Contents

1

Retargeting the Monitor 1

This chapter describes configuration procedures to build, or retarget, the Mon960 debug monitor. To retarget the monitor, you modify the source files provided and create a monitor that is specific to your target board. This chapter describes the following steps to retarget the monitor:

- modifying the source files
- recompiling and linking the monitor
- producing EPROMs containing the monitor program

If you are using the EV80960CA, EV80960SX, or QT960 board as your target, the source and hexadecimal files that Mon960 runs on are included with the monitor. You need only produce new EPROMs with the hexadecimal files and install those EPROMs on your target board. Section 1.2.2 has information about producing new EPROMs for your target board. After that, you can read Chapter 2 to learn about how to use the monitor.

If you are not using the EV80960CA, EV80960SX, or QT960 board as your target, read this chapter and retarget the monitor. Then you can connect your target board to a terminal or to a host running a terminal program and begin using the monitor.

NOTE

Although you could run the DB960 or GDB960 software debugger after you have finished retargeting, do not do so until you have used Mon960 sufficiently to be sure that your retargeting was successful. The debugger adds levels of complexity that can interfere with the task of verifying that the monitor is running correctly.

NOTE

The information in this chapter is presented with the assumption that you generate target code using iC-960 or gcc960.

1.1 Retargeting the Monitor

The monitor contains two classifications of files as follows. The source code for the monitor is in the mon960/common directory.

- files pertaining to all i960 targets (board-independent)
- files that depend on the target type (board-specific)

When you retarget, you only change the board-specific files. Changing only the board-specific files minimizes the number of files to change to port Mon960. In this chapter, *board* is a placeholder for the name of your board.

To configure the monitor for your board, duplicate the board-specific files that best match your hardware , renaming these files as:

board.h
*board*_dat.c
*board*_hw.c
mon*board*.ld

Then you can modify the hardware-specific files and change the makefile to reflect your new file names. For example, copy evca.h to myca.h, and evca_hw.c to myca_hw.c.

NOTE

The DOS operating system imposes an 8-character limit on the filename.

1.1.1 Code Areas Affected by Retargeting

When you retarget the monitor, the following areas of code can require modification:

- memory configuration, described in Section 1.1.2, including the following:
  - memory configuration in *board*.ld
  - bus configuration in *board*.h, (for the CA only)
  - hardware-dependent addresses and constants in *board*.h
- hardware-specific data in *board*_dat.c described in Section 1.1.4.
- hardware-dependent routines, including the following:
  - hardware-specific routines in *board*_hw.c described in Section 1.1.5
  - device-driver routines in i510.c or 16552.c described in Section 1.1.6

If your target board has two serial ports, you can use the additional port to help debug the monitor, using the routines in *board*_dbg.c. See Section 1.3.2 for information on the routines in *board*_dbg.c.

The following sections discuss the portions of the monitor code that require modification and suggests possible modifications to that code.

1.1.2 Memory Configuration

The monitor normally resides in EPROM on the target board. The monitor stores its own data and a copy of the i960 data structures in target RAM. Figures 1-1, 1-2, and 1-3 show the memory configuration for the EV80960CA board, QT960 board, and EV960SX board, respectively.

