if the board is executing code properly. If a '/' or '\' is received from the host while this routine is executing, it will terminate immediately.

```assembly
Select 586 ~

ld rism_addr, #OFFH

fl_wait0:
bhr insl,5, fl_wait0

fl_wait1:
bbc iosl,5, $ ; wait for a timer1 overflow

fl_loop1:
bbc iosl,5, fl_wait1

fl_loop1

jmp flash_leds ; check if char has been received

bne quit, zero ; if so exit

jmp rism_addr, zero ; else continue flashing pattern

quit:

ldb char, iosl

b rism_addr, $ ; if char was received, restore it

ret

$eject
```
; This is a RAM test for the EV80C196KB board in its 'shipped' configuration.
; The RAM from 2000H to 27FFH is not mapped during diagnostics, and therefore,
; is not tested. The test alternates between incrementing and decrementing
; the test data on even and odd cycles of the test so that a nonrepetitive
; pattern is produced in memory.

.CSEG

loop:

   ; enable PWM
   ldb loc1, #01H

   ; clear data register
   clr ax

   ; clear error register
   clrb

   ; clear test count register
   clr dx

   ; starting address of RAM in diag. mode.
   ld bx, #2800H

loop:

   ; save test data
   stb al, [bx]

   ; check if it is saved, and point to next byte
   cmpb [bx]+, al
   bne failed

   ; if not, test failed
   failed:

   ; check if test count is even or odd
   bne

   ; if it is odd, decrement test data
   decb al

   ; if it is even, increment test data
   inc

   ; has end of RAM been reached by pointer?
   cmp bx, #8000H
   bne loop

   ; is not continue,
   bne loop

   ; else, return pointer to starting address
   inc bx

   ; count the test as successful
   inc dx

   ; show completion to user on LEDs
   inc ioport1

   ; PWM LED gets brighter as ioport1
   ldb pwm控制, ioport1
   ; PWM LED gets brighter as ioport1
   ; value gets bigger
   br loop

   ; go back for another cycle
failed:

   ; set error register
   ld cx, #0FFFFH

   ; end test
   br $

$eject
ERR LOC | OBJECT | LINE | SOURCE STATEMENT
--------|--------|-----|--------------------------
A280    |        | 628 | cseg at (offset + 2280H)
A280    |        | 629 | ;------------------------
A283    |        | 637 |.cb_loop:                
A286 C6201C | 638 | SET_BIT IOPORT1,7
A28C B2201D | 640 | stb ax,[bx]
A28F 27F2 | 641 | CLR_BIT IOPORT1,7
A2A0    |        | 643 | ldb (ax+1),[bx]
A2A0    |        | 644 | br cb_loop
A2A3    |        | 646 | cseg at (offset + 22A0H)
A2A6 C2201C | 647 | ;------------------------
A2AC A2201E | 648 | cycle_word:
A2AF 27F2 | 649 | ;------------------------
A2A3    |        | 650 | ; does alternate read and write operation on the word specified by bx.
A2A6 C2201C | 651 | ;------------------------
A2AC A2201E | 662 | ;
A2AF 27F2 | 663 | $eject
ERR LOC OBJECT

VDOO

9D00

VDOO

9D00

9D00 A1000036

9D04 C301222036

9D09 A1802036

9D12 A13B1D36

9D16 C301102036

9D1B C3013E2000

VDOV

9D04 C301202036

VDOV

9D0D C301202036

VDOV

9D12 C3013E2000

VDOV

9D16 C301102036

VDOV

9D1B C3013E2000

VDOV

LINE

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

SOURCE STATEMENT

user at (offset + 1D00H)

-------------

user_setup:

----------

; This code completes changing the board into user mode. The PLD on the board (U12) automatically remaps memory when code from this address range is fetched.

ld tempw, #rism_psw ; value for rism and initial user value

st tempw, USR_PSW ; store rism psw as initial user psw

ld tempw, #USER_PSW ; set up user pc

st tempw, USER_PC

ld tempw, #(break-offset)

st tempw, (trap-offset)[0] ; initialize trap vector

st zero, (nmi-offset)[0] ; initialize nmi vector

monitor_pause:

br monitor_pause ; wait for a command from the host

$eject
SOURCE STATEMENT

START USER:

; Flush the pause routine off the stack
SET_BIT RISM_STAT, RUN_FLAG
CLR_BIT RISM_STAT, TRAP_FLAG

; update running signal to host

;reset sp to overwrite RISM pc & psw.

push USER_PC
; with user pc & user psw values

EXIT_RISH

break:

; This routine is invoked by a TRAP instruction used for breakpointing,
; it operates somewhat like a STOP_USER instruction.

ENTER_RISH

; update running signal to host

STOP_USER:

; Stops "user" execution by setting up the stack to return to pause with
; all interrupts but serial i/o locked out.

set_user:

; remove users psw & pc from stack

pop USER_PSW
pop USER_PC

; and save

set_rism_idle:

push #(monitor_pause_offset), the new program counter & psw

CLR_BIT RISM_STAT, RUN_FLAG

; update running signal to host

EXIT_RISH

;
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>736</td>
<td>command_table:</td>
</tr>
<tr>
<td>9D5E</td>
<td></td>
<td>737</td>
<td>;----------------</td>
</tr>
<tr>
<td>9D5E</td>
<td>3D00</td>
<td>739</td>
<td>dcw (SET_DLE_FLAG - offset) ; 00</td>
</tr>
<tr>
<td>9D60</td>
<td>1300</td>
<td>740</td>
<td>dcw (exit - offset) ; 01</td>
</tr>
<tr>
<td>9D62</td>
<td>4200</td>
<td>741</td>
<td>dcw (TRANSMIT - offset) ; 02</td>
</tr>
<tr>
<td>9D64</td>
<td>1300</td>
<td>742</td>
<td>dcw (exit - offset) ; 03</td>
</tr>
<tr>
<td>9D66</td>
<td>5700</td>
<td>743</td>
<td>dcw (READ_BYTE - offset) ; 04</td>
</tr>
<tr>
<td>9D68</td>
<td>5C00</td>
<td>744</td>
<td>dcw (READ_WORD - offset) ; 05</td>
</tr>
<tr>
<td>9D6A</td>
<td>6100</td>
<td>745</td>
<td>dcw (READ_DOUBLE - offset) ; 06</td>
</tr>
<tr>
<td>9D6C</td>
<td>6A00</td>
<td>746</td>
<td>dcw (WRITE_BYTE - offset) ; 07</td>
</tr>
<tr>
<td>9D6E</td>
<td>6F00</td>
<td>747</td>
<td>dcw (WRITE_WORD - offset) ; 08</td>
</tr>
<tr>
<td>9D70</td>
<td>7400</td>
<td>748</td>
<td>dcw (WRITE_DOUBLE - offset) ; 09</td>
</tr>
<tr>
<td>9D72</td>
<td>7C00</td>
<td>749</td>
<td>dcw (LOAD_ADDRESS - offset) ; 0A</td>
</tr>
<tr>
<td>9D74</td>
<td>8100</td>
<td>750</td>
<td>dcw (INDIRECT_ADDRESS - offset) ; 0B</td>
</tr>
<tr>
<td>9D76</td>
<td>9000</td>
<td>751</td>
<td>dcw (READ_PSW - offset) ; 0C</td>
</tr>
<tr>
<td>9D78</td>
<td>AC00</td>
<td>752</td>
<td>dcw (WRITE_PSW - offset) ; 0D</td>
</tr>
<tr>
<td>9D7A</td>
<td>B300</td>
<td>753</td>
<td>dcw (READ_SP - offset) ; 0E</td>
</tr>
<tr>
<td>9D7C</td>
<td>BA00</td>
<td>754</td>
<td>dcw (WRITE_SP - offset) ; 0F</td>
</tr>
<tr>
<td>9D7E</td>
<td>8000</td>
<td>755</td>
<td>dcw (READ_PC - offset) ; 10</td>
</tr>
<tr>
<td>9D80</td>
<td>8600</td>
<td>756</td>
<td>dcw (WRITE_PC - offset) ; 11</td>
</tr>
<tr>
<td>9D82</td>
<td>231D</td>
<td>757</td>
<td>dcw (START_USER - offset) ; 12</td>
</tr>
<tr>
<td>9D84</td>
<td>451D</td>
<td>758</td>
<td>dcw (STOP_USER - offset) ; 13</td>
</tr>
<tr>
<td>9D86</td>
<td>C000</td>
<td>759</td>
<td>dcw (REPORT_STATUS - offset) ; 14</td>
</tr>
<tr>
<td>9D88</td>
<td>4E00</td>
<td>760</td>
<td>dcw (MONITOR_ESCAPE - offset) ; 15</td>
</tr>
</tbody>
</table>

