A Serial Bootloader for Reprogramming the MC9S12DP256 FLASH Memory

By Gordon Doughman
Field Applications Engineer, Software Specialist
Dayton, Ohio

Introduction

The MC9S12DP256 is a member of the M68HC12 Family of 16-bit microcontrollers (MCU) containing 262,144 bytes of bulk or sector erasable, word programmable FLASH memory arranged as four 65,536 byte blocks. Including FLASH memory, rather than EPROM or ROM, on a microcontroller has significant advantages.

For the manufacturer, placing system firmware in FLASH memory provides several benefits. First, firmware development can be extended late into the product development cycle by eliminating masked ROM lead times. Second, when a manufacturer has several products based on the same microcontroller, it can help eliminate inventory problems and lead times associated with ROM-based microcontrollers. Finally, if a severe bug is found in the product’s firmware during the manufacturing process, the in-circuit reprogrammability of FLASH memory prevents the manufacturer from having to scrap any work-in-process.

The ability of FLASH memory to be electrically erased and reprogrammed also provides benefits for the manufacturer’s end customers. The customer’s products can be updated or enhanced with new features and capabilities without having to replace any components or return a product to the factory.
Unlike the M68HC11 Family, the MC9S12DP256 does not have a bootstrap ROM containing firmware to allow initial programming of the FLASH directly through one of the on-chip serial communications interface (SCI) ports. Initial on-chip FLASH programming requires either special test and handling equipment to program the device before it is placed in the target system or a background debug module (BDM) programming tool available from Freescale or a third party vendor.

The MC9S12DP256’s four on-chip FLASH arrays contain two variable size, erase protectable areas as shown in Figure 1. While the majority of the bootloader could be contained in any of the protected areas, the protected high area in the $C000–$FFFF memory range must at least contain reset and interrupt vectors that point to a jump table. In most cases, unless a complex or sophisticated communication protocol is required that will not fit into 16 K, it is easiest to place the entire bootloader into the protected high area of block zero.

Erasing and programming the on-chip FLASH memory of the MC9S12DP256 presents some unique challenges. Even though FLASH block zero has two separate erase protected areas, code cannot be run out of either protected area while the remainder of the block is erased or programmed. While it is possible to run code from one FLASH block while erasing or reprogramming another, adopting such a strategy would complicate the overall implementation of the bootloader. Consequently, during the erase and reprogram process, the code must reside in other on-chip memory or in external memory. In addition, because the reset and interrupt vectors reside in the erase protected area, they cannot be changed. This necessitates a secondary reset/interrupt vector table be placed outside the protected FLASH memory area.

The remainder of this application note explores the requirements of a serial bootloader and the implementation of the programming algorithm for the MC9S12DP256’s FLASH.
Overview of the MC9S12DP256’s FLASH

The MC9S12DP256’s 256 K of on-chip FLASH memory is composed of four 65,536 byte blocks. Each block is arranged as 32,768 16-bit words and may be read as bytes, words, or misaligned words. Access time is one bus cycle for bytes and aligned words reads and two bus cycles for misaligned word reads. Write operations for program and erase operations can be performed only as an aligned word. Each 64-K block is organized in 1024 rows of 32 words. An erase sector contains 8 rows or 512 bytes. Erase operations may be performed on a sector as small as 512 bytes or on the entire 65,536-byte block. An erased word reads $FFFF and a programmed word reads $0000.
The programming voltage required to program and erase the FLASH is generated internally by on-chip charge pumps. Program and erase operations are performed by a command driven interface from the microcontroller using an internal state machine. The completion of a program or erase operation is signaled by the setting of the CCIF flag and may optionally generate an interrupt. All FLASH blocks can be programmed or erased at the same time; however, it is not possible to read from a FLASH block while it is being erased or programmed.