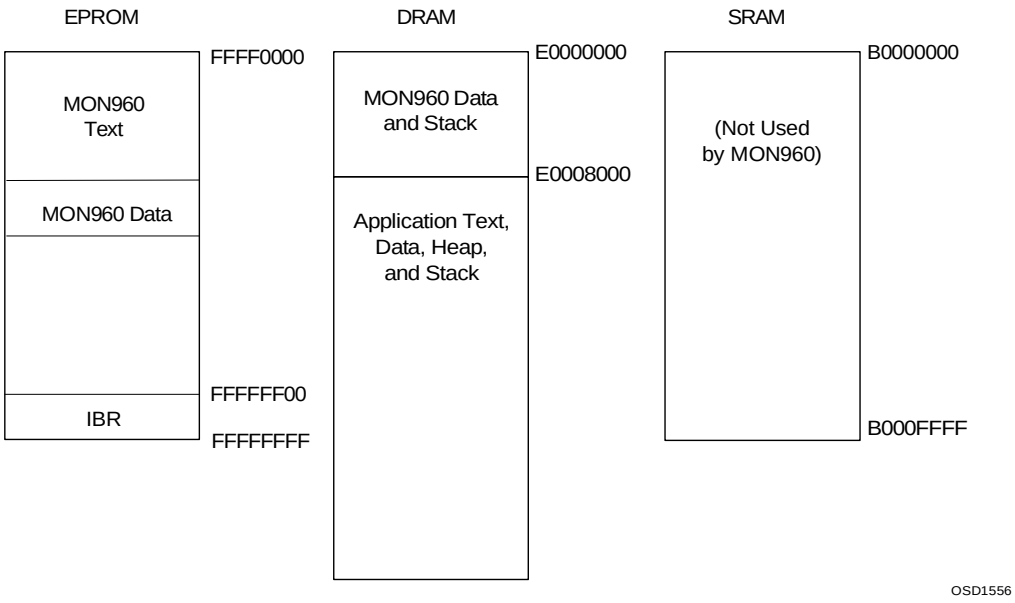

Figure 1-1 Memory Map for EV80960CA Board

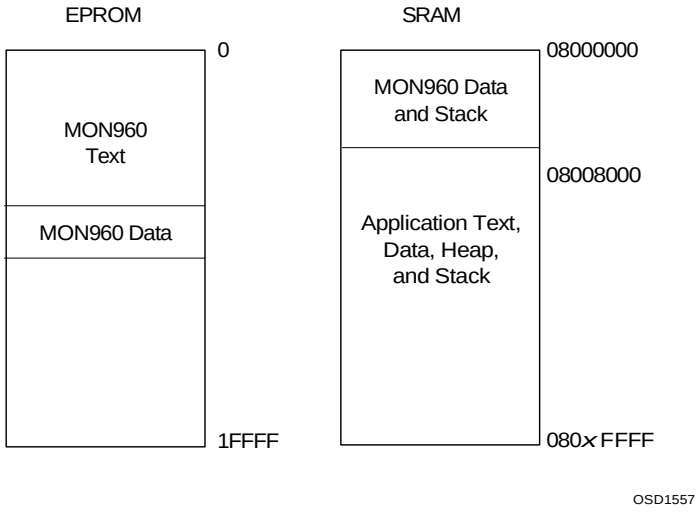

Figure 1-2 Memory Map for QT960 Board

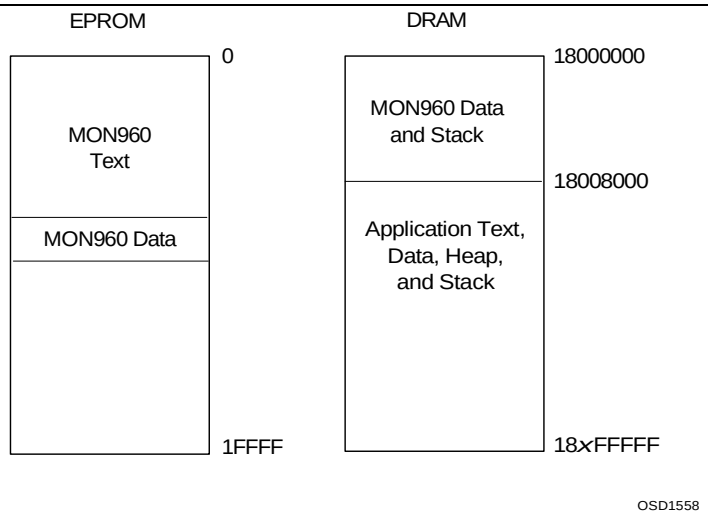

Figure 1-3 Memory Map for EV960SX Board

Define the target memory configuration by assigning values to the following memory configuration variables in the linker-directive file (for example, monevca.ld):

ibr	address of initialization boot record (CA only). The ibr address is 0xffffff00 and should never be changed.
rom	base address and size of EPROM space. The size requirement for the monitor is approximately 60 kilobytes. Be sure to allocate space in ROM after the end of the text for the initial values of the data section. You can reduce the monitor size by linking without some of the features, for example, the user interface. See section 1.2.1 for directions on using the makefile to relink the monitor.
data	base address and size of monitor initialized data space. The size requirement is about 1 kilobyte.
bss	base address and size of monitor uninitialized data space. The size requirement is about 12 kilobytes.

If you have room, reserve a portion of the beginning of RAM (for example, 32 kilobytes) to enable future expansion.

Several other symbols are defined in the linker-directive file:

pre_init	set to the address of the pre-initialization code, if it is required. See Section 1.1.3 for information on this pre-initialization code.
initial_stack	defines the stack that is used after initialization. Define it as monitor_stack.
_checksum	value placed in the IMI as part of the checksum.

