HOSTESS

COMTROL
Powerful Choices

2675 Patton Road
St. Paul, Minnesota 55113
(612) 631-7654
FAX: (612) 631-8117
1-800-926-6876

Part No.: 6254D March 1, 1995

Developer's Toolkit for the Hostess® *i* Series

COMTROL
Powerful Choices

Comtrol Corporation Document Number: 6254
Printed in the U. S. A.

Before You Begin

Scope

This guide describes the functionality of the Hostess *i* controller, along with information needed to program the controller.

This manual provides information for Hostess *i* controllers with a serial number of HI07-002409 or greater.

The EPROM on the controller changed for models HI07-002409 or greater to reflect a revision to the Borland® Turbo Debugger® (version 4.02). This means if you use this manual for earlier serial numbers of the controller:

- You will find that the steps for using the Turbo Debugger are not quite correct.
- You can not use the 4.02 or greater version of the Turbo Debugger with the EPROM on the controller; you must use version 3.2 or lower.

Note: *Unfortunately, Borland International did not make the 4.02 version of the Turbo Debugger backward compatible with previous levels.*

Prerequisites

This manual assumes that have also ordered the Development Board Option (discussed below). To effectively use this toolkit, the manual assumes the following conditions exist:

- The controller is installed in your system.
  If it is not installed, refer to the *User 's Guide* for this information.
- Your ISA personal computer system consists of the following:
  - DOS version 4.01 or higher
  - Optionally, Windows™ 3.1 or higher running in 386-enhanced mode
- You are running one of the following compilers on the development system:
  - Borland C++ (version 4.02 or later)
  - Microsoft® Visual C++ (version 1.0 or later)

Note: *For a detailed list of the requirements for the development system, refer to the documentation for the compiler.*

Audience

This guide is primarily for the programmer who is familiar with C language or 80286™ Assembly language.

What the Developer's Toolkit Contains

The *Developer's Toolkit* consists of the following pieces:

- This manual.
- A *Developer's Toolkit* diskette containing sample programs for your controller.
- The Advanced Micro Devices manual for the serial communications controller on your controller.

To readily use the Toolkit, you should have ordered the *Development Board Option* on your controller. This option is provided at no additional charge and includes the following pieces:

- A debug/reset header soldered to the controller
- A debug/reset box and cable

Note: *If you have any questions regarding the Toolkit or the Development Board Option, contact Comtrol using the information provided in Appendix A.*

Organization

This guide contains the following information:

Bibliography

Am8530H/Am85C30 Serial Communications Controller 1992 Technical Manual. U.S.A. Advanced Micro Devices, Inc., 1992.

NEC 16-Bit V-Series Microprocessor Data Book. U.S.A: NEC Electronics Inc., May 1990.

Table of Contents

List of Examples, Figures, Flowcharts

Examples

Figures

Flowcharts

List of Tables

Section 1. Controller Overview

1.1. Controller Features

The Hostess *i* is an 8-port intelligent serial controller, which can be upgraded to 16 ports with an 8-port upgrade module. Each port can be set for RS-232 or RS-422 mode. Ports 1 and 2 can be configured for synchronous mode and support full-duplex DMA.

The controller contains four or eight Am8530 Serial Communication Controllers (SCCs), depending on your model. This device implements the eight or sixteen serial ports found on the controller. The SCCs are mapped into the processor's I/O address space. For more information about programming the SCCs, refer to documentation for the 8530 SCCs listed in the *Bibliography* (in the *Before You Begin* section).

Figure 1-1 illustrates the architecture of the major components of the controller.

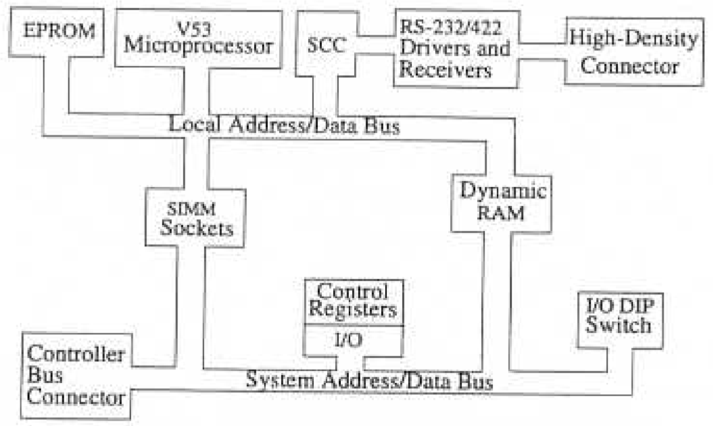


Figure 1-1. Major Controller Components

The following is a list of additional components and features of the controller:

- 12 MHz, zero-wait-state, NEC V53 16-bit microprocessor
- 128K dual-ported RAM
- SIMM slots to upgrade local RAM to 640K, 2MB, or 8MB
- Switch-definable I/O addresses
- Software-definable memory addresses, IRQs, and 8 or 16-bit memory transfers

- Three programmable timers
- A configuration control register, which controls the source of some serial signals

Note: *Refer to the* User's Guide *for information on the location of these components or for controller specifications.*

1.1.1. Microprocessor

The V53 microprocessor connects to many of the other components through the local address bus and the local data bus. It is binary code compatible with the 80286 processor, operating in *real* mode. This implies that it can address up to the one megabyte boundary. The V53 has a proprietary *extended address mode* that allows it to access memory above the one megabyte boundary.

The V53 microprocessor has the following integrated components:

- An 8237 four-channel DMA controller
- An 8251 UART communications controller
- Three 8254 timer-counters
- An 8259 interrupt controller
- A refresh controller

1.1.2. EPROM

The EPROM contains 64K bytes mapped at the top of the processor's one megabyte of memory space. This firmware contains code for the following:

- V53 bootstrap instruction
- V53 initialization
- Interrupt controller initialization
- Timer initialization
- Interrupt vector initialization
- Interrupt Service Routines (ISRs)
- Diagnostics for Serial Communication Controllers (SCCs) and memory
- Terminal debugger
- The Borland Turbo Debugger remote kernel

Although it is possible for users to produce their own EPROMs for the controller, Comtrol does not recommend it. Instead, users can customize the operation by developing their own control programs and downloading them to the controller.

1.1.3. Memory

The standard memory block contains 128K bytes of dual-ported RAM. This RAM is mapped at the bottom of the processor's memory space, which is the controller's base configuration.

A small portion of this memory is reserved for the interrupt vector table and for firmware usage. The remainder can be used to customize the controller, by developing and downloading a user's own control program into the dual-ported RAM.

Table 1-1 shows the standard memory map, as addressed by the local processor.

Table 1-1. Standard Memory Map

Description	Starting Address
Unused	10080h
Firmware user area (obsolete)	10000h
Firmware user area	00B80h
Unused	00C00h
Firmware work space	00400h
Interrupt vector table	00000h

Additional memory can be added to the controller by adding SIMM modules (640K, 2MB, or 8MB). All memory is contiguous, starting at address 0. When additional memory is added, the first 512K is dual-port and the remainder is local. Only the V53 can access local memory. All memory above the 1MB boundary is accessible to the V53 in extended mode.

The dual-ported RAM can be viewed by the system processor through a *sliding window*. This window represents the portion of dual-ported RAM that is visible to the system processor at any one time. The location and size of the window is software programmable through a set of control registers.

The I/O block contains the following functions, which can be performed by I/O writes or reads from the system's processor:

- Write a control register index
- Write to a control register
- Enable or disable dual-ported RAM
- Interrupt the on-board processor
- Select an interrupt request (IRQ)
- Enable or disable IRQs
- Reset the controller

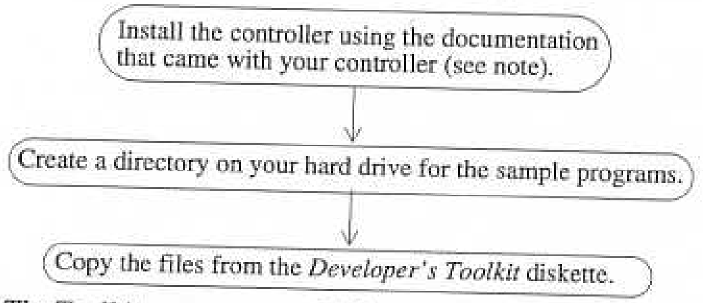
The control register block contains the functions that can be performed by I/O writes from the system processor. The control registers are accessed when an index, and then a control register value is written.

The control register functions select the following:

- The base address of dual-ported RAM in the system's memory space
- The size of the system's window in dual-ported RAM
- The portion of dual-ported RAM visible in the system's window
- The IRQ line used by the controller

1.2. Toolkit Installation

Figure 1-2 illustrates how to install the Developer's Toolkit.


Install the controller using the documentation that came with your controller (see note).
Create a directory on your hard drive for the sample programs.
Copy the files from the *Developer's Toolkit* diskette.

Note: *The Toolkit programs use 218h as the base I/O address. If you use a different address, you must edit the programs to reflect your address selection.*

Figure 1-2. Installing the Toolkit

See Section 2 for detailed information about the sample programs on the *Developer's Toolkit* diskette.

Section 2. Developer's Toolkit Sample Programs

2.1. Developer's Toolkit Sample Programs Overview

This section illustrates the sample programs included with this manual on the *Developer's Toolkit* diskette. The diskette contains the source listings and executable files for a simplified control program model that works on the Hostess *i*.

This manual and *Developer's Toolkit* use both C and 80286 assembly language examples. Control programs often are a mix of both high-level and low-level code.

Comtrol encourages you to use these files on the diskette and examine how the control program works. The control program, CPC.BIN, was written in the C and the 80286 assembly languages. This section lists the CPC.BIN source code, and the source code listings for the other programs found on the *Developer's Toolkit* diskette.

The executable control program model (CPC.BIN) runs on the Hostess *i*. It opens, closes, reads, and writes to any asynchronous line on the controller.

These files make up the control program model:

- CPC.C – the source code for the control program model.
- CPC.BIN – the executable control program model.
- CPC.H – the header file for CPC.C.
- CPCSTART.ASM – the startup code, in assembly language.
- CPC.TDS – the symbol table (for debugging).
- DPLOADER.C – the source code for the loader program.
- DPLOADER.EXE – the executable loader program.
- DPRAM.H – header file used by both the control program and by system applications.
- FIRMUSER.H – the C header file that defines the firmware user area.
- FIRMUSER.EQU – the assembly language file that defines the firmware user area.
- CLOCATE.EXE – a locator program that performs relocation of CPC.BIN file producing CPC.BIN as it's output.

2.2. How the Control Program Works

The following steps describe how the control program works:

- At power up, the local processor executes the initialization and diagnostic code out of the firmware.
- The system processor downloads (writes) the control program into dual-port memory (DPM) starting at the local processor's address C0:0 using the **COPY** firmware utility command.
- The system processor invokes the **EXEC** firmware utility command, which in turn invokes the control program, at the C0:0 entry point. The first section of the control program's code initializes the segment registers, stack, interrupt vectors, timer, and several fields of the firmware user area.

After initializing these data structures, the control program enters an infinite processing loop. *Line* refers to any one of the 16 serial lines (ports) on the controller. The sample programs number the lines from 0 to 15. Each serial line has a line-table entry associated with it.

Table 2-1. Line Table Number and Controller Ports

Line Tables	Controller Port
Line 0 table	Port 1
Line 1 table	Port 2
Line 2 table	Port 3
Line 3 table	Port 4
Line 4 table	Port 5
Line 5 table	Port 6
Line 6 table	Port 7
Line 7 table	Port 8
Line 8 table	Port 9
Line 9 table	Port 10
Line 10 table	Port 11
Line 11 table	Port 12
Line 12 table	Port 13
Line 13 table	Port 14
Line 14 table	Port 15
Line 15 table	Port 16

The main loop sequentially checks each line's line-table entry, line-status field. If the LINE_ACTIVE bit is not set, processing continues with the next line.

If the LINE_ACTIVE bit is set, the line status is checked to see if the TX_ACTIVE bit is set.

The TX_ACTIVE bit indicates that the SCC is busy sending a character and it cannot accept another character.

If the SCC is free, the main loop calls the **deq_Tx_data** routine to write the next character (if any) from the current line's transmit buffer to the SCC's internal transmit buffer.

The data is placed in the transmit buffer queue at the location pointed to by the transmit head pointer in the line table. Data is removed from the transmit buffer queue transmit tail pointer in the line table. (Consider these loop operations as background processing. Interrupt service routines (ISRs) handle all other processing in the control program.)

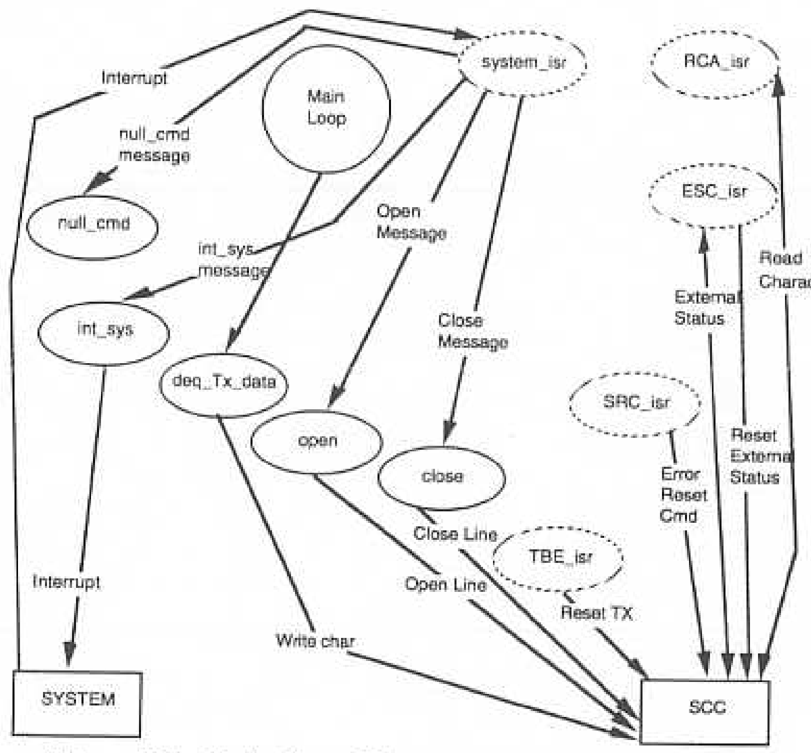


Figure 2-1. Data Flow Diagram for the CPC.C Program

```
void deq_tx_data(LINE_ENTRY_T far *lt_p)
{
  int tail;
  if(lt_p->line_status & ALLSENT_PEND)     /* waiting for last Tx bit to clear*/
    return;                         /* don't start next message till it does */
  tail = lt_p->Txq_tail;
  if(lt_p->Txq_head == tail)        /* Tx queue empty */
    return;
 if((lt_p->line_status & (RS485ENABLED|TXMODE)) == RS485ENABLED) /*new
     msg*/
 {
   disable();
   txmode485(lt_p,1);              /* set transceiver to transmit */
   enable();
 }
 lt_p->line_status |= TX_ACTIVE;              /* indicate transmit active */

 if(lt_p->line_status & RS485ENABLED)      /* doing RS-485 */
 {
   disable();       /* ensure ALLSENT_PEND gets set in timely manner */
   OUTB(lt_p->io_base + 2,*(lt_p->Txq_com + tail));  /* write char to SCC */

   if(lt_p->Txq_head == lt_p->Txq_tail)                /* Tx queue empty */
   {  /* the next TBE_isr indicates end of message */
     lt_p->line_status |= SEND_FLAG;              /* mark it to send flar char next */
   }
   enable();
 }
 else                                               /* not doing RS-485 */
 {
   OUTB(lt_p->io_base + 2,*(lt_p->Txq_com + tail)); /* write char to SCC */
 }
}
```

During the control program's initialization phase, an interrupt service routine (**system_isr**) replaces the firmware routine that first invoked the control program. The **system_isr** routine is responsible for processing messages sent by the system processor to the control program in the **Comq** buffer.

Four messages have been defined

- **null_cmd** (does nothing)
- **open**
- **close**
- **int_sys** (an example of interrupting the system).

Of these messages, only **open** and **close** are useful.

The **open** message includes the line number and the communication parameters. This information is passed to an **open** routine, which initializes the appropriate SCC and enables the line by setting the LINE_ACTIVE bit in the line-table entry, line-status field.

The **close** message includes the line number, which is passed to a **close** routine to disable the appropriate SCC and clear the LINE_ACTIVE bit.

```
void interrupt far system_isr()
{
  /* Set up dispatch table, one function for each Sys uP command */
  static void (*dispatch[NUM_SYSCMD])(void) =
  {
    null_cmd,              /* 0 */
    open,                  /* 1 */
    close,                 /* 2 */
    int_sys                /* 3 */
  };
  if(!deq_com_msg())                  /* get message from Com uP */
  {
    EOI(INTCTL,EOIVAL);            /* end of interrupt to PIC */
    return;                        /* no message, return */
  }
  if(msg_buf[0] < 0 || msg_buf[0] >= NUM_SYSCMD)   /* invalid command */
  {
    EOI(INTCTL,EOIVAL);               /* end of interrupt to PIC */
    return;                           /* no message, return */
  }
  (*dispatch[msg_buf[0]])();          /* execute command function */
  EOI(INTCTL,EOIVAL);                 /* end of interrupt to PIC */
}
```

Examples of system-side processing for **open** and **close** are contained in HILIB.C, the **hiopen** and **hiclose** functions illustrate the system-side processing for the **open** and **close** messages.

The **hiopen** function opens a serial line on the Hostess *i*. After a line has been opened, data may be transmitted to that line. To transmit a character, the system processor writes the character to that line's transmit buffer, using the normal queue operations, as used in the **HILIB.C**, **hiwrite** routine.

The control program's main loop removes the character from the queue and writes it to the SCC, and then sets the TX_ACTIVE bit in the line table. No more characters can be sent by the control program until TX_ACTIVE clears.

The system can continue adding characters to the transmit buffer until it is full. When the SCC has completed serially shifting the character out, it issues an interrupt to the Hostess *i* processor, which invokes the TBE_isr routine for that line. TBE_isr clears TX_ACTIVE, which allows the next character to be sent.

```
void TBE_isr(LINE_ENTRY_T far *lt_p)
{
  int scc;                                  /* SCC command register address */

  scc = lt_p->io_base;                      /* get SCC command register address */
  if(lt_p->line_status & SEND_FLAG)         /* time to send RS-485 flag char */
  {
    lt_p->line_status &= ~SEND_FLAG;
    lt_p->line_status |= ALLSENT_PEND;       /* next TBE_isr set Rx state */
    OUTB(lt_p->io_base + 2,FLAGCHAR);        /* write flag char to SCC */
    lt_p->line_status &= ~TX_ACTIVE;         /* indicate no char in SCC Tx buffer */
    OUTB(scc,WR0);                           /* end of interrupt to SCC */
    OUTB(scc,RESET_IUS);
    return;
  }
  if(lt_p->line_status & ALLSENT_PEND)       /* RS-485, set line to Rx */
  {
    txmode485(lt_p,0);                       /* set transceiver to receive state */
    lt_p->line_status &= ~ALLSENT_PEND;      /* no longer waiting for last char */
  }
  lt_p->line_status &= ~TX_ACTIVE;           /* indicate no char in SCC Tx buffer */
  OUTB(scc,WR0);                             /* reset pending Tx interrupt */
  OUTB(scc,RESET_TX_INT);
  OUTB(scc,WR0);                             /* end of interrupt to SCC */
  OUTB(scc,RESET_IUS);
}
```

After a line has been opened, data may also be received from that line. When the SCC receives a serial character, it issues an interrupt to the local processor, which invokes the RCA_isr routine for that line. RCA_isr reads the character from the SCC and places it in that line's receive buffer queue. The data is placed in the receive buffer queue at the location pointed to by the receive head pointer in the line table. Data is removed from the receive buffer queue receive tail pointer in the line table.

```
void RCA_isr(LINE_ENTRY_T far *lt_p)
{
  int scc;                                   /* SCC command register address */
  unsigned char ch;                          /* character read from SCC */
  int head;                                  /* Rx queue head pointer */
  int num_full;                              /* number Rx queue locations filled */
  scc = lt_p->io_base;                       /* get SCC command register address */
  ch = inp(scc+2);                           /* read character from SCC */
  head = lt_p->Rxq_head;
  if((num_full = head - lt_p->Rxq_tail) < 0)      /* num queue locations full */
  {
    num_full += RXB_SIZE;                    /* adjust for queue wrap */
  }
  if(num_full < RXB_SIZE - 1)                /* if Rx queue has empty space */
  {
    *(lt_p->Rxq_com + head) = ch;                 /* add received character to queue */
    lt_p->Rxq_head = (head + 1) % RXB_SIZE;       /* bump head pointer */
  }
  OUTB(scc,WR0);                             /* end of interrupt to SCC */
  OUTB(scc,RESET_IUS);
}
```

The system processor may then remove that character from the queue, as used in HILIB.C's **hiread** routine.

The SCC's are also capable of generating interrupts for external status changes or special receive conditions.