Each 64-K block contains hardware interlocks which protect data from accidental corruption. As shown in Figure 1, the upper 32 K of block zero can be accessed through the 16-Kbyte PPAGE window or at two fixed address 16-K address ranges. One protected area is located in the upper address area of the fixed page address range from $C000–$FFFF and is normally used for bootloader code. Another area is located in the lower portion of the fixed page address range from $4000–$7FFF. Additional protected memory areas are present in the three remaining 64-K FLASH blocks; however, they are only accessible through the 16-K PPAGE window.

FLASH Control Registers

The control and status registers for all four FLASH blocks occupy 16 bytes in the input/output (I/O) register area. To accommodate the four FLASH blocks while occupying a minimum of register address space, the FLASH control register address range is divided into two sections. The first four registers, as shown in Figure 2, apply to all four memory blocks. The remaining 12 bytes of the register space have duplicate sets of registers, one for each FLASH bank. The active register bank is selected by the BKSEL bits in the unbanked FLASH configuration register (FCNFG). Note that only three of the banked registers contain usable status and control bits; the remaining nine registers are reserved for factory testing or are unused.
FLASH Protection

The protected areas of each FLASH block are controlled by four bytes of FLASH memory residing in the fixed page memory area from $FF0A–$FF0D. During the microcontroller reset sequence, each of the four banked FLASH protection registers (FPROT) is loaded from values programmed into these memory locations. As shown in Figure 3, location $FF0A controls protection for block three, $FF0B controls protection for block two, $FF0C controls protection for block one, and $FF0D controls protection for block zero.

The values loaded into each FPROT register determine whether the entire block or just subsections are protected from being accidentally erased or programmed. As mentioned previously, each 64-K block can have two protected areas. One of these areas, known as the lower protected block, grows from the middle of the 64-K block upward. The other, known as the upper protected block, grows from the top of the 64-K block downward. In general, the upper protected area of FLASH block zero is used to hold bootloader code since it contains the reset and interrupt vectors. The lower protected area of block zero and the protected areas of the other FLASH blocks can be used for critical parameters that would not change when program firmware was updated.
The FPOPEN bit in each FPROT register determines whether the entire FLASH block or subsections of it can be programmed or erased. When the FPOPEN bit is erased (1), the remainder of the bits in the register determine the state of protection and the size of each protected block. In its programmed state (0), the entire FLASH block is protected and the state of the remaining bits within the FPROT register is irrelevant.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FF00–$FF07</td>
<td>Security back door comparison key</td>
</tr>
<tr>
<td>$FF08–$FF09</td>
<td>Reserved</td>
</tr>
<tr>
<td>$FF0A</td>
<td>Protection byte for FLASH block 3</td>
</tr>
<tr>
<td>$FF0B</td>
<td>Protection byte for FLASH block 2</td>
</tr>
<tr>
<td>$FF0C</td>
<td>Protection byte for FLASH block 1</td>
</tr>
<tr>
<td>$FF0D</td>
<td>Protection byte for FLASH block 0</td>
</tr>
<tr>
<td>$FF0E</td>
<td>Reserved</td>
</tr>
<tr>
<td>$FF0F</td>
<td>Security byte</td>
</tr>
</tbody>
</table>

**Figure 3. FLASH Protection and Security Memory Locations**

The FPHDIS and FPLDIS bits determine the protection state of the upper and lower areas within each 64-K block respectively. The erased state of these bits allows erasure and programming of the two protected areas and renders the state of the FPHS[1:0] and FPLS[1:0] bits immaterial. When either of these bits is programmed, the FPHS[1:0] and FPLS[1:0] bits determine the size of the upper and lower protected areas. The tables in **Figure 4** summarize the combinations of the FPHS[1:0] and FPLS[1:0] bits and the size of the protected area selected by each.
Figure 4. FLASH Protection Select Bits

The FLASH protection registers are loaded during the reset sequence from address $FF0D for FLASH block 0, $FF0C for FLASH block 1, $FF0B for FLASH block 2 and $FF0A for FLASH block 3. This is indicated by the “F” in the reset row of the register diagram in the MC9S12DP256 data book. This register determines whether a whole block or subsections of a block are protected against accidental program or erase. Each FLASH block can have two protected areas, one starting from relative address $8000 (called lower) toward higher addresses and the other growing downward from $FFFF (called higher). While the later is mainly targeted to hold the bootloader code since it covers the vector space (FLASH 0), the other area may be used to keep critical parameters. Trying to alter any of the protected areas will result in a protect violation error, and bit PVIOL will be set in the FLASH status register FSTAT.

