MOTOROLA
intelligence everywhere™

digitaldna™

By   Jim Williams
      8/16 Bit Applications Engineering
      Austin, Texas

Introduction

This application note describes a 2-Kbyte monitor program for the HCS12 series MCU. This program supports 23 primitive debug commands that allow FLASH/EEPROM programming and debugging through an RS-232 serial interface to a personal computer. The monitor supports primitive commands to reset the target MCU, read, or modify memory (including FLASH/EEPROM memory), read or modify CPU registers, GO, HALT, or TRACE single instructions. To allow a user to specify the address of each interrupt service routine, this monitor redirects interrupt vectors to an unprotected portion of FLASH just before the protected monitor program. This monitor is intended to be generic, so this single application (with very slight modification) should execute on any HCS12 derivative.

Using this monitor program provides users with a cost-effective method to evaluate their MCUs by writing programs, programming them into the MCU, then debugging their applications for the HCS12 MCU — using only a serial I/O cable and free software on their personal computer. The monitor does not use any RAM other than the stack itself. Because the monitor uses the COP watchdog for a cold reset function, user code should not enable or disable the COP function (by writing to COPCTL). This development environment assumes that the user resets to the monitor when performing debug operations. If user code takes control directly from reset, and then an SCI0 interrupt or a SWI attempts to enter the monitor, the monitor may not function because SCI0, the phase-locked loop (PLL), and memory initialization registers may not be initialized as they would be for a cold reset into the monitor.

There is no error handling for the PLL. If the frequency source is missing or broken, the monitor will not function. Through the use of equate definitions, the monitor sets the operating speed of the MCU to 24 MHz. User modification of

the MCU speed without consideration for the monitor program will render the monitor nonfunctional. If the PLL loses lock during operation, the monitor will fail.

NOTE: *With the exception of mask set errata documents, if any other Motorola document contains information that conflicts with the information in the device guide, the device guide should be considered to have the most current and correct data.*

Block Protection

To prevent accidental changes to the monitor program itself, the 2-Kbyte block of FLASH memory residing at $F800–$FFFF is block protected. The only way to change the contents of this protected block is to use a BDM-based development tool.

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.

Because the security and backdoor keys are stored in the protected block, these functions are disabled. Though it is possible to edit the source code to enable these HCS12 features and the monitor will still function with security enabled, it is recommended that a BDM pod be used for development of secure systems.

Oscillator Configuration

The monitor as written creates an operating frequency of 24 MHz. If crystal frequencies other than 4, 8, or 16 MHz are used, the user is responsible for creating definitions for these crystals and verifying that the SCI communication rate is within the ±4% tolerance allowed by the RS-232 standard. The oscillator definitions are stored in the S12SerMonxrx.def file.

COP Configuration

The monitor as written creates a hard reset function by using the COP watchdog timer. It does so by enabling the COP and waiting for a COP timeout reset to occur.

If the user application uses the COP, two issues must be considered:

- If the COP is disabled in the user application, the monitor will be unable to perform a hard reset and will soft reset to the start of the monitor instead.

- The monitor does not service the COP timer. If the user application implements COP timer servicing, upon re-entry into the monitor, a hard reset is likely to occur.

Memory Configuration

The monitor as written is intended to work with all HCS12 Family members available at the time of publication. Adding a new device requires creating a few memory definitions for the new device. The memory definition is stored in the S12SerMon<u>x</u>r<u>x</u>.def file. EB386/D should be studied for complete understanding of the theory behind the memory configuration chosen. A short description is given below:

- Register space is located at $0000–$03FF.
- FLASH memory is any address greater than $4000. All paged addresses are assumed to be FLASH memory.
- RAM ends at $3FFF and builds down to the limit of the device's available RAM.
- EEPROM (if the target device has any) is limited to the available space between the registers and the RAM ($0400–to start of RAM).
- External devices attached to the multiplexed external bus interface are not supported.

Serial Port Usage

For the serial port function of the monitor to work, the SCI0 serial interface is used. The monitor must have exclusive use of this interface. User application code should not implement communications on this serial channel.