These interrupts are handled by CPC.C's interrupt service routines ESC_isr and SRC_isr:

```
void ESC_isr(LINE_ENTRY_T far *lt_p)
{
  int scc;                              /* SCC command register address */
  unsigned char status;                 /* saves the external status */
  scc = lt_p->io_base;                  /* get SCC command register address */
  status = inp(scc);                    /* read the external status */
  status = status;                      /* prevent compiler warning */
 /* Do External Status Change processing here */
  OUTB(scc,WR0);                        /* reset external status interrupts */
  OUTB(scc,RESET_EXT);
  OUTB(scc,WR0);                        /* end of interrupt to SCC */
  OUTB(scc,RESET_IUS);
}
void SRC_isr(LINE_ENTRY_T far *lt_p)
{
    int scc;                          /* SCC command register address */
    unsigned status;                  /* saves the SRC status */

    scc = lt_p->io_base;              /*  get SCC command register address */
    OUTB(scc,RR1);                    /* read the SRC status */
    status = INB(scc);                /* read the external status */
    status = status;                  /* prevent compiler warning */

   /* Do Special Receive Condition processing here */

    OUTB(scc,WR0);                    /* for insurance */
    OUTB(scc,ERROR_RESET);            /* issue error reset command */
    OUTB(scc,WR0);                    /* end of interrupt to SCC */
    OUTB(scc,RESET_IUS);
}
```

The local processor's timer 1 is initialized by the control program to generate an interrupt 12 times a second. These are handled by the interrupt service routine **timer1_isr**, which does nothing except increment the "heartbeat" counter found in the firmware user area.

```
    void interrupt far timer1_isr()
    {
      fu_p->heartbeat++;              /* bump heartbeat counter */
      EOI(INTCTL,EOIVAL);             /* end of interrupt to PIC */
    }
```

2.3. Dual-Port RAM Configuration for CPC.BIN

Information stored in dual-port memory (DPM) includes:

- General information about the controller as defined by firmware (the firmware user area)
- Area for messages for the communications processor (**Comq**)
- Area for messages for the system processor (**Sysq**).
- Line tables for each of the 16 ports that describe the port.
- Transmit and receive buffer for each of the 16 lines.

The control program uses less than 64K of dual-port memory, beginning at the local processor's address 0000:0. The system processor views this same memory beginning at address D000:0. See Table 2-2 for a map of this area of dual-port memory.

Table 2-2. 64K Dual-Port Memory Map

Offset in Hex	Use	Length in Hex Bytes
0	Reserved	B80
	Firmware User Area	
B80	Processor interaction flag	2
B82	Boot/activity flag	2
B84	Configuration map (obsolete)	2
B86	Firmware release number	8
B8E	Control program release number	8
B96	Reserved	4
B9A	DRAM map	4
B9E	SCC map	4
BA2	Board ID	4
BA6	Invalid interrupt flag	1
BA7	Invalid interrupt type	1
BA8	Invalid interrupt count	2
BAA	Heartbeat	4
BAE	Utility command	1
BAF	Utility status	1
BB0	Utility message buffer	16
BC0	Balance of firmware area	64
C00	Control program and empty space	4400
	Comm Message Queue	
5000	Head pointer	2
5002	Tail pointer	2
5004	Message area	200
	System Message Queue	
5204	Head pointer	2
5208	Tail pointer	2
520A	Message area	200
5408	Filler	8

Table 2-2. 64K Dual-Port Memory Map (Continued)

Offset in Hex	Use	Length in Hex Bytes
	Line Tables	
5410	Line 0 table	20
5440	Line 1 table	20
5470	Line 2 table	20
54A0	Line 3 table	20
54D0	Line 4 table	20
5500	Line 5 table	20
5530	Line 6 table	20
5560	Line 7 table	20
5590	Line 8 table	20
55C0	Line 9 table	20
55F0	Line 10 table	20
5620	Line 11 table	20
5650	Line 12 table	20
5680	Line 13 table	20
56B0	Line 14 table	20
56E0	Line 15 table	20
	Transmit Buffers	
5710	Line 00	200
5910	Line 01	200
5B10	Line 02	200
5D10	Line 03	200
5F10	Line 04	200
6110	Line 05	200
6310	Line 06	200
6510	Line 07	200
6710	Line 08	200
6910	Line 09	200
6B10	Line 10	200
6D10	Line 11	200
6F10	Line 12	200

Table 2-2. 64K Dual-Port Memory Map (Continued)

Offset in Hex	Use	Length in Hex Bytes
7110	Line 13	200
7310	Line 14	200
7510	Line 15	200
	Receive Buffers	
7710	Line 00	800
7F10	Line 01	800
8710	Line 02	800
8F10	Line 03	800
9710	Line 04	800
9F10	Line 05	800
A710	Line 06	800
AF10	Line 07	800
B710	Line 08	800
BF10	Line 09	800
C710	Line 10	800
CF10	Line 11	800
D710	Line 12	800
DF10	Line 13	800
E710	Line 14	800
EF10	Line 15	800
F710	Unused	8F0
10000	End of DPM	

Table 2-3. Line Table Map

Offset in Hex	Use	Length in Hex Bytes
0	SCC base I/O address	2
2	Line status	2
4	Write register 2 value	1
5	Write register 3 value	1
6	Write register 4 value	1

Table 2-3. Line Table Map (Continued)

Offset in Hex	Use	Length in Hex Bytes
7	Write register 5 value	1
8	Write register 12 value	1
9	Write register 13 value	1
A	Write register 15 value	1
B	Filler, keep pointers on even address	1
C	Transmit buffer head pointer	2
E	Transmit buffer tail pointer	2
10	Transmit buffer local processor address	4
14	Transmit buffer system processor address	4
18	Receive buffer head pointer	2
1A	Receive buffer tail pointer	2
1C	Receive buffer local processor address	4
20	Receive buffer system processor address	4
24	Filler	C
30	End of DPM	

You can read through the listings to learn how a control program works with the Hostess *i* controller. You will see some of this code again as examples in the following subsections.

2.4. DPLOADER.C

DPLOADER is a DOS program written in C language. This program can:

- Reset the Hostess *i* controller.
- Remove header bytes before downloading.
- Download a binary file into dual-port RAM on the controller.
- Start the Turbo Debugger debugger kernel code on the controller.

Figure 2-2 shows the **DPLOADER.C**.

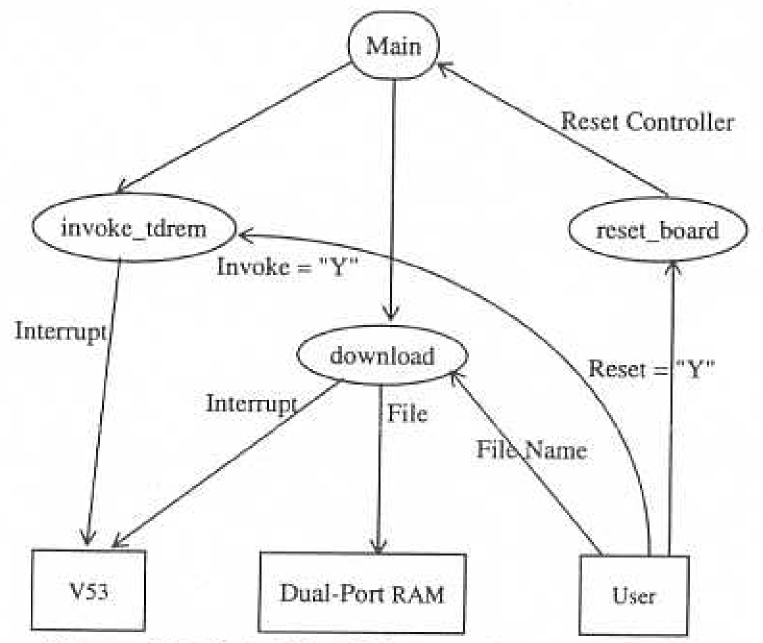


Figure 2-2. Data Flow Diagram for DPLOADER.C

2.5. HILIB.C

HILIB.C is the file that contains four significant Hostess *i* functions:

- **hiopen()**
- **hiclose()**
- **hiread()**
- **hiwrite()**

Compile and link the **HILIB.C** file with your application program to access Hostess *i* serial lines. The paragraphs that follow explain these routines, with examples in C syntax.

The **hiopen** function opens a requested serial line on the Hostess *i*, initializes the line to 9,600 baud, 8 data bits, 1 stop bit, and no parity.

```
int hiopen(int linenum)
{
  /* Default open message to controller */
  static char openmsg[MSG_LEN]=
  {
    1,                          /* command number parameter */
    0,                          /* line number parameters */
    0xc0,                       /* WR3 parameters (Rx character size) */
    0x44,                       /* WR4 parameters (stop bits, parity */
    0x60,                       /* WR5 parameters  (Tx character size) */
    0x0e,                       /* WR12 parameter   (lower byte of BRGTC) */
    0,                          /* WR13 parameter   (upper byte of BRGTC) */
    0,                     /* RS-485 parameter (0=disable RS485, 1=enable RS485) */
    0,0,0,0,0,0,0,0             /* unused */
  };
  openmsg[1] = (char)linenum;   /* set up line number in message */
  if(enq_com_msg(openmsg))      /* add message to COMQ */
  {
    OUTB(IO_SYS+2,0);           /* interrupt SYS uP */
    return(1);                  /* success */
  }
  else
    return(0);                  /* fail */
}
```

Returns 1 if successful, 0 if unsuccessful.

The function **hiclose** closes a requested serial line on the controller:

```
int hiclose(int linenum)
{
  /* Default close message to controller */
  static char closemsg[MSG_LEN] =
  {
    2,                               /* command number parameter */
    0,                               /* line number parameter */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0      /* unused */
  };
  closemsg[1] = (char)linenum;       /* set up line number in message */
  if(enq_com_msg(closemsg))          /* add message to COMQ */
  {
    OUTB(IO_SYS+2,0);                /* interrupt SYS uP */
    return(1);                       /* success */
  }
  else
    return(0);                       /* fail */
}
```

Returns 1 if successful, 0 if unsuccessful.

The **hiread** function reads up to a maximum **cnt** bytes into the line's receive buffer. The function does not wait for the bytes to read:

```
int hiread(char *sbuf,int cnt,int linenum)
{
  LINE_ENTRY_T far *lt_p;              /* ptr to line table entry */
  int i;                               /* balance of chars to copy after q wrap */
  int tail, head;                      /* Rx buffer head & tail ptrs */
  int heads;                           /* save copy of head ptr */
  int ccnt;                            /* count of chars copied */
  lt_p = line[linenum];                /* get ptr to line table entry */
  tail = lt_p->Rxq_tail;               /* read tail */
  head = lt_p->Rxq_head;               /* read head */
  heads = lt_p->Rxq_head;              /* read head again */
  /* Verify that head values match to prevent possibility that ctrlpgm
     modified it in between 8 bit SYS uP reads. Only need to do this
     if controller is in 8 bit mode. */
  while(head != heads)                 /* while the two head reads differ */
  {
    head = lt_p->Rxq_head;             /* read head */
    heads = lt_p->Rxq_head;            /* read head again */
  }
  /* Get number of characters in Rx buffer */
  if((ccnt = head - tail) < 0)
    ccnt += RXB_SIZE;                  /* adjust for queue wrap */
  else if(ccnt == 0)
    return(ccnt);                      /* nothing in Rx buffer */
  if(cnt < ccnt)
    ccnt = cnt;                        /* don't overflow SYS uP buffer */
  i = ccnt - (RXB_SIZE - tail);        /* i = whats left after wrap around */
  if(i < 0)
    i = 0;
  /* Copy to end of Rx buffer */
  BCOPY(lt_p->Rxq_sys + tail,sbuf,ccnt - i);
  /* Point to beginning of buffer if already at end of it */
  tail = (tail + (ccnt - i)) % RXB_SIZE;
  /* Copy the rest of the buffer, if any left */
  if (i != 0)
  {
    BCOPY(lt_p->Rxq_sys,sbuf + (ccnt - i),i);
    tail = i;
  }
  /* Update Rx buffer tail */
  lt_p->Rxq_tail = tail;
  return(ccnt);
}
```

Returns the number of bytes read (0 - 'cnt').

The **hiwrite** function writes up to a maximum **cnt** bytes from the line's receive buffer into dual-port memory. The function does not wait for enough space to write if the request is too large

```
int hiwrite(char *sbuf,int cnt,int linenum)
{
  LINE_ENTRY_T far *lt_p;              /* ptr to line table entry */
  int i;                               /* balance of chars to copy after q wrap */
  int numopen;                         /* num bytes open in Tx buffer */
  int head, tail;                      /* Tx buffer head & tail ptrs */
  int tails;                           /* save copy of tail ptr */
  lt_p = line[linenum];                /* get ptr to line table entry */
  head = lt_p->Txq_head;               /* read head */
  tail = lt_p->Txq_tail;               /* read tail */
  tails = lt_p->Txq_tail;              /* read head again */
  /* Verify that tail values match to prevent possibility that ctrlpgm
     modified it in between 8 bit SYS uP reads. Only need to do this
     if controller is in 8 bit mode. */
  while(tail != tails)                 /* while the two tail reads differ */
  {
    tail = lt_p->Txq_tail;             /* read tail */
    tails = lt_p->Txq_tail;            /* read tail again */
  }
  /* Get number bytes open in Tx buffer */
  if((numopen = tail - head - 1) < 0)
    numopen += TXB_SIZE;               /* adjust for q wrap */
  if(numopen > cnt)
    numopen = cnt;                     /* don't move more than are incoming */
  if(numopen == 0)
    return(0);                         /* no room in Tx buffer */
  i = numopen - (TXB_SIZE - head);        /* i = what's left after wrap around */
  if (i < 0)
    i = 0;
  /* Copy to end of Tx buffer */
  BCOPY(sbuf,lt_p->Txq_sys + head,numopen - i);
  /* Point to beginning of buffer if already at end of it */
  head = (head + (numopen - i)) % TXB_SIZE;
  /* Copy the rest of the buffer, if any left */
  if (i != 0)
  {
    BCOPY(sbuf + (numopen - i),lt_p->Txq_sys,i);
    head = i;
  }
  /* Update Tx buffer head */
  lt_p->Txq_head = head;
  return(numopen);
}
```

Returns the number of bytes written (0 - 'cnt').

2.6. HITERM.C

The HITERM program runs on the system and emulates a terminal. This DOS program works with the control program.

2.6.1. Invoking HITERM

To use the executable file HITERM.EXE with CPC.BIN, follow these steps:

1. Set the Hostess *i* controller for I/O address 218h.
2. Check that no other device occupies the D000 base memory address. The program uses 64K starting at D000:0.
3. Install the controller in the system.
4. Connect a non-intelligent ASCII terminal to the port on the Hostess *i* controller that you want to use. Set the terminal to:
   - 9,600 baud
   - 8 data bits
   - 1 stop bit
   - No parity
   - No flow control
5. Start-up DOS.
6. Execute DPLOADER.EXE.

   DPLOADER prompts you for values it needs to download the control program.
7. Execute HITERM.EXE.

   The HITERM application sends and receives any characters you type on either keyboard.

   Pressing the <F10> key terminates the transmittal.

2.7. Compiling the Sample Programs Using the Borland Make Utility

Included on the *Developer's Toolkit* diskette is a file called MAKEFILE. This file builds the executable programs, using the Borland **make** utility:

```
# This is the make file for the Hostess i.
CPC_RELOC_SEG = c0          # CPC relocation address segment
CC = bcc
MKF = makefile
STARTUP = cpcstart
CTLTYPE = HOSTESSi    # Must define one of SMARTH, HOSTESSi, or HOSTESS186
#Must define both of the following to represent the same control program model
CPMODEL = l              # choose s(small),m(medium),c(compact), or l(large)
CP_MODEL = MLARGE         # choose MSMALL,MMEDIUM,MCOMPACT or
    MLARGE
CPCLIB = \bc31\lib\c$(CPMODEL).lib   # library for Borland C++
all: cpc.bin hiterm.exe dploader.exe
#***** cpc control program ***********************
cpc.bin: $(STARTUP).obj cpc.obj $(MKF)
  tlink /s /c /v @&&!
$(STARTUP).obj $*.obj
$*.exe
$*.map
$(CPCLIB)
!
  clocate $*.exe $*.bin $(CPC_RELOC_SEG)
  tdstrip -s cpc.exe
  del cpc.exe
cpc.obj: cpc.c cpc.h dpram.h firmuser.h $(MKF)
  $(CC) -c -m$(CPMODEL) -v -D$(CTLTYPE) $*.c
$(STARTUP).obj: $(STARTUP).asm firmuser.equ $(MKF)
  tasm /l /zi /mx /d$(CTLTYPE) /d$(CP_MODEL) /dSTACK_SIZE=2048 $*.asm
#**** hiterm.exe ********************************
hiterm.exe: hiterm.c hilib.c dpram.h $(MKF)
  $(CC) -ml -v -D$(CTLTYPE) $*.c hilib.c
#**** dploader.exe *******************************
dploader.exe: dploader.c dpram.h firmuser.h $(MKF)
  $(CC) -v -D$(CTLTYPE) $*.c
```

2.7.1. Using the MAKEFILE

To use MAKEFILE:

1. Edit the MAKEFILE file and define the Comtrol controller type. The MAKEFILE entry must match your controller type:

   CTLTYPE = HOSTESSi......# Must define one of SMARTH, HOSTESSi, or HOSTESS186

2. Define the appropriate memory model for your system. The MAKEFILE entries must specify the same memory model.

   #Must define both of the following to represent the same control program model
   CPMODEL = l.........# choose s(small),m(medium),c(compact), or l(large)
   CP_MODEL = MLARGE.........# choose MSMALL,MMEDIUM,MCOMPACT or MLARGE

3. Save your changes to MAKEFILE.
4. Enter **make** at the DOS prompt.

   make

2.7.2. Building the Sample Program

To build the sample programs, **make** performs the following steps:

1. Compile DPLOADER.EXE from the source files DPLOADER.C, DPRAM.H, and FIRMUSER.H. The MAKEFILE entry includes a macro named CTLTYPE. The macro sets the controller type.

   ```
   dploader.exe: dploader.c dpram.h firmuser.h $(MKF)
   $(CC) -v -D$(CTLTYPE) $*.c
   ```

2. Compile HITERM.EXE from the source files HITERM.C HILIB.C, and DPRAM.H. The MAKEFILE entry also includes the CTLTYPE macro that sets the controller type.

   ```
   hiterm.exe: hiterm.c hilib.c dpram.h $(MKF)
   $(CC) -ml -v -D$(CTLTYPE) $*.c hilib.c
   ```

3. Assemble CPCSTART.OBJ from the assembly files CPSTART.ASM and FIRMUSER.EQU. The MAKEFILE entry includes the macros **STARTUP, CTLTYPE, CP_MODEL, and STACK_SIZE.**

   ```
   $(STARTUP).obj: $(STARTUP).asm firmuser.equ $(MKF)
     tasm /l /zi /mx /d$(CTLTYPE) /d$(CP_MODEL) /
   dSTACK_SIZE=2048 $*.asm
   ```

   Where:

   STARTUP is the name of the startup control module CPSTART.

   CTLTYPE sets the controller type.

   CP_MODEL sets the memory model.

   STACK_SIZE sets the size of the control program's stack.

4. Compile CPC.OBJ from the source files CPC.C, DPRAM.H, and FIRMUSER.H.

   The MAKEFILE entry also includes the CPMODEL and CTLTYPE macros that set the memory model and controller type.

   ```
   cpc.obj: cpc.c cpc.h dpram.h firmuser.h $(MKF)
   $(CC) -c -m$(CPMODEL) -v -D$(CTLTYPE) $*.c
   ```

5. Compile CPC.BIN from the object files CPCSTART.OBJ and CPC.OBJ. Link the output with the appropriate libraries to form to form **CPC.BIN**. The MAKEFILE entry includes the STARTUP and CPCLIB macros:

   Where:

   STARTUP is the name of the startup control module CPSTART.

   CPCLIB is the path to the Borland C language library for the memory model compiled for the control program.

   ```
   cpc.bin: $(STARTUP).obj cpc.obj $(MKF)
   tlink /s /c /v @&&!
   $(STARTUP).obj $*.obj
   $*.exe
   $*.map
   $(CPCLIB)
   ```

   In this example, CPCLIB is set to the **\cX.lib** ***path***:

   Where:

   X is the memory model (s for small, **m** for medium, **c** for compact, or **l** for large).

   path is the DOS path to the Borland C runtime library files.

   When you compile and link programs to run on the system processor under DOS, also link the **c0X.lib** library. This library contains the C startup code for DOS. Since the control program does not run under DOS, the CPCSTART module replaces **c0X.lib**.

6. Relocate CPC.BIN to the entry point address to create CPC.BIN. The MAKEFILE entry creates the downloadable binary image CPC.BIN.

   CLOCATE.EXE is a utility that performs the relocation. The CPC_RELOC_SEG macro sets the entry point segment address (in this case, C0h). This is the same address where DPLOADER.EXE downloads the control program.

