Application Note

AN2140/D
Rev. 1, 6/2003

Serial Monitor for MC9S08GB/GT

By Jim Sibigtroth
8/16 Bit Systems/Applications Engineering
Austin, Texas

Introduction

This application note describes a 1-Kbyte monitor program for the MC9S08GB60 MCU. This program supports 19 primitive debug commands to allow FLASH programming and debug through an RS-232 serial interface to a personal computer. This monitor supports primitive commands to reset the target MCU, read or modify memory (including FLASH memory), read or modify CPU registers, go, halt, or trace single instructions. In order to allow a user to specify the address of each interrupt service routine, this monitor enables a hardware logic feature that redirects interrupt vectors to an unprotected portion of FLASH just before the protected monitor program. This monitor will be modified for use with other members of the HCS08 Family as they are introduced.

A user on a tight budget can evaluate the MCU by writing programs, programming them into the MCU, and debugging their applications for the HCS08 using only a serial I/O cable and free software for their personal computer. This monitor does not use any RAM other than the stack itself.

The monitor turns off the COP watchdog. This development environment assumes you reset to the monitor when you are going to perform debug operations. If your code takes control directly from reset, and then an SCI1 interrupt or software interrupt (SWI) attempts to enter the monitor, the monitor may not function because the SCI, frequency-locked loop (FLL), FCLK, and COP may not be initialized as they would be for a cold reset into the monitor.

There is limited error handling for the FLL. If the frequency source is missing or broken, the monitor will not function. The crystal oscillator can take any amount of time to start. If the FLL loses clocks after the monitor is running, a reset is forced. If the FLL loses lock during operation, it may or may not fail — if it can regain lock, it can continue normally. If a user interrupt service routine (ISR) is present, the monitor uses that routine. If there is no user ISR, the monitor ICG

ISR acknowledges the error and does a real-time interrupt (RTI). If the loss of lock was due to noise, the monitor would resume normal operation. If there was a more serious problem such as a broken FLL, the monitor would stop working. Normally, interrupts are blocked while the monitor has control so this ISR is really just included for the case where a user clears I in their code and then the ICG experiences a loss of lock. (This reduces the risk that an ICG failure would lock up the user's application.)

Block Protection

In order to prevent accidental changes to the monitor program itself, the 1 Kbyte block of FLASH memory where it resides ($FC00–$FFFF), is block protected. The only way to change the contents of this protected block, is to use a BDM-based development tool (a BDM pod) to disable the block protection and then bulk erase the FLASH memory (or at least the last two 512-byte pages).

In the lowest-cost applications where the monitor is used with an SCI serial interface to the RS-232 serial port on a personal computer, there is no way to accidentally erase or modify the monitor software. Not even errors in a user program can cause changes to the monitor because the block protect can only be disengaged through a BDM command with a BDM-based debug pod.

Baud Rate Detection

The MC9S08GB60 version of this monitor assumes a 32.768-kHz crystal. It programs the FLL to produce a bus frequency of 18.874368 MHz and accommodates RS-232 serial communications through SCI1 at 115.2 kbaud. The monitor does not attempt to allow debugging of a user program that uses a different crystal frequency or sets up the FLL differently. For such systems, you should purchase a BDM pod which allows more sophisticated debugging.

Other variations of the monitor will use the internal oscillator on some HCS08 derivatives and/or other combinations of crystals and FLL settings. However, they will all send a break and wait for a carriage return before sending the first prompt sequence. In a later section of this application note, the parameters and control register settings that are related to the oscillator, FLL, and other frequency-sensitive features of the monitor will be discussed in greater detail.

During initialization after any cold reset, a long break is transmitted before other SCI communications. This break is about 30 bit-times to ensure that a Windows-based PC can recognize this as a break. In order to establish communications with the monitor, the host must send a carriage return ($0D) at the correct baud rate. If the monitor detects some other character, it implies

the host baud rate is not correct so it continues to wait in a loop for the $0D character before printing the first prompt sequence.

The host can detect the monitor baud rate by initially sending a carriage return at 115.2 kbaud. If this is the correct baud rate, the target MCU will respond with a prompt sequence of $E0, $08, and a ">" prompt character. If the host does not see the prompt sequence, it should try sending another carriage return at 57.6 kbaud, then 38.4 kbaud, then 19.2 kbaud, and finally 9600 baud, until the prompt sequence is received to indicate the correct communication rate.

Monitor Commands

The monitor recognizes 19 primitive binary commands that enable a third-party development tool vendor to develop a full-featured debug program. These commands use 8-bit command codes optionally followed by binary addresses, control, and data information depending upon the specific command.

In the following command descriptions, a shorthand notation is used to describe the command syntax. Each command starts with a binary command code. A slash (/) is used to separate parts of the command in these descriptions, but these slash characters are not sent as serial characters in commands. Underlined parts of the command are transmitted from the host PC to the target MCU while the portions that are not underlined are transmitted from the target MCU to the host PC.

The first two characters in each command sequence are the 1-byte command codes and are shown as a literal hexadecimal value such as A1. Other abbreviations used in the command sequences are shown here:

AA — The contents of the 8-bit accumulator

AAAA — A 16-bit address

CC — The contents of the 8-bit condition codes register

EADR — The 16-bit end address for an erase command

HH — The contents of the 8-bit H register (high half of H:X)

NN — The number of bytes (–1) for the block read and write commands

PH — The upper 8-bits of the user's program counter (PCH)

PL — The lower 8-bits of the user's program counter (PCL)

RD — One byte of read data

$RD_{(AAAA)}/RD_{(AAAA+1)}/.../RD_{(AAAA+NN)}$ — A series of 8-bit read data values from address locations AAAA through AAAA+NN.

SADR — The 16-bit start address for an erase command

SH — The upper 8-bits of the user's stack pointer (SPH)

SL — The lower 8-bits of the user's stack pointer (SPL)

WD — One byte of write data

$WD_{(AAAA)}/WD_{(AAAA+1)}/.../WD_{(AAAA+NN)}$ — A series of 8-bit write data values for address locations AAAA through AAAA+NN.

XX — The contents of the 8-bit index register X (low half of H:X)

Some monitor commands such as Read_Byte and Write_Byte can be executed at any time while others may only be executed while the monitor is active and waiting for commands (as opposed to running user code). Commands such as those that write to CPU registers would not make sense while user code is running because they would result in unexpected program execution. If these commands are attempted while user code is running, the command will not be executed and an error message will be returned to the host system as part of the next prompt sequence. Unless otherwise noted in the following command descriptions, the command can be executed at any time (while the monitor is active or while a user program is running).