| 761     |        |      |                  |
| 762     | $eject |      |                  |
ERR LOC OBJECT       LINE       SOURCE STATEMENT
     8000                        cseg at (offset + 0000H)
     8000                        ; ------------------------
     8000                        ; rism interrupt service routine
     8000                        ; ------------------------
     8000                        ; Control passes to this point when the rism gets a serial i/o interrupt
     8000                        ; from the host system.
     8000                        ;
     8000                        ; rism_isr:
     8000                        ;
     8001 B301021E36            ;
     8001                        ; ENTER_RISM
     8001                        ;
     8006 950436                ;
     8006                        ; ldb tempb, general_int[0] ; read uart interrupt status
     8006                        ;
     8009 DFOA                  ;
     8009                        ; be receive_ready
     800A B10136                ;
     800A                        ; ldb tempb, #01H        ; enable only receive fifo interrupt...
     800E C701011E36            ;
     800E                        ; stb tempb, gener_enabl[0] ; of the uart, mask all others
     8013                        ;
     8013                        ; exit:                   
     8013                        ;
     8015                        ;
     8015                        ; receive_ready:
     8015 AF01001R36            ;
     8015                        ; ldbhse tempw, txd_rxd[0] ; "char" is low byte of tempw
     801A 3B3803E7C420           ;
     801A                        ; bbc RISM_STAT,USER_MAP, not_user
     8020 38380F                ;
     8020                        ; bbs RISM_STAT, DLE_FLAG, force_load_data
     8023 991F36                ;
     8023                        ; cmpb char, #1FH          ; check if byte is a command
     8026 D90D                  ;
     8026                        ; hh load_data            ; commands are <= 1FH
     802A                        ;
     802A                        ; process_command:
     8028 643636                ;
     8028                        ; add tempw, tempw         ; convert "char" to word index
     802A A3775R1D36            ;
     802A                        ; ld tempw, (command_table-offset)[tempw]
     8030 E336                  ;
     8030                        ; br [tempw]               
     8037                        ;
     8037                        ; $eject
force_load_data:  
CLR_BIT RISM_STAT, DLE_FLAG
load_data:

set_DLE_flag:

set_bit RISM_STAT, DLE_FLAG

transmit:

send_data_byte

monitor_escape:

if RISM_DATA=1 then execute reset

cmp RISM_DATA, #01

bne exit

rst $ 

br $ 

$eject
ERR LOC OBJECT

8057 848 READ_BYTE:
849  
850 ; RISM_DATA:=byte at RISM_ADDR
851 ;

8057 B23430 852 ldb RISM_DATA, [RISM_ADDR]
853 EXIT_RISM
854 ;

805C 855 READ_WORD:
856  
857 ; RISM_DATA:=word at RISM_ADDR
858 ;

805C A25450 861 ld RISM_DATA, [RISM_ADDR]
862 EXIT_RISM
863 ;

8061 864 READ_DOUBLE:
865  
866 ; RISM_DATA:=double-word at RISM_ADDR
867 ;

8061 A23430 870 ld RISM_DATA, [RISM_ADDR]
871  
872 EXIT_RISM
873 ;

8064 A3340232 874  
875 ;

806A 876 WRITE_BYTE:
877  
878 ; byte at RISM_ADDR:=RISM_DATA
879 ; RISM_ADDR:=RISM_ADDR+1
880 ;

806A C63530 881 stb RISM_DATA, [RISM_ADDR]+
882 EXIT_RISM
883 ;

806F 884 WRITE_WORD:
885  
886 ; word at RISM_ADDR:=RISM_DATA
887 ; RISM_ADDR:=RISM_ADDR+2
888 ;

806F C23530 892 st RISM_DATA, [RISM_ADDR]+
893 EXIT_RISM
894 ;

8074 897 WRITE_DOUBLE:
898  
899 ; double-word at RISM_ADDR:=RISM_DATA
900 ;

8074 C23530 901 st RISM_DATA, [RISM_ADDR]+
902 EXIT_RISM
903 ;

8077 C23532 907 st (RISM_DATA+2), [RISM_ADDR]+
ERR LOC OBJECT

LINE SOURCE STATEMENT

807C LOAD_ADDRESS:
811 ;-------------
812 ; RISM_ADDR:=RISM_DATA
813
815 ld RISM_ADDR, RISM_DATA
816 EXIT_RISM
817
8081 INDIRECT_ADDRESS:
821 ;--------------
822 ; RISM_ADDR:=[RISM_ADDR]
823
825 ld RISM_ADDR, [RISM_ADDR]
826 EXIT_RISM
827
8086 WRITE_PC:
830 ;----------
831 ; user_pc:=RISM_DATA. (Assumes user is not running)
832
834 st RISM_DATA, USER_PC
835 EXIT_RISM
836
808D READ_PC:
840 ;--------
841 ; RISM_DATA:=user_pc
842
844 bbs RISM_STAT, RUN_FLAG, rpc_running
845
847 EXIT_RISM
848
8097 rpc_running:
849
850 ld RISM_DATA, 2{sp}
851 EXIT_RISM
852
855 $eject
SOURCE STATEMENT

READ_PSW:
\[ \text{; \ldots} \]
\[ \text{bbs RISM_STAT, RUN_FLAG, rpsw\_running} \]
\[ \text{ld RISM\_DATA, USER\_PSW} \quad ; \text{user is not running} \]
\[ \text{EXIT\_RISM} \]

WRITE_PSW:
\[ \text{; \ldots} \]
\[ \text{user\_psw:=RISM\_DATA} \quad \text{(Assumes user is not running)} \]
\[ \text{EXIT\_RISM} \]