You can edit the linker directive file to change the addresses for some of these variables to match your target board. If you are retargeting the CA version, the *board*.h file defines the bus configuration values for the processor. When you change the settings in the .ld file, make parallel changes to the *board*.h file as well.

The ca_ibr.c file contains the initialization boot record for the i960 CA processor. You need not change this file because the code is recompiled using the definitions in *board*.h. See the *80960CA User's Manual*, listed in the Preface, for information on the initialization boot record.

NOTE

The i960 K*x* processor has no memory configuration table or initialization boot record. Therefore, the K*x* version of the monitor does not include the ca_ibr.c file.

1.1.3 Board Initialization

At reset, the i960 processor does a checksum self-test, reads pointers from the initial memory image (IMI), then begins executing user instructions at the boot address. Boards designed for the i960 architecture normally include a mechanism to signal assertion of the i960 processor FAILURE# pin. The Intel evaluation boards implement this mechanism with an LED. This mechanism enables you to determine if the processor successfully read the IMI and reached the boot address.

The boot address is _start_ip in the init.s file. Normally, this file needs no modification. The code in init.s checks the symbol pre_init to see whether any code must be executed before monitor initialization. If pre_init is not zero, init.s calls this code via the instruction balx pre_init,g14.

The pre_init routine performs any functions that must be completed before monitor initialization. For example, it may do DRAM initialization or board-level self tests, such as memory tests or ROM checksum verification.

The pre_init routine must be written in assembly language, since it is entered via a branch-and-link instruction. The routine must preserve the global registers g14 and g11. The g14 register is used as the return address, and the g11 register contains processor revision information for processors with this functionality.

NOTE

If you want to call a C language routine, you must save the value of g14 in another register and then zero the g14 register before the call. The pre_init routine must not call any C library routines, because they are not yet initialized.

Put any initialization routines that need not be completed before monitor initialization in the init_hardware routine. The init_hardware routine is called from main and can use C and C library support.

After the pre-initialization code returns, the init.s file sets up the stacks, initializes the C run-time environment, copies the processor data structures to RAM, and re-initializes the processor for the new data structure locations.

Finally, init.s branches to _main, which calls the init_hardware routine in the *board*_hw.c module. If necessary, you can modify the init_hardware routine and other hardware-dependent routines in *board*_hw.c.

The following sections describe each of the routines that might need modification.

1.1.4 Board-specific Data

The *board*_dat.c file contains variables that specify information about your board. These variables are defined as const because their values do not change at run time. You must change the values of these variables to describe the target board.

`const int arch;`

arch specifies the processor architecture. The value of arch is one of ARCH_CA, ARCH_KA, ARCH_KB, ARCH_SA, or ARCH_SB. The value of arch is used by the monitor and host to determine what registers and other capabilities your board has.

`const char arch_name[];`

arch_name contains the name of the processor architecture ("CA", "KA", etc.). It is printed in a banner across the screen when the monitor starts up.

`const char board_name[];`

board_name contains the name of your board. It is printed in a banner across the screen when the monitor starts up.

`const char board_version[];`

board_version contains the version number of your board-specific code. It is printed in the banner when the monitor starts up. You can use this variable to track the revision of your retargeting code in use.

1.1.5 Board-specific Routines

The *board*_hw.c file contains the following initialization routines that you might need to change when retargeting the monitor:

`void init_hardware(void)`

`int get_int_vector(void)`

`void board_reset (void)`

`int clear_break_condition(void)`

void fatal_error(char *format, ...)

void blink(int val)

void board_go_user(void)

void board_exit_user(void)

These routines isolate target-hardware dependencies in the monitor. If necessary, modify these files to conform to the capabilities of your target board. Below is an explanation of each of the routines in *board*_hw.c.

init_hardware void init_hardware(void)

This routine initializes the target board. The routine writes the processor interrupt control registers with values applicable to the board. It also initializes other board resources. See the programmer's reference manual for your processor, listed in the Preface, for information on programming the interrupt control registers.

Before the monitor calls the init_hardware routine, the monitor establishes most of its own default variables. Thus, the values of these default variables can be changed, if necessary, in the init_hardware routine.

The init_hardware routine must not initialize any hardware devices that are not required for the operation of the monitor. Initialization of these other devices should be left to the application that is downloaded to the target board once the monitor is running.

get_int_vector int get_int_vector(void)

This routine returns the vector number of the interrupt that is generated when a break is received by the serial port.

When the host debugger needs to interrupt the application, it sends a break signal to the target system. You can program a universal asynchronous receiver transmitter (UART) to generate an interrupt when it receives a break. The priority of this interrupt must be as high as possible to ensure that the debugger can interrupt the application and regain control even when the processor is running at high priority. If possible, use priority 31 for the K*x* version and NMI for the CA version.