This monitor accommodates RS-232 serial communications through SCI0 at 115.2 kbaud. For systems requiring the use of SCI0, use a BDM development system that allows more sophisticated debugging. Alternatively, this monitor program may be modified by the user to check the state of the switches used to enter the monitor and adjust vector relocation accordingly. This modification would enable the user to use the monitor SCI port, although debugging of SCI0 routines would be very difficult or impossible using the monitor.

During initialization after any cold reset, a long break is transmitted before any other SCI communication takes place. This break is about 30 bit-times to

ensure that a Windows®-based PC can recognize this as a break. 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, 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 must verify 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. The prompt sequence is detailed more completely in following sections.

Defining AllowSci0 in the S12SerMonxrx.def file enables the monitor to release the SCI0 port back to the user code in the system as a run/load switch function. In load mode, the monitor will have full access of the SCI0 port. In RUN mode, the monitor will check for a valid user SCI0 interrupt vector and jump to this vector if it is valid.

Vector Redirection and Interrupt Use

Access to the user vectors is accomplished via a jump table located within the monitor memory space. This jump table points all interrupt sources to a duplicate vector table located just below the monitor ($F780–$F7FE). The monitor will automatically redirect vector programming operations to these user vectors. The user code should therefore keep the normal (non-monitor) vector locations ($FF80–$FFFE). The monitor also checks interrupts as they occur, and it will re-enter monitor mode if execution of an interrupt with an unprogrammed vector is attempted. If this occurs, the **$E3 — Stack Pointer Out of Range** error will be returned. The user is strongly encouraged to implement a software response for all vectors (which is a good programming practice).

The monitor depends on interrupts being available for monitor re-entry after GO or TRACE commands. Therefore, the user application must normally have interrupts enabled. If user code blocks interrupts such as during interrupt service routines, the serial monitor cannot execute memory access or HALT commands until interrupts are enabled again.

Run/Load Switch

It is recommended that systems using the serial monitor incorporate a run/load switch attached to a general-purpose I/O (GPIO) pin. Definitions for GPIO uses are located in the S12SerMonxrx.def file. They are:

- SwPort — defines which port the switch is connected to
- Switch — defines which bit of the above port the switch is connected to
- SwPullup — defines which register controls the pullup for the switch
- mSwPullup — defines the switch bit position in the pullup control register

Monitor Commands

The monitor recognizes 23 primitive binary commands that enable a third-party development tool vendor to create a full-featured debug program. These commands use 8-bit command codes optionally followed by binary address, 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 in **Table 1**.

Table 1. Abbreviations Used in Command Sequences

Abbreviation	Definition
AAAA	A 16-bit address
CCR	The contents of the 8-bit condition codes register
D	The contents of the 16-bit D register, consisting of the A and B accumulators
EADR	The 16-bit end address for an erase or block command
IDID	The 2-byte device ID from PARTID register
IX	The contents of the 16-bit X index register
IY	The contents of the 16-bit Y index register

Table 1. Abbreviations Used in Command Sequences (Continued)

Abbreviation	Definition
NN	The number of bytes (−1) for block read and write commands
PC	The contents of the 16-bit user program counter
RD	One byte of read data
RDW	One word of read data
$RDB_{(AAAA)}$ / $RDB_{(AAAA+1)}$ /.../ $RDB_{(AAAA+NN)}$	A series of 8-bit read data values from address locations AAAA through AAAA+NN
SADR	The 16-bit start address for an erase or block command
SP	The 16-bits of the user stack pointer (SP)
WD	One byte of write data
WW	One word of write data
$WB_{(AAAA)}$ / $WB_{(AAAA+1)}$ /.../ $WB_{(AAAA+NN)}$	A series of 8-bit write data values for address locations AAAA through AAAA+NN

Some monitor commands such as Read_Byte and Write_Byte can be executed at any time while others may be executed only while the monitor is active and waiting for commands (as opposed to while running user code). Executing commands such as those that write to CPU registers 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 before issuing a new prompt.

$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. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page.

$A2 — Write_Byte   A2/AAAA/WD — Writes the supplied byte of data to the specified 16-bit address. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. All writes are processed through an intelligent routine that programs FLASH/EEPROM 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 details.

$A3 — Read_Word   A3/AAAA/RDW — Reads a word of data from the specified 16-bit address and sends the 16-bit data back to the host PC. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page.