READ_SP:
\[ \text{; \ldots} \]
\[ \text{add RISM\_DATA, sp, \#4} \quad ; \text{add four to account for PC and PSW...} \]
\[ \text{EXIT\_RISM} \quad ; \text{on the stack during this interrupt} \]

WRITE_SP:
\[ \text{; \ldots} \]
\[ \text{user\_sp:=RISM\_DATA} \quad \text{(Assumes user is not running)} \]
\[ \text{br set\_rism\_idle} \]

REPORT\_STATUS:
\[ \text{; \ldots} \]
\[ \text{report user status:} \]
\[ \text{stopped equ 0} \]
\[ \text{running equ 1} \]
\[ \text{trapped equ 2} \]
\[ \text{\ldots} \]
\[ \text{ld RISM\_DATA, \#running} \]
\[ \text{bbs RISM\_STAT, RUN\_FLAG, exit} \]
\[ \text{ld RISM\_DATA, \#trapped} \]
\[ \text{bbs RISM\_STAT, TRAP\_FLAG, exit} \]
\[ \text{ld RISM\_DATA, \#stopped} \]
\[ \text{EXIT\_RISM} \quad ; \text{else report stopped} \]
\[ \text{\ldots} \]
\[ \text{end} \]

\[ \text{\ldots} \]
\[ \text{\ldots} \]
<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_COMMAND</td>
<td>0002H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>AD_DONE</td>
<td>A002H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>AD_BREAK_HI</td>
<td>0003H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>ADRESULT_HI</td>
<td>0002H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>ADDR_CONTROL</td>
<td>1807H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>AH</td>
<td>001DH</td>
<td>REG ABS BYTE</td>
</tr>
<tr>
<td>AL</td>
<td>001CH</td>
<td>REG ABS BYTE</td>
</tr>
<tr>
<td>AROUND</td>
<td>A220H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>AX</td>
<td>001CH</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>BAUD_A_HI</td>
<td>1E01H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>BAUD_A_LO</td>
<td>1E00H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>BAUD_REG</td>
<td>000EH</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>BL</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>BREAK</td>
<td>9D3BH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>BX</td>
<td>0020H</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>BYTE_PROTECT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>CB_LOOP</td>
<td>A283H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>CHAR</td>
<td>0036H</td>
<td>REG ABS BYTE</td>
</tr>
<tr>
<td>CHIP_CONFIG</td>
<td>0A19H</td>
<td>CODE ABS BYTE</td>
</tr>
<tr>
<td>CLOCK_CONFIG</td>
<td>1E00H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>CLR_BIT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>COMMAND_TABLE</td>
<td>9D5EH</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>CW_LOOP</td>
<td>A2A3H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>CX</td>
<td>0022H</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>CYCLE_BYTE</td>
<td>A2C0H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>CYCLE_WORD</td>
<td>A2A0H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DEFINE_BIT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>DIAG_COMMAND</td>
<td>A13EH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DIAG_MODE</td>
<td>A130H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DIAG_PAUSE</td>
<td>A122H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DIAG_PAUSE_LOOP</td>
<td>A125H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DIAG_TABLE</td>
<td>A14CH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DIAGNOSTIC_FLAG</td>
<td>0007H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>DLEG_FLAG</td>
<td>0000H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>DREAD_PC</td>
<td>A1A6H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DREAD_PSW</td>
<td>A1B4H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DUSER_RUNNING</td>
<td>A1A6H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DRP_SW_RUNNING</td>
<td>A1BCH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DSTART_USER</td>
<td>A191H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DSTOP_USER</td>
<td>A178H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DUSER_PSW</td>
<td>A180H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DWHITE_PSW</td>
<td>003AH</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>DWRITE_PC</td>
<td>003CH</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>DWRITE_PSW</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>DWREG_PROTECT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>DWREG_RUNNING</td>
<td>A1C0H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DWREG_PSW</td>
<td>A1A1H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>A1C1H</td>
<td>CODE ABS ENTRY</td>
<td></td>
</tr>
<tr>
<td>NAME</td>
<td>VALUE</td>
<td>Attributes</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>DWRITE SP</td>
<td>A16EH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>DX</td>
<td>001EH</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>ENTER RISM</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>EV96</td>
<td>0013H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>EXIT RISM</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>EXIT RISM</td>
<td>0013H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>EXTERNAL INT</td>
<td>A006H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>EXTERNAL INT_PIN</td>
<td>A03AH</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>FAILED</td>
<td>A233H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>FL_LOOP1</td>
<td>A1DDH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>FL_WAIT1</td>
<td>A1D7H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>FL_WAIT2</td>
<td>A1E3H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>FL_WAIT3</td>
<td>A1DDH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>FORCE_LOAD_DATA</td>
<td>8032H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>GENER_ENABLE</td>
<td>1E01H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>GENERAL_INT</td>
<td>1E02H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>HERE</td>
<td>A21EH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>HSI_DATA</td>
<td>A004H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>HSI_ENTRY 4</td>
<td>A034H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>HSI_FIFO_FULL</td>
<td>A03CH</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>HSI_MODE</td>
<td>0003H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>HSI_STATUS</td>
<td>0006H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>HSI_TIME</td>
<td>0004H</td>
<td>NULL ABS WORD</td>
</tr>
<tr>
<td>HSI_ZERO</td>
<td>A008H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>HSO_COMMAND</td>
<td>0006H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>HSO_EVENT</td>
<td>A006H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>HSO_TIME</td>
<td>0004H</td>
<td>NULL ABS WORD</td>
</tr>
<tr>
<td>INDIRECT_ADDRESS</td>
<td>8081H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>INT_MASK</td>
<td>0008H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>INT_PENDING</td>
<td>0009H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>INVVALID_OPCODE</td>
<td>A012H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>IO_MODE</td>
<td>1E04H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>IOC0</td>
<td>0015H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>IOC1</td>
<td>0016H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>IOPORT1</td>
<td>000FH</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>IOPORT2</td>
<td>0010H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>IOS0</td>
<td>0015H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>IOS1</td>
<td>0016H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>LINK_CONFIG</td>
<td>1E03H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>LINE_STATUS</td>
<td>1E05H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>LOAD_ADDRESS</td>
<td>B41CH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>LOAD_DATA</td>
<td>8035H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>LAXUF</td>
<td>A20FH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>MEM_TST</td>
<td>A200H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>MEM_TST</td>
<td>1E04H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>MODEM_STATS</td>
<td>1E05H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>MONITOR.Escape</td>
<td>804EH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>MONITOR_PAUSE</td>
<td>9D20H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>NMI</td>
<td>A03EH</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>NOT_USER</td>
<td>A0E4H</td>
<td>CODE ABS ENTRY</td>
</tr>
</tbody>
</table>
## MCS-96 MACRO ASSEMBLER EV96