This feature is not necessary for the basic operation of the monitor. If get_int_vector returns 0, this feature is disabled. You can code this routine to return 0 until the rest of the monitor is working properly. If this routine returns 0, then do not program the UART to generate any interrupts.

The init_hardware routine configures the interrupt mechanism. The get_int_vector routine returns the interrupt vector number assigned to the UART, and the monitor fills in the appropriate interrupt vector into the interrupt table. The serial_open routine configures the UART to generate an interrupt when a break is received.

If your target board requires special processing to clear the UART or an interrupt controller, do this processing in the clear_break_condition routine. The monitor calls this routine to do any special processing required to clear an interrupt condition.

board_reset `void board_reset (void)`

This routine resets the monitor target. The routine does not return a value. If the target hardware does not support resetting the processor, this routine restarts the i960 processor.

To reset the CA processor, this routine calls the following routine:

```
send_sysctl(0x300, reinit, &rom_prcb)
```

To restart the K*x*-series processor, this routine calls the following routine:

```
send_iac(0, iac)
```

The iac argument is a four-word array containing a type 0x93 iac (reinitialize processor). See the programmer's reference manual for your processor, listed in the Preface, for information on the sysctl and iac arguments.

clear_break_condition `int clear_break_condition(void)`

This routine clears an interrupt from the serial port. The routine is called when an interrupt is received from the serial port. It does any special processing required to clear the interrupt condition in the UART or interrupt controller. It must ensure that any null characters or framing errors created by the start or end of the break are cleared in the UART. In the example code, this work is done by the serial_intr routine in the i510.c file.

If the interrupt was caused by a break received from the host, the routine returns TRUE. If the interrupt was caused by some other condition, such as overrun, the routine returns a FALSE.

NOTE

You need not program the UART to generate an interrupt on overrun or other error conditions. However, some UARTs do not allow you to disable these interrupts, so the monitor ignores them.

fatal_error **void fatal_error(char *format, ...)**

Parameter: format printf-style arguments

The monitor calls this function if an unrecoverable error occurs. This routine does not return a value.

If the monitor is not connected to a host debugger, the fatal_error routine prints the message given by format and the other arguments to the serial port. If the monitor is connected to a host debugger and the debug port is enabled, the monitor prints the information to the debug port. See Section 1.3.2 for information on the debug port.

blink **void blink(int val)**

This function blinks an LED on the target once. The monitor calls this routine at specified locations during initialization to show the current execution point.

You can use the val argument to select an LED to blink or a value to display. The code supplied for the EV80960CA board uses the seven segment LED display on that board to display val as a digit.

board_go_user **void board_go_user(void)**

The monitor calls this routine each time execution returns to the application program. You can use this routine to light an LED to indicate that the application program is running, or do other board-specific requirements to execute an application program. On the CA, if the break interrupt priority is not NMI, this routine enables the appropriate interrupt in the saved copy of the imsk register, as in the following code:

```
register_set[REG_IMSK] |= IMSK_VAL;
```

board_exit_user **void board_exit_user(void)**

This function performs any required board-specific actions when execution returns from the application to the monitor. The routine is called when execution returns to the monitor.

The remaining routines in *board*_hw.c are used internally by other routines within the module and are not required by the monitor.

leds **void leds(int value, int mask)**

Use this function with an 8-segment bar LED. The mask specifies which segments are being changed, and the value specifies the new value for the segments. A value of 1 indicates the segment is lit. For example, the call leds(4,5) extinguishes segment 0 (bit 0 of mask is 1 and bit 0 of value is 0) and lights segment 2 (bit 2 of mask and bit 2 of value are both 1). All other segments are unchanged. This function is for display purposes only and may be a stub with no effect on monitor operation. The segments currently used are:

bit 0 in serial_read

bit 1 in serial_write

bit 2 application is executing

1.1.6 Serial Device Driver Routines

The monitor code includes I/O device drivers for the Intel 82510 UART and the National Semiconductor 16552 DUART. The driver routines are in the i510.c and 16552.c files, respectively.

If your board uses the 82510 UART, you must modify the definitions of the following constants in the board-specific include file (*board*.h):

I510BASE	the base address of the 82510
I510DELTA	the spacing between the hardware registers
XTAL	the frequency of the baud rate crystal

If your board uses the 16552 DUART, you must modify the definitions of the following constants in the board-specific include file:

DUART	the base address of the 16552
DUART_DELTA	the spacing between the hardware registers
XTAL	the frequency of the baud rate crystal
DFLTPORT	the port to use, defined as CHAN1 or CHAN2

The serial driver is separated into two parts: the routines in serial.c and the routines in i510.c and 16552.c.