$A4 — Write_Word   A4/AAAA/WW — Writes the supplied word of data to the specified 16-bit address. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. All writes are processed through an intelligent routine that programs FLASH/EEPROM 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 details.

$A5 — Read_Next   A5/RDW — Pre-increments the user IX register (by 2), reads the word of data from the address pointed to by IX, and sends the 16-bit data back to the host PC. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. This command is not allowed when the user program is running. If executed during run mode, this command will return an **$E2 — Command Not Allowed in Run Mode** error and $0000 data.

$A6 — Write_Next   A6/WW — Pre-increments the user IX register (by 2) and writes the supplied word to the address pointed to by IX. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. All writes are processed through an intelligent routine that programs FLASH/EEPROM or writes to RAM or registers based on the address being written. If any error occurs during an attempt to program a nonvolatile memory location, an error code is transmitted before a new prompt is issued. See **Intelligent Writes** for details. This command is not allowed when the user program is running. If executed during run mode, this command will return an **$E2 — Command Not Allowed in Run Mode** error and the write data will be ignored.

$A7 — Read_Block   A7/AAAA/NN/RDB$_{(AAAA)}$ /RDB$_{(AAAA+1)}$ /.../RDB$_{(AAAA+NN)}$ — 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. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. Although this command can be executed while a user program is running, it is not recommended because it could slow down operation of the user program.

$A8 — Write_Block <u>A8</u>/<u>AAAA</u>/NN/<u>WB</u>$_{(AAAA)}$ /<u>WB</u>$_{(AAAA+1)}$ /.../<u>WB</u>$_{(AAAA+NN)}$ — Writes a series of NN+1 (1 to 256) bytes of data into the target MCU memory starting at address AAAA and ending with address AAAA+NN. This routine assumes that accesses to the paged memory area have been preceded by a PPAGE register access (Write_Byte 0030 = page) to select the appropriate page. All writes are processed through an intelligent routine that programs FLASH/EEPROM 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 details. Although this command can be executed while a user program is running, it is not recommended because it could slow down operation of the user program.

$A9 — Read_Regs <u>A9</u>/SP/PC/IY/IX/D/CCR — Sends the current contents of user registers SP, PC, IY, IX, D, and CCR (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 9 less due to the user register stack frame. Although this command can be executed while a user program is running, it is not recommended because these register values change much more quickly than they could be read.

$AA — Write_SP <u>AA</u>/<u>SP</u> — Adjusts the specified 16-bit data to compensate for the user register stack frame (–9) and writes this adjusted value to the stack pointer register. The monitor uses stack space below the user register stack frame. When a GO or TRACE1 command is executed, the monitor exits to the user program by executing an RTI instruction. The monitor doesn't move the user register values to the new user register stack frame, so the host should rewrite 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.

$AB — Write_PC <u>AB</u>/<u>PC</u> — 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.

$AC — Write_IY <u>AC</u>/<u>IY</u> — Writes the specified 16-bit data to the Y index register in the user register stack frame. This command is not allowed when the user program is running.

$AD — Write_IX AD/IX — Writes the specified 16-bit data to the X index register in the user register stack frame. This command is not allowed when the user program is running.

$AE — Write_D   AE/D — Writes the specified 16-bit data to accumulator A and accumulator B in the user register stack frame. This command is not allowed when the user program is running.

$AF — Write_CCR   AF/CCR — 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 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 program, you can first use a Write_PC command to set the user program counter to a new location. In this mode, the user application program will execute until it is interrupted by a breakpoint, an SCI0 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 and waiting for further commands from the host. In most cases of an SCI0 interrupt, the run flag is set (or remains set) to indicate that the monitor should return to the user application program after completing the current command. The exception is the case of the SCI0 interrupt due to the HALT command. In this special case, the run flag gets cleared.

$B2 — Trace1   B2 — The monitor sets up the on-chip breakpoint or debug module to force a CPU breakpoint immediately after executing a single instruction in the user program. It then executes an RTI to copy user CPU register values from the 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 is in the user register stack frame. After executing a single user instruction, an SWI is forced, which causes 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 and waiting for additional commands from the host.