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET</td>
<td>8000H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>PORT0</td>
<td>0000H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>PROCESS COMMAND</td>
<td>8028H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>PWM CONTROL</td>
<td>0017H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>QUIT</td>
<td>A1F5H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ BYTE</td>
<td>8057H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ_DOUBLE</td>
<td>8061H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ PC</td>
<td>8080H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ_PSW</td>
<td>8090H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ SP</td>
<td>80B3H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>READ.Word</td>
<td>805CH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RECEIVE READY</td>
<td>8015H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>REPORT_STATUS</td>
<td>800CH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RESET_VECTOR</td>
<td>A080H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RISM_ADDR</td>
<td>0034H</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>RISM_DATA</td>
<td>0030H</td>
<td>REG ABS LONG</td>
</tr>
<tr>
<td>RISM_ISR</td>
<td>0000H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RISM_PSW</td>
<td>0000H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>RISM_STAT</td>
<td>0038H</td>
<td>REG ABS BYTE</td>
</tr>
<tr>
<td>RPC_RUNNING</td>
<td>8097H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RFSW_RUNNING</td>
<td>80A7H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>RUN_FLAG</td>
<td>0002H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>RUNNING</td>
<td>0001H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>SBUF</td>
<td>0007H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>SEND_DATA_BYTE</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>SERIAL_PORT</td>
<td>A000H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>SERIAL_RXD</td>
<td>A032H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>SERIAL_TXD</td>
<td>A030H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>SET_BIT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>SET_DLE_FLAG</td>
<td>8030H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>SET_RISN</td>
<td>9040H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>SOFTWARE_TIMER</td>
<td>A00AH</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>SP</td>
<td>0016H</td>
<td>NULL ABS WORD</td>
</tr>
<tr>
<td>SCON</td>
<td>0011H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>SPOSTAT</td>
<td>0011H</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>START_USER</td>
<td>9022H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>STOP_USER</td>
<td>9045H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>STOPPED</td>
<td>0000H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>TENV</td>
<td>0036H</td>
<td>REG ABS BYTE</td>
</tr>
<tr>
<td>TEMPOW</td>
<td>0036H</td>
<td>REG ABS WORD</td>
</tr>
<tr>
<td>TIMER1_OV</td>
<td>A000H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>TIMER2</td>
<td>0000H</td>
<td>NULL ABS WORD</td>
</tr>
<tr>
<td>TIMER2_C</td>
<td>0000H</td>
<td>NULL ABS WORD</td>
</tr>
<tr>
<td>TIMER2_CAPTURE</td>
<td>0036H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>TIMER2_OV</td>
<td>A000H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>TRANSMIT</td>
<td>8042H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>TRAP</td>
<td>A010H</td>
<td>CODE ABS WORD</td>
</tr>
<tr>
<td>TRAP_FLAG</td>
<td>0001H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>TRAPPED</td>
<td>0002H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>NAME</td>
<td>VALUE</td>
<td>ATTRIBUTES</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>TXD_RXD</td>
<td>1600H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>UART</td>
<td>1500H</td>
<td>DATA ABS BYTE</td>
</tr>
<tr>
<td>USER MAP</td>
<td>0003H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>USER PC</td>
<td>2020H</td>
<td>DATA ABS WORD</td>
</tr>
<tr>
<td>USER_PSW</td>
<td>2022H</td>
<td>DATA ABS WORD</td>
</tr>
<tr>
<td>USER_SETUP</td>
<td>9000H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WATCHDOG</td>
<td>000AH</td>
<td>NULL ABS BYTE</td>
</tr>
<tr>
<td>WORD_PROTECT</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>WRITE_BYTE</td>
<td>806AH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WRITE_DOUBLE</td>
<td>8074H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WRITE_PC</td>
<td>8086H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WRITE_PSW</td>
<td>808CH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WRITE_SP</td>
<td>80B0H</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>WRITE_WORD</td>
<td>806FH</td>
<td>CODE ABS ENTRY</td>
</tr>
<tr>
<td>ZERO</td>
<td>0000H</td>
<td>NULL ABS WORD</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
Appendix D.

Timing Analysis
Timing analysis of the EV80C196KB board.

All values used are based on the 80C196KB operating at 12MHz. They are taken from the October 1988 version of the 80C196KB data sheet, Intel order number 270634-001.

80C196KB A.C. Characteristics

$T_{avv} = 81 \text{ ns MAX.}$

$T_{avv(WAIT)} = 11 \text{ ns (AC373 Dn to On Tplh MAX)} + 35 \text{ ns (PAL/EPLD Tpd MAX)}$

$\quad + 9 \text{ ns (AC08 Tplh MAX)} + 12 \text{ ns (AC112 RES to Q Tphl MAX)}$

$\quad = 67 \text{ ns.}$

$T_{llv}$ is irrelevant in this design.

$T_{clyx} = 53 \text{ ns MAX.}$

$T_{clyx(WAIT)} = 10 \text{ ns (AC112 CLOCK to Q Tplh MAX).}$

$T_{llx}$ is irrelevant in this design.

$T_{avgv} = 81 \text{ ns MAX.}$

$T_{clyx(BUSWIDTH)} = 11 \text{ ns (AC373 Dn to On Tplh MAX)} + 35 \text{ ns (PAL/EPLD Tpd MAX)}$

$\quad = 46 \text{ ns.}$

$T_{llgv}$ is irrelevant in this design.

$T_{clgx}$ is irrelevant in this design.

$T_{avdv} = 183 \text{ ns MAX, for zero wait states.}$

$T_{avdv(ROMsim)} = 11 \text{ ns (AC373 Dn to On Tplh MAX)} + 35 \text{ ns (PAL/EPLD Tpd MAX)}$

$\quad + 100 \text{ ns (RAM Tco1 MAX)}$

$\quad = 146 \text{ ns.}$

$T_{avdv} = 349 \text{ ns MAX, for one wait state.}$

$T_{avdv(EPROM)} = 11 \text{ ns (AC373 Dn to On Tplh MAX)} + 35 \text{ ns (PAL/EPLD Tpd MAX)}$

$\quad + 200 \text{ ns (EPROM Tco MAX)}$

$\quad = 246 \text{ ns.}$

$T_{avdv} = 516 \text{ ns MAX, for two wait states.}$

$T_{avdv(UART)} = 11 \text{ ns (AC373 Dn to On Tplh MAX)} + 35 \text{ ns (PAL/EPLD Tpd MAX)}$

$\quad + 288 \text{ ns (UART Tavrl MIN + Trldv MAX)}$

$\quad = 334 \text{ ns.}$
Trldv = 60 ns MAX, for zero wait states.
Trldv(ROMsim) = 50 ns (RAM Toe MAX).

Trldv = 226 ns MAX, for one wait state.
Trldv(EPROM) = 75 ns (EPROM Toe MAX).

Trldv = 393 ns MAX, for two wait states.
Trldv(UART) = 281 ns (UART Trldv MAX).

Tcoldv is irrelevant in this design.

Trhdz = 63 ns MAX.
Trhdz(ROMsim) = 35 ns (RAM Tohz MAX).
Trhdz(EPROM) = 55 ns (EPROM Tdh MAX).
Trhdz(UART) = 40 ns (UART Trhdz MAX).

Trxdx = 0 ns MIN.
Trxdx(ROMsim) = 0 ns (RAM Tohz MIN).
Trxdx(EPROM) = 0 ns (EPROM Toh MIN).
Trxdx(UART) is not specified.

Txhch is irrelevant in this design.