$A1 — Read_Byte

A1/AAAA/RD — Reads a byte of data from the specified 16-bit address and sends the 8-bit data back to the host PC.

$A2 — Write_Byte

A2/AAAA/WD — Writes the supplied byte of data to the specified 16-bit address. All writes are processed through an intelligent routine that programs FLASH or writes to RAM or registers based on the address being written. If any error occurs during an attempt to program a nonvolatile memory location, an error code is transmitted before a new prompt is issued. See **Intelligent Writes** for more detail.

$A5 — Read_Next

A5/RD — Pre-increments the user H:X register (by 1), reads the byte of data from 0,X, and sends the 8-bit data back to the host PC. This command is not allowed when the user program is running because the monitor cannot control the contents of the H:X index register.

$A6 — Write_Next

A6/WD — Pre-increments the user H:X register (by 1) and writes the supplied byte of data to 0,X. All writes are processed through an intelligent routine that programs FLASH or writes to RAM or registers based on the address being written. If any error occurs during an attempt to program a nonvolatile memory location, an error code is transmitted before a new prompt is issued. See **Intelligent Writes** for more detail. This command is not allowed when the user program is running.

$A7 — Read_Block

A7/AAAA/NN/$RD_{(AAAA)}$/$RD_{(AAAA+1)}$/.../$RD_{(AAAA+NN)}$ — Reads a series of NN+1 (1 to 256) bytes of data starting at address AAAA and returns the data one byte at a time to the host (starting with the data read from address AAAA and ending with the data from address AAAA+NN). Although this command can be executed while a user program is running, it is not usually recommended because it could slow down operation of the user program.

$A8 — Write_Block

A8/AAAA/NN/$WD_{(AAAA)}$/$WD_{(AAAA+1)}$/.../$WD_{(AAAA+NN)}$ — Writes a series of NN+1 (1 to 256) bytes of data into the target MCU's memory starting at address AAAA and ending with address AAAA+NN. All writes are processed through an intelligent routine that programs FLASH or writes to RAM or registers based on the address being written. If any error occurs during an attempt to program a nonvolatile memory location, an error code is transmitted before a new prompt is issued. See **Intelligent Writes** for more detail. This command is capable of programming locations in FLASH memory even while a user program is running from within the same FLASH memory. Although this command can be executed while a user program is running, it is not usually recommended because it could slow down operation of the user program.

$A9 — Read_Regs

A9/SH/SL/PH/PL/HH/XX/CC/AA — Sends the current contents of user registers SPH, SPL, PCH, PCL, H, X, CCR, A (in that order) to the host PC. The SP value is the user SP value and while the monitor is active and waiting for commands, the real SP is 6 less due to the user register stack frame. Although this command can be executed while a user program is running, it is not usually recommended because these register values change much more quickly than they can be read.

$AA — Write_SP

AA/SH/SL — Adjusts the specified 16-bit data to compensate for the user register stack frame (–6) and writes this adjusted value to the stack pointer register. The monitor uses stack space below the user register stack frame. When you execute a Go or Trace1 command, the monitor exits to the user program by pulling H and then executing an RTI instruction. Because of the monitor requirements for stack space, the valid range of values for SP in the Write_SP command is from RamStart+$45 to RamLast.

The monitor doesn't move the user register values to the new user register stack frame so the host should re-write all user register values after changing the SP value, and before attempting to display current user register values. This command is not allowed when the user program is running. The first AA in this command sequence is the Write_SP command code.

$AB — Write_PC

AB/PH/PL — Writes the specified 16-bit data to PCH:PCL in the user register stack frame. This command is not allowed when the user program is running.

$AD — Write_HX

AD/HH/XX — Writes the specified 16-bit data to the H:X index register pair in the user register stack frame. This command is not allowed when the user program is running.

$AE — Write_A

AE/AA — Writes the specified 8-bit data to accumulator A in the user register stack frame. This command is not allowed when the user program is running.

$AF — Write_CCR

AF/CC — Writes the specified 8-bit data to the condition codes register (CCR) in the user register stack frame. This command is not allowed when the user program is running.

$B1 — Go

B1 — The monitor pulls H then executes an RTI to copy user CPU register values from user register stack frame into the actual CPU registers. Processing resumes in the user program at the location specified by the user program counter that was in the user register stack frame. To go to an arbitrary address in the user's program, you can first use a Write_PC command to set the user's program counter to a new location.

In run mode, the user's application program will execute until it is interrupted by a breakpoint, an SCI1 interrupt, or a Halt command. In the case of a breakpoint or Halt command, the monitor will clear the run mode flag to indicate to the monitor that it should remain active waiting for further commands from the host. In the case of an SCI1 interrupt, the run flag is set (or remains set) to indicate that the monitor should return to the user's application program after completing the current command.

Since the monitor requires the SWI and SCI1 interrupt vectors to be programmed with specific values, the Go and Trace1 commands perform a check to make sure these vectors are good before exiting the monitor to run user code. If the vector locations are erased ($FF), the verification routine tries to program the correct values into the FLASH vector locations. If this is successful, or if the vectors were already correct, the monitor finishes with a Go or Trace1 command. If the vectors cannot be programmed correctly, the

monitor aborts the Go or Trace1 command and reports the error with the next prompt sequence.

If the Go command is executed while a user program is running, control returns to the running user program.

$B2 — Trace1

B2 — The monitor sets up the on-chip debug module to force a CPU breakpoint immediately after executing a single instruction in the user's program. It then pulls the H register and executes an RTI to copy user CPU register values from user register stack frame into the actual CPU registers. Processing resumes in the user program at the location specified by the user program counter that was in the user register stack frame.

After executing a single user instruction, an SWI is forced to cause control to return to the monitor program. In response to the SWI, the monitor clears the run flag (or leaves it cleared) to indicate that the monitor should remain active waiting for additional commands from the host.

Since the monitor requires the SWI and SCI1 interrupt vectors to be programmed with specific values, the Go and Trace1 commands perform a check to make sure these vectors are good before exiting the monitor to run user code. If the vector locations are erased ($FF), the verification routine tries to program the correct values into the FLASH vector locations. If this is successful, or if the vectors were already correct, the monitor finishes with a Go or Trace1 command. If the vectors cannot be programmed correctly, the monitor aborts the Go or Trace1 command and reports the error with the next prompt sequence.

If the Trace1 command is executed while a user program is running, one additional user program instruction will be executed and then the monitor will become active and wait for additional commands.