   ```
   clocate $*.exe $*.bin $(CPC_RELOC_SEG)
   ```

7. Strip the symbol table information from CPC.BIN. Place this information in CPC.TDS. The MAKEFILE entry removes the symbol table information and places this information in the CPC.TDS file. Turbo Debugger uses the symbol table information to find addresses for variables and other data structures.

   tdstrip -s cpc.exe

Section 3. System I/O Addresses

3.1. Overview

This section discusses the following issues:

- Setting system I/O addresses
- Reading the controller identification byte
- Resetting and initializing the controller
- Initializing control registers
- Control register features
- Control register #1
- Control register #2
- Control register #3
- Control register #4

3.2. Setting System I/O Addresses

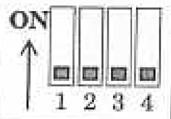
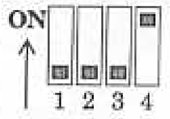
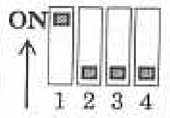
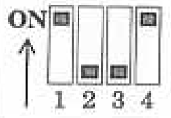
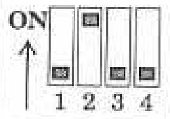
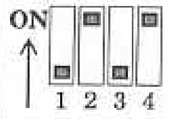
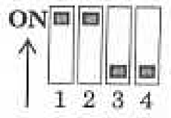
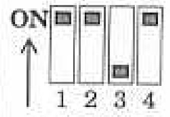
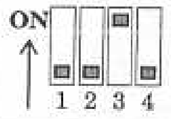
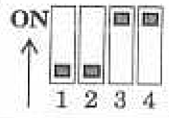
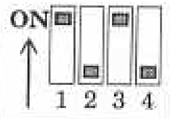
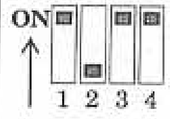
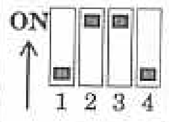
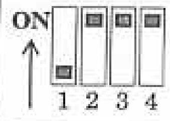
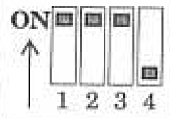
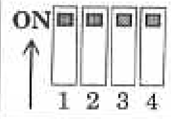
The four-position DIP switch block on the controller sets the system I/O addresses. The controller reserves four consecutive I/O addresses, starting with the address set by the switches. These addresses are used to

- Reset and initialize the controller
- Initialize control registers
- Enable memory on the controller

The following subsections explain how these actions occur.

Table 3-1 shows the possible I/O addresses and their switch settings.

Table 3-1. Switch Settings

I/O Address Range	DIP Switch Settings	I/O Address Range	DIP Switch Settings
218 - 21B hex		618 - 61B hex	
21C - 21F hex		61C - 61F hex	
238 - 23B hex		638 - 63B hex	
23C - 23F hex		63C - 63F hex	
318 - 31B hex		718 - 71B hex	
31C - 31F hex		71C - 71F hex	
338 - 33B hex		738 - 73B hex	
33C - 33F hex		73C - 73F hex	

The controller reserves four consecutive system I/O addresses:

- I/O_base+0
- I/O_base+1
- I/O_base+2
- I/O_base+3

Table 3-2 shows the I/O map and refers you to the subsection where it is discussed in more detail.

Table 3-2. Input/Output Map

I/O Address	Description	Detailed Discussion
I/O_base+0	Writes to control registers	Subsection 3.7.1
I/O_base+1	Enables/disables memory, control register index	Subsection 3.6
I/O_base+2	Interrupts controller	Subsection 3.5
I/O_base+3	Resets controller	Subsection 3.4

3.3. Reading the Controller Identification Byte

Reading (byte read) from address **I/O_base + 2** gets the controller identification byte. This byte can be used to identify the type of Comtrol controller installed. This value is 01h for the controller. The controller identification read is supported only on the Revision B (and later) controllers. See the *Before You Begin* discussion to find the version level of your controller.

3.4. Resetting and Initializing the Controller

Writing to the address **I/O_base+3** resets the controller, then it initializes the controller by causing the firmware start-up code to execute. Write the value 00h, delay one-tenth of a second, then write the value 0FFh. The controller's memory comes up as disabled after the controller is reset. System reads or writes to dual-port RAM are not allowed between these two I/O writes. (For device drivers for the UNIX® operating system, you can protect these two I/O writes using an **spl 7 ()** kernel call.)

This example shows how to reset a controller whose I/O base address is 218h:

```
outp (0x21b,0x00);      /* Set the reset */
delay(HZ/10);           /* Delay 1/10 second */
outp (0x21b,0xff);      /* Remove the reset */
```

After removing the reset, you must wait between approximately five seconds (for the 128 Kbytes of memory) to 25 seconds (for 2 MB) to allow the reset diagnostics to complete.

3.5. Interrupting the Controller

Use **I/O_base+2** to interrupt the controller. Writing any byte value generates an interrupt to the controller. It is the controller's responsibility to service this interrupt. This example shows how to interrupt a controller whose I/O base address is 218h:

```
outp (0x21a,0);
```

3.6. Enabling and Disabling Dual-Port Memory

Use **I/O_base+1** to enable or disable memory and as an index register when writing to the control registers. A write of value 1 to bit 2 disables the memory. A write of value 0 to bit 2 enables the memory. This example shows how to disable and then enable the memory, using the I/O base address 218h:

```
outp (0x219,4);          /* OFF */
outp (0x219,0);          /* ON */
```

See Subsection 3.7.1 for information on indexing the control registers.

3.7. Control Register Overview

There are four control registers on the controller. These write-only registers

- Control the memory addressing
- Select the memory window size, interrupts, and mode of operation (either PC (8-bit) or AT (16-bit)).

You access the control registers by writing an index value to the **I/O_base + 1** address, then the register contents to the **I/O_base + 0** address (see Subsection 3.7.1). The four-position DIP switch SW1 selects the I/O base address (see Subsection 3.2).

Overall, the registers function in this manner:

- Control register #1 selects the "above one megabyte" system address.
- Control register #2 selects the "below one megabyte" system address.
- Control register #3 selects the "sliding window" of dual-port memory.
- Control register #4 selects the interrupt request (IRQ).

3.7.1. Writing Control Registers

The control registers are written through a two-step process:

- First an index value is written out to **I/O_base+1** to select the control register:

Table 3-3. Writing to Control Registers

Control Register	Index with RAM Disabled	Index with RAM Enabled
1	05h	01h
2	06h	02h
3	0Ch	08h
4	14h	10h

- Then the register contents are written out to **I/O_base+0**. The index will remain fixed until an **I/O_base+1** is written again. Subsequent writes to the same control register are permitted without intervening index writes (this is useful for applications that use a sliding window into dual-port RAM).

Initialize all control registers before enabling the controller's memory. This means that the data bit D2 must be set to a 1 whenever you write out to **I/O_base+1** (see Subsection 3.2).

For example:

```
outp (I/O_base+1, 05h);          Setup for Control Register #1
outp (I/O_base+0, <value>)

outp (I/O_base+1, 06h);          Setup for Control Register #2
outp (I/O_base+0, <value>)

outp (I/O_base+1, 0Ch);          Setup for Control Register #3
outp (I/O_base+0, <value>)

outp (I/O_base+1, 14h);          Setup for Control Register #4
outp (I/O_base+0, <value>)
```

After initializing the control registers, enable memory by executing the following:

```
outp (I/O_base+1, 00h)
```

Once the control registers are initialized, you access the registers with new addresses using the *Index with RAM Enabled* values in Table 3-3.

3.7.2. Control Register #1

This write-only register selects the system memory address above one megabyte for the controller.

- If you want to address the controller above one megabyte, write 00h to control register #2, and write the value to select the desired address to control register #1, as determined by Table 3-6. Control register #1, bit D7 must be set to one (1).
- If you want to address the controller below one megabyte, write zeros to bits D6 to D0.

Table 3-4 illustrates the format of control register #1. Writing a value to data bits D0 to D6 sets the address; writing a value to data bit D7 determines the mode of data transfer between the controller and the system:

Table 3-4. Control Register #1 Format

Data Bit	Field
D7	AT/PC mode
D6	SA23
D5	SA22
D4	SA21
D3	SA20
D2	SA19
D1	SA18
D0	SA17

Note: *When AT/PC MODE is equal to 1, 16-bit memory transfer is set. When AT/PC MODE is equal to 0, 8-bit memory transfer is set.*

Bit D7 of control register #1 determines if the system accesses the dual-port memory using 8-bit or 16-bit transfers.

If the dual-port memory is mapped above 1 megabyte, always set D7 to 1 (16-bit mode).

If the memory is mapped below 1 megabyte, you must set D7 to 0 (8-bit mode) unless you have another 16-bit peripheral addressed in the same 128K block of system memory as the controller. The 128K block begins on an even 128K boundary. Table 3-5 summarizes this information.

Table 3-5. Control Register #1 (Bit D7)

System Memory Address	Other Peripherals in the Same 128K Block	D7 Value
Above 1 megabyte	Not applicable	1 (16 bit)
Below 1 megabyte	8 bit	0 (8 bit)
Below 1 megabyte	16 bit	1 (16 bit)

The following example selects the above one megabyte base address FA0000h, using the I/O base 218h:

```
outp (219h,05h);        /*Access CR#1 */
outp (218h,0FDh);       /* Set address, AT mode */
outp (219h,06h);        /* Access CR#2 */
outp (218h,00h);        /* Zero out */
outp (219h,0Ch);        /* Access CR#3 */
outp (218h,00h);        /* Zero out */
...
...
outp (219h,00h);        /* Enable DPRAM */
```

Table 3-6 defines all the memory base locations for controllers addressed above one megabyte:

Table 3-6. Memory Above One Megabyte

Address	Value for Control Register #1**
FC0000h	0FEh
FA0000h	0FDh
F80000h	0FCh
F60000h	0FBh
F40000h	0FAh
F20000h	0F9h
F00000h	0F8h
EE0000h	0F7h
EC0000h	0F6h
EA0000h	0F5h
E80000h	0F4h
E60000h	0F3h
E40000h	0F2h
E20000h	0F1h
E00000h	0F0h
DE0000h	0EFh
DC0000h	0EEh

Table 3-6. Memory Above One Megabyte (Continued)

Address	Value for Control Register #1**
DA0000h	0EDh
D80000h	0ECh
D60000h	0EBh
D40000h	0EAh
D20000h	0E9h
D00000h	0E8h

*These values assume that you are using 16-bit data transfers

3.7.3. Control Register #2

This write-only register selects the system memory address below one megabyte for the controller.

- If you want to address the controller above one megabyte, write 00h to control register #2.
- If you want to address the controller below one megabyte
  - Write 00h to D6 through D0 of control register #1
  - Write either 0 or 1 to bit D7 on control register #1 to choose 8- or 16-bit mode (see Table 3-5)
  - Write the value to select the desired address to control register #2, as determined by Table 3-8.

Table 3-7 illustrates the format of control register #2. Writing a value to data bits D0 through D5 sets the address.

Table 3-7. Control Register #2 Format

Data Bit	Field
D7	Not used
D6	Not used
D5	SA19
D4	SA18
D3	SA17
D2	SA16
D1	SA15
D0	SA14

This example selects the below one megabyte base address **D000:0h**, using the I/O base **218h**, with a 64K window:

```
outp (219h,05h);        /* Access CR#1, */
outp (218h,00h);        /* Zero out, PC mode */
outp (219h,06h);        /* Access CR#2 */
outp (218h,34h);        /* Below 1 MB RAM address */
outp (219h,0Ch);        /* Access CR#3 */
outp (218h,24h);        /* Set 64K upper window */
...
...
outp (219h,00h);        /* Enable DPRAM */
```

Table 3-8 defines all the memory base locations for controllers addressed under one megabyte. The addresses and offsets displayed in the table are valid for both the PC and AT mode of operation (see Subsection 3.7.4).

Table 3-8. Below One Megabyte Addressing

Memory Address and Offset	Control Register #2 (D5 to D0)	Control Register #1 (D6 to D0)	Valid System Window Sizes (Control Register #3)
8000:0000	20h	00h	16K, 32K, 64K
8000:4000	21h	00h	16K
8000:8000	22h	00h	16K, 32K
8000:C000	23h	00h	16K
9000:0000	24h	00h	16K, 32K, 64K
9000:4000	25h	00h	16K
9000:8000	26h	00h	16K, 32K
9000:C000	27h	00h	16K
A000:0000	28h	00h	16K, 32K, 64K
A000:4000	29h	00h	16K
A000:8000	2Ah	00h	16K, 32K
A000:C000	2Bh	00h	16K
B000:0000	2Ch	00h	16K, 32K, 64K
B000:4000	2Dh	00h	16K
B000:8000	2Eh	00h	16K, 32K
B000:C000	2Fh	00h	16K
C000:0000	30h	00h	16K, 32K, 64K
C000:4000	31h	00h	16K

Table 3-8. Below One Megabyte Addressing (Continued)

Memory Address and Offset	Control Register #2 (D5 to D0)	Control Register #1 (D6 to D0)	Valid System Window Sizes (Control Register #3)
C000:8000	32h	00h	16K, 32K
C000:C000	33h	00h	16K
D000:0000	34h	00h	16K, 32K, 64K
D000:4000	35h	00h	16K
D000:8000	36h	00h	16K, 32K
D000:C000	37h	00h	16K
E000:0000	38h	00h	16K, 32K, 64K
E000:4000	39h	00h	16K
E000:8000	3Ah	00h	16K, 32K
E000:C000	3Bh	00h	16K

Note: *When using under one megabyte addresses, choosing the AT or PC mode depends on whether other boards are addressed in the same 128K block of system memory space. All boards within a 128K block that begins on a 128K boundary must use the same mode of operation.*

3.7.4. Control Register #3

This write-only register selects and controls the dual-port memory window size and offset. This "window" is the portion of the dual-port memory the system processor sees at any one time. Table 3-9 illustrates the format of control register #3.

Table 3-9. Control Register #3 Format

Data Bit	Field
D7	WA18
D6	WA17
D5	ENBLSA16
D4	ENBLSA15
D3	ENBLSA14
D2	WA16
D1	WA15
D0	WA14

Data bits D0, D1, D2, D6, and D7 control the window's offset from the beginning of the 512K block of dual-port memory. (Bits D6 and D7 are set to zero (0) when only 128K of DRAM is present. This is the default case, when no additional SIMMs are added.)

Table 3-10 illustrates the format of the control register #3 window offset (bits D0 through D2, D6, and D7).

Table 3-10. Control Register #3 Window Offset

Data Bits					16K Window Offset	32K Window Offset	64K Window Offset	128K Sliding Window
D7	D6	D2	D1	D0				
0	0	0	0	0	+ 0	+ 0	+ 0	+ 0
0	0	0	0	1	+ 16K	+ 0	+ 0	+ 0
0	0	0	1	0	+ 32K	+ 32K	+ 0	+ 0
0	0	0	1	1	+ 48K	+ 32K	+ 0	+ 0
0	0	1	0	0	+ 64K	+ 64K	+ 64K	+ 0
0	0	1	0	1	+ 80K	+ 64K	+ 64K	+ 0
0	0	1	1	0	+ 96K	+ 96K	+ 64K	+ 0
0	0	1	1	1	+ 112K	+ 96K	+ 64K	+ 0
0	1	0	0	0	+ 128K	+ 128K	+ 128K	+ 128K
0	1	0	0	1	+ 144K	+ 128K	+ 128K	+ 128K
0	1	0	1	0	+ 160K	+ 160K	+ 128K	+ 128K
0	1	0	1	1	+ 176K	+ 160K	+ 128K	+ 128K
0	1	1	0	0	+ 192K	+ 192K	+ 192K	+ 128K
0	1	1	0	1	+ 208K	+ 192K	+ 192K	+ 128K
0	1	1	1	0	+ 224K	+ 224K	+ 192K	+ 128K
0	1	1	1	1	+ 240K	+ 224K	+ 192K	+ 128K
1	0	0	0	0	+ 256K	+ 256K	+ 256K	+ 256K
1	0	0	0	1	+ 272K	+ 256K	+ 256K	+ 256K
1	0	0	1	0	+ 288K	+ 288K	+ 256K	+ 256K
1	0	0	1	1	+ 304K	+ 288K	+ 256K	+ 256K
1	0	1	0	0	+ 320K	+ 320K	+ 320K	+ 256K
1	0	1	0	1	+ 336K	+ 320K	+ 320K	+ 256K
1	0	1	1	0	+ 352K	+ 352K	+ 320K	+ 256K
1	0	1	1	1	+ 368K	+ 352K	+ 320K	+ 256K
1	1	0	0	0	+ 384K	+ 384K	+ 384K	+ 384K

Table 3-10. Control Register #3 Window Offset (Continued)

Data Bits					16K Window Offset	32K Window Offset	64K Window Offset	128K Sliding Window
D7	D6	D2	D1	D0				
1	1	0	0	1	+ 400K	+ 384K	+ 384K	+ 384K
1	1	0	1	0	+ 416K	+ 416K	+ 384K	+ 384K
1	1	0	1	1	+ 432K	+ 416K	+ 384K	+ 384K
1	1	1	0	0	+ 448K	+ 448K	+ 448K	+ 384K
1	1	1	0	1	+ 464K	+ 448K	+ 448K	+ 384K
1	1	1	1	0	+ 480K	+ 480K	+ 448K	+ 384K
1	1	1	0	0	+ 496K	+ 480K	+ 448K	+ 384K

Note: *Bits D3 through D5 define the size of the system processor's window.*

Data bits D3 through D5 control the size of the system's window into dual-port memory. Table 3-11 illustrates the window size format of control register #3.

Table 3-11. Control Register #3 Sliding Window Sizes

Data Bits			Window Size
D5	D4	D3	
0	0	0	128K
1	0	0	64K
1	1	0	32K
1	1	1	16K

The following example selects the below one megabyte base address **D000:0h**, using the I/O base 218h, and sets a 64K sliding window with a 64K offset:

```
outp (219h,05h);        /* Access CR#1, */
outp (218h,00h);        /* Zero out, PC mode */
outp (219h,06h);        /* Access CR#2 */
outp (218h,34h);        /* Below 1 MB RAM address */
outp (219h,0Ch);        /* Access CR#3 */
outp (218h,24h);        /* Set 64K upper window + offset */
...
...
outp (219h,00h);        /* Enable DPRAM */
```

Setting the sliding window size determines how much of the dual-port RAM the system may access at one time. A 16K window allows four controllers to be configured under one megabyte within 64K of system memory as shown in Figure 3-1.

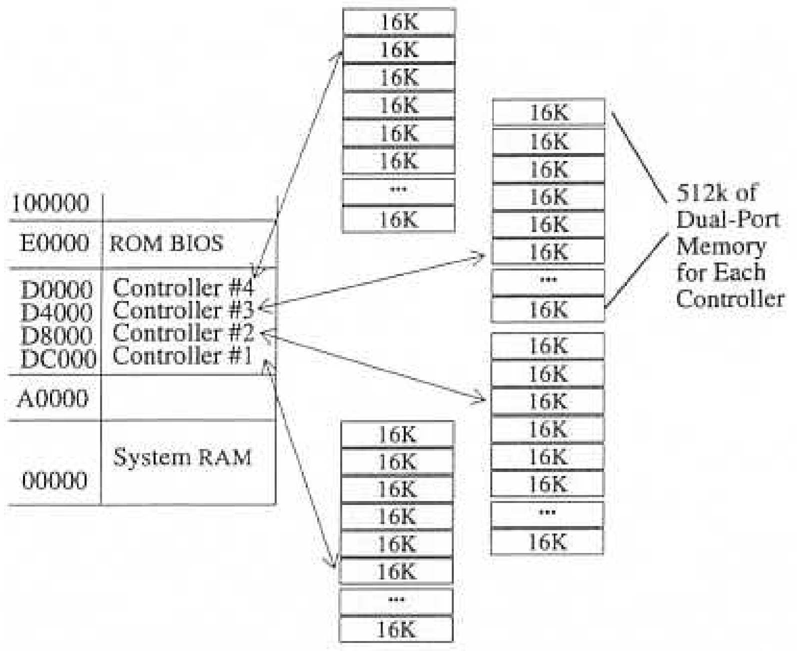


Figure 3-1. Four Controllers Addressed Under One Megabyte

The 512K of dual-port RAM in Figure 3-1 is with the optional SIMMS memory installed. Without SIMMS, only 128K of dual-port RAM is available.

3.7.5. Control Register #4

This write-only register selects the IRQ used to interrupt the system. Open-collector outputs allow more than one controller to share the same IRQ. (The open-collector output is a feature not used by Comtrol™ device drivers.)

The appropriate value for each IRQ appears in Table 3-12.

Table 3-12. Control Register #4 Interrupt Values

Interrupt	Control Register #4
IRQ3	08h
IRQ4	09h
IRQ5	0Ah
IRQ9	0Bh
IRQ10	0Ch

Table 3-12. Control Register #4 Interrupt Values (Continued)

Interrupt	Control Register #4
IRQ11	0Dh
IRQ12	0Eh
IRQ15	0Fh
Disabled	00h

The following example selects the below one megabyte base address **D000:0h**, using the I/O base 218h, with a 64K window and a 64K offset, and selects IRQ11:

```
outp (219h,05h);        /* Access CR#1, */
outp (218h,00h);        /*Zero out, PC mode */
outp (219h,06h);        /* Access CR#2 */
outp (218h,34h);        /* Below 1 MB RAM address */
outp (219h,0Ch);        /* Access CR#3 */
outp (218h,24h);        /* Set 64K upper window + offset */
outp (219h,14h);        /* Access CR#4 */
outp (218h,0Dh);        /* Set IRQ11 */
outp (219h,00h);        /* Enable DPRAM */
```

Section 4. Controller I/O Addresses

4.1. Overview

This section discusses the following issues:

- Controller internal I/O addresses
- Configuration control register
  - Transmit clock source
  - RS-232 and RS-422 synchronous support
  - DTR source
  - EPROM enable
- Configuration control register interrupts
  - Int 24h — control register read
  - Int 25h — control register write

4.2. Controller Internal I/O Addresses

Table 4-1 shows the internal I/O addresses for controller's devices.