Tclcl = 166 ns.
Tclcl(WAIT) = 55 ns (PAL/EPLD Tp MIN).
\[= 10 \text{ ns (AC112 1/Fmax MIN)}\]

Tchcl = 73 ns MIN.
Tchcl(WAIT) = 25 ns (PAL/EPLD Tco MAX) + 35 ns (PAL/EPLD Tpd MAX)
\[+ 4 \text{ ns (AC112 Tsu MIN)}\]
\[= 64 \text{ ns.}\]
or = 25 ns (PAL/EPLD Tco MAX) + 35 ns (PAL/EPLD Tpd MAX)
\[+ 8 \text{ ns (AC08 Tplh MAX)} + 2 \text{ ns (AC112 Trem MIN)}\]
\[= 70 \text{ ns.}\]

Tcllh is irrelevant in this design.

Tllch is irrelevant in this design.

Tlhlh is irrelevant in this design.

Tlhl = 73 ns MIN.
Tlhl(A0-A15) = 5 ns (AC373 Tw MIN).
Tavl1 = 68 ns MIN.
Tavl1(A0-A15) = 5 ns (AC373 Ts MIN).
Tavl1(WAIT) = 11 ns (AC373 Dn to On Tplh MAX) + 35 ns (PAL/EPLD Tpd MAX)
+ 8 ns (AC00 Tphi MIN) + 5 ns (AC112 Tw MIN)
= 59 ns.
Tavl1(BHE#) = 11 ns (AC14 Tplh MAX) + 4 ns (AC112 Tsu MIN)
= 15 ns.

Tllax = 43 ns MIN.
Tllax(A0-A15) = 0 ns (AC373 Th MIN).
Tllax(BHE#) = 0 ns (AC112 Th MIN).

Tllrl = 43 ns MIN.
Tllrl(UART) = 7 ns (UART Tavlrl MIN).

Trcl is irrelevant in this design.

Trlh = 411 ns MIN, for two wait states.
Trlh(UART) = 281 ns (UART Trlh MIN).

Trhll = 83 ns MIN.
Trhll(STALE) = 9 ns (74AC08 Tplh MAX) + 3 ns (74AC112 Trem MIN)
= 12 ns.

Tllwl = 73 ns MIN.
Tllwl(UART) = 7 ns (UART Tavlwl MIN).

Tclwl is irrelevant in this design.

Tqvwh = 60 ns MIN, for zero wait states.
Tqvwh(ROMsim) = 40 ns (RAM Tdw MIN).

Tqvwh = 393 ns MIN, for two wait states.
Tqvwh(UART) = 90 ns (UART Tdvwh MIN).

Tchwh is irrelevant in this design.

Twlwh = 53 ns MIN, for zero wait states.
Twlwh(ROMsim) = 50 ns (RAM Twp MIN).

Twlwh = 386 ns MIN, for two wait states.
Twlwh(UART) = 231 ns (UART Twlwh MIN).
$T_{\text{whqx}} = 73 \text{ ns MIN.}$  
$T_{\text{whqx}}(\text{ROMsim}) = 9 \text{ ns (74AC32 Tplh MAX) } + 0 \text{ ns (RAM Tdh MIN)}$  
$= 9 \text{ ns.}$  
$T_{\text{whqx}}(U14) = 0 \text{ ns (RAM Tdh MIN).}$  
$T_{\text{whqx}}(\text{UART}) = 12 \text{ ns (UART Twhdx MIN).}$

$T_{\text{whlh}} = 73 \text{ ns MIN.}$  
$T_{\text{whlh}}(\text{ROMsim}) = 9 \text{ ns (74AC32 Tplh MAX) } + 0 \text{ ns (RAM Twr MIN)}$  
$= 9 \text{ ns.}$  
$T_{\text{whlh}}(\text{UART}) = 0 \text{ ns (UART Twhax MIN).}$  
$T_{\text{whlh}}(\text{STALE}) = 9 \text{ ns (74AC08 Tplh MAX) } + 3 \text{ ns (74AC112 Trem MIN)}$  
$= 12 \text{ ns.}$

$T_{\text{whbx}}$ is irrelevant in this design.
Appendix E.

Programmable Logic Equations
Doug Yoder
Intel
January 19, 1989
EV80C196KB 002
5AC312

Generates mapping signals for the target processor on the 80C196KB evaluation board.
OPTIONS: TURBO=ON PART: 5AC312

% Input declarations %

INPUTS:  CLOCKOUT,  % MCS96 system CLOCKOUT  
         STALE@2,  % STretched MCS96 Address Latch Enable  
         nHLDA@3,  % 80C196KB Hold Acknowledge  
         A8@4,  % MCS96 latched A8 - A15  
         A9@5,  %  
         A10@6,  %  
         A11@7,  %  
         A12@8,  %  
         A13@9,  %  
         A14@10,  %  
         A15@11,  %  
         nRESET@13  % MCS96 RESET pin  

% Output declarations %

OUTPUTS:  nCS510@14,  % 0V => enable uart, U20  
          nCE2@15,  % 0V => enable U14 memory  
          nBUSWIDTH@16,  % 0V => put processor in 8 bit mode  
          SB0@17,  % wait-state counter bit 0  
          SB1@18,  % wait-state counter bit 1  
          nWAIT@19,  % 0V => hold MCS96 in wait_state  
          SB2@20,  % wait-state counter bit 2  
          nCEO@21,  % 0V => enable U1 and U8 memory  
          nCE1@22,  % 0V => enable U6 and U13 memory  
          MAP@23  % 5V => map RAM as romsim  

% I/O Architecture declarations %

NETWORK:
  MAP,MAP = RORF(MAPd, CLOCKOUT, RESET, GND, VCC)
  nWAIT = CONF(nWAITd, VCC)
  nCS510 = COCF(UART, VCC)
  nCE2 = COCF(EIFROM, VCC)
  nCE1 = CONF(RAM, VCC)
  nCEO = CONF(EIFROM, VCC)
  nBUSWIDTH = CONF(nBWd, VCC)
% Intermediate variable definitions %

EQUATIONS:

\[
\begin{align*}
\text{RESET} &= \neg \text{nRESET}; \\
\text{HLDA} &= \neg \text{nHLDA}; \\
\text{MAP}^d &= \text{MAP} + (\text{RANGE3} \times \neg \text{STALE}); \\
\text{EPROM}' &= (\neg \text{MAP} \times \text{RANGE6}) \\
&\quad + \text{RANGE1} \\
&\quad + \text{RANGE4}; \\
\text{RAM}' &= (\text{MAP} \times \text{RANGE6}) \\
&\quad + \text{RANGE7}; \\
\text{EEPROM}' &= \text{RANGE8}; \\
\text{UART}' &= \text{RANGE5}; \\
\text{OPEN0} &= \text{RANGE2} \\
&\quad + \text{RANGE10}; \\
\text{OPEN1} &= \text{RANGE9}; \\
\text{nBWd}' &= \neg \text{EEPROM} + \neg \text{UART}; \\
\text{WAIT}_1 &= \text{STALE} \times \neg \text{HLDA} \times (\text{WAIT}_2 + \neg \text{EPROM} + \text{OPEN1}); \\
\text{WAIT}_2 &= \text{STALE} \times \neg \text{HLDA} \times (\text{WAIT}_3 + \neg \text{UART}); \\
\text{WAIT}_3 &= \text{WAIT}_4; \\
\text{WAIT}_4 &= \text{WAIT}_5; \\
\text{WAIT}_5 &= \text{WAIT}_6; \\
\text{WAIT}_6 &= \text{WAIT}_7; \\
\text{WAIT}_7 &= \text{GND}; \\
\text{nWAITd} &= \neg \text{WAIT};
\end{align*}
\]
% Address Range Equations %