$B3 — Halt

B3 — This command is used to force the user's application program to stop executing and the monitor to gain control and remain active to wait for additional commands from the host. This command requires an enabled SCI1 interrupt so it can only be recognized if the user application program has cleared the I bit in the CCR. If the user program temporarily blocks interrupts, such as during execution of another interrupt service routine, the Halt command will not be recognized until the user application program re-enables interrupts (typically by executing the RTI at the end of an interrupt service routine).

$B4 — Reset

B4 — If there is a programmed user reset vector, the level on the user/monitor switch (PTA7 in the MC9S08GB60), and the level on the RxD1 line; this will cause a reset to user code or to the monitor. The sequence of checks at reset include:

1. Check for possible stop2 mode recovery (indicated when the PPDF bit in SPMSC2 register is 1). If PPDF = 1, the MCU pins remain latched to the states whey had when stop2 mode was entered. Control is passed to the user's reset initialization code so pin and peripheral states can be reconfigured before PPDACK is written to restore I/O pins to normal operation.
2. Check for warm start (which implies reset was caused by an SCI1 Rx interrupt or a break in the user code (SWI) where the SP was not valid in the interrupt service routine.) A warm start is indicated when all of the following conditions are true...
    a. SRS register indicates reset was caused by an illegal opcode
    b. A 16-bit signature at RamLast – 3 = warmSig
    c. The old saved baud rate at RamLast matches the normal baud rate setting for the monitor (baud115200 in the MC9S08GB60)

Warm start skips the long break and restores the previous baud rate instead of waiting for the next carriage return. The warm reset is needed in the case of an SCI1 Rx interrupt or SWI with an invalid SP value because the interrupt stacking could have corrupted RAM or register values and because the monitor cannot function without a valid stack.

3. If the user/monitor switch (PTA7 in the MC9S08GB60) = 0 (logic low), force monitor reset
4. If RxD1 = 0 (logic low) force monitor reset. If no RS-232 device is connected, or if RxD is being driven by an idle level, RxD1 will be 1 so monitor mode is not selected.
5. If the first byte of the user reset pseudo-vector = $FF (unprogrammed) force monitor reset

If none of the above, use the reset pseudo-vector to jump to the user's reset startup routine. Two pullup enable registers are modified during the monitor startup, but they are restored to their reset values before going to the user reset location. The user has complete control over all control registers (including write-once registers and bits) as if the user's reset routine had started immediately after the actual reset.

$B5 — Erase

B5/SADR/EADR — Erase pages of FLASH memory starting with the FLASH page that includes address SADR and ending with the FLASH page that includes address EADR. SADR and EADR must specify a valid range of addresses in FLASH memory. Before checking for a valid FLASH address range (in the MC9S08GB60), the low order half of SADR is forced to $80 and the low order half of EADR is forced to $FF. Multi-page erase commands are accomplished by starting with a page address of SADR (after modification) and repeating a loop that erases one page of FLASH memory, adds $200 (512) to the page address, and repeats the loop until the page address is past the address EADR (after modification). This unusual algorithm prevents any possibility of unintended changes to high-page registers in case the range of FLASH addresses being erased overlaps the high-page register space from $1800–$182B and prevents the possibility that the starting address would fall in the inaccessible portion of the first FLASH page ($1000–$107F) where internal RAM takes priority over the FLASH memory.

$B6 — Erase_All

B6 — Erase all FLASH memory except the 1-Kbyte monitor at $FC00–$FFFF. If this command is issued while an application program is running, the run flag is cleared so the monitor retains control after the FLASH memory is erased.

$B7 — Device_Info

B7 — Returns the HCS08 device revision and ID code followed by a normal prompt sequence. In HCS08 devices, the information for the mask revision and device ID comes from the system device identification register (SDIDH:SDIDL) at $1806:$1807 which contains a 4-bit revision number and a 12-bit device identification code. For the original MC9S08GB60, this code is $0:$002, so the complete response to the $B7 Device_Info command is: $00, $02, $E0, $00 (or $01 if a user program is running), $3E (the prompt symbol “>”).

Command Error Codes

A 3-character prompt is issued after monitor initialization and after each command is completed. A prompt is not issued after a Go command until a breakpoint is encountered or a Halt command stops execution of the user’s application program. The prompt sequence consists of a 1-byte error code, a 1-byte status code, and a ‘>’ prompt symbol ($3E). After initialization or after a command is executed successfully, the error code is $E0 indicating no error. After a cold reset initialization, the status code is $08 indicating a hard reset has occurred and the monitor is in active monitor mode waiting for additional commands from the host. Therefore the complete 3-character prompt after a cold reset is $E0, $08, $3E.

Some commands are not allowed while the target MCU is in run mode because they would interfere with proper execution of the application program (see error

code $E2). In addition, other commands are not recommended while the target MCU is in run mode, but since these commands do not interfere with proper execution of the application program, other than to slow it down, they are allowed to execute and do not result in an error.

The following paragraphs describe each of the possible error codes in more detail.

$E0 — No Error

This code is used after any successful command. It indicates there are no pending errors.

$E1 — Command not recognized

This code indicates the previous command code was not one of the recognized command codes. If the monitor was in run mode, control returns to the user's application program. If the monitor was not in run mode, control returns to the top of the command loop to wait for the next command from the host.

$E2 — Command not allowed in Run Mode

This code indicates that the requested command is only legal when the monitor is halted (active monitor mode). In the case of a command request that is not legal in run mode, the command is not executed to avoid corrupting the running user application program. The commands that are not allowed in run mode are:

- Read_Next and Write_Next are not allowed because these commands use the user's H:X register value as a pointer and this value constantly changes during run mode.
- Write_SP, Write_PC, Write_HX, Write_A, and Write_CCR are not allowed in run mode because they change much faster than the host could check their contents. Therefore there is no way to predict how these changes would affect the application program so it would not make sense to execute these commands while the application program is running.

Although commands that write to memory or erase FLASH memory (Write_Byte, Write_Block, Erase, or Erase_All) are allowed while running a user program, care should be used to avoid changing the program itself while it is executing.

$E3 — Stack Pointer Out of Range

This error code indicates that when the monitor program took over control from a running user program, the stack pointer was not pointing to a valid RAM location. This is an unrecoverable error because the interrupt that caused the monitor to gain control wrote to several memory locations above the current invalid stack pointer location. If the stack pointer was pointing into the on-chip registers, this could have corrupted important system configuration settings. In addition, the user PC value may not have been written to a read/write location so the monitor would not know where to return to the user program.