NOTE: A mass or bulk erase of the full 64-Kbyte block is only possible when the FPLDIS and FPHDIS bits are in the erased state.

FLASH Security

The security of a microcontroller’s program and data memories has long been a concern of companies for one main reason. Because of the considerable time and money that is invested in the development of proprietary algorithms and firmware, it is extremely desirable to keep the firmware and associated data from prying eyes. This was an especially difficult problem for earlier M68HC12 Family members as the background debug module (BDM) interface provided easy, uninhibited access to the FLASH and EEPROM contents using a 2-wire connection. Later revisions of the original D Family parts provided a method that
allowed a customer’s firmware to disable the BDM interface (BDM lockout) once the part had been placed in the circuit and programmed. While this prevents the FLASH and EEPROM from being easily accessed in-circuit, it does not prevent a D Family part from being removed from the circuit and placed in expanded mode so the FLASH and EEPROM can be read.

The security features of the MC9S12DP256 have been greatly enhanced. While no security feature can be 100 percent guaranteed to prevent access to an MCU’s internal resources, the MC9S12DP256’s security mechanism makes it extremely difficult to access the FLASH or EEPROM contents. Once the security mechanism has been enabled, access to the FLASH and EEPROM either through the BDM or the expanded bus is inhibited. Gaining access to either of these resources may be accomplished only by erasing the contents of the FLASH and EEPROM or through a built-in back door mechanism. While having a back door mechanism may seem to be a weakness of the security mechanism, the target application must specifically support this feature for it to operate.

Erasing the FLASH or EEPROM can be accomplished using one of two methods. The first method requires resetting the target MCU in special single-chip mode and using the BDM interface. When a secured device is reset in special single-chip mode, a special BDM security ROM becomes active. The program in this small ROM performs a blank check of the FLASH and EEPROM memories. If both memory spaces are erased, the BDM firmware temporarily disables device security, allowing full BDM functionally. However, if the FLASH or EEPROM are not blank, security remains active and only the BDM hardware commands remain functional. In this mode, the BDM commands are restricted to reading and writing the I/O register space. Because all other BDM commands and on-chip resources are disabled, the contents of the FLASH and EEPROM remain protected. This functionality is adequate to manipulate the FLASH and EEPROM control registers to erase their contents.

NOTE: Use of the BDM interface to erase the FLASH and EEPROM memories is not present in the initial mask set (0K36N) of the MC9S12DP256. Great care must be exercised to ensure that the microcontroller is not programmed in a secure state unless the back door mechanism is supported by the target firmware.
The second method requires the microcontroller to be connected to external memory devices and reset in expanded mode where a program can be executed from the external memory to erase the FLASH and EEPROM. This method may be preferred before parts are placed in a target system.

As shown in Figure 5, the security mechanism is controlled by the two least significant bits in the security byte. Because the only unsecured combination is when SEC1 has a value of 1 and SEC0 has a value of 0, the microcontroller will remain secured even after the FLASH and EEPROM are erased, since the erased state of the security byte is $FF.

As previously explained, even though the device is secured after being erased, the part may be reset in special single-chip mode, allowing manipulation of the microcontroller via the BDM interface. However, after erasing the FLASH and EEPROM, the microcontroller can be placed in the unsecured state by programming the security byte with a value of $FE. Note that because the FLASH must be programmed one aligned word at a time and because the security byte resides at an odd address ($FF0F), the word at $FF0E must be programmed with a value of $FFFE.