The serial.c file contains high-level routines that are not specific to a particular UART, but can be implemented differently on some boards. You normally do not change serial.c unless your board or UART has unusual requirements. The routines in serial.c are listed in Appendix B. The calling conventions for the routines in serial.c are listed below:

```
int serial_baud(int port, unsigned long baud)
int serial_open (CONFIG *config)
int serial_read(int port, unsigned char *buf, int len, unsigned timo)
int serial_write(int port, const unsigned char *buf, int len)
int calc_looperms(void)
```

1.1.6.1 Routines in i510.c and 16552.c

The files i510.c and 16552.c contain low-level, device-specific routines. These files have to be changed to work with any other type of UART. The routines in i510.c and 16552.c are listed below:

```
int serial_getc(void)
void serial_init(void)
```

int serial_intr(void)
void serial_loopback(int flag)
void serial_putc(int c)
int serial_set(unsigned long baud)

serial_getc **int serial_getc(void)**

This routine returns a character received if one is immediately available; otherwise it returns a -1.

serial_init **void serial_init(void)**

This routine initializes the serial port. It is called by serial_open.

serial_intr **int serial_intr(void)**

The clear_break_condition routine calls this routine when the monitor is entered because of a break interrupt from the serial port. It waits for the end of the break and clears the FIFO.

serial_loopback **void serial_loopback(int flag)**

If flag is true, this routine enables loopback mode in the UART; otherwise disables it. This routine is called by calc_looperms. If your UART does not support loopback mode, you must change calc_looperms in serial.c.

serial_putc **void serial_putc(int c)**

This routine transmits the character.

serial_set **int serial_set(unsigned long baud)**

This routine sets the baud rate for the serial port. Called by serial_open and serial_baud.

1.1.7 Routines in flash.c

The routines in the flash.c file are:

int check_eeprom(ADDR addr, unsigned long length)
int erase_eeprom(ADDR addr, unsigned long length)

```
void init_eeprom(void)
int is_eeprom(ADDR addr, unsigned long length)
```

Flash memory is electrically erasable and programmable memory (EEPROM). Flash is useful for testing boot code and other non-volatile code that would usually be programmed into EPROM. The QT960 and EV80960SX evaluation boards feature flash memory. These boards use the Intel 28F256 devices so that the programming algorithms in the monitor are for the Intel family of flash memory. Flash memory requires special programming and erasure algorithms. The monitor uses algorithms taken from the *Using the 28F256 Flash Memory for In-System Reprogrammable Nonvolatile Storage* application note. These algorithms can be modified to program larger flash devices using the algorithms found in the *28F010 1024K (128K x 8) CMOS Flash Memory* data sheet. The *28F256 256K (32K x 8) CMOS Flash Memory* data sheet also contains relevant information. You can find a list of these documents with their order numbers in the Preface.

The routines in flash.c can be used with any board that has a single bank of flash EPROM. The memory can be 1, 2, or 4 bytes wide. The driver erases and programs all devices in parallel. You must define the symbol hardware on the compiler command line as the name of the hardware-specific include file (*board*.h), which defines the following symbols:

FLASH_ADDR	the base address of the flash memory
FLASH_WIDTH	the number of devices that are accessed in parallel
PROC_FREQ	the processor frequency in Mhz
TIMER_CNTL	the address of the 8254 timer control register
TIMER_0	the address of the 8254 count register for timer 0
TIMER_XTAL	the frequency in Hz of the timer crystal

If you do not define the symbol PROC_FREQ, the flash initialization code calibrates the timing loop using timer 0 of the 8254. This process works properly even if you change to a processor with a different frequency. Note that the timer is used only during initialization; it is available to the application after that.

To use this driver with a board that does not have an 8254-compatible timer, define the symbol PROC_FREQ as the processor frequency in Mhz. (i.e., if the processor frequency is 25 Mhz, PROC_FREQ is defined as 25.) The timing loop is calibrated for this frequency.

To use this driver with a board that has more than one bank of flash, you must modify the code in init_eeprom that determines the flash type and size and the code in check_eeprom, is_eeprom, erase_eeprom, and write_eeprom that validates the address and length of the region to be programmed or erased.

check_eeprom `int check_eeprom(ADDR addr, unsigned long length)`

This function checks to see if the memory at the specified address is flash memory and then checks to see if that memory is erased.