Since the monitor requires certain control register settings and a valid stack to function correctly, a bad SP error results in the monitor forcing a reset to restore required settings. Most of the time, this can be a warm reset that skips the long break and the checks for going directly to the user reset vector. After the warm reset, clocks and control registers are initialized and the prompt will include the bad SP error message and the warm start status code ($E3, $0C, $3E).

A warm reset is also forced if the monitor gets an unexpected interrupt from the SCI1 Tx or Error interrupt sources. If the user program has experienced code runaway and has changed the SCI1 control registers to allow these interrupts, the safest response is to force a (warm) reset to allow the monitor to regain control.

$E4 — Write_SP Attempted with an Invalid SP value

This error code indicates that the host attempted a Write_SP command with an invalid 16-bit SP value. In order for the monitor program to function correctly, the SP must always point into a valid area of RAM to support monitor functions. The Write_SP command adjusts the supplied SP value (by subtracting 6) to compensate for the user register stack frame. In addition the monitor needs a certain amount of stack space for stacking return addresses for nested subroutine calls and for temporary storage of register contents during the normal course of executing the monitor program. Because of these monitor requirements, the valid range of values for SP is a little less that the whole range of RAM addresses. See **$AA — Write_SP** for more detail.

$E6 — FLASH Error

This code indicates that an error occurred during an attempt to write or erase FLASH memory. Possible errors that would trigger this error code are:

- An FACCERR (access error)
- An FPVIOL (protection violation error)
- An attempt was made to program a FLASH location that was not previously erased

In cases where a single byte was being programmed or a single page was being erased, the attempted FLASH operation would not be performed. In cases where multiple bytes were being programmed or multiple pages were being erased, this error indicates that at least one location or page erase operation was not performed. This monitor does not provide more detailed information about these errors. The debug tool or programmer running in the host PC can perform additional memory reads to get more detailed information about the error.

$E7 — Erase Range Error

The FLASH memory in an HCS08 is organized into pages of 512 bytes each. The starting address SADR and the ending address EADR for the Erase command must specify a valid range of addresses in the FLASH memory or

this $E7 error will be generated and no FLASH locations will be erased. Refer to **$B5 — Erase** for more information.

$E8 — Go or Trace1 with No Vectors This code indicates that a Go or Trace1 command was attempted, but critical interrupt vectors for SWI or SCI1 were not properly initialized in the unprotected area of FLASH memory at $FBxx. If possible, the monitor will try to program correct vectors into these locations. However, if the locations contained non-FF values, programming is not possible to fix the vectors without erasing user FLASH memory.

Monitor Status Codes

The second character of a 3-character prompt is a status code that tells the host debug program the current state of the monitor.

$00 — Monitor Active This code indicates that the user's application program is not currently running and the monitor is active and waiting for further commands from the host.

$01 — User Program Running This code indicates that the user's application program is currently running. In this mode, the host may still issue commands to read or write memory locations or halt the user program so the monitor regains control. However, the monitor can only honor such command requests if/when the user program has cleared the I bit in the CCR because these commands rely on an SCI1 receive interrupt to gain the attention of the monitor program.

$02 — Halt This code indicates that a Halt command was executed to stop the running user program. After the Halt command, the monitor remains active waiting for additional commands.

$04 — Trace Done A Trace1 command was just executed and the monitor is active waiting for additional commands.

$06 — Breakpoint An SWI was encountered and a Trace1 command was not the cause of the SWI. Other possible causes of the SWI include a breakpoint or the end of a debug run (both caused by the on-chip DBG module hardware).

$08 — Cold Reset This code indicates a cold reset has just occurred. The usual causes of cold reset are a power-on event, a Reset command, or the user pressing the reset

button on the target system. A cold reset can also result if errors in a user program cause code runaway. In the case of code runaway, an attempt to execute an illegal opcode would generate a system reset.

$0C — Warm Reset

This code indicates a warm reset has just occurred. The monitor makes a distinction between a cold reset and a warm reset in order to allow faster recovery when the reset is caused by a known event such as an SWI or SCI1 Rx interrupt where the stack pointer was out of range. A warm reset can also be caused by an unexpected interrupt from the SCI1 Tx or error systems whether the stack pointer is good or not. Although the stack pointer may be valid, an error must have occurred because these interrupts should never occur unless a user program disturbed the SCI1 control settings.

Command Decode

The code in lines 517 through 528 implements a simple lookup table to decode monitor commands. The key to this routine is the command table which consists of a 3-byte entrSy for each command. The first byte of each entry is the command code such as $A1 for the Read_Byte command. The last two bytes of each entry are the high and low halves of the address where the command routine can be called. For example the 3-byte entry for the Read_Byte command is $A1,$FC06 in lines 227 and 266. The $A1 is the command code, and $FC06 is where the Read_Byte command routine (RdByteCmd) is located. Lines 264 and 265 are the command table entries for the Device_Info command. The label tableEnd in line 266 marks the end of the command table. This structure makes it easy to add or remove commands.

```
227 FC06 A1        commandTbl:  dc.b   $A1
228 FC07 FD7A                   dc.w   RdByteCmd     ;read byte
229
230 FC09 A2                     dc.b   $A2
231 FC0A FDA2                   dc.w   WtByteCmd     ;write byte
 "   "    "                     "      "             "
264 FC3C B7                     dc.b   $B7
265 FC3D FD27                   dc.w   DeviceCmd     ;Return device information
266      0000 FC3F tableEnd:    equ    *             ;end of command table marker
 "   "    "           "         "      "               "
517 FD5D AD 6C     prompt1:     bsr    GetChar       ;get command code character
518 FD5F 45 FC06                ldhx   #commandTbl   ;point at first command entry
519 FD62 F1        CmdLoop:     cmp    ,x            ;does command match table entry?
520 FD63 27 0B                  beq    doIt          ;branch if command found
521 FD65 AF 03                  aix    #3            ;advance to next table entry
522 FD67 65 FC3F                cphx   #tableEnd     ;see if past end of table
523 FD6A 26 F6                  bne    CmdLoop       ;if not, try next entry
524 FD6C A6 E1                  lda    #ErrCmnd      ;code for unrecognized command
525 FD6E 20 D8                  bra    Prompt        ;back to prompt; command error
526
527 FD70 9ECE 01   doIt:        ldhx   1,x           ;get pointer to command routine
528 FD73 FC                     jmp    ,x            ;go process command
```

Monitor versus Run Mode

The target MCU operates in either monitor active mode or run mode. Monitor active mode refers to the mode of operation where the target MCU is executing code within the monitor and keeps active control waiting for additional commands through the serial interface. Run mode refers to the mode of operation where the target MCU is executing the user's application program. Some monitor commands such as Read_Byte and Write_Byte can be executed while the target MCU is in run mode. In this case, an SCI1 interrupt causes the monitor to temporarily gain control to decode and execute the requested command, but when the command is completed the monitor automatically returns control to the user's application program. Throughout such a sequence, the target MCU is said to be in run mode even though software in the monitor is executed to complete the requested command.