RANGE1 = !A15 * !A14 * !A13 * !A12 * !A11 * !A10 * !A9 * !A8; % 0000-00FF %

RANGE2 = !A15 * !A14 * !A13 * A12 * !A10 * !A8
+ !A15 * !A14 * !A13 * !A10 * !A9 * A8
+ !A15 * !A14 * !A13 * !A12 * A10
+ !A15 * !A14 * !A13 * A11 * !A9 * !A8
+ !A15 * !A14 * !A13 * A12 * !A11
+ !A15 * !A14 * !A13 * !A12 * !A9;

RANGE3 = !A15 * !A14 * !A13 * A12 * !A9
+ !A15 * !A14 * !A13 * A12 * !A10
+ !A15 * !A14 * !A13 * !A12 * !A11;

RANGE4 = !A15 * !A14 * !A13 * A12 * !A11 * A10 * !A9 * !A8; % 1D00-1DFF %

RANGE5 = !A15 * !A14 * !A13 * A12 * !A11 * A10 * !A9 * !A8; % 1E00-1EFF %

RANGE6 = !A15 * !A14 * A13 * !A12 * !A11;

RANGE7 = !A15 * !A14 * A13 * A12
+ !A15 * !A14 * A13 * A11
+ !A15 * A14 * !A13;

RANGE8 = !A15 * A14 * A13; % 6000-7FFF %

RANGE9 = A15 * !A14; % 8000-BFFF %

RANGE10 = A15 * A14; % C000-FFFF %
% State machine %

MACHINE: WAIT_STATE
CLOCK: CLOCKOUT
CLEAR: RESET

STATES: 
  [ SB2 SB1 SB0 ] ASYNC_START [ 0 0 0 ]
  HOLD_2 [ 0 0 1 ]
  HOLD_3 [ 0 1 1 ]
  HOLD_4 [ 1 1 1 ]
  HOLD_5 [ 1 1 0 ]
  HOLD_6 [ 1 0 0 ]
  HOLD_7 [ 1 0 1 ]

REMOVE HOLD [ 0 1 0 ]

ASYNC_START: IF WAIT_1 & !WAIT_2 THEN REMOVE_HOLD
  IF WAIT_2 THEN HOLD_2
  ASSERT: IF WAIT_1 THEN WAIT

HOLD_2: IF WAIT_3 THEN HOLD_3
  REMOVE_HOLD
  ASSERT: WAIT

HOLD_3: IF WAIT_4 THEN HOLD_4
  REMOVE_HOLD
  ASSERT: WAIT

HOLD_4: IF WAIT_5 THEN HOLD_5
  REMOVE_HOLD
  ASSERT: WAIT

HOLD_5: IF WAIT_6 THEN HOLD_6
  REMOVE_HOLD
  ASSERT: WAIT

HOLD_6: IF WAIT_7 THEN HOLD_7
  REMOVE_HOLD
  ASSERT: WAIT

HOLD_7: REMOVE_HOLD
  ASSERT: WAIT

REMOVE_HOLD: ASYNC_START

ENDS
Name KBBUSCON;
Partno EV80C196KB;
Revision 01; Date 1/18/89;
Designer Doug Yoder;
Company Intel ECO;
Assembly 80C196KB evaluation board;
Location U12;
Device 22V10;

RESET Generate mapping signals for the target processor on the 80C196KB evaluation board.
Allowable Target Device Types: 22V10

** Inputs **

** Outputs **

 Declarations and Intermediate Variable Definitions **

FIELD memaddr = [a15..8];
eprom = (!MAP & memaddr:[2000..27FF]) # memaddr:[0..FF] # memaddr:[1D00..1DFF];
ram = (MAP & memaddr:[2000..27FF]) # memaddr:[2800..5FFF];
eeprom = memaddr:[6000..7FFF];
uart = memaddr:[1E00..1EFF];
open0 = memaddr:[100..1CFF] # memaddr:[C000..FFFF];
open1 = memaddr:[8000..BFFF];
bw = eeprom # uart;
wait_1 = STALET !HLDA & (wait_2 # eprom # open1);
wait_2 = STALET !HLDA & (wait_3 # uart);
wait_3 = wait_4;
wait_4 = wait_5;
wait_5 = wait_6;
wait_6 = wait_7;
wait_7' = 'b'0;
FIELD state_count = [state_bit_0..2];

$DEFINE async_start 'b'000
$DEFINE hold_2 'b'001
$DEFINE hold_3 'b'011
$DEFINE hold_4 'b'111
$DEFINE hold_5 'b'110
$DEFINE hold_6 'b'100
$DEFINE hold_7 'b'101
$DEFINE remove_hold 'b'010

/** Wait-State Machine **/
SEQUENCE state_count
{
PRESENT async_start
IF wait_1 OUT WAIT;
IF wait_1 & !wait_2 NEXT remove_hold;
IF wait_2 NEXT hold_2;
DEFAULT NEXT async_start;

PRESENT hold_2
OUT WAIT;
IF wait_3 NEXT hold_3;
DEFAULT NEXT remove_hold;

PRESENT hold_3
OUT WAIT;
IF wait_4 NEXT hold_4;
DEFAULT NEXT remove_hold;
PRESENT hold_4
   OUT WAIT;
   IF wait_5
      DEFAULT
      NEXT hold_5;
      NEXT remove_hold;
PRESEN hold_5
   OUT WAIT;
   IF wait_6
      DEFAULT
      NEXT hold_6;
      NEXT remove hold;
PRESEN hold_6
   OUT WAIT;
   IF wait_7
      DEFAULT
      NEXT hold_7;
      NEXT remove_hold;
PRESEN hold_7
   OUT WAIT;
   NEXT remove_hold;
PRESEN remove_hold
   NEXT async_start;
} */ Logic Equations */

MAP.D = (memaddr: [1000..1DFF] & !STALE) # MAP;
MAP.AR = RESET;
MAP.SP = 'b'0;
MAP.OE = 'b'1;

state_bit_0.AR = RESET;
state_bit_0.SP = 'b'0;
state_bit_0.OE = 'b'1;
state_bit_1.AR = RESET;
state_bit_1.SP = 'b'0;
state_bit_1.OE = 'b'1;
state_bit_2.AR = RESET;
state_bit_2.SP = 'b'0;
state_bit_2.OE = 'b'1;

CE0 = eprom;
CE1 = ram;
CE2 = eeprom;
CS510 = uart;

BUSWIDTH = bw;
Appendix F.

Standard Memory-I/O Connector
for
EvalBoards
### General Purpose Memory Expansion Connector

**Compatibility with Other Intel Evaluation Boards**

2x30 Pin Molex 39-51-2604 or Equiv.