If addr is equal to the constant NOADDR, this routine checks all of the EEPROM. Otherwise, it checks from the specified address.

This routine sets cmd_stat to E_EEPROM_ADDR if the memory specified is not EEPROM. The routine sets cmd_stat to E_VERSION if flash memory is not supported by this monitor or no flash memory is installed on the board. If the EEPROM is not erased, this routine sets cmd_stat to E_EEPROM_PROG and sets the following global variables:

eeprom_prog_first
eeprom_prog_last

erase_eeprom `int erase_eeprom(ADDR addr, unsigned long length)`

This function erases the specified flash memory.

If addr is equal to the constant NOADDR, this routine erases all of the EEPROM. Otherwise, it erases from the specified address.

If length is 0, erase_eeprom erases the smallest erasable block starting with addr. If length is not 0, it must match exactly the length of one or more erasable blocks starting at addr. If these conditions are not met, the routine sets cmd_stat to E_EEPROM_ADDR and returns ERROR without attempting any erasure.

init_eeprom **void init_eeprom(void)**

The init_hardware routine calls this function to determine the amount of flash memory on the board and initialize the values needed to program the flash memory.

This routine sets the variable eeprom_size to the size of EEPROM available. The init_eeprom routine is not required if init_hardware does not call it.

is_eeprom **int is_eeprom(ADDR addr, unsigned long length)**

Returns: TRUE if region is EEPROM
FALSE if region is not
ERROR if region is partially EEPROM and partially not

This function checks whether the region of memory specified by addr and length is EEPROM. It returns TRUE if the region is EEPROM, FALSE if it is not, and ERROR if it is partially EEPROM and partially not. This function is called by check_eeprom.

1.1.7.1 Local Routines in flash.c

The local routines in the flash.c file are:

long loopcnt(int t)
int program_zero(ADDR addr, unsigned long length)
int program_word(ADDR addr, flash_type data, flash_type mask)
long time(int loops)

loopcnt long loopcnt(int t)

This routine returns the delay constant required to delay t microseconds and uses the time constants calculated by init_eeprom.

program_zero int program_zero(ADDR addr, unsigned long length)

This function programs the specified region of memory with zeros. The erase_eeprom routine calls this function to clear the flash memory before programming it.

program_word int program_word(ADDR addr, flash_type data, flash_type mask)

This function programs the specified address of flash with the specified data value. The mask parameter indicates the bytes to program. This function is called by program_zero and write_eeprom.

time long time(int loops)

This function returns the length of delay time in nanoseconds for a delay loop of the specified length. The function uses timer 0 of the 8254-compatible timer. Called by init_eeprom.

write_eeprom int write_eeprom(ADDR addr, const void *data, int length)

This function programs the specified flash memory.

1.2 Building the Monitor

After modifying source files as necessary for your target board, complete the task of retargeting the monitor by doing the following:

1. Build the ROM image of the monitor. Use the provided makefile in the mon960/common/ directory to coordinate the compiling, assembling, linking, and image-building as described in Section 1.2.1.
2. Write the monitor image into any EPROMs needed for your target using a PROM programmer and any .hex files generated by the makefile.
3. Install the EPROM(s) on your target board. Then connect your target board to a terminal using an RS-232C or RS-422 cable. Ensure that the monitor is running properly on your target board as described in Section 1.3.1.

1.2.1 Making the ROM Image

A single makefile that can build a monitor for any of the evaluation boards using any supported set of tools running on any supported host computer is in the /mon960/common directory. To use this makefile, do the following:

1. If you have retargeted the monitor, first edit the makefile as follows:
   a. Duplicate the group of make commands for the evaluation board that is most similar to your target board. For example, if your target board uses the i960 CA processor, you would duplicate the group of make commands under the heading "EVCA Evaluation Board (EV80960CA)".
   b. Replace all the *BOARD* strings in your new commands with the appropriate string for your target board. For example, replace all occurrences of "EVCA" with "MYCA" and all occurrences of "evca" with "myca".
   c. Edit the line "*BOARD*_OBJS=..." to match the characteristics of your target board and desired monitor. For example, the EVCA line is as follows:

      ```
      EVCA_OBJS= ${CA_OBJS} ${BASE_OBJS} ${UI_OBJS} ${NOFP_OBJS}
      ${NOFLASH_OBJS} ... ${COMM_OBJS}
      ```

      If you don't want to link in the user interface, replace ${UI_OBJS} with ${NOUI_OBJS}. Therefore, a reasonable line might be:

      ```
      MYCA_OBJS= ${CA_OBJS} ${BASE_OBJS} ${NOUI_OBJS}
      ${NOFP_OBJS} ${NOFLASH_OBJS} ${COMM_OBJS}
      ```

2. If you have retargeted the monitor, you also need to make a copy of the linker directives file that reflects the memory configuration of your target board. The linker directives file is specified in the makefile by the line "*BOARD*_ROM_LD= *filename*". For example, make a copy of monevca.ld in which you have modified the memory ranges specified at the beginning of the file.