Table 4-1. Internal I/O Addresses

Device	I/O Address
DMA Registers:	
DICM	9060h
DCH	9061h
DBC/DCC (Low-order byte)	9062h
DBC/DCC (High-order byte)	9063h
DBA/DCA (Low-order byte)	9064h
DBA/DCA (Middle-order byte)	9065h
DBA/DCA (High-order byte)	9066h
Reserved	9067h
DDC (Low-order byte)	9068h
DDC (High-order byte)	9069h
DMD	906Ah
DST	906Bh

Table 4-1. Internal I/O Addresses (Continued)

Device	I/O Address
Reserved	906Ch
Reserved	906Dh
Reserved	906Eh
DMK	906Fh
Interrupt control register	9071h
Timer 0 count register	9074h
Timer 1 count register	9075h
Timer 2 count register	9076h
Timer control word	9077h
SCC Port 1 (controller Port 2) command register	E1F0h
SCC Port 1 (controller Port 2) data register	E1F2h
SCC Port 0 (controller Port 1) command register	E1F4h
SCC Port 0 (controller Port 1) data register	E1F6h
SCC Port 3 (controller Port 4) command register	E3F0h
SCC Port 3 (controller Port 4) data register	E3F2h
SCC Port 2 (controller Port 3) command register	E3F4h
SCC Port 2 (controller Port 3) data register	E3F6h
SCC Port 5 (controller Port 6) command register	E5F0h
SCC Port 5 (controller Port 6) data register	E5F2h
SCC Port 4 (controller Port 5) command register	E5F4h
SCC Port 4 (controller Port 5) data register	E5F6h
SCC Port 7 (controller Port 8) command register	E7F0h
SCC Port 7 (controller Port 8) data register	E7F2h
SCC Port 6 (controller Port 7) command register	E7F4h
SCC Port 6 (controller Port 7) data register	E7F6h
SCC Port 9 (controller Port 10) command register	E9F0h
SCC Port 9 (controller Port 10) data register	E9F2h
SCC Port 8 (controller Port 9) command register	E9F4h
SCC Port 8 (controller Port 9) data register	E9F6h
SCC Port 11 (controller Port 12) command register	EBF0h
SCC Port 11 (controller Port 12) data register	EBF2h

Table 4-1. Internal I/O Addresses (Continued)

Device	I/O Address
SCC Port 10 (controller Port 11) command register	EBF4h
SCC Port 10 (controller Port 11) data register	EBF6h
SCC Port 13 (controller Port 14) command register	EDF0h
SCC Port 13 (controller Port 14) data register	EDF2h
SCC Port 12 (controller Port 13) command register	EDF4h
SCC Port 12 (controller Port 13) data register	EDF6h
Security GAL	EE7Eh
SCC Port 15 (controller Port 16) command register	EFF0h
SCC Port 15 (controller Port 16) data register	EFF2h
SCC Port 14 (controller Port 15) command register	EFF4h
SCC Port 14 (controller Port 15) data register	EFF6h
Configuration Control register	*

4.3. Configuration Control Register

This 16-bit register is accessible at the internal I/O address FE5Eh (see Subsection 4.3.4). It defines the configuration of the following programmable functions:

- RS-232 and RS-422 synchronous support
- DTR source
- EPROM enable

Table 4-2 defines the register bits.

4.3.1. RS-232 and RS-422 Synchronous Support

Setting the controller for RS-422 synchronous communication requires writing a value of 1 to Bit D11 of the configuration control register. (The default value of 0 sets the register for RS-232 synchronous.) In RS-232 synchronous mode, TxClk is always an input. In RS-422 mode, TxClk is always an output.

4.3.2. DTR Source

When Ports 1 or 2 are used in the full-duplex DMA mode, DTR modem control is unavailable as an SCC output. Bits D1 and D5 provide two choices for the DTR source for Ports 1 and 2. Clearing these bits to 0 allows the SCC Write Register to control DTR. Setting bits D1 or D5 to 1 allows bits D2 and D6 to control DTR for Ports 1 and 2.

Note: *The default source is the SCC DTR output.*

4.3.3. EPROM Enable

The 64K memory segment beginning at F0000h is reserved for EPROM. With two megabytes of DRAM installed on the controller, the 64K of DRAM at this location is not accessible. Disabling EPROM allows this 64K of DRAM to be accessed. Clearing bit D8 to 0 enables the EPROM, this is the default state. Setting bit D8 to 1 disables the EPROM and enables the DRAM from F0000h to FFFFFh.

Table 4-2. Configuration Control Register Bits

Register Bits	Value	
	0	1
D15	Reserved	
D14	Reserved	
D13	Reserved	
D12	Reserved	
D11	RS-232 synchronous	RS-422 synchronous
D10	Reserved	
D9	Reserved	
D8	EPROM enabled	EPROM disabled
D7	Reserved	Reserved
D6	Port 2 alternate DTR active	Port 2 alternate DTR inactive
D5	Port 2 DTR SCC source	Port 2 alternate DTR source
D4	Reserved	Reserved
D3	Reserved	Reserved
D2	Port 1 alternate DTR active	Port 1 alternate DTR inactive
D1	Port 1 DTR SCC source	Port 1 DTR alternate DTR source
D0	Reserved	Reserved

Note: *Always use int 24h and int 25h to communicate with this register. See Subsection 4.3.4 for guidelines for this register.*

4.3.4. Configuration Control Register Interrupts

The firmware contains two interrupts to read and write the configuration control register. These interrupts are:

Interrupt	Function
int 24h	Control register read
int 25h	Control register write

Always read the configuration control register first with **Int 24h**, set or clear the desired bits in the AX register, and then write the configuration control register with **Int 25h**. Do not modify any reserved bits.

4.3.4.1. Int 24h — Configuration Control Register Read

This interrupt reads the configuration control register. The value stored in the AX register is the value read from the configuration control register.

This example sets up Port 1 to use alternate DTR:

```
int 24h           ; Read register
or AX, 0002h      ; Set bit one
int 25h           ; Write value to register
```

4.3.4.2. Int 25h — Configuration Control Register Write

This interrupt writes the configuration control register. The value stored in the AX register is the value written to the configuration control register.

The following example disables EPROM:

```
int 24h           ; Read register
or AX, 0100h      ; Disable EPROM
int 25h;          ; Write value to register
```

Section 5. Dual-Port Memory

5.1. Overview

As discussed in the previous section, writing to the control registers sets the dual-port memory addresses in the system's address space. The controller reserves a 16K, 32K, 64K, or 128K block of system memory space, which begins at the base address set by control registers #1 and #2.

System base addresses can range from 13MB to 16MB (D00000 to FE00D0h), and under 1MB (080000 to 0FC000h). Refer to Tables 3-6 and 3-8 for the possible memory addresses found above and below one megabyte of memory.

5.2. Dual-Port Memory Map

Of the 128K of dual-port memory in a base configuration of the controller, 125K is available for control programs. Expanding local RAM does not change the basic mapping of dual-port memory (see Table 5-1).

Table 5-1. Dual-Port Memory Map

System Memory Address	Controller Memory Address	Description	Length (Bytes)
Base + C00h	00C00h	Unused	1F400h
Base + B80h	00B80h	Firmware user area	80h
Base + 400h	00400h	Firmware work space	780h
Base* + 0	00000h	Interrupt vector table	400h

* *Base memory address using a 128K window.*

The lower 400h bytes are reserved for the interrupt vector table. The firmware uses 400h to B80h for miscellaneous work space.

The 80h bytes from 00B80h to 00C00h are called the firmware user area. The firmware stores information about the controller in this area. The rest of the unused memory is available for control programs to use.

Figure 5-1 illustrates a controller installed above the first megabyte at the F00000 system base memory address.

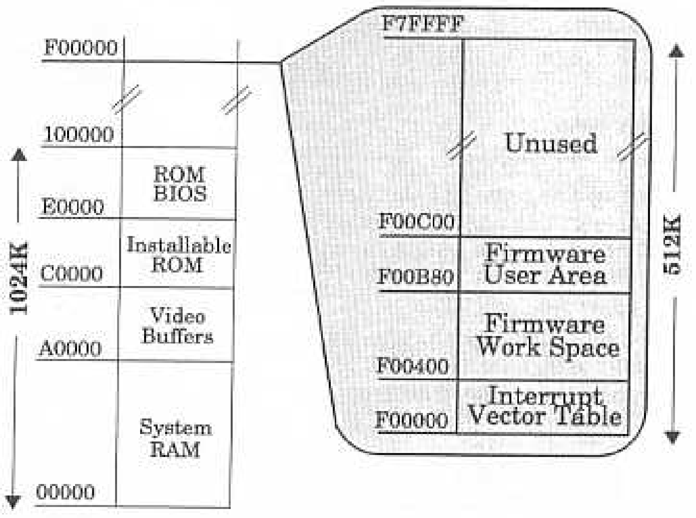


Figure 5-1. System View of the Controller's Dual-Port RAM

Figure 5-2 illustrates the arrangement of RAM on the controller.

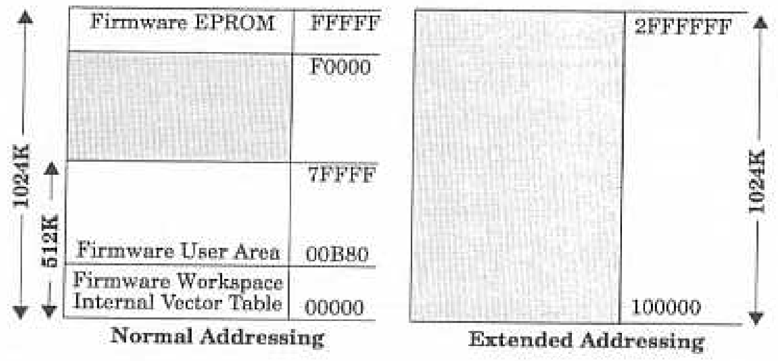


Figure 5-2. Controller's View of Its RAM

5.3. Firmware User Area Map

The firmware user area, located at the B80h controller memory address, is 80h bytes long. As the firmware executes, the user area fills with the information listed in Table 5-2.

Table 5-2. Firmware User Area Map

Offset	Description	Length
0h	Interaction flag 55AAh = controller active	2h bytes
2h	Boot flag 0000h = hard boot FFFFh = soft boot	2h bytes
4h	Old configuration map 0000h = not currently used	2h bytes
6h	Firmware release number	8h bytes
Eh	Open for control program release number	8h bytes
16h	Unused	4h bytes
1Ah	Local RAM map, V53 normal mode, one bit set per 64K	2h bytes
1Ch	Local RAM map, V53 extended mode, one bit set per 512K	2h bytes
1Eh	SCC port map, one bit set per port found	4h bytes
22h	Identification number 00081048h = Hostess *i*	4h bytes
26h	Invalid interrupt field	4h bytes
2Ah	Heartbeat counter	4h bytes
2Eh	Firmware utility command	1h byte
2Fh	Firmware utility status	1h byte
30h	Firmware utility message buffer	10h bytes
40h	Reserved for future use	40h bytes

The following list describes the information found in Table 5-2:

- **0h Offset**
  The *interaction flag* equals 55AAh when the controller is functioning properly.
- **2h Offset**
  The *boot flag* equals 0000h when the system powers up. It changes to FFFFh when the controller is reset by the software.
- **4h Offset**
  The two-byte *old config map* is not used, but space is allocated so the firmware user area is compatible with the firmware user area of other controllers.
- **6h Offset**
  The *firmware release number* is an ASCII string.
- **Eh Offset**
  Eight bytes are open for an ASCII string that identifies the control program release.
- **1Ah Offset**
  The *local RAM map, normal mode* is a word (16 bits) that has one bit set for every 64K block of memory found on the controller in the normal address mode (0h to FFFFFh range). This starts with the lowest memory block in the low-order bit.
- **1Ch Offset**
  The *local RAM map, extended mode* is a word (16 bits) that has one bit set for every 512K block of memory found on the controller in the extended address mode (100000h to 7FFFFFh range). This starts with the 512K block at the 100000h address in the low-order bit.
- **1Eh Offset**
  The *SCC port map* is a double word (32 bits) that has one bit set for each SCC channel that passes the SCC internal diagnostic test (see Table 5-3). This starts with the lowest memory block in the low order bit. A value of FFFFh in this map indicates that 16 channels passed.

Table 5-3. SCC Port Map

Bit	Hex Offset	SCC Chip	Channel	Port
0	0001	1	A	1
1	0002	1	B	2
2	0004	2	A	3
3	0008	2	B	4
4	0010	3	A	5
5	0020	3	B	6
6	0040	4	A	7
7	0080	4	B	8
8	0100	5	A	9
9	0200	5	B	10
10	0400	6	A	11
11	0800	6	B	12
12	1000	7	A	13
13	2000	7	B	14
14	4000	8	A	15
15	8000	8	B	16

- **22h Offset**
  The *identification number* for the Hostess *i* is 00081048h.
- **26h Offset**
  The *invalid interrupt field* marks any spurious interrupts that come into the interrupt controller. The firmware recovers from spurious interrupts, so the control program does not have to handle it.
- **2Ah Offset**
  The *heartbeat counter* is a simple counter. The default is 00000000. For example, the sample program's **timer1_isr** routine (on the *Developer's Toolkit* diskette) increments this counter to record interrupts generated by the local processor's timer.
- **2Eh Offset**
  The *firmware utility command* executes upon interrupt. The utility commands are listed in Table 5-4. See Subsection 11.2 for details on using the firmware utilities.

Table 5-4. Utility Commands

Command	Action
00h	A **null** command that is set for status and return.
01h (default)	Executes a control program at a vector in the message buffer: 30h = segment 32h = offset When complete, it sets the command status to *finished* (status=01).
02h	A copy command that uses the following message buffer parameters: 30h = source segment 32h = source offset 34h = destination segment 36h = destination offset 38h = count (two bytes) When complete, it sets the command status to *finished* (status=01)
03h through **FFh**	Reserved, currently uses a null command.

- **2Fh Offset**

  The *firmware utility status* holds the status of the firmware utility command. The values include:
  - **00h** default (command processing)
  - **01h** (command processing finished)
  - **02h** through FFh (reserved)
- **30h Offset**

  The *firmware utility message buffer* is a message buffer for commands. This buffer is initialized before the controller is interrupted. The controller can also return information in this buffer.

The remaining 40h bytes in the firmware user area are reserved for future use.

The following C data type, from the FIRMUSER.H file, defines the firmware data area:

```
typedef struct
{
    unsigned i_flag;                /* processor interaction flag */
    unsigned boot_flag;             /* boot/activity flag */
    unsigned cfg_map;               /* configuration map */
    char fw_release[8];             /* firmware release number */
    char sw_release[8];             /* control program release number */
    unsigned long dram_map1;        /* DRAM map */
    unsigned long dram_map2;        /* DRAM map */
    unsigned long scc_map;          /* SCC map */
    unsigned long board_id;         /* board ID */
    char ii_flag;                   /* invalid interrupt flag */
    char ii_type;                   /* invalid interrupt type */
    unsigned ii_cnt;                /* invalid interrupt count */
    unsigned long heartbeat;        /* heartbeat counter */
    char cmd;                       /* firmware utility command */
    char status;                    /* firmware utility status */
    char msg[16];                   /* firmware utility message buffer */
    char reserved1[0x40];           /* reserved for future use */
} FIRMUSER_T;
```

Example 5-1. Defining the Firmware Data Area

Section 6. Extended Addressing Mode

Most of the information in this section originated in the *V-Series μPD70236 (V53™) User's Manual* from NEC (July 1989).

6.1. Overview

The controller can have up to 8MB of local RAM. The lower megabyte is accessed by the V53 microprocessor in its *normal* addressing mode. The upper megabytes of RAM can be accessed only by using the *extended* addressing mode.

Extended addressing involves expanding the 20-bit physical addresses generated by the processor's Effective Address Generator (EAG) to 24-bit addresses. This is done by relocating the addresses.

Expanding an address uses the processor's address conversion table (see Table 6-1). The V53 microprocessor allocates 64 page registers (PGR1 to PGR64) to this table.

Note: *Extended addressing mode can not be used for I/O, DMA, or refresh cycles.*

6.2. Relocating Addresses

Relocating addresses involves executing certain instructions to switch the addressing mode. Relocation occurs in 16K bytes-per-page units. Memory space can be expanded up to 16MB (64 pages). Extended addresses are managed in 16K units because the 14 low-order address bits remain unchanged.

Figure 6-1 displays what happens when memory is expanded.

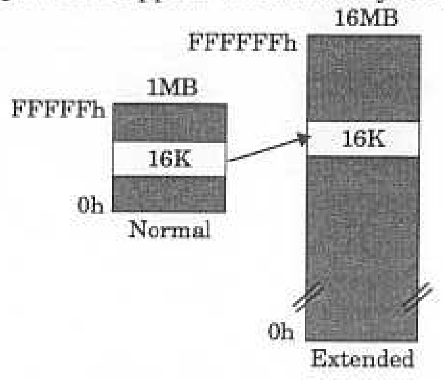
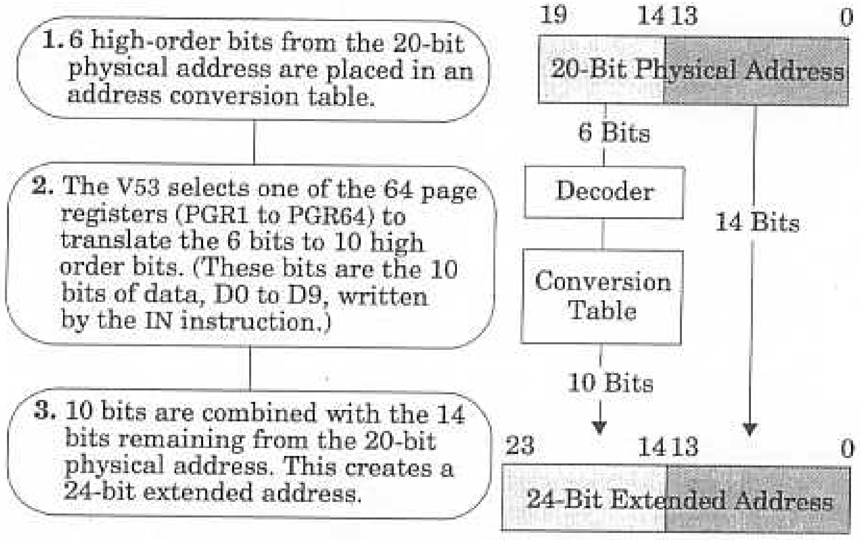


Figure 6-1. Expanding Memory

Flowchart 6-1 shows the method used to expand the processor's addresses by relocation.


1. 6 high-order bits from the 20-bit physical address are placed in an address conversion table.
2. The V53 selects one of the 64 page registers (PGR1 to PGR64) to translate the 6 bits to 10 high order bits. (These bits are the 10 bits of data, D0 to D9, written by the IN instruction.)
3. 10 bits are combined with the 14 bits remaining from the 20-bit physical address. This creates a 24-bit extended address.
19
14 13
0
20-Bit Physical Address
6 Bits
Decoder
14 Bits
Conversion Table
10 Bits
23
14 13
0
24-Bit Extended Address

Flowchart 6-1. Relocating Addresses

Note: *When the extended address mode is not specified, the 20-bit physical address is output directly, and 0 (zero) is output to the 4 high-order bits (A20 to A23).*

Although address expansion is transparent to executing code, the hardware does see the extended address mode.

If you use this mode of addressing, there is a performance penalty. Extended addressing requires one bus cycle to generate the extended address. This results in a 10 to 20 percent decrease in performance, as compared to a normal addressing mode.

6.2.1. Page Registers

Each of the 64 page registers are 16 bits wide. The page registers are placed at the FF00h to FF7Eh I/O addresses. The OUT instruction writes, and the IN instruction reads these 16-bit registers.

The effective bits of the page registers are the 10 low-order bits (D9 to D0). The ineffective bits are the 6 high-order bits (D15 to D10). These 6 bits are seen as 0 (zero) during a read operation, and are ignored during a write operation. These bits are unaffected by reset input.

Note: *The page registers should not be accessed while in extended address mode.*

Table 6-1 lists page registers and I/O addresses, along with corresponding bit values.