The Halt command causes an SCI1 interrupt which causes the CPU registers (except H) to be pushed onto the stack. The ISR for SCI1 then sets the run mode flag (even though this flag will be cleared again if the command that caused the SCI1 interrupt turns out to be the Halt command). The H register is pushed to complete the user register stack frame, and the monitor proceeds to the command decode routine.

Breakpoints use the SWI interrupt mechanism (the Trace1 command also uses a hardware breakpoint), which is not blocked when the I bit in the CCR is set.

This implies that Trace1 and breakpoints always work independent of what the user program does to the I bit. Other commands use the SCI1 interrupt, therefore these other commands cannot execute unless the user program clears the I bit. There are some cases where it would be natural for a user program to set the I bit (or leave it set) such as during reset initialization routines and when the user program is executing an interrupt service routine. Most such cases would only block interrupts for a very short period of time so the user would not notice that a command request was delayed. If a user program erroneously fails to clear the I bit or gets stuck in an interrupt service routine, commands through the SCI serial link cannot be recognized. If this condition persists, the user must force a reset or cycle the power to the target system so it gets a power-on reset so the monitor can regain control.

Intelligent Writes

The intelligent write routine uses the address in H:X to decide what to do. If the location is in FLASH, it checks to see if the location is already correct (if so it skips the program operation and signals success). If the FLASH location is different than desired, the routine checks to see if it is erased (it is considered an error if you try to change a location in nonvolatile memory that is not blank). If those checks pass, the routine does a byte program operation and finally checks FACCERR and FPVIOL to make sure there was no access error or protection violation during programming (it is considered an error if there was). If the location is not FLASH, the routine writes the requested data to the specified address. See **WriteA2HX Subroutine** and listing lines 1055–1088 for more detail.

DoOnStack Subroutine

This unusual subroutine is used to program FLASH locations or perform page-erase operations in the FLASH memory. Like most nonvolatile memories, you cannot execute a program out of the FLASH while a program or erase operation is being performed on the same nonvolatile memory. Because of this, the DoOnStack subroutine (which is located in the FLASH), copies a small routine onto the stack (in RAM) and then passes control to that subroutine on the stack. When the operation is finished, an RTS returns control to the DoOnStack routine which deallocates the space used by the small stack routine and then returns to the program from where DoOnStack was called.

Prior to calling DoOnStack, the main program started a FLASH operation by writing to FSTAT to clear out any previous error flags. Also, the A accumulator is pre-loaded with the command code for Byte_Prog or Page_Erase.

The first two lines in DoOnStack save H and X on the stack, and the third line stores the command code on the stack. Lines 1140 through 1144 copy the SpSub routine onto the stack (with a series of PSHA instructions) starting with the last byte of SpSub and ending with the push of the first byte of SpSub onto the stack. At this point SP points to the location just before the first byte of the stacked SpSub routine. The TSX in line 1145 copies SP+1 into H:X so H:X points at the copy of SpSub on the stack. While the monitor program is active, the I bit in the CCR is set to mask interrupts. Since this routine can also be called as a utility subroutine from a user program, the I bit may or may not be set at the time DoOnStack is called. Lines 1146–1148 check to see if I is set or clear. If the I bit was already set, lines 1154–1155 load the data for the command and call the subroutine on the stack to complete the requested FLASH operation. If the I bit was clear, line 1149 is used to set I before calling the subroutine on the stack and line 1152 is used to restore it to 0 to re-enable interrupts when the FLASH memory is back in the map.

Line 1150 or 1154 preloads A with the data for the FLASH operation. The JSR in line 1151 or 1155 calls the copy of SpSub that is now located on the stack. The SpSub subroutine was written in a position-independent manner so it could be copied to a new location (on the stack) and would still execute as expected. In this case, SpSub is such a short subroutine that it was easy to make it position independent.

```
1136 FF72 89        DoOnStack:   pshx
1137 FF73 8B                     pshh                ;save pointer to flash
1138 FF74 87                     psha                ;save command on stack
1139 FF75 45 FFB0                ldhx   #SpSubEnd    ;point at last byte to move to stack
1140 FF78 F6        SpMoveLoop:  lda    ,x           ;read from flash
1141 FF79 87                     psha                ;move onto stack
1142 FF7A AF FF                  aix    #-1          ;next byte to move
1143 FF7C 65 FF98                cphx   #SpSub-1     ;past end?
1144 FF7F 26 F7                  bne    SpMoveLoop   ;loop till whole sub on stack
1145 FF81 95                     tsx                 ;point to sub on stack
1146 FF82 85                     tpa                 ;move CCR to A for testing
1147 FF83 A4 08                  and    #$08         ;check the I mask
1148 FF85 26 09                  bne    I_set        ;skip if I already set
1149 FF87 9B                     sei                 ;block interrupts while FLASH busy
1150 FF88 9ED6 001E              lda    SpSubSize+6,sp ;preload data for command
1151 FF8C FD                     jsr    ,x           ;execute the sub on the stack
1152 FF8D 9A                     cli                 ;ok to clear I mask now
1153 FF8E 20 05                  bra    I_cont       ;continue to stack de-allocation
1154 FF90 9ED6 001E I_set:       lda    SpSubSize+6,sp ;preload data for command
1155 FF94 FD                     jsr    ,x           ;execute the sub on the stack
1156 FF95 A7 1B     I_cont:      ais    #SpSubSize+3 ;deallocate sub body + H:X + command
1157                                                 ;H:X flash pointer OK from SpSub
1158 FF97 48                     lsla                ;A=00 & Z=1 unless PVIOL or ACCERR
1159 FF98 81                     rts                 ;to flash where DoOnStack was called
```

Although SpSub appears to be located at $FF99–$FFB0, during execution we actually execute a copy of SpSub that is temporarily located on the stack in RAM. SpSub completes the FLASH command by performing the following steps:

1. Write to a FLASH location. The data for this write was in A when SpSub was started. The address was previously stored on the stack and is loaded into H:X with a stack-pointer-relative LDHX instruction at line 1175.
2. Write the code for Byte_Prog or Page_Erase to the FCMD register. The command code is also retrieved from the stack with a stack-pointer-relative load instruction.
3. Write a 1 to FCBEF to initiate the command.
4. Wait for FCCF = 1 to indicate the operation is complete and the FLASH is again visible in the normal memory map.