<table>
<thead>
<tr>
<th>EV80C51FB</th>
<th>EV80C196KB</th>
<th>EV80C196</th>
<th>EV80C186</th>
<th>EV80C196KB</th>
<th>EV80C51FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCC</td>
<td>VCC</td>
<td>VCC</td>
<td>1</td>
<td>2</td>
<td>VCC</td>
</tr>
<tr>
<td>Addr 0</td>
<td>Addr 0</td>
<td>Addr 0</td>
<td>3</td>
<td>4</td>
<td>Addr/Data 0</td>
</tr>
<tr>
<td>Addr 1</td>
<td>Addr 1</td>
<td>Addr 1</td>
<td>5</td>
<td>6</td>
<td>Addr/Data 1</td>
</tr>
<tr>
<td>Addr 2</td>
<td>Addr 2</td>
<td>Addr 2</td>
<td>7</td>
<td>8</td>
<td>Addr/Data 2</td>
</tr>
<tr>
<td>Addr 3</td>
<td>Addr 3</td>
<td>Addr 3</td>
<td>9</td>
<td>10</td>
<td>Addr/Data 3</td>
</tr>
<tr>
<td>Addr 4</td>
<td>Addr 4</td>
<td>Addr 4</td>
<td>11</td>
<td>12</td>
<td>Addr/Data 4</td>
</tr>
<tr>
<td>Addr 5</td>
<td>Addr 5</td>
<td>Addr 5</td>
<td>13</td>
<td>14</td>
<td>Addr/Data 5</td>
</tr>
<tr>
<td>Addr 6</td>
<td>Addr 6</td>
<td>Addr 6</td>
<td>15</td>
<td>16</td>
<td>Addr/Data 6</td>
</tr>
<tr>
<td>Addr 7</td>
<td>Addr 7</td>
<td>Addr 7</td>
<td>17</td>
<td>18</td>
<td>Addr/Data 7</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>VSS</td>
<td>19</td>
<td>20</td>
<td>VSS</td>
</tr>
<tr>
<td>Addr 8</td>
<td>Addr 8</td>
<td>Addr 8</td>
<td>21</td>
<td>22</td>
<td>Addr/Data 8</td>
</tr>
<tr>
<td>Addr 9</td>
<td>Addr 9</td>
<td>Addr 9</td>
<td>23</td>
<td>24</td>
<td>Addr/Data 9</td>
</tr>
<tr>
<td>Addr 10</td>
<td>Addr 10</td>
<td>Addr 10</td>
<td>25</td>
<td>26</td>
<td>Addr/Data 10</td>
</tr>
<tr>
<td>Addr 11</td>
<td>Addr 11</td>
<td>Addr 11</td>
<td>27</td>
<td>28</td>
<td>Addr/Data 11</td>
</tr>
<tr>
<td>Addr 12</td>
<td>Addr 12</td>
<td>Addr 12</td>
<td>29</td>
<td>30</td>
<td>Addr/Data 12</td>
</tr>
<tr>
<td>Addr 13</td>
<td>Addr 13</td>
<td>Addr 13</td>
<td>31</td>
<td>32</td>
<td>Addr/Data 13</td>
</tr>
<tr>
<td>Addr 14</td>
<td>Addr 14</td>
<td>Addr 14</td>
<td>33</td>
<td>34</td>
<td>Addr/Data 14</td>
</tr>
<tr>
<td>Addr 15</td>
<td>Addr 15</td>
<td>Addr 15</td>
<td>35</td>
<td>36</td>
<td>Addr/Data 15</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>VSS</td>
<td>37</td>
<td>38</td>
<td>VSS</td>
</tr>
<tr>
<td>N.C.</td>
<td>CLKOUT</td>
<td>CLK</td>
<td>39</td>
<td>40</td>
<td>VSS</td>
</tr>
<tr>
<td>PSEN/RD</td>
<td>RD#</td>
<td>RD#</td>
<td>41</td>
<td>42</td>
<td>WR#</td>
</tr>
<tr>
<td>N.C./TP6</td>
<td>BREQ#</td>
<td>ES#</td>
<td>43</td>
<td>44</td>
<td>BHE#</td>
</tr>
<tr>
<td>ALE</td>
<td>ALE</td>
<td>ALE</td>
<td>45</td>
<td>46</td>
<td>SRDY</td>
</tr>
<tr>
<td>N.C./TP7</td>
<td>NMI</td>
<td>IO#</td>
<td>47</td>
<td>48</td>
<td>DRQ0</td>
</tr>
<tr>
<td>RESET#</td>
<td>RESERET#</td>
<td>RESERET#</td>
<td>49</td>
<td>50</td>
<td>INTO</td>
</tr>
<tr>
<td>PAL Disable#</td>
<td>Note 2</td>
<td>T0OUT</td>
<td>51</td>
<td>52</td>
<td>TOIN</td>
</tr>
<tr>
<td>N.C.</td>
<td>HLDA#</td>
<td>HLDA</td>
<td>53</td>
<td>54</td>
<td>HOLD</td>
</tr>
<tr>
<td>12VDC</td>
<td>-12VDC</td>
<td>-12V</td>
<td>55</td>
<td>56</td>
<td>+12VDC</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>VSS</td>
<td>57</td>
<td>58</td>
<td>VSS</td>
</tr>
<tr>
<td>VCC</td>
<td>VCC</td>
<td>VCC</td>
<td>59</td>
<td>60</td>
<td>VCC</td>
</tr>
</tbody>
</table>

**Note 1**

N.C = No Connect

N.C./TPx = No Connect, but routed to an on-board test point for the user.

**Note 2:**

Pin 51 of the EV80C196KB will be connected to U12 pin 20 on future revisions of this board.
Appendix G.

Sample Session
This list file was produced by using the command "list demo.lst" before invoking demo.log with the command "include demo.log" as described below. This list file can be used to compare to the screen of your own PC while you are running demo.log.