<table>
<thead>
<tr>
<th>SEC[1:0]</th>
<th>Security State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:0</td>
<td>Secured</td>
</tr>
<tr>
<td>0:1</td>
<td>Secured</td>
</tr>
<tr>
<td>1:0</td>
<td>Unsecured</td>
</tr>
<tr>
<td>1:1</td>
<td>Secured</td>
</tr>
</tbody>
</table>

Figure 5. Security Bits

Utilizing the FLASH Security Back Door

In normal single-chip or normal expanded operating modes, the security mechanism may be temporarily disabled only through the use of the back door key access feature. Because the back door mechanism requires support by the target firmware, it is impossible for the back door mechanism to be used to defeat device security unless the capability is designed into the target application. To disable security, the firmware must have access to the 64-bit value stored in the security back door comparison key located in FLASH memory from $FF00–$FF07. If
operating in single-chip mode, the key would typically be provided to the firmware through one of the on-chip serial ports. In addition, back door security bypass must be enabled by leaving the most significant bit of the Security byte at $FF0F erased. To disable the back door security bypass feature, this bit should be programmed to zero.

Once the application receives the 64-bit key, it must set the KEYACC bit in the FCNFG register. After setting the KEYACC bit, the firmware must write the received 64-bit key to the security back door comparison key memory locations ($FF00–$FF07) as four 16-bit words, in sequential order. Finally, the KEYACC bit must be cleared. If all four 16-bit words written to the comparison key memory area matched the corresponding values stored in FLASH, the MCU will be unsecured by forcing the SEC[1:0] bits in the FSEC register to the unsecured state. Note that this operation only temporarily disables the device security. The next time the MCU is reset, the SEC[1:0] bits will be loaded from the security byte at $FF0F.

FLASH Program and Erase Overview

All FLASH program and erase timings are handled by a hardware state machine, freeing the CPU to perform other tasks during these operations. The timebase for the state machine is derived from the oscillator clock via a programmable down counter. Program and erase operations are accomplished by writing values to the FCMD register. Four commands are recognized in the current implementation and are summarized in Figure 6.

<table>
<thead>
<tr>
<th>Command</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$20</td>
<td>Memory program</td>
<td>Program 1 aligned word, 2 bytes</td>
</tr>
<tr>
<td>$40</td>
<td>Sector erase</td>
<td>Erase a 512-byte sector</td>
</tr>
<tr>
<td>$41</td>
<td>Mass erase</td>
<td>Erase a 64-Kbyte block</td>
</tr>
<tr>
<td>$05</td>
<td>Erase verify</td>
<td>Verify erasure of a 64-Kbyte block</td>
</tr>
<tr>
<td>Other</td>
<td>Illegal</td>
<td>Generate an access error</td>
</tr>
</tbody>
</table>

Figure 6. FLASH Program and Erase Commands
The command register and the associated address and data registers are implemented as a 2-stage first in, first out (FIFO) command buffer. This configuration allows a new command to be issued while the hardware state machine completes the previously issued command. The main reason for this design is to decrease programming time. Without the 2-stage FIFO command buffer, the programming voltage would have to be removed from the FLASH array at the end of each program command to avoid exceeding the high voltage active time, $t_{HV}$, specification. Applying and removing the programming voltage after each program command would double the time required to program an aligned word. If program commands are continuously available to the state machine, it will keep high voltage applied to the array if the program command operates on the same 64-byte row. If the command in the second stage of the FIFO buffer has changed, the address is not within the same 64-byte row or the command buffer is empty, the high voltage will be removed and reapplied with a new command if required.

To aid the development of a multitasking environment where the CPU can perform other tasks while performing program and erase operations, the FLASH module control registers provide the ability to generate interrupts when a command completes or the command buffer is empty. When the command buffers empty interrupt enable (CBEIE) bit is set, an interrupt is generated whenever the command buffers empty interrupt flag (CBEIF) is set. When the command complete interrupt enable (CCIE) bit is set, an interrupt is generated when the command complete interrupt flag (CCIF) is set. Note that the CCIF flag is set at the completion of each command while the CBEIF is set when both stages of the FIFO are empty.