3. You are now ready to execute the makefile. Enter the command:

   make make HOST=*host* TOOL=*tool*

   where:

host	signifies the host computer on which you are running the make utility. Examples of *host* are sun4, i386v, and dos. See the makefile itself for a complete list.
tool	signifies the toolset you are using to build the monitor. The *tool* option can be intel, gnu960, or mri.

   This command creates a new makefile that enables the appropriate make commands for your combination of host and toolset. The original makefile is saved as Makefile.old.

4. Now make the monitor files. Enter the command:

   make *board*

   where:

board	is the name you used in step 1 (for example, myca).

   This command creates the following output files:

board	the COFF file containing the complete monitor.
board.ima	the binary image file of the complete monitor.
board.hex	the Intel MCS-86 hexadecimal-format file that contains the complete monitor.

   This command creates additional, smaller .hex files if the specified target board is the QT960 or EV960SX, which use multiple EPROMs. These .hex files contain the same information as *board*.hex, but split the information across the EPROMs.

   Most PROM programmers can use hexadecimal-format files.

   Finally, this command also creates a *board*flsh file, which is the COFF file prepared for downloading into FLASH memory using the NINDY monitor on the QT960 or EV960SX target boards.

1.2.2 Producing New EPROMs

After using the makefile to produce the .hex file, you can program any EPROMs needed for your target board using the .hex file and a PROM programmer. If you are using the EV80960CA board, a .hex file is provided. For the QT960 and EV960SX boards, the monitor is provided as both a single .hex file, and four files for the four EPROMs. You can use the single .hex file if your PROM programmer can split the code among four EPROMs.

Finally, install the EPROMs on your target board.

NOTE

Although you could run the DB960 or GDB960 debugger software at this time, do not do so until you have used Mon960 enough to be sure that your retargeting was successful. The debugger adds levels of complexity that can interfere with the task of verifying that the monitor is running correctly.

1.3 Debugging the Monitor

Once the monitor is in EPROM on the target board, you can connect a terminal to the target board and turn on the power on the target board. (You can also use a host system running Comm960 or a terminal program. Comm960 is a terminal emulation program supplied with the monitor.) If your terminal is set to 9600 baud, you should see the string "Mon960" after the system initializes. If your terminal is set to some other baud rate, this string displays only nonsensical characters. In either case, press Enter one, two, or three times. If everything works properly, the monitor signs on with the monitor version number listed in the sign-on line in the form:

Mon960 monitor for the Intel i960 *processor*
Version *version board_name date*
Copyright 1992, Intel Corporation

Where:

processor	is the name of the processor type (CA, CF, KA, KB, SA, or SB).
version	is the version number of the core of the monitor.

board_name	is the value of the variable *board_name* specified in the board_dat.c file.
date	is the build date of the monitor.

The following sections can help you isolate problem areas in the code.

1.3.1 Verifying Monitor Operation

To verify that the monitor is running correctly on your target board, do the following operations with the user-interface commands listed in Chapter 3. Many of these operations are also discussed in Chapter 2.

1. Read memory from EPROM space.
2. Read memory from FLASH space.
3. Read memory from RAM space.
4. Write data to RAM space.
5. Write data to FLASH space.
6. Check FLASH memory.
7. Erase FLASH memory.
8. Download code to RAM space.
9. Set a breakpoint.
10. Execute to the breakpoint.
11. Display the registers.
12. Single step an instruction (step command).
13. Single step a procedure (pstep command).
14. Delete the breakpoint.
15. Download code to the FLASH space.

NOTE

On the QT960 and EV960SX boards, you can download the *board*flsh file, and then rejumper the board to boot the monitor out of FLASH memory instead of EPROM. See Section 3.6 for instruction on loading Mon960 into FLASH memory.

1.3.2 Troubleshooting Host-target Communication Problems

The following problems can occur between the host and the target.