$B3 — Halt   B3 — This command is used to force the user application program to stop executing and the monitor to gain control and remain active and waiting for additional commands from the host. This command requires an enabled SCI0 interrupt, so it can only be recognized if the user application program has cleared the I bit in the CCR (executed the CLI instruction). 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 — When a user reset vector is programmed, the levels on the run/load switch and the RxD0 line could cause a reset to either the user code or the monitor. The sequence of checks to determine the type of reset is listed and illustrated in **Figure 1**.

1. **If the first byte of the user reset pseudo-vector = $FF (unprogrammed), force monitor reset.**
2. **If the run/load switch = 0 (logic low), force monitor reset.**
3. **If RxD0 = 0 (logic low), force monitor reset.**
4. **If none of the above, use the reset pseudo-vector to jump to the user reset start-up routine.**

Warm start skips the long break output to SCI0. The warm reset is needed in the case of an SCI0 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.

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 INITRG, INITRM, INITEE registers are configured to promote compatibility as described in EB386/D. The single write available to these registers is used so that a single monitor program will support all HCS12 Family devices.
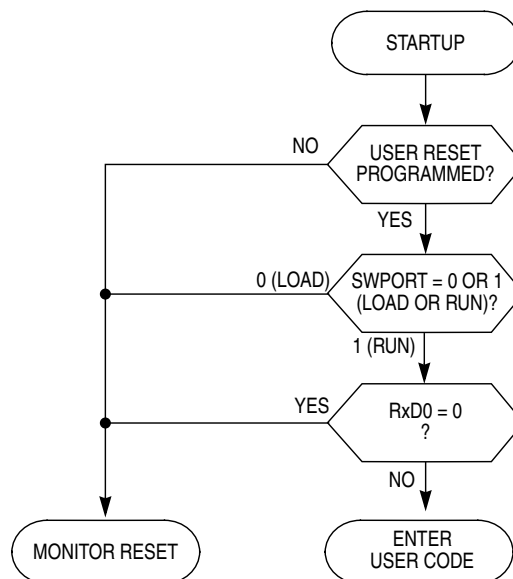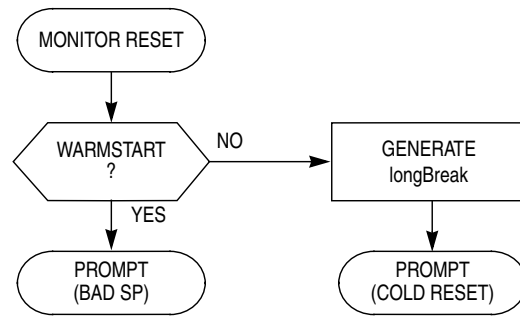

Figure 1. Determining Cause of Reset

Figure 2. Determining Cause of Monitor Reset

$B5   Command not supported. Range erasing is left to the user application.

$B6 — Erase_ALL   B6 — Erase all of FLASH and EEPROM memory, preserving only the 2K monitor area. The $E6 error code will be returned if the command does not complete successfully.

$B7 — Device_info   B7/DC/IDID — Returns the constant $DC (<u>D</u>evice <u>C</u>=12) and the 2-byte HCS12 device ID register. Please refer to selected device guides for device ID register contents.

$B8 — Erase_Page   B8 — Erase one page of FLASH memory selected by the current PPAGE register. $E6 error code will be returned if the command does not complete successfully. PPAGE must be preloaded with desired page to erase.

$B9 — Erase_EEPROM   B9 — Erase all EEPROM memory. The $E9 error code will be returned if the command does not complete successfully.

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 application program. The prompt 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 $00 indicating 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, $00, $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 **$E2 — Command Not Allowed in Run Mode**). Other commands are not recommended while the target MCU is in run mode. However, because these commands in most cases do not interfere with proper execution of the application program, other than slowing it down, they are allowed to execute and do not result in an error.

$E0 — No Error

This code is used after cold reset initialization and 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 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 not allowed in run mode are:

- Read_Next and Write_Next — Not allowed because these commands use the user IX register value as a pointer and this value may change during run mode.
- Write_SP, Write_PC, Write_IY, Write_IX, Write_D, and Write_CCR — Not allowed in run mode because these registers change much faster than the host checks their contents. 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.

$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 below 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. This error code will also be returned if an interrupt occurs with a non-programmed user vector ($FFFF).