**NOTE:** Because the interrupt vectors are located in FLASH block zero, memory locations in block zero cannot be erased or programmed when utilizing FLASH interrupts in a target application.
FLASH Erasure

As previously discussed, each 64-K block is organized in 1024 rows of 32 words. An erase sector contains 8 rows or 512 bytes. Erase operations may be performed on a sector as small as 512 bytes or on the entire 65,536 byte block. An erased word reads $FFFF and a programmed word reads $0000. Program and erase operations are very similar, differing only in the command written to the FCMD register and the data written to the FLASH memory array. The FLASH state machine erase and verify command operation is depicted in the flowchart of Figure 7.

![Flowchart](image)

**Figure 7. Erase and Verify Flowchart**

Before beginning either an erase or program operation, it is necessary to write a value to the FCLKDIV register. The value written to the FCLKDIV register programs a down counter used to divide the oscillator clock, producing a 150-kHz to 200-kHz clock source used to drive the FLASH memory’s state machine. The most significant bit of the FCLKDIV register, when set, indicates that the register has been...
initialized. If FDIVLD is clear, it indicates that the register has not been written to since the part was last reset. Attempting to erase or program the FLASH without initializing the FCLKDIV register will result in an access error and the command will not be executed.

A combination of the PRDIV8 and FDIV[5:0] bits is used to divide the oscillator clock to the 150-kHz to 200-kHz range required by the FLASH's state machine. The PRDIV8 bit is used to control a 3-bit prescaler. When set, the oscillator clock will be divided by eight before being fed to the 6-bit programmable down counter. Note that if the oscillator clock is greater than 12.8 MHz, the PRDIV8 bit must be set to obtain a proper state machine clock source using the FDIV[5:0] bits. The formulas for determining the proper value for the FDIV[5:0] bits are shown in Figure 8.

```plaintext
if (OSCCLK > 12.8 MHz)
  PRDIV8 = 1
else
  PRDIV8 = 0

if (PRDIV 8 == 1)
  CLK = OSCCLK / 8
else
  CLK = OSCCLK

FCLKDIV[5:0] = INT((CLK / 1000) / 200)
FCLK = CLK / (FCLKDIV[5:0] + 1)
```

**Figure 8. FCLKDIV Formulas**

In the formulas, OSCCLK represents the reference frequency present at the EXTAL pin, NOT the bus frequency or the PLL output. The INT function always rounds toward zero and FCLK represents the frequency of the clock signal that drives the FLASH's state machine.

**NOTE:** Erasing or programming the FLASH with an oscillator clock less than 500 kHz should be avoided. Setting FCLKDIV such that the state machine clock is less than 150 kHz can destroy the FLASH due to high voltage over stress. Setting FCLKDIV such that the state machine clock is greater than 200 kHz can result in improperly programmed memory locations.
After initializing the FCLKDIV register with the proper value, the PPAGE register and the BKSEL[1:0] bits must be initialized. The PPAGE register must be written with a value that places the correct 16-K memory block in the PPAGE window that contains the memory area to be erased. If a mass (bulk) erase operation is performed on one of the 64-K blocks, the PPAGE register may be written with any one of the four PPAGE values associated with a 64-K block. Note that when performing a mass or sector erase in the address range of one of the two fixed pages, $4000–$7FFF or $C000–$FFFF, the value of the PPAGE register is unimportant.

The BKSEL[1:0] bits, located in the FCNFG register, are used to select the banked status and control registers associated with the 64-K FLASH block in which the erase operation is to be performed. As shown in Figure 1, the value of the FLASH block number decreases with increasing PPAGE values. Closely examining Figure 1 reveals that the correct value for the BKSEL[1:0] bits is the one's complement of the PPAGE[3:2] register bits. Even though the flowchart shows the block select bits being written before the PPAGE register, these registers may be written in reverse order. This makes the code implementation straightforward since the value of the block select bits may be easily derived from the value written to the PPAGE register.