Table 6-1. Address Conversion Table

Bit						Page Register	I/O Address
A19	A18	A17	A16	A15	A14		
0	0	0	0	0	0	PGR1	FF00
0	0	0	0	0	1	PGR2	FF02
0	0	0	0	1	0	PGR3	FF04
0	0	0	0	1	1	PGR4	FF06
0	0	0	1	0	0	PGR5	FF08
0	0	0	1	0	1	PGR6	FF0A
0	0	0	1	1	0	PGR7	FF0C
0	0	0	1	1	1	PGR8	FF0E
0	0	1	0	0	0	PGR9	FF10
0	0	1	1	0	1	PGR10	FF12
0	0	1	0	1	0	PGR11	FF14
0	0	1	0	1	1	PGR12	FF16
0	0	1	1	0	0	PGR13	FF18
0	0	1	1	0	1	PGR14	FF1A
0	0	1	1	1	0	PGR15	FF1C
0	0	1	1	1	1	PGR16	FF1E
0	1	0	0	0	0	PGR17	FF20
0	1	0	0	0	1	PGR18	FF22
0	1	0	0	1	0	PGR19	FF24
0	1	0	0	1	1	PGR20	FF26
0	1	0	1	0	0	PGR21	FF28
0	1	0	1	0	1	PGR22	FF2A
0	1	0	1	1	0	PGR23	FF2C
0	1	0	1	1	1	PGR24	FF2E
0	1	1	0	0	0	PGR25	FF30

(Continued)

Table 6-1. Address Conversion Table (Continued)

Bit						Page Register	I/O Address
A19	A18	A17	A16	A15	A14		
0	1	1	0	0	1	PGR26	FF32
0	1	1	0	1	0	PGR27	FF34
0	1	1	0	1	1	PGR28	FF36
0	1	1	1	0	0	PGR29	FF38
0	1	1	1	0	1	PGR30	FF3A
0	1	1	1	1	0	PGR31	FF3C
0	1	1	1	1	1	PGR32	FF3E
1	0	0	0	0	0	PGR33	FF40
1	0	0	0	0	1	PGR34	FF42
1	0	0	0	1	0	PGR35	FF44
1	0	0	0	1	1	PGR36	FF46
1	0	0	1	0	0	PGR37	FF48
1	0	0	1	0	1	PGR38	FF4A
1	0	0	1	1	0	PGR39	FF4C
1	0	0	1	1	1	PGR40	FF4E
1	0	1	0	0	0	PGR41	FF50
1	0	1	0	0	1	PGR42	FF52
1	0	1	0	1	0	PGR43	FF54
1	0	1	0	1	1	PGR44	FF56
1	0	1	1	0	0	PGR45	FF58
1	0	1	1	0	1	PGR46	FF5A
1	0	1	1	1	0	PGR47	FF5C
1	0	1	1	1	1	PGR48	FF5E
1	1	0	0	0	0	PGR49	FF60
1	1	0	0	0	1	PGR50	FF62
1	1	0	0	1	0	PGR51	FF64

(Continued)

Table 6-1. Address Conversion Table (Continued)

Bit						Page Register	I/O Address
A19	A18	A17	A16	A15	A14		
1	1	0	0	1	1	PGR52	FF66
1	1	0	1	0	0	PGR53	FF68
1	1	0	1	0	1	PGR54	FF6A
1	1	0	1	1	0	PGR55	FF6C
1	1	0	1	1	1	PGR56	FF6E
1	1	1	0	0	0	PGR57	FF70
1	1	1	0	0	1	PGR58	FF72
1	1	1	0	1	0	PGR59	FF74
1	1	1	0	1	1	PGR60	FF76
1	1	1	1	0	0	PGR61	FF78
1	1	1	1	0	1	PGR62	FF7A
1	1	1	1	1	0	PGR63	FF7C
1	1	1	1	1	1	PGR64	FF7E

6.3. Initializing the Extended Addressing Mode

The $\overline{\text{RESET}}$ signal sets the extended address mode flag to 0 (normal address mode). The address conversion table is set to the following:

- Power-on (undefined)
- Normal reset (hold its state before reset)

A power-on (hard-boot), or a software reset from an **I/O_base+3 write**, generates the $\overline{\text{RESET}}$ signal.

6.4. Setting and Clearing the Extended Addressing Mode

Before setting or clearing the extended address mode, determine its status by reading the extended address mode register. This register is located at FF80h, bit 0 (extended address mode flag).

Note: *The extended address mode flag can only be read by an IN byte type instruction.*

The flag status includes:

- 0 (normal addressing mode)
- 1 (extended addressing mode)

The $\overline{\text{RESET}}$ signal clears the flag to 0 (the normal address mode).

The extended address mode is set and cleared by the following instructions:

- **BRKXA*n*** (sets the extended address mode)

  Reads vector ***n*** and branches to ISR (flag is set to 1)
- **RETXA*n*** (clears the extended address mode)

  Reads vector ***n*** and branches to ISR (flag is cleared to 0)

Note: *The BRKXA and RETXA instructions execute from the branch destination fetch cycle.*

The firmware defines interrupt vector **26h** to use with both the **BRKXA** and **RETXA** instructions. This ISR consists of one IRET instruction that causes the V53 processor to execute the instruction that follows **BRKXA** or **RETXA**.

Example 6-1 shows how to set and clear the extended addressing mode for the V53 microprocessor.

```
;This macro implements the V53 BRKXA instruction, which puts the processor
;in extended addressing mode
brkxa       MACRO
     dw     0e00fh   ; opcode for BRKXA instruction
     db     26h      ; interrupt vector type that BRKXA uses for
                     ; return. Interrupt service routine
                     ; is in the firmware
     ENDM
;This macro implements the V53 RETXA instruction, which puts the processor
;in normal addressing mode
retxa       MACRO
     dw     0f00fh   ; opcode for RETXA instruction
     db     26h      ; interrupt vector that RETXA uses for
                     ; return.  Interrupt service routine
                     ; is in the firmware
     ENDM
;This code fragment shows how to enter and exit extended addressing mode.
;Page registers PGR9 through PGR12 are used to translate addresses in the
;range 20000h - 2ffffh to addresses in the range 200000h - 20ffffh
PGR9         equ    0ff10h       ; page register 9 address
EXT_ADDR     equ    0080h        ; page reg contents for absolute address 200000h
NORM_ADDR equ       2000h        ; segment to reach PGR9-12 when in extended mode
        mov   cx,4
        mov   dx,PGR9            ; start with PGR9
        mov   ax,EXT_ADDR        ; bits 23-14 of extended address
ext_10:
        out   dx,ax              ; out to page register
        inc   ax                 ; extended address
        add   dx,2               ; next page register
        loop  ext_10
        mov   ax,NORM_ADDR       ; set ES segment to use PGR9-12
        mov   es,ax
        pushf                    ; set up stack for simulated int return
        push  cs                 ; segment of address to return to
        mov   ax,offset ext_20   ; offset of address to return to
        push  ax
        brkxa                    ; enable extended mode
ext_20:
        ;Code to access extended mode memory using segment in ES goes here
        ;Sets V53 back to normal mode when finished with extended mode
        pushf                    ; set up stack for simulated int return
        push  cs                 ; segment of address to return to
        mov   ax,offset ext_90   ; offset of address to return to
        push  ax
        retxa                    ; disable extended mode
ext_90:
```

Example 6-1. Extended Addressing Mode

Section 7. Direct Memory Access

Most of the information in this section originated in the *V-Series μPD70236 (V53™) User's Manual* from NEC (July 1989).

7.1. DMA Channels

The V53 microprocessor has four direct memory access (DMA) channels that transmit and receive data on Ports 1 and 2. Each channel either transmits or receives data on one port only. This allows for full-duplex DMA on Ports 1 and 2.

Table 7-1 lists the DMA channel port functions.

Table 7-1. DMA Channels

DMA Channel	Function
0	Transmit Port 1
1	Receive Port 1
2	Transmit Port 2
3	Receive Port 2

The DMA controller can access all RAM installed on the controller below 1 Mbyte. It cannot access extended addressed memory (see Section 6).

Note: *The information in this section is based on the μPD71071 mode.*

For information that describes how the Serial Communication Controller (SCC) handles DMA requests, refer to documentation for the 8530 SCC. However, disregard any reference to a potential problem when using DMA with an NMOS SCC. Hardware on the controller prevents the referenced condition from occurring.

Note: *When the DTR/REQ pin is used for DMA operation, the SCC pin cannot be used to provide the DTR modem control signal for the selected port.*

7.2. DMA Addressing

The V53 microprocessor's DMA control unit (DMAU) has 24 registers. Once a particular channel's register is preset by the channel register (DCH), then the DMAU channel registers can be accessed.

The following is a list of the DMAU registers.

- Channel register (DCH)
- Device control register (DDC)
- Status register (DST)
- Mask register (DMK)
- Base address register * (DBA)
- Current address register * (DCA)
- Base count register * (DBC)
- Current count register * (DCC)
- Mode control register * (DMD)

Note: ** One for each of the four DMA channels.*

To access the DMAU registers, set the following:

- System control register's (SCTL's) DMAM bit
  Set to 0 for µPD71071 mode
- On-chip peripheral selection register's (OPSEL's) DS bit
  Set to 1 to enable DMAU operation
- High-order address (A15 to A8)
- Low-order address (A7 to A4)
- Address signal bits (A3 to A0)

The high and low-order addresses should be set to a 906xh value, see Table 7-2.

Table 7-2 lists the addresses that access the DMAU registers.

Table 7-2. DMAU Register Addresses (µPD71071 Mode)

High-Order Address (A15 to A8)	Low-Order Address (A7 to A4)	Bits				Register	Operation
		A3	A2	A1	A0		
10010000 (90h)	0110 (6h)	0	0	0	0	DICM	Write
		0	0	0	1	DCH	Read/Write
		0	0	1	0	DBC/DCC (Low-order byte)	Read/Write
		0	0	1	1	DBC/DCC (High-order byte)	Read/Write
		0	1	0	0	DBA/DCA (Low-order byte)	Read/Write
		0	1	0	1	DBA/DCA (High-order byte)	Read/Write
		0	1	1	0	DBA/DCA (Upper-order byte)	Read/Write
		0	1	1	1	Reserved	
		1	0	0	0	DDC (Low-order byte)	Read/Write
		1	0	0	1	DDC (High-order byte)	Read/Write
		1	0	1	0	DMD	Read/Write
		1	0	1	1	DST	Read
		1	1	0	0	Reserved	
		1	1	0	1	Reserved	
		1	1	1	0	Reserved	
		1	1	1	1	DMK	Read/Write

Note: *The IOGA (Internal I/O Address) value in the SCTL register does not affect registers set for µPD71071 mode.*

As shown in the previous table, A3 to A0 selects the mode commands that read or write DMAU registers. These commands are issued by I/O instructions for the addresses set at the system I/O area.

Table 7-3 displays the addresses (A3 to A0) that access the µPD71071 mode commands.

Table 7-3. Accessing µPD71071 Mode Commands

Address	Command	Operation
0h	Initialize (DICM)	* Write
1h	Channel register (DCH)	* Read
	Channel register (DCH)	* Write
2h	Count register (DBC/DCC) Low-order byte	Read/Write
3h	Count register (DBC/DCC) High-order byte	Read/Write
4h	Address register (DBA/DCA) Low-order byte	Read/Write
5h	Address register (DBA/DCA) High-order byte	Read/Write
6h	Address register (DBA/DCA) Upper-order byte	* Read/Write
7h	Reserved	
8h	Device control register (DDC) Low-order byte	Read/Write
9h	Device control register (DDC) High-order byte	Read/Write
0Ah	Mode control register (DMD)	* Read/Write
0Bh	Status register (DST)	* Read
0Ch-0Eh	Reserved	
0Fh	Mask register (DMK)	* Read/Write

* Carried out by the byte IN/OUT instructions.

Note: *Only the address and operation combinations listed in this table are permitted; all others are prohibited.*

The following subsections describe the DMAU registers (µPD71071 mode commands).

7.2.1. DICM (Initialize Command Register)

The hardware and software initializes the DMAU with the DMA initialize command register's (DICM's) $\overline{\text{RESET}}$ signal.

Using the byte OUT instruction for address 0h, the 0 bit (RES bit) is either cleared or initialized with the following values:

- 0 - cleared (no operation)
- 1 - initialized (set)

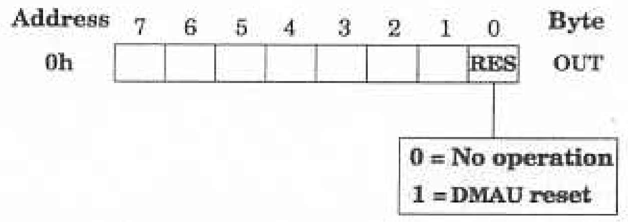


Figure 7-1. DICM Register

At the end of initialization, the RES bit is automatically cleared.

Table 7-4 displays the changes that occur to the DMAU registers when they are initialized.

Table 7-4. DMAU Register Initialization Changes

Register Name	Changes
Address register	No change
Count register	No change
Channel register	(CH0 selection) 7 6 5 4 3 2 1 0: | - | - | - | 0 | 0 | 0 | 0 | 1 |
Mode control register	All bits clear
Device control register	All bits clear
Status register	All bits clear
Mask register	All bits set (all channels masked)

7.2.2. DCH (Channel Register)

The DMA channel register (DCH) responds to both read and write operations. Set this register for a particular channel before setting the address, count, and mode control registers for that channel. The format for the DCH register differs for read and write operations.

7.2.2.1. DCH Read

The DCH read command is performed by a byte IN instruction at the 1h address, using 5 of the 8 bits.

Figure 7-2 shows the four SEL bits (SEL0 to SEL3) that display the *current* channel.

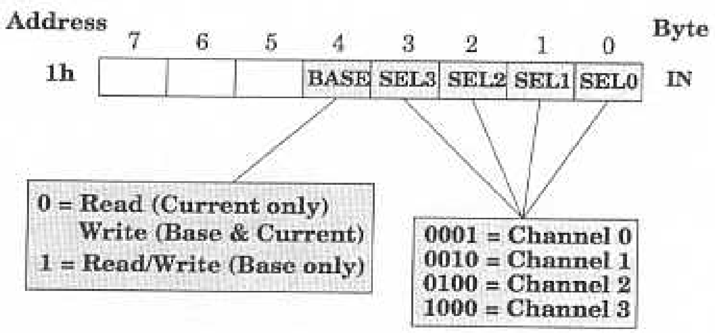


Figure 7-2. DCH Register (Read)

A BASE bit (bit 4) shows the *current* read/write access conditions for both the count and address registers (register access mode).

When the BASE bit is equal to 1 (set), the base registers (DBC and DBA) have both read and write access. When this bit is equal to 0 (cleared), the current registers (DCC and DCA) have read access, and the base and current registers (DBC/DCC and DBA/DCA) have write access.

7.2.2.2. DCH Write

The DCH write command is performed by a byte OUT instruction at the 1h address, using 3 of the 8 bits.

Figure 7-3 shows the two SELCH bits (bits 0 and 1) that select one of the four DMA channels for CPU programming.

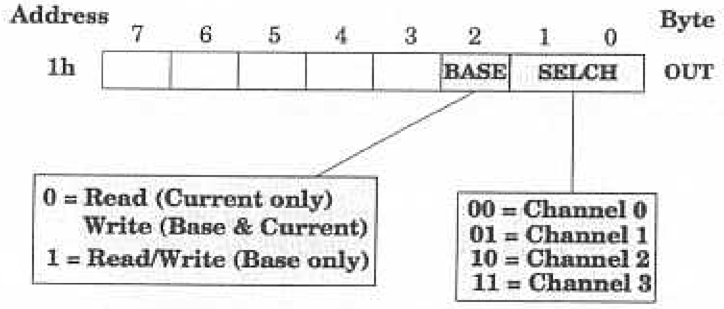


Figure 7-3. DCH Register (Write)

The BASE bit (bit 2) specifies the read/write access conditions for both the count and address registers (register access mode).

When the BASE bit is equal to 1 (set), the base registers (DBC and DBA) have both read and write access. When this bit is equal to 0 (cleared), the current registers (DCC and DCA) have read access, and the base and current registers (DBC/DCC and DBA/DCA) have write access.

7.2.3. DBC/DCC (Base/Current Count Register)

The DBC/DCC count register consists of the following 16-bit registers for each of the four DMA channels:

- DBC (DMA base count register)
- DCC (DMA current count register)

The DBC and the DCC both have the following addresses:

- 2h (low-order byte)
- 3h (high-order byte)

The DCH (DMA channel register) selects the read/write access conditions for the DBC and DCC.

The DBC holds a count value until a new count is set. This value transfers to the DCC during autoinitialization when a terminal count or $\overline{\text{END}}$ condition occurs.

Figure 7-4 shows the DBC/DCC format.

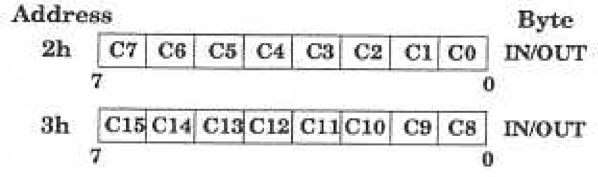


Figure 7-4. DBC/DCC Read/Write Command Register

Note: *The word IN/OUT instruction reads or writes this register.*

7.2.4. DBA/DCA (Base/Current Address Register)

The DBA/DCA address register consists of the following 24-bit registers for each of the four DMA channels:

- DBA (DMA base address register)
- DCA (DMA current address register)

The DBA and the DCA both have the following addresses:

- 4h (low-order byte)
- 5h (middle-order byte)
- 6h (high-order byte)

The DCH (DMA channel register) selects the read/write access conditions for the DBA and DCA.

The DBA holds an address value until a new address is set. This value transfers to the DCA register during autoinitialization.

During each DMA transfer, the DCA is updated by the following values:

- 2 (during word transfers)
- 1 (during byte transfers)

Figure 7-5 shows the DBA/DCA format.

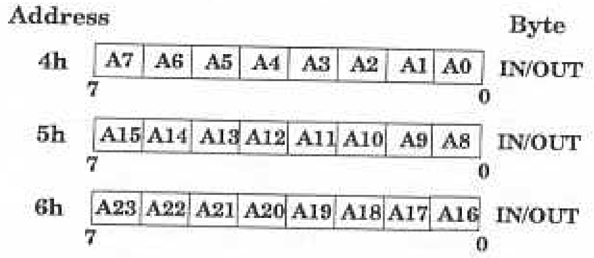


Figure 7-5. DBA/DCA Read/Write Command Format

Note: *Either the word or byte IN/OUT instructions read and write the low-order and high-order bytes (addresses 4h and 5h) of this register.*

Only the byte IN/OUT instruction reads and writes the upper-order byte (address 6h) of this register.

The DBA/DCA address register defines the physical memory address. This register does not use the extended addressing mode that is described in a previous section.

7.2.5. DDC (Device Control Register)

The DMA device control register (DDC) programs the DMA operations for all channels. It is a 16-bit register located at 8h and 9h, and it is accessed by word IN/OUT instructions.

Figure 7-6 shows the bits used at 8h and 9h.

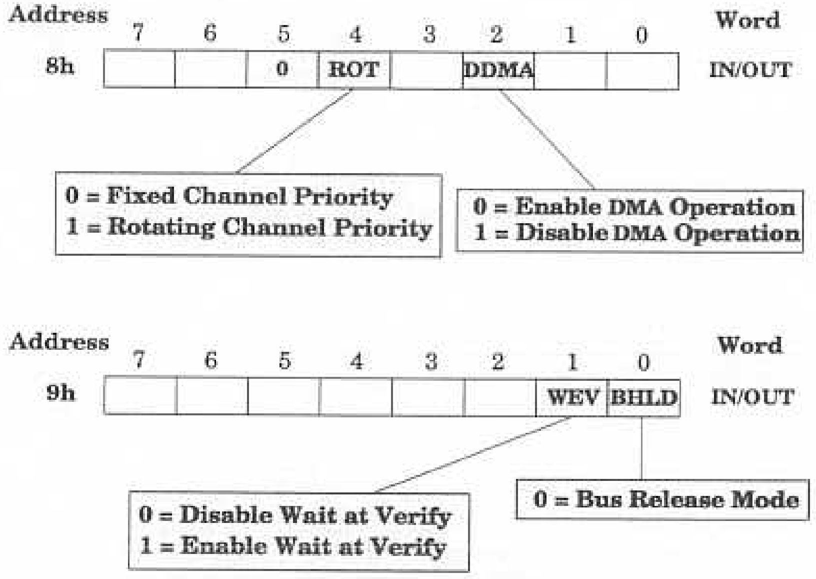

Address
7 6 5 4 3 2 1 0
Word
8h
0
ROT
DDMA
IN/OUT
0 = Fixed Channel Priority
1 = Rotating Channel Priority
0 = Enable DMA Operation
1 = Disable DMA Operation
Address
7 6 5 4 3 2 1 0
Word
9h
WEV
BHLD
IN/OUT
0 = Disable Wait at Verify
1 = Enable Wait at Verify
0 = Bus Release Mode

Figure 7-6. DDC Register

The following list describes the bits located at 8h:

- **DDMA** (bit 2)
  This bit enables or disables the DMAU. Setting the DDMA bit temporarily stops DMA operation. When DDMA is set to:
  - **0**, the DMA is enabled, and operation resumes in the same state before it was disabled.
  - **1**, the DMA is disabled.
- **ROT** (bit 4)
  This bit selects the channel priority. When ROT is set to:
  - **0**, a fixed priority is set. With a fixed priority, Channel 0 always has the highest priority, and channel 3 always has the lowest priority.
  - **1**, a rotating priority is set. With a rotating priority, the channel last served becomes the lowest priority. This insures service to the low-priority channels, as well as to the high-priority channels.