The NOP in line 1181 is used to ensure that there are at least four cycles between the write to FCBEF (line 1180) and the first read of FSTAT (line 1182). This delay is required so the internal FLASH command sequencer can properly update the FCBEF and FCCF flags in FSTAT. Execution stays in the ChkDone loop until the command finishes (FCCF becomes set). At this point the FLASH is back in the memory map and we can return to DoOnStack (which is in the FLASH).

```
1175 FF99 9EFE 1C   SpSub:       ldhx   <SpSubSize+4,sp ;get flash address from stack
1176 FF9C F7                     sta    ,x              ;write to flash; latch addr and data
1177 FF9D 9ED6 001B              lda    SpSubSize+3,sp ;get flash command
1178 FFA1 C7 1826                sta    FCMD            ;write the flash command
1179 FFA4 A6 80                  lda    #mFCBEF         ;mask to initiate command
1180 FFA6 C7 1825                sta    FSTAT           ;[pwpp] register command
1181 FFA9 9D                     nop                    ;[p] want min 4~ from w cycle to r
1182 FFAA C6 1825   ChkDone:     lda    FSTAT           ;[prpp] so FCCF is valid
1183 FFAD 48                     lsla                   ;FCCF now in MSB
1184 FFAE 2A FA                  bpl    ChkDone         ;loop if FCCF = 0
1185 FFB0 81        SpSubEnd:    rts                    ;back into DoOnStack in flash
1186      0000 0018 SpSubSize:   equ    (*-SpSub)
```

The RTS in line 1185 returns to DoOnStack at line 1152 or 1156. The AIS instruction in line 1156 deallocates the stack space used by the SpSub subroutine and temporary storage locations that resulted from pushes at lines 1136–1138 and leaves SP pointing at the return address to the main program. The ASLA in line 1158 moves the FCCF flag, which was set, into the carry bit. This leaves the FPVIOL and FACCERR flag bits in the two most significant bits of A. A should be $00 unless there was a protection violation or an access error as a result of the FLASH operation that was just performed. A simple BEQ or BNE can be used to check for errors after returning to the main program.

Vector Redirection

HCS08 devices support an optional hardware vector redirection mechanism that can automatically redirect vectors (except reset) if a portion of the FLASH memory is block protected. This monitor uses that mechanism rather than a software pseudo-vector mechanism that is traditionally used in ROM monitors. This monitor resides at $FC00–$FFFF, this 1-Kbyte block is protected, and the nonvolatile FNORED bit is programmed to 0 to enable hardware vector redirection. When an interrupt occurs, the vector is fetched from $FBCC–$FBFD rather than from $FFCC–$FFFD, but all timing and cycle-by-cycle activity is the same as when the vectors were not redirected. This vector redirection mechanism configures the interrupt vectors so they appear in unprotected space which in turn allows the user to control the contents.

Because the vectors are in unprotected space, it is possible for a user to intentionally or unintentionally erase the vectors for SWI and SCI1 that the monitor needs to perform some operations. After power-on, reset, or while the monitor is active, the I bit in the CCR is set, which prevents interrupts from being recognized so the vectors are not needed at those times. Interrupts only become critical when the monitor passes control to the user program through a Go or Trace1 command. As part of the Go and Trace1 commands, the monitor checks for valid SWI, SCI1, and ICG interrupt vectors. If the vectors are erased, this routine attempts to correct them (reprogram them). If the vectors have been erased and reprogrammed with incorrect information, the monitor would not be able to regain control after exiting to the user program. Consequently, the Go or Trace1 command is not executed and the prompt sequence reflects the error.

The user routinely erases the unprotected FLASH memory in order to program new application programs into it. This operation also erases the monitor's SWI, SCI1, and ICG vectors. When programming the application code into the FLASH, leave SWI, SCI1_Rx, SCI1_Tx, SCI1_Error, and optionally the ICG vectors unprogrammed ($FFFF). The first time you try to do a Go or Trace1 command, the monitor vectors will get programmed automatically to the correct values.

If a third-party development tool wants to force these monitor locations to be reprogrammed after an Erase_All command, it can set the user PC value to $FC00 and execute a Trace1 command. The instruction at $FC00 is a jump instruction to the start of the Prog1flash utility subroutine. The only negative consequence of doing this is that the option to use a more sophisticated ISR for the ICG would not be available.

The routine that checks for valid monitor vectors compares the vectors for SCI1_Rx, SCI1_Tx, SCI1_Error, and SWI against the monitor values stored at

$FFF4–$FFFD. In the special case of the ICG vector, this verification routine only checks to make sure the ICG vector is not blank. If a user has already programmed a vector for a user-supplied ICG ISR, this routine accepts that vector. If the ICG vector location is unprogrammed ($FFFF), the verification routine programs a default vector to a simple ISR in the monitor.

Stack Usage Details

Worst case stack usage by the monitor determines how much extra space a user must allow below the application stack to support debugging with the monitor. Typically, the functions for programming and erasing nonvolatile memory are done before attempting to do any debugging on user programs. Because of this, the amount of stack needed for these commands is generally not important. The monitor was written to try to minimize the amount of stack needed to support debugging.

During reset initialization, the stack pointer is set to the end of RAM ($107F in the MC9S08GB60) and a 6-byte user register stack frame is set up with all 0s except the user CCR which is initially set to $68 (I bit = 1) and the user PC which is initially loaded with the user's reset vector from $FBFE:FBFF. An additional two bytes of stack are used while printing the prompt sequence. This is a worst case stack depth of eight bytes before processing any commands. While waiting for new commands, the stack is eight bytes deep — six bytes for the user register stack frame and two bytes for the JSR to the GetChar routine.

When nonvolatile memory is not being programmed, monitor commands use up to 17 bytes of stack space including the 6 bytes used for the user register stack frame. If a Write_Byte or Write_Next command is used to program a FLASH byte, 45 bytes of stack are used including the 6 bytes for the user register stack frame. If a Write_Block command is used to program a FLASH byte, 47 bytes of stack are used including the 6 bytes for the user register stack frame.

The reset command does not use any extra stack except the six bytes for the user register stack frame and the two bytes for JSR GetChar. The erase commands use 50 bytes including the 6 bytes for the user register stack frame.

The bottom line for worst case stack usage is that an application should allot 50 bytes of extra stack space in their application programs to allow for debugging with the monitor program. If no nonvolatile memory needs to be programmed, only 17 bytes of extra stack space are needed. Failure to allow for this extra space could result in the monitor overwriting other user resources such as RAM variables or registers. For applications that cannot tolerate this extra stack space, use a BDM-based debug pod which does not need any user memory or I/O resources.