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. After the warm reset, the PLL and control registers are initialized and the prompt indicates there was a bad SP error message ($E3, $00, $3E).

$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 9) 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 details.

$E5 — Byte Write Access to Nonvolatile Memory

This code indicates that a byte write access to a FLASH or EEPROM location was attempted. Byte writing to the nonvolatile memory of the HCS12 devices is not permitted.

$E6 — FACCERR or FPVIOL FLASH Error

This code indicates that an access error or protection violation error occurred during an attempt to write or erase FLASH/EEPROM memory. In cases where a single word was being programmed or a single page was being erased, the attempted FLASH/EEPROM operation might not have been 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 flagged with an error. 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. This error code will also be returned if the debug tool attempts to program over the protected monitor space ($F800–$FFFF).

$E7 and $E8

Error codes not implemented

$E9 — EACCERR or EPVIOL EEPROM Error

This code indicates that an access error or protection violation error occurred during an attempt to write or erase EEPROM memory. In cases where a single word was being programmed or a single page was being erased, the attempted EEPROM operation might not have been 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 flagged with an error. 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.

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 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 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 SCI0 receive interrupt to gain the attention of the monitor program.

$02 — User Program Halted

This code indicates that the user application program has been halted by the host. Control has been returned to the monitor.

$04 — TRACE1 Command Returned from User Program

This code indicates that a TRACE1 (single step) command has been executed successfully and the monitor is now active and waiting for a new command.

$08 — Cold Reset Executed

This code indicates that a complete device reset and monitor restart has occurred.

$0C — Warm Reset Executed

This code indicates that the monitor has detected a fatal error and has restarted.

Software Detail

To create applications software that will run both with the monitor program and without, several simple modifications must be made to the environment:

1. Memory must be configured per EB386/D.

For assembly programs, the following lines should be added to the start of the source code and executed:

```
movb  #$00,INITRG    ;set registers at $0000
movb  #$39,INITRM    ;set ram to end at $3fff
movb  #$09,INITEE    ;set eeprom to end at $0fff
```

For C programs, the following lines should be added to the Start12.c file at the beginning of the _Startup() function, before any calls or jumps occur:

```
/* for Monitor based software remap the RAM & EEPROM to adhere
   to EB386. Edit RAM and EEPROM sections in PRM file to match these. */

  *((unsigned char *) 0x0011)=0x00;  /* lock registers block to 0x0000 */
  *((unsigned char *) 0x0012)=0x09;  /* lock EEPROM block to 0x0000 */
  *((unsigned char *) 0x0010)=0x39;  /* lock Ram to end at 0x3FFF */
/* Here user defined code could be inserted, the stack could be used */
```

2. Memory definition should be adjusted to match above configurations.

For assembly programs — All ORG statements in the application should be adjusted, such that they reflect actual memory of their device with the above setting.

For C programs — The projects linker file should be adjusted, such that they reflect actual memory of their device with the above setting. In CodeWarrior, this is the .prm file. Shown below is an example:

```
//INTERNAL_RAM = READ_WRITE 0x0400 TO 0x1FFF; /* Default for E128 */
```

Change to:

```
INTERNAL_RAM = READ_WRITE 0x2000 TO 0x3FFF; /* For Monitor compatibility*/
```

3. Vectors should remain defined at $FF80–$FFFF. The monitor will create a copy of the application's vectors in a jump table from $F780–$F7FF.

4. Application code should exclude the $F780–$FF7F memory.

For assembly programs — In the CodeWarrior assembler, the following definition may be added to generate an automatic error if this occurs:

```
IF EndOfApplication >= $F780
    FAIL "Application code overflows into Monitor area"
  ENDIF
```

The application just needs to define the EndOfApplication label to reference the end of the application code.

For C programs in the CodeWarrior environment, FLASH settings for the upper bank should be modified in the .prm file. Shown below is an example:

```
//ROM_C000 = READ_ONLY  0xC000 TO 0xFF7F; /* upper bank of 16K FLASH */
```

Change to:

```
  ROM_C000 = READ_ONLY  0xC000 TO 0xF77F; /* upper bank of 16K FLASH */
```

Monitor Mode versus Run Mode

At any given time, the target MCU is operating in either monitor mode or run mode. Monitor 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 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 SCI0 interrupt causes the monitor to temporarily gain control to decode and execute the requested command. When the command is completed, the monitor automatically returns control to the user 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 SCI0 interrupt which causes the CPU registers to be pushed onto the stack. The ISR for SCI0 then sets the run mode flag (even though this flag will be cleared again if the command that caused the SCI0 interrupt turns out to be the HALT command).