- **EXW** (bit 5)

  This bit sets the write timing, and is always set to **0** (normal timing).

The following list describes the bits located at 9h:

- **BHLD** (bit 0)

  The BHLD bit sets the DMA transfer bus mode, and is always set to **0** (bus release mode).

  In the bus release mode, the right to use the bus returns to the CPU at the end of each service. When multiple DMA requests occur simultaneously, another bus cycle can intervene between these requests; the response to DMA requests is slow.

- **WEV** (bit 1)

  The WEV bit enables and disables wait state insertion by the external $\overline{\text{READY}}$ signal and programmable wait-at-verify transfer. This bit is set to **1** (enable wait at verify).

7.2.6. DMD (Mode Control Register)

The DMA mode control register (DMD) sets the operation mode for each channel. It is located at 0Ah and is accessed by byte IN/OUT instructions.

The DMD register uses the following 7 bits of this 8 bit register:

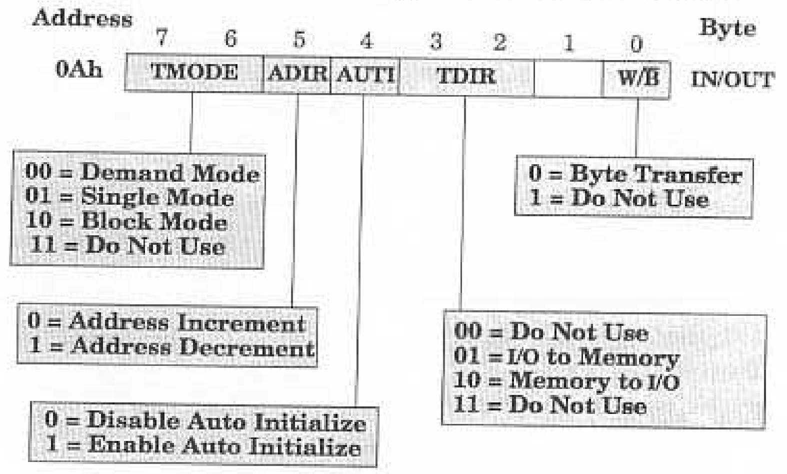


Figure 7-7. DMD Register

- **TMODE** (bits 6 and 7)

  These bits select the transfer direction for each channel, activating an appropriate control signal. This occurs during memory-to-I/O transfer.

  These bits should always be set to single mode (**01**), where a channel transfers a single byte or word and then the DMAU enters an idle state.

- **ADIR** (bit 5)

  This bit specifies the direction of the current address register update. When ADIR is set to:

  - **0**, the address increments by 1.
  - **1**, the address decrements by 1.

- **AUTI** (bit 4)

  This bit disables (0) or enables (1) the autoinitialization function. The autoinitialization function automatically initializes the address and count registers when a terminal count ($\overline{\text{TC}}$) or $\overline{\text{END}}$ is generated.

  In this situation, the following occurs:

  - The contents of the base address register transfer to the current address register.
  - The contents of the base count register transfer to the current count register.
  - The applicable bit of the mask register clears.

- **TDIR** (bits 2 and 3)

  These bits specify the direction of the memory and I/O transfer. When TDIR is set to:

  - **00**, a verify occurs (a transfer does not take place-**do not use**).
  - **01**, a write occurs from I/O to memory (use for Channels 1 and 3).
  - **10**, a read occurs from memory to I/O (use for Channels 0 and 2).
  - **11**, a transfer is not allowed (**do not use**).

- **W/$\overline{\text{B}}$** (bit 0)

  This bit specifies a byte or word transfer. When **W/$\overline{\text{B}}$** is set to:

  - **0**, a byte is transferred.
  - **1**, a word is transferred (**do not use**).

During byte transfers, address registers are updated by +/-1 and count registers are updated by -1.

7.2.7. DST (Status Register)

The DMA status register (DST) holds the status information for each channel. The byte IN instruction reads this 8 bit register, located at 0Bh (see Figure 7-8).

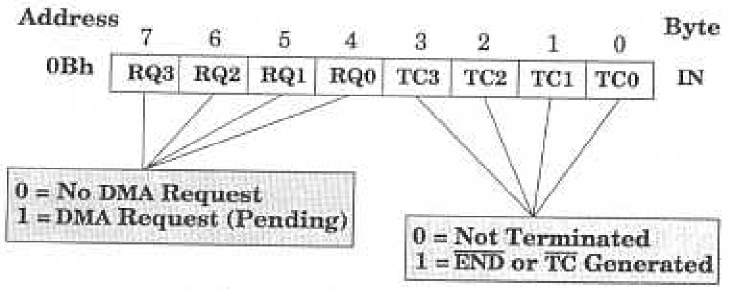


Figure 7-8. DST Register

DST contains the following bits:

- **TC0-TC3** (bits 0-3)
  These bits determine if termination has occurred. If TC*n* is set to:
  - **0**, the service has not ended (for each read).
  - **1**, service has ended with a terminal count ($\overline{\text{TC}}$) or an $\overline{\text{END}}$.
- **RQ0-RQ3** (bits 4-7)
  These bits determine if a DMA service request exists. If RQ*n* is set to:
  - **0**, an active DMA service request does not exist.
  - **1**, an active DMA service request exists.

7.2.8. DMK (Mask Register)

The DMA mask register (DMK) disables or enables masking for DMA channels. The byte IN/OUT instruction accesses this 8 bit register, located at 0Fh (see Figure 7-9).

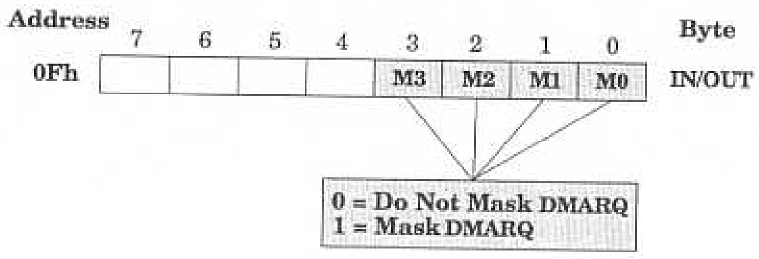


Figure 7-9. DMK Register

When DMK (M0-M3) is set to:

- **0**, the channel is not masked.
- **1**, the channel is masked.

The mask bit is not set for a channel that was autoinitialized.

Section 8. Interrupts

8.1. Interrupting the System Processor

The controller uses IRQs 3-5, 9-12, or 15. A system write to control register #4 sets the IRQ that the controller uses to interrupt the system processor.

The control program interrupts the system processor on the IRQ line by writing 0008h to the EF60h I/O address. After a two-microsecond delay, clear the interrupt by writing 0000h to EF60h. (The delay is executed with three consecutive **jmp short $+2** statements.)

Example 8-1 sets and clears an interrupt to the system processor.

```
mov     dx,ef60h      ; dx = interrupt address
mov     ax,0008h      ; ax = value to set interrupt low
out     dx,ax         ; set the interrupt

jmp     short $+2     ; delay
jmp     short $+2     ; delay
jmp     short $+2     ; delay

mov     ax,0000h      ; ax = value to clear interrupt high
out     dx,ax         ; clear the interrupt
```

Example 8-1. Setting and Clearing a System Processor Interrupt

Multiple controllers can share the same IRQ line. To share an IRQ, the Interrupt Service Routine (ISR) on the system computer must include code that identifies the controller generating the interrupt.

8.2. Interrupting the Controller

Writing to the **I/O base+2** address causes the system processor to interrupt the controller on the V53 microprocessor's interrupt line 3:

```
outp(0x218+2,0);          /* write anything to io_base+2 */
```

This generates an interrupt vector type **33h**.

The control program's ISR must clear the interrupt after processing it. The interrupt is cleared by writing **20h** to the interrupt control register at address 9070h:

```
mov     dx,09070h     ; dx = interrupt control register
mov     al,20h        ; al = 20h to clear the interrupt
out     dx,al         ; clear the interrupt
iret                  ; return from interrupt
```

Example 8-2. Clearing a Controller Interrupt

Note: *The firmware uses this interrupt to invoke the firmware utility commands. The control program must change the 33h interrupt vector table entry to the system ISR vector before using this interrupt.*

8.3. Internal Interrupt Service Routine

The control program must have interrupt service routines (ISRs) for all interrupts it uses. The interrupt vector table stores the address of the ISR, so when the interrupt comes in, execution immediately jumps to the correct ISR.

The ISR processes and clears the interrupt, and executes the **iret** instruction to return from the interrupt. The processing of the interrupt is specific to the control program. To clear the interrupt, write **20h** to the interrupt control register at address 9070h.

Do not disable or enable other interrupts with the **cli** and **sti** instructions while in an ISR, as it would let another interrupt come in before the current interrupt clears.

Example 8-3 shows a sample timer ISR.

```
timer_isr  proc
    push   ax           ; save registers
    push   dx

                        ; do internal processing

    mov    dx,09070h    ; dx = interrupt control register
    mov    al,20h       ; al = value to clear interrupt
    out    dx,al        ; clear the interrupt

    pop    dx           ; restore registers
    pop    ax
    iret                ; return from the interrupt routine
timer_isr  endp
```

Example 8-3. Timer ISR

8.4. Interrupt Vectors

Each interrupt has a vector type number. The address of the ISR requires four bytes and is placed in the interrupt vector table at *vector_type* * 4. The two-byte offset address is the first to be stored in the interrupt vector table, followed by the two-byte segment address.

Example 8-4 stores the system ISR address in the interrupt vector table.

```
xor   ax,ax                  ; zero ax
mov   es,ax                  ; segment of vector table = 0
mov   bx,33h*4               ; get vector table location
mov   ax,offset system_isr   ; offset of isr
mov   es:[bx],ax             ; store in vector table
mov   ax,cs                  ; segment of isr routine
mov   es:[bx+2],ax           ; store in vector table
```

Example 8-4. Storing the System ISR Address

Table 8-1 lists the interrupts the controller can use, along with the interrupt vector types.

Table 8-1. Interrupt Vectors

Interrupt	Vector Type Number	Vector Table Location	Control Program Modifiable	Hardware/ Software Generated
NMI	2h	8h	No	H/W
DEBUGGER	20h	80h	No	S/W
RAM_QUERY	21h	84h	No	S/W
DEBUG_PORT	22h	88h	No	S/W
CONFIG_QUERY	23h	8Ch	No	S/W
CONFIGURATION_ CONTROL_REGISTER_READ	24h	90h	No	S/W
CONFIGURATION_ CONTROL_REGISTER _WRITE	25h	94h	No	S/W
ENTER/EXIT_ EXTENDED_MODE	26h	98h	Yes	S/W
TURBO_DEBUGGER _REMOTE	27h	9Ch	No	S/W

* *Operates in cascade mode, so it does not use this vector table entry.*

** *Generated by the system to the controller.*

Table 8-1. Interrupt Vectors (Continued)

Interrupt	Vector Type Number	Vector Table Location	Control Program Modifiable	Hardware/ Software Generated
DMA terminal count	30h	C0h	Yes	H/W
8530 bank 1	31h	*	No	H/W
8530 bank 2	32h	*	No	H/W
SYSTEM** (I/O + 2 write)	33h	CCh	Yes	H/W
TIMER 0	34h	D0h	Yes	H/W
TIMER 1	35h	D4h	Yes	H/W
TIMER 2	36h	D8h	Yes	H/W
IRQ7 (Catches invalid interrupts)	37h	DCh	No	H/W
SCC_base	80h	200h	Yes	H/W

**Operates in cascade mode, so it does not use this vector table entry.*

*** Generated by the system to the controller.*

The firmware sets up eighteen interrupt vectors, eleven of which should not be changed and seven that can be modified by the control program.

The following list describes each of the interrupts listed in Table 8-1:

- **NMI** interrupt (Non-Maskable Interrupt, type 2h)
  This external interrupt occurs only on a controller that is set up for development, which has reset and debug switches. The debug switch triggers an NMI, which invokes the debugger.
- **DEBUGGER** interrupt (type 20h)
  This software interrupt invokes the firmware debugger.
- **RAM_QUERY** interrupt (type 21h)
  This software interrupt returns the first segment that is open for control program use in the AX register. It can be used to determine where to load the control program.
- **DEBUG_PORT** interrupt (type 22h)
  This software interrupt changes the firmware's debugging port (the first serial port) to the one specified in the AL register.

- **CONFIG_QUERY** interrupt (type 23h)
  This software interrupt returns information in the firmware data area about the number of ports and amount of dual-ported RAM on the controller. It is dependent on the AL register's *on entry* value:
  - If the AL register = 0 on entry, the **old config map** is returned in the AX register.
  - If the AL register = 1 on entry, the **dual-ported RAM** map is returned. The low word is in the AX register, and the high word is in the BX register.
  - If the AL register = 2 on entry, the SCC **port map** is returned. The low word is in the AX register, and the high word is in the BX register.
- **CONFIGURATION_CONTROL_REGISTER_READ** interrupt (type 24h)
  This software interrupt reads the configuration control register value and stores it in the AX register.
- **CONFIGURATION_CONTROL_REGISTER_WRITE** interrupt (type 25h)
  This software interrupt writes the value specified in the AX register to the configuration control register.
- **ENTER/EXIT_EXTENDED_MODE** interrupt (type 26h)
  This interrupt is used to set the V53 into or out of extended addressing mode.
- **TURBO_DEBUGGER_REMOTE** interrupt (type 27h)
  This interrupt is used to invoke the remote Borland Turbo Debugger kernel on the controller.
- **DMA terminal count** interrupt (type 30h)
  This interrupt is used by the DMA controller to indicate that the DMA transfer is complete.
- **8530** interrupt (type 31h and type 32h)
  These interrupts are cascaded from the SCC interrupt. They should not be used or modified. They do not use the controller's vector table entries. For more information on SCC interrupt types, see Subsection 8.6.
- **SYSTEM** interrupt (type 33h)
  This interrupt is generated when the system processor writes to the **I/0 base+2** address to interrupt the controller. This vector should be replaced with the control program's vector to process system interrupts.
- **TIMER 0** interrupt (type 34h)
  This interrupt is generated by **timer 0**. This vector should be replaced with the control program's vector if the control program uses **timer 0**.

- **TIMER 1** interrupt (type 35h)
  This interrupt is generated by **timer 1**. This vector should be replaced with the control program's vector if the control program uses **timer 1**.
- **TIMER 2** interrupt (type 36h)
  This interrupt is generated by **timer 2**. This vector should be replaced with the control program's vector if the control program uses **timer 2**.
- **IRQ7** interrupt (type 37h)
  This interrupt collects all invalid interrupts.
- **SCC_base** interrupt
  These interrupts are placed every eight bytes (for every two type numbers) in the interrupt vector table, beginning with type 80h. The control program must initialize the SCC interrupts, because the firmware does not initialize them.

8.5. Interrupt Mask Register (IMR)

The V53 has an interrupt mask register (IMR), located at 9071h, that is functionally equivalent to the Intel 8259 mask register. Use this register to individually mask a hardware interrupt request:

- **0** resets the interrupt channel.
- **1** sets the mask for an interrupt channel (INT0 through INT7).

If you mask an interrupt channel, it does not affect the operation of other channels.

Table 8-2 lists hardware interrupts and the corresponding IMR bits.

Table 8-2. Hardware Interrupt IMR Bits

Hardware Interrupt	IMR Bit
DMA terminal count	INT0
8530 bank 1	INT1
8530 bank 2	INT2
SYSTEM (I/O + 2 write)	INT3
TIMER 0	INT4
TIMER 1	INT5
TIMER 2	INT6
IRQ7 (Catches invalid interrupts)	INT7

8.6. SCC Interrupt Vector Types

Each Serial Communications Controller (SCC) port generates the following four types of interrupts, which are daisy-chained:

- Transmit buffer empty
- Receive character available
- Receive special condition
- External/status change

In write register 9 of the SCC, set **VIS=1**, **NV=0**, and **STATUS_HIGH/STATUS_LOW=0**. When the processor requests an interrupt vector, the SCC places the interrupt vector specified in Write Register 2 on the bus. This vector is modified to contain status information in Bits 1, 2, and 3 that shows the type of interrupt generated.

Table 8-3 displays SCC interrupt vector binary values.

Table 8-3. SCC Interrupt Vector Binary Values

Interrupt Type	Interrupt Vector (Binary)
Even Numbered Ports	
Transmit Buffer Empty	xxxx0000
External/Status Change	xxxx0010
Receive Character Available	xxxx0100
Special Receive Condition	xxxx0110
Odd Numbered Ports	
Transmit Buffer Empty	xxxx1000
External/Status Change	xxxx1010
Receive Character Available	xxxx1100
Special Receive Condition	xxxx1110

The SCC interrupt vectors are placed at eight-byte increments in the interrupt vector table.

Use the following steps to find the interrupt vector table location for a receive character available interrupt on Port 5:

1. Combine the base vector's binary value with the interrupt vector's binary value to arrive at the modified vector's value:

   A0h (Base Vector) = **10100000**(binary value)

   xxxx110(interrupt vector's binary value for the receive character available interrupt on an odd numbered port)

   Modified Vector's Binary Value **10101100= 0ACh** (modified vector)

2. Multiply the modified vector by 4 to find the interrupt vector table location.

   0ACh * 4 = **2B0h** (interrupt vector table location)

Table 8-4 lists the available interrupt vector table locations.

Table 8-4. Interrupt Vector Table Locations

Port	Base Vector Type	Transmit Buffer Empty	External/ Status Change	Receive Character Available	Special Receive Condition
1	80h	220h	228h	230h	238h
2	80h	200h	208h	210h	218h
3	90h	260h	268h	270h	278h
4	90h	240h	248h	250h	258h
5	A0h	2A0h	2A8h	2B0h	2B8h
6	A0h	280h	288h	290h	298h
7	B0h	2E0h	2E8h	2F0h	2F8h
8	B0h	2C0h	2C8h	2D0h	2D8h
9	C0h	320h	328h	330h	338h
10	C0h	300h	308h	310h	318h
11	D0h	360h	368h	370h	378h
12	D0h	340h	348h	350h	358h
13	E0h	3A0h	3A8h	3B0h	3B8h
14	E0h	380h	388h	390h	398h
15	F0h	3E0h	3E8h	3F0h	3F8h
16	F0h	3C0h	3C8h	3D0h	3D8h

8.7. Initializing SCC Interrupt Vectors

The following sample, from the CPC.C sample file, initializes SCC interrupt vectors. Each vector requires 4 bytes, and every second vector is not used.

The SCC modifies Bits 3, 2, and 1 of the base vector type, but does not modify Bit 0. The unused vectors are already initialized to point to an invalid interrupt ISR by the firmware, so they are not altered.

```
void vector_init(void)
{
                         /* Table of SCC interrupt vectors */
  static void interrupt far (*vectors[NUMLINES][4])() =
  {
      {line01_TBE, line01_ESC, line01_RCA, line01_SRC},
      {line00_TBE, line00_ESC, line00_RCA, line00_SRC},
      {line03_TBE, line03_ESC, line03_RCA, line03_SRC},
      {line02_TBE, line02_ESC, line02_RCA, line02_SRC},
      {line05_TBE, line05_ESC, line05_RCA, line05_SRC},
      {line04_TBE, line04_ESC, line04_RCA, line04_SRC},
      {line07_TBE, line07_ESC, line07_RCA, line07_SRC},
#if defined HOSTESS186 || SMARTH
      {line06_TBE, line06_ESC, line06_RCA, line06_SRC}
#else if defined HOSTESSi
      {line06_TBE, line06_ESC, line06_RCA, line06_SRC},
      {line09_TBE, line09_ESC, line09_RCA, line09_SRC},
      {line08_TBE, line08_ESC, line08_RCA, line08_SRC},
      {line11_TBE, line11_ESC, line11_RCA, line11_SRC},
      {line10_TBE, line10_ESC, line10_RCA, line10_SRC},
      {line13_TBE, line13_ESC, line13_RCA, line13_SRC},
      {line12_TBE, line12_ESC, line12_RCA, line12_SRC},
      {line15_TBE, line15_ESC, line15_RCA, line15_SRC},
      {line14_TBE, line14_ESC, line14_RCA, line14_SRC}
#endif
  };
int linenum;       /* line number */
  int i;
  int vector_type;/* interrupt vector type number */
set_vector(SYS_TYPE,system_isr);/* initialize system ISR vector */
                         /* Initialize the vector table entries for each SCC */
  vector_type = SCCBASE_TYPE;/* type for 1st SCC */
  for(linenum = 0;linenum < NUMLINES;linenum++)  /* for each line */
  {
    for(i = 0;i < 4;i++)/* there are 4 vectors for each line */
    {
      set_vector(vector_type,vectors[linenum][i]);
      vector_type += 2;/* skip unused vector */
    }
  }
}
```

Section 9. Timers

Most of the information in this section originated in the *V-Series μPD70236 (V53™) User's Manual* from NEC (July 1989).

9.1. TCU Operation Procedure

After the power is turned on, the state of the timer control unit (TCU) is defined by the firmware. The TCU consists of three sets of 16-bit timer/counters (TCT2-TCT0), which are initialized by the timer mode register (TMD) and by the timer clock selection register (TCKS).