You do not see the Mon960 string when connected at 9600 baud

Check the following:

- The baud rate of the terminal is 9600 baud. At other baud rates you do not see the mon960 message, because it is printed before autobaud is done.
- The terminal program, if you are using one, is configured to use the proper communications port.
- The serial connection is correct. See the user manual for the board for information on the serial-cable connection. Be sure to use a null-modem adapter, if necessary.

If you have retargeted the monitor, check the following:

- Monitor initialization routines.
- Your serial_init routine, and serial_open if you changed it.
- Your serial_putc routine, and serial_write if you changed it.
- If you changed serial_open routine, ensure that it sets the baud rate to 9600 and calls serial_write with the Mon960 string.

You see the Mon960 string, but do not see the sign-on message when you press the Enter key.

Check the following:

- Your serial_getc routine, and serial_read if you changed it.

- Your serial_set routine, and serial_baud if you changed it.

At baud rates other than 9600, you must press the Enter key two or three times to get a response. The autobaud mechanism cannot recognize the baud rate fully when you enter only one character. You must enter the subsequent Enter key presses within 1 second after the first for the multiple entry to be recognized.

Comm960 works properly, but exe960 does not connect

Comm960 is the terminal emulation program included with the monitor. The exe960 file is the "download-and-go" program included with the monitor.

- Check the specification of communications port to exe960.
- Try a lower baud rate. Some hosts cannot keep up at the maximum supported baud rate.
- Check that the code in serial_read that handles timeouts is the same as the code provided.
- Check the routine calc_looperms.
- If you have a spare serial port, use it as a debug port, as described in Section 1.3.2.

If you are using a non-standard communications board in your DOS or Windows host

Check the –d invocation option.

The -d option sets the divisor for the serial crystal frequency. A non-standard communications board is one which does not use a 1.8432 Mhz crystal..

1.3.3 Debugging with a Debug Port

If your target board has an additional serial port, other than the one used to communicate with the host, you can use this port to help debug the monitor. The monitor prints debugging output to a terminal connected to the

additional port. The routines to support this feature are in the evca_dbg.c and evsx_dbg.c files.

The evca_dbg.c file contains routines that work with the Intel 82510 UART. The evsx_dbg.c file contains routines that work with the 16552 UART. If you have a different UART on your target board, change these routines. The code in these files resembles the code in the serial driver files (i510.c and 16552.c), used for communication with the host. If you change the serial driver file, make parallel changes to the debug code.

To enable debugging output, delete the object files for the serial driver and hardware-specific routines. For example, evca_hw.c, as well as any module from which you want debug output produced. Change the makefile by deleting the comment character in the makefile on the following lines:

```
# DEBUG = –DDEBUG
# DEBG_OBJS = debug.o
# BOARD_DEBG_OBJS = list-of-files
```

Recall that you might want to edit the Makefile.old file, rather than just Makefile. See Section 1.2.1 for more information on making the ROM image.

Then, rebuild the monitor using the makefile and reprogram your EPROMs, as described in Section 1.2.

When you boot the monitor, debugging output appears on the terminal attached to the debug port. The monitor operates normally with the debugging code enabled. However, for best performance, rebuild the monitor without the debugging code after it is working properly.

The debug.h file contains macros that print debugging information. These macros keep track of the current nesting level, so you can see which routines call a given routine.

The debug.h file contains the following macros:

HDEBUG	is called at the entry point of a routine and is given the name of the routine. Any routine that calls HDEBUG must call the RETURN or VRETURN macro in all places where it returns.
RETURN	returns a value.
VRETURN	returns from void routines.

ADEBUG — general-purpose output macro, that takes the following parameters:

- a printf-style format string with no more than one format specification
- the argument for the format specification, (NULL if the file contains no format specification)

Throughout the code, you can add calls to the ADEBUG macro as required to help isolate bugs.

For examples of the use of these routines, see the evca_hw.c or i510.c files that came with the monitor.

The dbg_openport routine in the board_dbg.c file initializes the debug port. This routine is called automatically when the first debug macro is called. However, the init_hardware routine can call it again to initialize it for interrupt-driven operation. The latter call is made by passing it a non-zero argument, which is the interrupt vector number to use. It places the appropriate vector in the interrupt table.

The debug port must be interrupt-driven to be used in the serial_read or serial_write routines, or any of the routines they call. If the debug port is not interrupt-driven, the delays while waiting to send characters cause characters from the host to be lost.

If you have debugging output only in initialization routines, you need not make the debug port interrupt-driven. However, you must change the serial_read and serial_write routines by deleting the calls to HDEBUG and changing RETURN to return.