;---List file opened on 01/24/1989 at 16:43:15
*include demo.log
;---INCLUDE FILE OPEN
*
*; This is a demo of some of the features of iECM-96 for use with the
*; EV80C196KB board. In order to run the demo, place the software disk in a
*; drive. Then select that drive by typing "A:" or "B:," whichever corresponds
*; to that drive, and a carriage return. Type "ECM96" and carriage return.
*; At the asterisk prompt type "INCLUDE DEMO.LOG" and carriage return.
*;
*; For additional information, please see the EV80C196KB Microcontroller
*; Evaluation Board USER'S MANUAL.
*;
*pause
;  Hit the space bar to continue...
*;
*; This command loads 96KBDEMO.OBJ from disk.
*;
*load 96kbdemo.obj
;
;  mod name is: IDFM096KBI
;  mod date stamp is: 01/24/89 16:34:47
*;
*pause
;  Hit the space bar to continue...
*;
*dasm 2080,8 ; This disassembles 8 lines of code starting at 2080H
   | RESET_VECTOR:
   2080:  A1000118  |  LD     18,#0100
   2084:  011C      |  CLR    AX
   2086:  0120      |  CLR    CX
   2088:  0122      |  CLR    DX
   208A:  B10116    |  LDB    16,#01
   208D:  110F      |  CLRB   IPORT1
   208F:  1117      |  CLRB   17
   2091:  A1BF201E   |  LD     BX,#20BF
*pause
;  Hit the space bar to continue...
*;
*pc ; This displays the current value of the Program counter.
  PC=RESET_VECTOR
*;
*; To change the Program Counter use "pc = 2080<cr>".
*;
*pause
; Hit the space bar to continue...
*;
*go from 2080 forever ; This command clears all breakpoints and executes code.
*;
>; The LED's for I/O Port 1 should be incrementing regularly.
*;
>pause
; Hit the space bar to continue...
*;
>dasm .past,8 ; The disassembler and all other memory read commands can be...
; | PAST:
; 20A6: 8900801E  | CMP  BX,#8000
; 20AA: D7E9      | JNE  LOOP
; 20AC: A1BF201E  | LD   BX,#20BF
; 20B0: 0722      | INC  N
; 20B2: 170F      | INCB IOPORT1
; 20A4: R00F17    | LDR  17,IOPORT1
; 20B7: 27DC      | SJMP LOOP
; | FAILED:
; 20B9: A1FFFF20  | LD   CX,#0FFFF
;
>; used while code is running on the board.
;
>;pause
; Hit the space bar to continue...
*;
>asm 20b2 ; start assembling code at address 20b2H, see disassembly listing.
; | Single Line Assembler activated, exit with "end" directive
; 20B2H: decb .ioprt1
; 20B4H: end
>pause
; Hit the space bar to continue...
*;
>; The LED's for I/O Port 1 should now be decrementing.
*;
>; Note that not only is there an assembler, it and all other memory modifying commands can be used while the board is executing user code. However, use caution when modifying code while it is running, the resulting code may cause errors due to variable length instructions.
*;
>pause
Halting...

*dasm .loop,9

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2095</td>
<td>C61E1C</td>
<td>STB AL, [1E]</td>
</tr>
<tr>
<td>2098</td>
<td>9A1F1C</td>
<td>CMPB AL, [1E]+</td>
</tr>
<tr>
<td>209B</td>
<td>D71C</td>
<td>JNE FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209D</td>
<td>382204</td>
<td>JBS 22,00, BACK</td>
</tr>
<tr>
<td>20A0</td>
<td>171C</td>
<td>INCB AL</td>
</tr>
<tr>
<td>20A2</td>
<td>2002</td>
<td>SJMP PAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20A4</td>
<td>151C</td>
<td>DECB AL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20A6</td>
<td>8900801E</td>
<td>CMP BX, #8000</td>
</tr>
<tr>
<td>20AA</td>
<td>D7E9</td>
<td>JNE LOOP</td>
</tr>
</tbody>
</table>

*pause

Hit the space bar to continue...

*go from 2080 till 20a6; This go command sets a breakpoint[0] = 20a6H.

*pause

Hit the space bar to continue...

*pc; Code has stopped at the breakpoint! Note that 20a6 has not executed yet.

PC=PAST

*pause

Hit the space bar to continue...

*br ; This command displays all breakpoints, 20a6 has been set.

BREAKPOINT[0] = PAST

*pause

Hit the space bar to continue...

*br[0]=0 ; This command clears breakpoint[0].

*pause

Hit the space bar to continue...

*br ; As can be shown.

NO BREAKPOINTS ARE ACTIVE

*pause

Hit the space bar to continue...

*br[0]=20a6; This command sets breakpoint[15] = 20a6.

*pause

Hit the space bar to continue...

*br ; Set?

BREAKPOINT[15] = PAST

*pause
; Hit the space bar to continue...
*
*; This concludes the demo, we hope you enjoy using the EV80C196KB board.
*
*pause
; Hit the space bar to continue...
*
*; Type "QUIT" and carriage return to exit iECM-96.
*
*quit
## NORTH AMERICAN SALES OFFICES

**ALABAMA**
- Intel Corp., 600 Boulevard South, Suite 104-L, Huntsville 35802
  - Tel: (205) 950-3521
  - FAX: (205) 950-3521
- Intel Corp., 410 North 44th Street, Suite 500, Birmingham 35212
  - Tel: (205) 871-4500
  - FAX: (205) 871-4500
- Intel Corp., Phoenix 85008
  - Tel: (602) 287-3588
  - FAX: (602) 2444488

**ARIZONA**
- Intel Corp., 12515 Vanowen Street, Suite 116, Canoga Park 91303
  - Tel: (818) 750-8500
  - FAX: (818) 750-8500
- Intel Corp., 145 Sierra Gato Plaza, Suite 280C, San Diego 92123
  - Tel: (619) 498-2901
  - FAX: (619) 498-2901

**CALIFORNIA**
- Intel Corp., 145 Sierra Gato Plaza, Suite 280C, San Diego 92123
  - Tel: (619) 498-2901
  - FAX: (619) 498-2901

**COLORADO**
- Intel Corp., 1445 Northpark Drive, Suite 100, Colorado Springs 80907
  - Tel: (719) 246-9300
  - FAX: (719) 246-9300
- Intel Corp., 600 S. Cherry St, Suite 700, Denver 80202
  - Tel: (303) 621-8006
  - FAX: (303) 621-8006

**CONNECTICUT**
- Intel Corp., 301 Lee Farm Corporate Park, 85 Webster Rd, Danbury 06810
  - Tel: (203) 748-3130
  - FAX: (203) 748-3130
- Intel Corp., 800 Fairway Drive, Suite 160, Deerfield Beach 33441
  - Tel: (561) 421-5050
  - FAX: (561) 421-2444

**FLORIDA**
- Intel Corp., 850 Crossingway Office Park, Fort Lauderdale 33304
  - Tel: (954) 425-2750
  - FAX: (954) 222-5081

**GEORGIA**
- Intel Corp., 20 Technology Park, Suite 150, Norcross 30092
  - Tel: (404) 449-0541
  - FAX: (404) 605-9762

**ILLINOIS**
- Intel Corp., Woodfield Mall, 300 N. Mannings Road, Suite 430
  - Schaumburg 60173
  - Tel: (847) 965-0861
  - FAX: (847) 706-9762

**INDIANA**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**MARYLAND**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**Massachusetts**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**MINNESOTA**
- Intel Corp., 3500 W 80th St, Suite 390, Bloomington 55421
  - Tel: (612) 818-8277
  - FAX: (612) 818-8277

**NEW JERSEY**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**NEW YORK**
- Intel Corp., 20 Technology Park, Suite 150, Norcross 30092
  - Tel: (404) 449-0541
  - FAX: (404) 605-9762

**OHIO**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**OREGON**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**PENNSYLVANIA**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**WASHINGTON**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**WISCONSIN**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**ENDESA**
- Intel Corp., 14500 Midwest Rd, Suite 116, Westfield 46074
  - Tel: (317) 875-8000
  - FAX: (317) 875-8000

**Quebec**
- Intel Corp., 1 Rue Holiday, Suite 115, Varennes East, Quebec H9R 5N3
  - Tel: (514) 675-1705
  - FAX: (514) 675-2358

**Sales and Service Office**
- *Field Service Location*
Intel embedded architectures and flash memory are supported by an array of development tools solutions. Use the World Wide Web, FaxBack, Literature Centers, and Intel Hotline for comprehensive tools information.