The timer mode register (TMD), which selects the operating mode for each timer/counter, by default is set to the following:

- Count = binary count
- Count mode = mode 2
- Read/write mode = write lower byte then write high byte

The TMD also issues latch commands for the timer/counters.

The timer clock selection register (TCKS), which selects the clock source for the timer/counters, by default is set to the following:

- Clock input = internal clock
- Divisor = 32

See the TMD (Subsection 9.2.1) and TCKS (Subsection 9.2.2) descriptions for more details on these settings.

9.2. TCU Registers

TCU read/write operations and command issuing are performed by I/O instructions. Table 9-1 lists the TCU register/command addresses.

Table 9-1. TCU Register/Command Addresses

Address	Register/ Command	Operation
9074h	TCT0 TST0	Read/Write Read
9075h	TCT1 TST1	Read/Write Read
9076h	TCT2 TST2	Read/Write Read

Table 9-1. TCU Register/Command Addresses

Address	Register/ Command	Operation
9077h	TMD	Write
FFF0h	TCKS	Read/Write

Writing to the timer mode register (TMD) issues a **Count Latch**, or **Multiple Latch** command (see Subsection 9.2.1). This sets the operation mode (binary/BCD count mode, count mode, and read/write mode) and latches the counter value of each counter in the TCU.

The timer/counter registers (TCT2-TCT0) write the number of counts to each timer/counter and read the *count data* from each timer/counter. Usually, a **Count Latch** or **Multiple Latch** command is issued, and the *count data* is latched before being read.

The timer status registers (TST2-TST0) read the *counter status*. The status information is read after the *counter status* is latched by a **Multiple Latch** command.

When both the *status* and *count data* for one counter are latched, the first read obtains the status, and the following read obtains the *count data*.

9.2.1. TMD (Timer Mode Register)

The TMD write register selects the operating mode, the **Count Latch** command, and the **Multiple Latch** command for each timer/counter. To initialize a timer/counter, write the mode word to TMD and set the mode for each counter.

Figure 9-1 shows the bits for the TMD register.

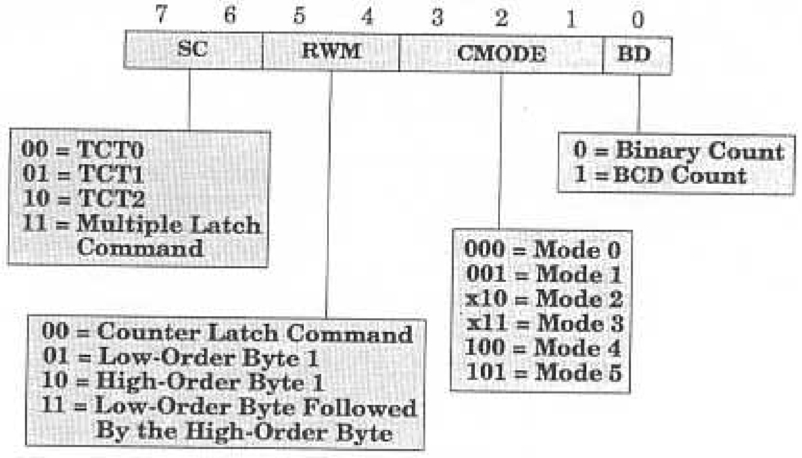


Figure 9-1. TMD Register

Note: *Under CMODE, the x is arbitrary.*

The following list describes the TMD bits:

- **SC** (bits 6 and 7)

  The SC bits specify the mode setting objective timer/counter (TCT2-TCT0) or **Multiple Latch** command. When a timer/counter is specified, setting the RWM, CMODE, and BD bits is only valid for the specified timer/counter.When the **Multiple Latch** command is specified, the meaning of bits 0-5 is different. See Figure 9-5 for a description of these bits.
- **RWM** (bits 4 and 5)

  The RWM bits specify the read/write mode which writes to the timer/counter register and reads the count latch or specifies the **Count Latch** command. For a description of the **Count Latch** command, refer to Figure 9-3.
- **CMODE** (bits 1-3)

  The CMODE bits specify the count mode (0 to 5). The default setting is mode 2 (rate generator).
- **BD** (bit 0)

  The BD specifies binary count or BCD count. When binary count is specified, binary counting is performed and the number of counts can be set from 0000h to FFFFh. When BCD count is specified, decimal counting is performed and the number of counts can be set from 0 to 9999.

9.2.2. TCKS (Timer Clock Selection Register)

TCKS selects the clock source and the clock prescaler divisor for the timer/counters. The clock supplied to the three counters (TCT2-TCT0) in the TCU is selected from one of the following:

- Input from the external TCLK pin (4.9152 MHz)
- Created by the internal clock (24 MHz)

The TCKS address is fixed at FFF0h in the system I/O area, which is different from other TCU internal registers.

Figure 9-2 shows the TCKS register.

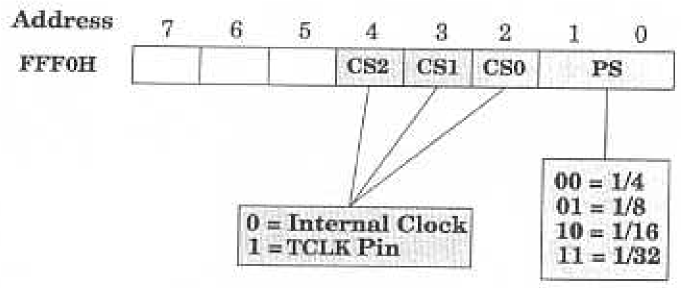


Figure 9-2. TCKS Register

The following list describes the TCKS bits:

- **CS2** (bit 4)
  This bit sets the clock input to TCT2.
- **CS1** (bit 3)
  This bit sets the clock input to TCT1.
- **CS0** (bit 2)
  This bit sets the clock input to TCT0.
- **PS** (bits 0 and 1)
  These bits set the oscillation frequency division ratio.

9.2.3. TCT (Timer/Counter Registers)

There is one 16-bit timer/counter register for each channel. These registers are written and read in accordance with the read/write mode set by the mode word.

When the low-order 1 byte and the high-order 1 byte are set, one low-order byte and one high-order byte are each written by one write operation. In this case, the remaining high-order and low-order bytes become 00h.

The low-order byte is written by the first write, and the high-order byte is written to the same address by the second write.

The low-order and high-order bytes are listed in Table 9-2.

Table 9-2. TCT Registers

Read/ Write Mode	Number of Writes	High-Order Byte	Low-Order Byte
Low-order	1 byte	00H	xxH
High-order	1 byte	xxH	00H
Low-order, high-order	2 bytes	xxH (2nd time)	xxH (1st time)

Reading from a timer/counter is basically the same as writing a timer/counter. In the low-order, high-order (2 bytes mode), the low-order byte is read by the first read. The high-order byte is read from the same address by the second read. (See Table 9-1 for the TCT addresses.)

Timer/counter read procedures include the following:

- Direct read from a timer/counter
- Read after a **Count Latch** command
- Read after a **Multiple Latch** command

Since direct read from a timer/counter reads the count latch in the down counter tracking state, its value can change during the read operation.

Reading should be performed after a **Count Latch** or **Multiple Latch** command has been issued. (A **Multiple Latch** command latches the status as well as the count.)

9.3. Count Latch Command

The **Count Latch** command is issued by writing 0 (zero) to Bits 0-5 in the TMD register. Bits 6 and 7 (SC) choose the timer/counter to be latched (refer to Figure 9-3).

The latched count data is held until it is read or a new mode is set. The **Count Latch** command reads the accurate count data at the time it is issued, without affecting the counting operation.

If a timer/counter has an issued **Count Latch** command that has not been read, any new **Count Latch** commands for that same timer/counter are ignored. When the latched count data is read, the latch is cleared and returned to its original down counter tracking state.

Figure 9-3 shows the format for the **Count Latch** command.

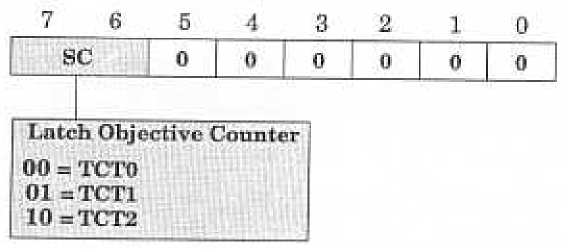


Figure 9-3. Count Latch Command Format

9.4. Multiple Latch Command

The **Multiple Latch** command is issued by writing 11 to bits 6 and 7 in the TMD register. When a **Multiple Latch** command is issued, the count data and status of the selected counters are latched. When a timer/counter in the latched state is read, the counter operation is not affected.

Figure 9-4 displays the format for the **Multiple Latch** command.

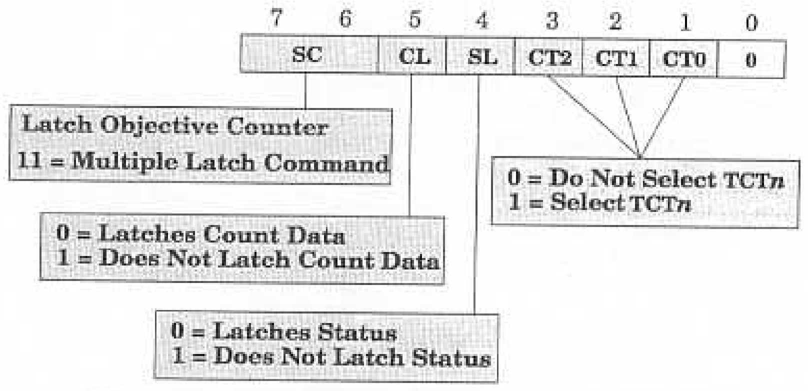


Figure 9-4. Multiple Latch Command Format

The following is a list of the timer/counters that CT2 to CT0 specify:

- CT2 specifies TCT2
- CT1 specifies TCT1
- CT0 specifies TCT0

The count data and status of multiple timer/counters can be latched simultaneously. The status shows the operating status of the timer/counter when the **Multiple Latch** command is issued.

Figure 9-5 shows the format for the timer status register (TST) latch status format.

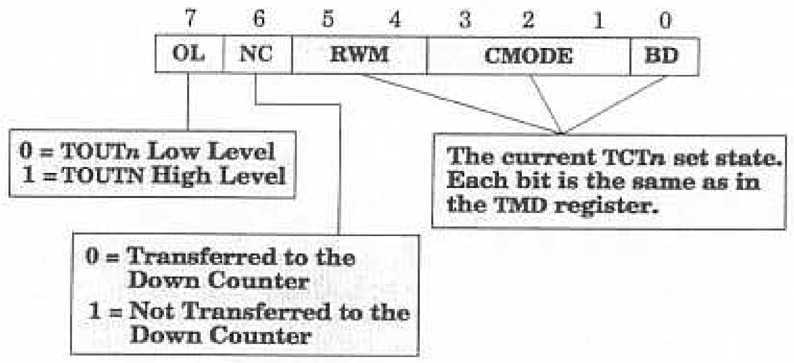

7 6 5 4 3 2 1 0
OL
NC
RWM
CMODE
BD
0 = TOUTn Low Level
1 = TOUTN High Level
The current TCTn set state. Each bit is the same as in the TMD register.
0 = Transferred to the Down Counter
1 = Not Transferred to the Down Counter

Figure 9-5. TST Format

The OL bit shows the output state of the timer/counter when the **Multiple Latch** command was issued.

The NC bit is the invalid count flag. It shows whether or not the newest written number of counts has been transferred to the down counter.

Table 9-3 and Figure 9-6 describe when the NC flag is changed.

Table 9-3. NC Flag Change

Operation (to) Counter	NC Flag
Writing of the mode word (to corresponding counter)	1
* Writing of the number of counts to the count register	1
Transfer of the number of counts from the count register to the down counter	0

* *In the read / write 2 bytes mode, the flag becomes 1 when the second byte is written.*

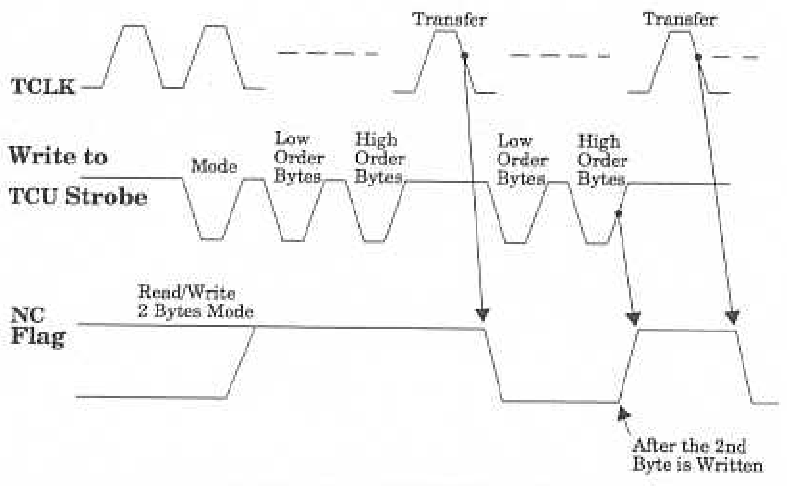


Figure 9-6. NC Flag Change

9.4.1. State of Multiple Latch Commands

After being latched, the count and status latches ignore other **Multiple Latch** commands until they are read, or until a new mode is set. Once the count or status latch is read, it is cleared.

The state of **Multiple Latch** commands is shown in Table 9-4.

Table 9-4. State of Multiple Latch Commands

	TCT0		TCT1		TCT2	
	Count	Status	Count	Status	Count	Status
1. Start	C	C	C	C	C	C
2. TCT0, Count Latch	L	C	C	C	C	C
3. TCT1, Status Latch	L	C	C	L	C	C
4. TCT0 & TCT2 Count & Status Latch	L*	L	C	L	L	L

C = Latch Clear State
L = Latched State
* = Command Ignored

Table 9-4. State of Multiple Latch Commands (Continued)

	TCT0		TCT1		TCT2	
	Count	Status	Count	Status	Count	Status
5. TCT0, Count & Status Read	C	C	C	L	L	L
6. TCT0 & TCT1 Count & Status Latch	L	L	L	L*	L	L
7. TCT0-TCT2 Count & Status Latch	L*	L*	L*	L*	L*	L*

C = Latch Clear State
L = Latched State
* = Command Ignored

A **Multiple Latch** command always reads the status at the first read operation, regardless of whether the count data or status was latched first. The count data is read at the next 1 or 2 reads (differs with the read/write mode). When reading is continued, the tracking state count value of the unlatched down counters is read.

9.5. Using Timers

Three general-purpose timers are available. The following steps explain how to use the timers:

1. Clear/disable each timer first by writing the appropriate value out to the timer control word at the 9077h I/O address.

   Table 9-5 lists the timer control word values.

   Table 9-5. Timer Control Word Values

Timer	Control Word Value
0	34h
1	74h
2	B4h

2. Set up the desired frequency by writing to the appropriate timer count register.

   To do this, write the least significant byte (LSB) of the frequency, followed by the most significant byte (MSB) of the frequency.

   Table 9-6 lists the timer count register addresses.

Table 9-6. Timer Count Register Addresses

Timer	Count Register Address
0	9074h
1	9075h
2	9076h

The following formula calculates the timer count register value (in decimal) for a certain frequency. It is assumed that the firmware set up the TCKS register for an internal clock and a divisor of 32.

$$\textbf{Count Value} = \frac{750{,}000}{\text{Desired Frequency}}$$

Convert this value to hexadecimal before using it in code.

9.5.1. Timer Frequencies

Table 9-7 lists the possible timer frequencies.

Table 9-7. Timer Frequencies

Frequency Times Per Second	Count Register Hexadecimal Value
11.4	FFFF
20	927C
30	61A8
40	493E
50	3A98
60	30D4
70	29DA
80	249F
90	208D
100	1D4C
110	1AA2
120	186A
130	1689
140	14ED
150	1388

Table 9-7. Timer Frequencies (Continued)

Frequency Times Per Second	Count Register Hexadecimal Value
...	...
200	0EA6

Example 9-1 sets timer 1 to a frequency of 20 times per second.

```
mov     dx,9077h        ; dx = timer control word
mov     al,74h          ; to clear timer 1
out     dx,al           ; clear timer 1

mov     dx,9075h        ; dx = timer 1 count register
mov     al,7ch          ; 20 times per second - LSB
out     dx,al           ; write LSB out
mov     al,92h          ; 20 times per second - MSB
out     dx,al           ; write MSB out
```

Example 9-1. Setting Timer 1

Example 9-2 sets timer 2 to a frequency of 120 times per second.

```
mov     dx,9077h        ; dx = timer control word
mov     al,0b4h         ; to clear timer 2
out     dx,al           ; clear timer 2

mov     dx,9076h        ; dx = timer 2 count register
mov     al,6ah          ; 120 times per second - LSB
out     dx,al           ; write LSB out
mov     al,18h          ; 120 times per second - MSB
out     dx,al           ; write MSB out
```

Example 9-2. Setting Timer 2

9.5.2. Disabling Timers

The three general-purpose timers are disabled by writing the appropriate value to the timer control word at the 9077h I/O address. See Table 9-5 for the timer control word values.

Example 9-3 clears Timer 1 and Timer 2.

```
mov     dx,9077h        ; dx = timer control word
mov     al,74h          ; to disable (clear) timer 1
out     dx,al           ; disable timer 1

mov     dx,9077h        ; dx = timer control word
mov     al,0b4h         ; to disable (clear) timer 2
out     dx,al           ; disable timer 2
```

Example 9-3. Clearing Timers 1 and 2

Section 10. SCC Port Communications

10.1. Command and Data Register I/O Addresses

Each Serial Communications Controller (SCC) has two ports. Each port has preassigned command and data register I/O addresses, which are accessible from the controller side only.

The command register sets up the communication parameters (baud rate, parity, data bits, stop bits, flow control, and so forth.). The data register transmits and receives data.

Note: *Refer to the 8530 technical manual for specifics on setting up the SCC port.*

Table 10-1 lists the command and data register I/O addresses.

Table 10-1. SCC I/O Addresses

Controller Port	SCC Port	Command Register	Data Register
1	0	E1F4h	E1F6h
2	1	E1F0h	E1F2h

10.2. Writing a Value to Port 1

The following example writes a value (3) to Port 1's command register:

```
outp (0xE1F4, 4);      /* Setup register 4 index on port 1 */
outp (0xE1F4, 3);      /* Write value (3) to register 4 */
```

The following example writes a value (31h) to Port 1's data register:

```
outp (0xE1F6, 0x31);  /* Write value (31h) to data register on port 1 */
```

Section 11. Downloading and Executing a Control Program

11.1. Overview

The concepts in this section are demonstrated in the sample program, **DPLOADER.C** on the *Developer's Toolkit* diskette.

To download and execute a control program, use the following steps:

1. Write the control program's executable code to dual-ported RAM, starting at the controller memory address C00h.

   Use the **copy** firmware utility command (**02**h) to place the control program at any controller memory address below 1MB.

   Note: *You can use the* ***DPLOADER*** *program, which is located on the toolkit diskette, to download a control program. To execute* ***DPLOADER.EXE****, enter* ***dploader*** *at the prompt.* ***DPLOADER*** *prompts you for the information it needs to download the control program.*

   DPLOADER*, which is a program for the MS-DOS operating system, sets the I/O address at 218h and the memory address at D0000h.*

2. Compliment the two-byte *interaction flag* at the controller memory address B80h from 55AAh to AA55h.
3. Invoke the execute control program firmware utility (**01**h).

   The control program should immediately perform the following tasks:

   - Disable interrupts
   - Allocate a local stack
   - Initialize the interrupt vectors for the system's interrupts, timers, and SCCs

Following this initialization, enable interrupts and continue with normal operation.

Comtrol recommends that the control program sets up its interrupt vectors and then compliments the bytes of the *interaction flag* back to 55AAh. This notifies the system that the control program is functioning properly.

11.2. Using Firmware Utilities

Using firmware utilities allows you to place and invoke a control program anywhere in the controller's memory below 1MB.

The following is a list of the firmware utility buffer fields, taken from the firmware user area map in Table 5-2:

- **2Eh Offset**
  The *firmware utility command* executes upon interrupt. The utility commands are listed in Table 11-1.

Table 11-1. Utility Commands

Command	Action
00h	A null command that is set for status and return.
01h (default)	Executes a control program at a vector in the message buffer: 30h = segment 32h = offset A control program sets the command status to *finished* (status=01).
02h	A copy command that uses the following message buffer parameters: 30h = source segment 32h = source offset 34h = destination segment 36h = destination offset 38h = count (two bytes) When complete, it sets the command status to *finished* (status=01)
03h through FFh	Reserved, currently uses a null command.

- **2Fh Offset**
  The *firmware utility status* holds the status of the firmware utility command. The values include:
  - **00h** default (command processing)
  - **01h** (command processing finished)
  - **02h** through **FFh** (reserved)

- **30h Offset**
  The *firmware utility message buffer* is a message buffer for commands. This buffer is initialized before the controller is interrupted. The controller can also return information in this buffer.