The monitor checks the value of SP in the SWI service routine to make sure it can support normal monitor activity. If SP is less than RamStart+$45 or greater than RamLast, the monitor forces a warm reset to get the stack pointer back into a legal range. If SP is outside this range, the stacking operation for the SWI could have written over other system resources including program variables or control and status registers.The lower limit (RamStart+$45) could have been set at RamStart+50, but some guard band was allowed in case changes are made to the monitor program some time in the future.

Tricks to Save Code Space

This section describes several techniques that were used to reduce code space. Some of these techniques are good programming practices while others should only be used as a last resort. This 1-Kbyte monitor is compact and is usually used without modification so a few unusual tricks were used to make the code fit within a 1-Kbyte protected block. Care was taken to document these techniques to avoid future problems when this program is modified for other HCS08 derivatives.

Utility Subroutines

One of the most common ways to reduce code space is to develop a good set of utility subroutines. A good utility subroutine is one that can be used in several different contexts to perform some common task. A few obvious choices are GetChar and PutChar routines to receive and send characters through the SCI serial interface. Other utility routines that were used in this bootloader are PCrLf, PrintCrLf, PrintMsg, and WriteA2HX.

Partition Common Blocks of Code into Subroutines

This technique is similar to the idea of making utility subroutines except that these blocks of code are not as general purpose as something like the GetChar routine. The usual way these sequences are detected is that a programmer will notice they are doing something very similar in two or more places in a program. When this happens, they can study the code and try to make a subroutine out of the common parts of the sequence.

One example of this technique is the preInc subroutine which is used in the Write_Next and Read_Next commands. In both of these commands, the user's H:X register must be retrieved from the user register stack frame, get incremented by one, and be updated in the user register stack frame. The incremented H:X value must be in the H:X register to perform the requested read or write operation. The common subroutine saved 10 bytes of code space compared to doing a 5-line in-line sequence twice.

Other examples of this technique in the monitor include DoOnStack, and WriteA2HX. These routines are discussed in more detail elsewhere in this application note.

WriteA2HX Subroutine

The WriteA2HX subroutine saves code space similar to the way other subroutines save space, but there is another, more important result from combining this function into a subroutine. The Write_Byte command, the Write_Next command, and the Write_Block command all change the contents of memory locations. By building this function into a subroutine, it was possible to make the operation smart so that it could use the address to intelligently decide whether to program FLASH memory or simply write the data to the requested RAM or register location. The routine also performs error checks to detect improper attempts to program nonvolatile locations.

The resulting routine ensures that FLASH programming will be performed in exactly the same way no matter which monitor command is responsible for the change. This helps improve code reliability and reduces the amount of testing for the final program. If a change is needed for the nonvolatile programming algorithm, it can be changed in this one routine rather than having to locate three different places to correct the program.

User-Accessible Utility Subroutines

This monitor includes two user-accessible utility subroutines that can be called through a jump table. The jump table is located in the first 6 bytes of the 1-Kbyte protected monitor memory. The jump table provides a way to call these routines at a predictable address that doesn't change even if the monitor program is changed so that the actual subroutines are located at different addresses. For example, when a user program executes a JSR to uPrg1Flash (at $FC00), the JMP instruction at $FC00 passes control to the actual Prg1Flash subroutine wherever it happens to be located in the 1-Kbyte monitor program.

The two utility routines are:

uPrg1Flash($FC00) — Writes the value in accumulator A to the address pointed to by H:X. If the Z condition code is set (.EQ.) after returning from this subroutine, it indicates the operation was successful. If Z is clear (.NE.), it indicates an FACCERR or FPVIOL error was detected during an attempted FLASH programming operation, or the FLASH location that was being written had some bits already programmed before the programming operation was attempted.

uErasePages ($FC03) — Erases one or more 512-byte pages of FLASH memory. Before calling this subroutine, push the end address onto the stack and load H:X with the start address for the range of locations you want to erase.

If the operation is successful, the Z bit is set on return from this subroutine. If the range of addresses is not within the unprotected portion of the FLASH, or if there is an FACCERR or FPVIOL error during the erase operation, the Z bit will be cleared on return.

For additional information about the detailed operation of the utility subroutines, refer to the complete listing for the monitor program.

Modifications for Clock Speed and Memory Map Variations

This section describes the changes that would be needed to adapt this monitor to another MC9S08GB60 system that uses a different frequency source such as a 4-MHz crystal. For other HCS08 devices, refer to the specific data sheet for detail about where to make similar changes. There are several frequency-related dividers that would need to be adjusted. In the most extreme case, this monitor may be adapted to some new HCS08 derivative that has a different type of clock generator module. In the case of a different clock generator, you would need to change the initialization code in lines 402 through 408. Most other changes can be made by simply adjusting some of the initialization constants at the top of the program.

Setting Bus Frequency

The first change for a different frequency source would be to adjust the settings in initICGC1 and initICGC2. Refer to the device data sheet for detailed information about how these registers should be set up.

ICGC1 includes settings for:

- RANGE selects the crystal frequency range (for example, low range for 32-kHz crystals or internal self-clocked frequency source and high range for a 4-MHz crystal)
- Setting REFS to 1 enables the crystal oscillator amplifier if a crystal will be used; REFS can be set to 0 if the internal self-clocked reference will be used.
- The CLKS1:CLKS0 bits select the FLL operating mode which should be FEE for a crystal or FEI for the internal self-clocked oscillator. If the internal self-clocked frequency source is used, you should also adjust the trim in the ICGTRM register to establish a relatively accurate 243-kHz frequency source.