Breakpoints use the SWI exception (the TRACE1 command also uses this 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 SCI0 interrupt. 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 forgets to clear the I bit or gets stuck in an interrupt service routine, commands through the SCI0 serial link cannot be recognized. If this condition persisted, the user would have to force a reset or cycle the power to the target system so it would get a power-on reset and the monitor could regain control.

Intelligent Writes

The intelligent write routine uses the address in IX register to decide what to do. If the location is in nonvolatile memory, it checks to see if the location is already correct (if so it skips the program operation and signals success). If the nonvolatile memory 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 word program operation and finally checks FACCERR, FPVIOL, EACCERR and EPVIOL to make sure there was no access or protection violation during programming (it is considered an error if there was). If the location is not nonvolatile memory, the routine simply writes the requested data to the specified address. See **WriteD2IX Subroutine** for details.

DoOnStack Subroutine

This unusual subroutine is used to program FLASH locations or perform erase operations in the FLASH memory. You cannot execute a program out of the FLASH array while a program or erase operation is being performed on the same nonvolatile memory array. 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 from the DoOnStack routine. This de-allocates the space used by the small stack routine and then returns to the program where DoOnStack was called.

Prior to calling DoOnStack, the main program started a FLASH operation by writing data to a FLASH address and by writing an erase or word program command code to the FCMD register. These steps can be performed by code executing in the FLASH, but the final step of setting the CBEIF bit in FSTAT must not be executed from within the FLASH. The FLASH is removed from the memory map as soon as this write is executed.

The first line in DoOnStack saves the X index register on the stack. The SpMoveLoop copies the SpSub routine onto the stack (with a series of PSHD instructions) starting with the last word of SpSub and ending with the push of the first word of SpSub onto the stack. At this point, the stack pointer points to the location of the first word of the stacked SpSub routine. The TFR in the next line copies the SP into the IX register so X points to the start of the copy of

SpSub on the stack. The next line preloads A with a mask corresponding to the CBEIF bit which will be used to complete the FLASH command.

At this point if the monitor program is active, the I bit in the CCR is already set to mask interrupts; however, 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. The next two lines save the CCR (and the I bit) for restoration after the JSR calls the SpSub routine.

The JSR ,x instruction 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. SpSub is such a short subroutine that it was easy to make it position independent.

SpSub completes the FLASH command by writing 1 to the CBEIF bit in FSTAT. The series of NOPs in the routine is to used to ensure that the FLASH state machine has registered the command before polling begins. This delay is required so the internal FLASH command sequencer can properly update the CBEIF and CCIF flags in FSTAT. Execution stays in the ChkDone loop until the command finishes (CCIF becomes set). At this point, the FLASH is back in the memory map and we can return to DoOnStack (which is in FLASH). The RTS in the SpSub returns to the DoOnStack routine.

NOTE: *Due to the RTS (in lieu of an RTC instruction), any user calls to this routine from banked memory are prohibited. This means that user access to this routine must be called from the non-banked areas of FLASH ($4000 or $C000 areas).*

The CCR can be saved if needed, but not the interrupt state. CCR should be restored (as well as the interrupt state), and the stack used for the SpSub is deallocated. Next the FSTAT register is read and masked for the FACCERR and FPVIOL violations are recorded to be returned to the calling program. The IX register is then restored with the status of the routine preserved so that a simple BEQ or BNE can be used to check for errors after returning to the main program.

Vector Redirection   This monitor uses a software pseudo-vector mechanism that is traditionally used in ROM monitors. Because the monitor resides at $F800–$FFFF, and this 2-Kbyte block is protected, when an interrupt occurs, the vector is fetched from $FF80–$FFFE as normal, but is redirected to the pseudo-vector table located at $F780–$F7FE. This vector redirection mechanism places the pseudo-vector interrupt vectors in unprotected space so the user can control the contents.