11.2.1. Using the Copy Command (02h)

The following steps explain how to use the copy command (**02h**) to download the control program at the execution address in the controller's memory:

1. Write the controller memory segment and offset of the source buffer at offsets 30h and 32h in the firmware user area.
2. Write **02h** (copy) in the firmware user area.
3. Write the first (or next) block of the control program into the dual-ported memory source buffer.
4. Write the number of bytes in the source buffer at offset 38h in the firmware user area.
5. Write the controller memory first (or next) destination segment and offset at offsets 34h and 36h in the firmware user area.
6. Write **00h** at 2Fh (status field) in the firmware user area.
7. Interrupt the controller.
8. Wait for the status field to change to 01h, indicating that the copy is complete.
9. Repeat steps 3 through 8 until the entire control program is downloaded.

11.2.2. Using the Execute Command (01h)

The following steps explain how to use the execute command (**01h**) to start executing a previously downloaded control program:

1. Write AA55h at offset 0 (interaction flag) in the firmware user area.
2. Write **01h** (execute) at offset 2Eh (command field) in the firmware user area.
3. Write the control program entry point segment and offset at offset 30h and 32h in the firmware user area.
4. Interrupt the controller.
5. Wait for the control program to change the interaction flag to 55AAh.

11.3. Using the DPLOADER Program

You can use the **DPLOADER** DOS program found on the *Developer's Toolkit* diskette, to download a control program. This program uses the following format:

DPLOADER [***control-program-name***] [***Turbo Debugger (Y/N)***]

Where:

control-program-name is an optional name of the control program file to download. If not given this defaults to **CPC.BIN**.

Turbo Debugger (Y/N) is an optional Yes or No switch indicating whether or not to invoke the Turbo Debugger program remotely to allow debugging of the control program.

The **DPLOADER** program uses the 218h I/O address and D000:0 memory address for the controller.

Section 12. Debugging Tools

12.1. Debugging Tools Overview

This section contains information on a variety of debugging tools available to you. These tools include the following:

- The Borland Turbo Debugger
- Firmware debugger

12.2. Turbo Debugger Overview

The Borland Turbo Debugger is a source-level debugger that provides a windowing user interface. Comtrol supports the use of this debugger, which allows you to execute and debug programs that operate on the controller.

There are two possible debugging environments:

- A single PC with the controller and Turbo Debugger.
- Two PCs, one with the Turbo Debugger and the other with the controller. The following subsections describe this method.

You can debug only code that resides in RAM. The Borland Turbo Debugger cannot make firmware debugging calls (for example **int 21h**, the firmware RAM query) to the controller.

Note: *The firmware on the controller only supports the Borland C++ 4.02 version of the Turbo Debugger.*

The controller is not backwards compatible with earlier versions of the Turbo Debugger.

12.2.1. Setting Up the Hardware Environment

The following subsections discuss a debugging environment that consists of a

- Development PC that displays the Borland Turbo Debugger.
- Remote PC that executes the controller's control programs under the Borland Turbo Debugger environment. The controller is installed in this system.

See the documentation that came with the Turbo Debugger for detailed information about system requirement to run the Turbo Debugger.

To use this subsection, you should have ordered the *Development*

Board Option on your controller. This option is provided at no additional charge and includes the following pieces:

- A debug/reset header soldered to the controller
- A debug/reset box and cable

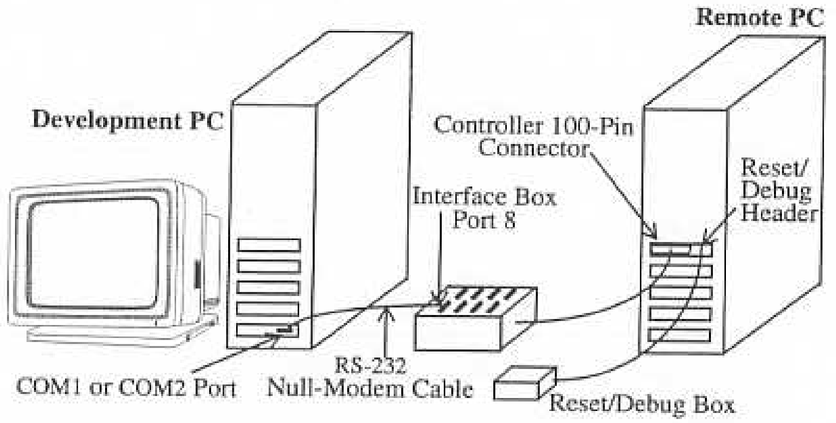
Note: *If you have any questions regarding the Toolkit or the Development Board Option, contact Comtrol using the information provided in Appendix A.*

Use the next subsection to connect the debugging environment.

12.2.2. Connecting a Two-PC Environment

To physically connect the debugging environment, perform the following steps (see Figure 12-1):

1. Connect the development PC to the remote PC with an RS-232 null-modem cable.
   a. Attach one end of the cable to port 8 of the controller's interface box.
   b. Attach the other end of this cable to the COM1 or COM2 port of the development PC.
2. Connect the cable attached to the reset/debug box to the reset/ debug header on the controller.



Note: *This assumes that the interface box has already been attached to the controller (see the* Interface Reference Card).

Figure 12-1. Cabling between the Development System and the Remote System

12.2.3. Configuring Symbol Tables

To use the Borland Turbo Debugger, you must generate a symbol table to accompany your program on the development PC.

Use the following steps to create a symbol table:

1. Use the compiler's command options to compile and link your program (refer to your compiler documentation for specifics).
2. Run the resulting .EXE file through the Turbo Debugger symbol table separating utility, which is called TDSTRIP.EXE.

   This utility removes the symbol table from the executable file and places it in a separate file. The symbol table file has a .TDS extension.
3. Run the stripped CPC.EXE file through the Comtrol locate utility called CLOCATE.EXE. This converts the load module for the MS-DOS operating system into an executable download file called CPC.BIN.

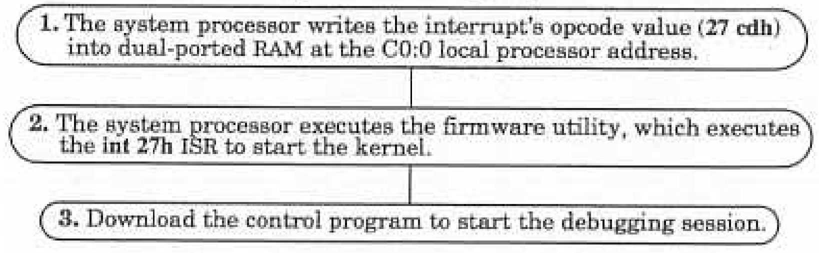
The sample MAKEFILE.BC make file on the *Developer's Toolkit* diskette gives an example of how this is done.

12.2.4. Invoking the Remote Kernel

The controller's firmware contains a Turbo Debugger remote kernel that must be invoked before the development PC can establish communications with the controller.

Invoke the kernel by executing an **int 27h** interrupt to the controller's processor. This interrupt can be executed in one of two ways:

- **Before downloading the control program**



1. The system processor writes the interrupt's opcode value (**27 cdh**) into dual-ported RAM at the C0:0 local processor address.

2. The system processor executes the firmware utility, which executes the **int 27h** ISR to start the kernel.

3. Download the control program to start the debugging session.

Flowchart 12-1. Executing int 27h Before Downloading the Control Program

- **Within the downloaded control program**

  To embed the **int 27h** interrupt within the downloaded control program, enter the starting address of the instruction immediately following the **int 27h** interrupt (do this in the **<Ctrl> G** step of the Turbo Debugger start-up sequence).

Example 12-1 shows a sample in assembly language.

```
        int 27h             ; invoke Turbo Debugger remote kernel
    start_debug:            ; symbolic address for
                            ; Turbo Debugger <CTRL G>
```

Example 12-1. Embeddingint 27h

Note: *When debugging code remotely, do not use or initialize line 7 (Port 8) of the SCC channel because it is used by the Turbo Debugger.*

12.2.5. Running the Turbo Debugger

The *Developer's Toolkit* diskette contains sample C programs. Before you can run the Turbo Debugger

1. Create directories on both the remote and the development PCs.
2. Copy the following programs to the appropriate systems:
   - Remote System
     - DPLOADER, CPC.BIN, and HITERM.EXE (executable files)
   - Development System
     - CPC.C, CPC.H, and CPCSTART.ASM (source files)
     - CPC.TDS (symbol table)

Note: *If you have not done so already, install the Borland Turbo Debugger Version 4.02 on the development system.*

Use the following steps to invoke the debugger on both systems.

Note: *Because this procedure can be tedious, steps have been included to invoke the Turbo Debugger* ***macro recording*** *function. This saves time and keystrokes when performing subsequent debugging sessions.*

1. Invoke DPLOADER on the remote system, by entering the following at the prompt:

   dploader

2. Authorize DPLOADER to reset the controller, identify the control program to download, and to invoke the Turbo Debugger remote kernel.
3. Invoke the Borland Turbo Debugger on the development system, by entering the following at the prompt:

   td -rp# -rs3

   where:

   -rp1 specifies the COM1 port and **-rp2** specifies the COM2 port.

   -rs3 specifies 38.4K baud for the Turbo Debugger, Version 4.02.

An opening window appears first, followed by the CPU window.

Note: *If your control program becomes large, the debugger may need additional memory to hold the symbol table. In that case, add the* **-smxx** *option to the* **td** *command, where* **xx** *is the number of Kilobytes to be used for the symbol table.*

See the CPC.EXE and CPCSTART sample files for examples of the following steps. Your macros will differ, depending on the files to debug.

4. Press **<Alt> O** and choose **Macros** and then **Create**.

   The Turbo Debugger prompts you for the macro keystroke sequence.

5. Set the stack sequence to 98h:

   a. Press **<Alt> V** to view.
   b. Press **R** or press the **<Arrow key>** to **Registers**.
   c. Press the **<Arrow key>** to choose the **Stack Segment** by highlighting **SS** and press **<Alt> F10**.
   d. Press **C** or press the **<Arrow key>** to Change
   e. Enter the value (in this case, 98h), press **<Enter>**, **<Alt>W** for **Window**, and then **Close**.

Note: *The Stack Segment (SS) must be set to 98h before executing the startup program. Failure to do so might result in your code getting stepped on by the stack.*

6. Press **<Alt> F** and choose **Symbol load**.
7. Choose the file to load, and enter the symbol table's file name. The example file uses the **cpc.tds** symbol table file.
8. Press **<Alt> F** and choose **Table relocate**.
9. Enter the new segment value.

   This specifies the segment to execute on the controller's processor. This value should always be 0C0.

10. Press **<Ctrl> G** and enter the address of the entry point, in this example. Type **start**.
11. Press **<Ctrl> N**.

    This updates the registers for the CS:IP (current segment:instruction pointer).

12. Enter the first instruction displayed to assemble, in this example. Type **cli**.

    To bring up the *Enter instruction to assemble* window, you must type in the first character of the instruction (**c** in the example). If you do not enter the first instruction displayed, the debugger does not acknowledge that your program is loaded.

At this point, you can customize your environment, but remember that the macro facility records all of your actions. To stop recording, open the **Options** menu and choose **Macros** and **Stop recording**. To save the macro, open the **Options** menu and choose **Save options**.

For future debugging sessions, use this macro to automatically replay steps 6 through 11.

To invoke the macro, type the specified macro keystrokes when the debugger starts.

Note: *If the Borland Turbo Debugger is running and a breakpoint is never reached, press the Reset switch to stop execution.*

12.2.6. Single-Stepping Instructions

Single-stepping through hardware interrupt instructions with the <F7> and <F8> function keys may not generate interrupts reliably.

For example, **outp(0xe1f6,0x31)** writes a character out to Line 0. This normally results in a Transmit Buffer Empty (TBE) interrupt. However, if this instruction is single-stepped, the interrupt may not occur. To avoid this, set a breakpoint after **outp()** and run to it, rather than single stepping over it.

12.3. Firmware Debugger Background

This section explains the commands to use with the firmware debugger. The debugger is part of the firmware installed on the controller, and it provides the following functions:

- Displays/changes memory
- Disassembles instructions
- Performs input and output to I/O ports
- Displays registers
- Single steps
- Sets breakpoints

The debugger console is initially assigned to the first serial port on the controller. The serial communications parameters are defined as follows:

- 9,600 bits/second
- Eight (8) bits per character
- No parity
- One (1) stop bit

Because these parameters are fixed, a program cannot alter them.

Note: *This assumes that the interface box has already been attached to the controller in the development PC (see the* Interface Reference Card*).*

12.3.1. Invoking the Firmware Debugger

The firmware debugger essentially operates as an interrupt service routine (ISR). The controller firmware provides access to debugger functions through the following software interrupts:

- **int 20h**
  A program can invoke the debugger through this interrupt. The firmware configures Port 1 as the debug console during system initialization.
- **int 22h**
  After the system initializes, a program can change the debug console by executing this interrupt. Load a valid device number from 0 to F (Ports 1 through 16) into the AL register before executing the software interrupt.

Note: *To access the firmware debugger, make sure that you have your system connected as shown in Subsections 12.5.1 and 12.5.2.*

To invoke the firmware debugger, press the Debug switch on the box.

12.3.2. Firmware Debugger Commands

Table 12-2 lists the commands the firmware debugger supports.

Table 12-1. Debugger Commands

Command	Name	Function
B	Byte	Formats all succeeding input or output commands to read or write 8 bit values.
D	Dump	Displays the contents of a specified memory region.
E	Edit	Changes the contents of a specified memory byte.
F	Fill	Changes the contents of a specified memory region.
G	Go	Continues execution from the current location with or without breakpoints.
I	Input	Inputs and displays a byte or word from the specified I/O port.
O	Output	Outputs a byte or word to the specified I/O port.
R	Register	Displays the contents of all registers.
T	Trace	Executes the next instruction (single step).
U	Unassemble	Disassembles a specified memory region.
W	Word	Formats all succeeding input or output commands to read or write 16 bit values.

12.3.3. Firmware Debugger Command Definitions

This subsection describes how to use each of the debugger commands listed in Tables 12-2 and 12-3. The following is a list of guidelines that apply to the debugger commands:

- Each command consists of a single letter, followed by one or more parameters.
- Optional parameters are displayed inside parenthesis ().
- Enter commands and parameters in uppercase, lowercase, or a combination of both.
- Commands executed after pressing **<Enter>**.
- The debugger prompt is a hyphen (-).
- The location of syntax errors is indicated by the pointer error.

Table 12-2. Debugger Command Definitions

Format	Where Clause
B	Arguments are not required.
D (*starting address*) (*ending address*) or D (*starting address*) (*l length*)	***starting address*** specifies the first address of a range of addresses to display and takes any of the following forms: • A segment value, offset value pair separated by a colon (:) • A segment register mnemonic and an offset value separated by a colon (:) • An offset value only (a default segment is used) ***ending address*** is an offset value within the segment specified by the ***starting* address**, which specifies the last address of a range of addresses to display. ***l length*** specifies the number of bytes to display. **Note:** *If* **ending address** *or* **l length** *is not specified, the default display length is 128 bytes.* If arguments are not specified and a **D** command has not been entered, display starts at the current CS:IP location. If a **D** command has been entered, display starts with the byte following the last byte displayed. The default length is 128 bytes. The following commands display the contents of memory from 0040h:000h through 0040h:00ffh: **Continued**

Table 12-2. Debugger Command Definitions (Continued)

Format	Where Clause
D (*starting address*) (*ending address*) or D (*starting address*) (*l length*) (cont.)	**D 40:0** FF or **D 40:0** L **100**The following command displays the contents of memory from the 1000h offset. This is within the segment that the ES register currently points to. The ES register uses the 107Fh offset, which is the default length. **D ES:1000**
E (*starting address*) (*byte value*)	***starting address*** specifies the first address of a range of addresses to change and takes any of the following forms: • A segment value, offset value pair separated by a colon (:) • A segment register mnemonic and an offset value separated by a colon (:) • An offset value only (a default segment is used) ***byte value*** specifies the value to change. If no ***byte value*** is issued, you enter edit mode. The following commands change the contents of memory 0040h:000h: **E40:0 55** **Note:** *In edit mode, to continue editing the next byte, press the space bar. To exit edit mode, press the return key.*
F (*starting address*) (*ending address*) (*byte value*) or F(*starting address*) (*l length*) (*byte value*)	***starting address*** specifies the first address of a range of addresses to change and takes any of the following forms: • A segment value, offset value pair separated by a colon (:) • A segment register mnemonic and an offset value separated by a colon (:) • An offset value only (a default segment is used) ***ending address*** is an offset value within the segment specified by the ***starting* address**, which specifies the last address of a range of addresses to change. ***l length*** specifies the number of bytes to change. ***byte value*** specifies the value to change. **Continued**

Table 12-2. Debugger Command Definitions (Continued)

Format	Where Clause
F *(starting address) (ending address) (byte value)* or F*(starting address) (l length) (byte value)* (cont.)	The following commands change the contents of memory from 0040h:000h to 0040h:FFh: **F40:0 FFh 55** or **F40:0 L 100 55**
G *(breakpoint address)*	***breakpoint address*** specifies an address where program execution is interrupted, and control is returned to the debugger. If a ***breakpoint address*** is not specified, the program continues normal execution. The following command allows program execution to continue from the current location (CS:IP), and sets a breakpoint at 4000h:0007h. If the program attempts to execute the instruction at this address, execution is interrupted, and control is returned to the debugger. **G 4000:7**
I *portaddress*	*portaddress* specifies a 16 bit I/O address for input data. The size of the input data (byte or word) depends on the current I/O mode (see B and W commands). The following command inputs data from the I/O port at 202h: **I 202**
O *portaddress value*	*portaddress* specifies a 16 bit I/O address for output data. The size of the output data specified by *value* (byte or word) depends on the current I/O mode (see the B and W commands). The following command outputs 55h to the I/O port at 200h: **O 200 55**
R	Arguments are not required.
T	Arguments are not required.
U *(starting address) (count)*	*starting address* specifies the first address of a range of addresses to disassemble and takes any of the following forms: • A segment value, offset value pair separated by a colon (:)

Continued

Table 12-2. Debugger Command Definitions (Continued)

Format	Where Clause
U *(starting address) (count)* (Cont.)	• A segment register mnemonic and an offset value separated by a colon (:) • An offset value only (a default segment is used) ***count*** specifies the number of instructions to disassemble. If ***count*** is not specified, a default of 16 instructions are disassembled. If arguments are not specified and a **U** command has not been entered, disassembly begins at the current CS:IP location. If a U command has been entered, disassembly begins with the instruction following the last instruction previously displayed. The following command disassembles eight instructions starting at 4000h:0003h: **U 4000:3 8** The following command disassembles 16 instructions (default count). Disassembly begins at the 0003h offset, within the segment that the CS register points to: **U CS:3**
W	Arguments are not required.

12.4. Using the Firmware Debugger

The following is a list of things to remember when using the firmware debugger:

- Jump instructions display the next instruction's address as a relative address, not as an absolute address.
- Non-8086 instructions do not disassemble correctly, but they do execute correctly.

  The instructions appear as follows:

  *** data ***
- Timer, system, and SCC interrupts continue to occur when using the firmware debugger.

  These ISRs cannot make any assumptions about the state of any registers. This includes the segment registers, which the firmware debugger modifies for its own use. Because of this, the ISRs must save and initialize the registers. Then, the registers must be restored before exiting the ISRs.

Appendix A. Developer's License Agreement and Contacting Comtrol

A.1. Developer's License Agreement

At Comtrol, we want to encourage you to develop software products for our hardware products, so we developed this no-nonsense Developer's License Agreement.

The software supplied with the Developer's Toolkit is protected by United States copyright law and international copyright treaties. In order for Comtrol to protect its copyrights, we have some limitations on reproduction and distribution:

1. The software may only be used to develop software products that will operate with Comtrol brand hardware and software.
2. You may not reproduce nor distribute the source code contained in the Developer's Toolkit.
3. Any reproduced or modified Developers's Toolkit software distributed in executable object code form must bear either Comtrol's copyright notice (for example, Copyright 1991, 1992, 1993, 1994, 1995 Comtrol Corporation), or your own copyright notice.

Other than these restrictions, programs that you write using the materials in the Developer's Toolkit may be used, distributed, modified, or licensed by you as you decide.

Sample programs are provided to help you start programming right away. You may edit, modify, or otherwise incorporate these programs and routines; and you may redistribute and license them for use by your customers without any other license fee or restriction.

Of course, you are solely responsible for your own programming and you agree to hold us harmless from all claims, liability, and damage arising from your own products which include any Comtrol Software. Remember that this software is designed for use only with Comtrol Products. It will not function properly with any other brand of controller.

A.2. Contacting Comtrol

If you need assistance or have questions about any of our products, contact Comtrol using one of the following methods. Comtrol has a staff of hardware and software engineers, and technicians available to help you.

Corporate Headquarters:

email: support@Comtrol.com

FAX: (612) 631-8117

BBS (for device driver updates): (612) 631-8310

Note: *The BBS supports modem speeds up to 28.8 Kbps (V.FC) with 8 bits and no parity.*

Toll free: (800) 926-6876

Phone: (612) 631-7654

Comtrol Europe:

BBS: +44 (0) 1* 869-243-687

Note: *The BBS supports modem speeds up to 14.4 Kbps with 8 bits and no parity.*

FAX: +44 (0) 1* 869-323-211

Phone: +44 (0) 1* 869-323-220

**Dependent upon the telephone carrier until April 16, 1995.*

Index