ICGC2 is used to set the FLL dividers to multiply the source frequency to get the ICGOUT frequency, which is 2x the bus frequency. The MFD control field sets a multiplier factor N. The RFD field sets a post FLL divider. When the internal self-clocked frequency source is used, the source frequency for these calculations is (243 kHz / 7). The overall relationship between source frequency and bus frequency is:

- For a low frequency crystal source such as 32.768 kHz (RANGE = 0, REFS = 1, CLKS = 1:1 in ICGC1)
  For this monitor set ICGC1 = %00111000

  Source_Frequency * 64 * (N / 2R) = Bus_Frequency

  ex. 32.768 kHz * 64 * (18 / 2) = 18.874368 MHz where...
  N = 18 (MFD = 1:1:1) and R = 1 (RFD = 0:0:0)
  For this monitor set ICGC2 = %01110000

- For the internal self-clocked frequency source (RANGE = REFS = 0, CLKS = 0:1 in ICGC1)
  For this monitor set ICGC1 = %00001000

  Source_Frequency * 64 * (N / 2R) = Bus_Frequency

  ex. (243 kHz / 7) * 64 * (18 / 2) = 19.995427 MHz where...
  N = 18 (MFD = 1:1:1) and R = 1 (RFD = 0:0:0)
  For this monitor set ICGC2 = %01110000

- For a high frequency crystal source such as 4 MHz (RANGE = 1, REFS = 1, CLKS = 1:1 in ICGC1)
  For this monitor set ICGC1 = %01111000

  Source_Frequency * 1 * (N / 2R) = Bus_Frequency

  ex. 4 MHz * 1 * (10 / 2) = 20 MHz where...
  N = 10 (MFD = 0:1:1) and R = 1 (RFD = 0:0:0)
  For this monitor set ICGC2 = %00110000

Setting Up SCI Baud Rate

SCI baud rates are derived by dividing the bus frequency, so if the bus frequency changes, the divider for the baud rate generator must change. This monitor uses 115.2 kbaud (for the MC9S08GB60). Some other variations of this monitor for HCS08 devices that operate from a slower clock source may use slower baud rates. For reliable SCI communications, the baud rate must be within about ± 4.5% of the ideal rate. A modulo divider is controlled by the SBR (SBR12..SBR0) setting in the SCI1BDH and SCI1BDL registers. Since this program does not use any divisor that is greater than 8 bits, the SCI1BDH register can be left at its default $00 value and all baud rate adjustment will be done using the SCI1BDL register. When SBR = 0, the baud rate generator is off.

The formula for computing baud rate from bus frequency is

Baud_Rate = Bus_Frequency / (SBR * 16)

The following table summarizes the bus frequency, SBR settings, and frequency tolerance for 9600 baud and 115.2 kbaud for this monitor assuming a 32.768 kHz crystal, a 4 MHz crystal, or the internal self-clocked frequency source. Bus frequency values assume the FLL settings shown in **Setting Bus Frequency**.

Table 1. Baud Rate Setup

Frequency Source	Bus Frequency (MHz)	SBR	Baud Rate (Baud)	Frequency Tolerance
32.768 kHz	18.874368	123	9600	–0.1%
32.768 kHz	18.874368	10	115.2 k	+2.4%
4 MHz	20	130	9600	+1.6%
4 MHz	20	11	115.2 k	+1.36%
(internal)	19.995427	129	9600	+0.9%
(internal)	19.995427	11	115.2 k	+1.4%

Setting longBreak Constant

This monitor sends a break to the host PC to get its attention when the target system is powered up. Because some Windows-based PCs need a long period of break to detect the condition, this monitor sends a break that is about 30 bit-times at the selected baud rate. This time delay is not critical, but it should be at least 30 bit-times.

The monitor generates this break delay time by executing a software loop that is eight bus cycles long. The number of times this loop is executed is set by the

value in longBreak. One bit-time is equal to 16 times the baud rate constant. To get the number of loops to execute for 32 bit-times (for example), you would take (the baud rate constant) * 16 * 32 / 8. To compute the value for longBreak to get a 30 to 32 bit-time break, multiply the baud rate constant first by 60 and then 64, and then choose a convenient number between these two values.

For a 32.768 kHz-based system with bus speed 18.874368 MHz, longBreak is set to 625. For a 4 MHz-based system with bus speed 20 MHz, longBreak is set to 700. In a system that uses the internal frequency source and a bus speed of 19.995427 MHz, set longBreak to 700.

FLASH System Clock Speed

The FLASH memory system uses an internal state machine to execute programming and erase commands. The timing of these commands is determined by the speed of a clock in the FLASH module and this clock must be between 150 kHz and 200 kHz for proper operation. The FLASH clock (FCLK) speed is

Bus_Frequency / (8 * (FCDIV + 1)) = FCLK

Table 2. FCDIV Settings

Frequency Source	Bus Frequency	FCDIV	FCLK
32.768 kHz	18.874368 MHz	11	196.608 kHz
4 MHz	20 MHz	12	192.308 kHz
(internal)	19.995427 MHz	12	192.264 kHz

Memory Map Changes

Most changes to the memory map from one HCS08 derivative to another will be taken into account automatically by replacing the 9S08GB60_v1.equ file with the equate file for the appropriate derivative MCU. This file specifies the start and end address locations for the RAM and FLASH memory and for all status and control registers. You may wish to change the port pins for the switch that forces monitor versus user mode (PTA7 in the initial version for the MC9S08GB60), and for the SCI RxD pin. These changes can be made by modifying the equate directives at the beginning of the monitor program.

Conclusion

This application note has described a 1-Kbyte serial monitor program. In addition to functioning as a debug monitor program, the serial monitor program serves as a good programming example for the HCS08 Family of microcontrollers and demonstrates a number of useful programming techniques.

Routines for erasing and programming nonvolatile FLASH memory are described in detail. The unusual DoOnStack subroutine copies a routine onto the stack and executes it there since it is not possible to program or erase the FLASH memory from code executing within the same FLASH. This routine can easily be adapted for use in other user programs. The WriteA2HX subroutine decides whether to use FLASH or simple RAM write operations, depending on the address of the location to be programmed.

The reset initialization routines show how to setup the FLL and SCI subsystems. A technique for differentiating between a warm reset compared to a cold reset is also described.

A set of 19 primitive monitor commands support third-party FLASH programming and debug tools. This monitor allows for debugging through an on-chip SCI serial interface instead of using a more expensive background debug interface pod.

Vector redirection through a new hardware mechanism is described, and techniques to reduce code size are discussed.

Code Listing

Two zip files accompany this application note, AN2140SW1.zip and AN2140SW2.zip. AN2140SW1.zip contains just the assembly files for both the 32-kHz crystal and 4-MHz crystal versions of the serial monitor program along with the necessary equate file. This zip file does not contain compiled versions of the programs.

AN2140SW2.zip contains two complete project folders, one for each version of the serial monitor program. These project directories are for use with MetroWerks CodeWarrior for CW08_V3.0. The folders are named “32k_9S08GB60_Monitor” and “4m_9S08GB60_Monitor.” Inside the first level of each project folder is a CodeWarrior project file with a “.mcp” filename extension. Double clicking these files will open the project if CodeWarrior has been installed. Each project has been assembled and listings (“.lst” file extension) are available in the “bin” subfolders. Also, the s records (“.s19” file extension) are available in the same folder.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2140/D
Rev. 1
6/2003