Because the real vectors are in protected space, it is not possible for a user to unintentionally erase the real vectors.

After power-on, reset, or while the monitor is active, due to ISR or startup entry the I bit in the CCR is set. This 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.

The monitor also redirects programming commands for the $FF80–$FFFE area to the $F780–$F7FF pseudo vector area. This allows programs that are developed with the serial monitor program to execute normally if loaded stand alone.

Stack Usage Details
Worst-case stack usage by the monitor determines how much extra space a user needs to allow below their application stack to support debugging with the monitor. With the large RAM space available in the HCS12 Family, this is not as important as with small MCUs with less RAM. Typically, the bootloader 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+1 ($4000).

• A 9-byte user register stack frame is set up with all zeros except the user CCR which is initially set to $D0 (SX-IN---).

• The I bit is set, which disables all maskable interrupts.

• The user PC is initially loaded with the user reset vector at $F7FE:$F7FF assuming that the lsb of the vector is not $FF.

Users should allow 35 bytes of extra space in the bottom-most area of the stack for worst-case stack usage. This space will be needed for debugging application programs with the monitor program. Because the smallest HCS12 Family member to date has 2K of RAM available, the 2% overhead is considered to be a minor limitation.

Failure to allow this extra space could result in the monitor overwriting other user resources such as RAM variables or register space. For applications that cannot tolerate this extra stack space, use a BDM-based debug development system that does not use any user memory or resources.

The monitor checks the value of the SP in the SWI service routine to make sure it can support normal monitor activity. If SP is less than RamStart+35 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 write over other system resources including program variables or control and status registers.

User-Accessible 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. The following utility subroutines are provided for the user:

- PutChar
- GetChar
- EraseAllCmd
- DoOnStack
- WriteD2IX

These routines are documented in the source code and previous pages. It is recommended that users wishing to add to this jump table of instructions do so in a manner that does not affect currently implemented commands.

WriteD2IX Subroutine

The WriteD2IX subroutine saves code space similar to the way other subroutines save space, but there is more important benefit from combining this function into a subroutine. The Write_Byte command, the Write_Next command, and the block write command all change the contents of memory locations. By building these functions into a subroutine, it is possible to make the operation use the address to intelligently decide whether to program FLASH memory or simply write the data to the requested RAM or register location. This 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.

Setting Up the 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. The baud rate must be within about ±4.5% of the ideal rate. A modulo divider is controlled by the SCIBR (SBR12..SBR0) setting in the SCIBDH and SCIBDL registers.

The formula for computing baud rate from bus frequency is:

Baud_Rate = Bus_Frequency / (SBR $\times$ 16)

Setting the longBreak Constant

The monitor sends a break to the host PC to get its attention when the target system is powered up. Since some Windows terminal programs (HyperTerminal) need a long period of break to be detected, this monitor sends a break that is about 30 bit-times at the selected baud rate. This time delay is not critical but should be at least 30 bit-times.

The monitor generates this break delay time by executing a software loop that is five 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, you would take the (baud rate constant) $\times 16 \times 32 \div 5$. Compute the value for longBreak to get a 30 to 32 bit-time break.

For a 24-MHz based system, longBreak is set to 1500. If the user application modifies the system speed from 24 MHz, longBreak should be adjusted to compensate.

FLASH/EEPROM 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 speed is set to 200 kHz by the monitor based on the OscFreq setting in the S12SerMonxrx.def file.

Monitor Run/Load Switch

You may wish to change the port pins for the switch that forces monitor versus user mode 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 2-Kbyte serial monitor program. In addition to the use of this program as a debug monitor program, it also serves as a good programming example for the HCS12 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 small 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 array. This routine can easily be adapted for use in other user programs. The WriteD2IX 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 PLL and SCI subsystems. A technique for differentiating between a warm reset compared to a cold reset is also described.

A set of 23 primitive monitor commands has been developed to support third-party FLASH programming and debug tools. This monitor enables debugging through an on-chip SCI serial interface instead of using a more expensive background debug interface development system.

Code Listing

Companion software for this program is available from the Motorola web site, http://motorola.com/sps:

AN2548SW1.zip — Complete Assembly source files

AN2548SW2.zip — Full CodeWarrior project

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors

(M) MOTOROLA

AN2548/D