After initializing the PPAGE register and the block select bits, the command buffer empty interrupt flag (CBEIF) bit should be checked to ensure that the address, data and command buffers are empty. If the CBEIF bit is set, the buffers are empty and a program or erase command sequence can be started. The next three steps in the flowchart must be strictly adhered to. Any intermediate writes to the FLASH control and status registers or reads of the FLASH block on which the operation is being performed will cause the access error (ACCERR) flag to be set and the operation will be immediately terminated. For a mass erase operation, the address of the aligned data word may be any valid address in the 64-K block. For a sector erase, only the upper seven address bits are significant, the lower eight bits are ignored. For all erase operations, the data written to the FLASH block is ignored.
After writing a program or erase command to the FCMD register, the CBEIF bit must be written with a value of 1 to clear the CBEIF bit and initiate the command. After clearing the CBEIF bit, the ACCERR and PVIOL bits should be checked to ensure that the command sequence was valid. If either of these bits is set, it indicates that an erroneous command sequence was issued and the command sequence will be immediately terminated. Note that if either or both of the ACCERR and PVIOL bits are set, they must be cleared by writing a 1 to each flag’s associated bit position before another command sequence can be initiated. Five bus cycles after the CBEIF bit is cleared, the CCIF flag will be cleared by the state machine indicating that the command was successfully begun. If a previous command has not been issued, the CBEIF bit will become set, indicating that the address, data, and command buffers are available to begin a new command sequence.

Once the erase command has completed, erasure of the sector or block should be verified to ensure that all locations contain $FF. When erasing a 512-byte sector, each byte or word must be checked for an erased condition using software. Fortunately, however, the state machine has a verify command built into the hardware to perform an erase verify on the contents of any of the 64-K blocks. The command sequence used to perform an erase verify is identical to that of performing an erase command except that the erase verify command ($05) is written to the FCMD register and the block select bits and the PPAGE register need not be rewritten. If all locations in a 64-K block are erased, a successful erase verify will cause the BLANK bit in the FSTAT register to be set. Note that the BLANK bit must be cleared by writing a 1 to its associated bit position before the next erase verify command is issued.

FLASH Programming

As mentioned in the previous section, the erase and program operations follow a nearly identical flow. There are, however, some minor changes to the flow that can improve the efficiency of the programming process. To take advantage of the decreased programming time provided by the 2-stage FIFO command buffer, it must be kept full with programming commands. As the flowchart in Figure 9 shows, rather than waiting for each programming command to complete, a new programming command is issued as soon as the CBIEF flag is set. This allows the programming voltage to remain applied to the array as long as the next...
aligned word address remains within the same 64-byte row. Therefore, to minimize programming times, blocks of data to be programmed into the FLASH array should begin on a 64-byte boundary and be a multiple of 64 bytes.

Verification of programmed data should be performed only after a block of data has been programmed and all programming commands have completed. Performing a read operation on the FLASH array while a programming command is executing will cause the ACCERR flag to be set and all current and pending commands are terminated.

Figure 9. Programming Flowchart
A program such as the FLASH serial bootloader has two important requirements. First, it must have minimal impact on the final product’s software performance. Second, it should add little or no cost to the hardware design. Because the MC9S12DP256 includes a variety of on-chip communications modules, five CAN modules, one J1850 module, two SCI ports, and three SPI modules, no additional external hardware should be required. Designs incorporating a CAN or J1850 network connection could easily incorporate the existing connection into the bootloader to download the new FLASH data. For applications not utilizing a network connection in the basic design, one of the two SCI ports can be used. In many systems, the SCI may be a part of the hardware design since it is often used as a diagnostic port. If an RS232 level translator is not included as part of the system design, a small adapter board can be constructed containing the level translator and RS232 connector. This board can then be used by service personnel to update the system firmware. Using such an adapter board prevents the cost of the level translator and connector from being added to each system. In addition to the SCI port, a single input pin is required to notify the serial bootloader startup code to execute the bootloader code or jump to the system application program.

As mentioned previously, because the MC9S12DP256’s interrupt and reset vectors reside in the protected bootblock, they cannot be changed without erasing the bootblock itself. Even though it is possible to erase and reprogram the bootblock, it is inadvisable to do so. If anything goes wrong during the process of reprogramming the bootblock, it would be impossible to recover from the situation without the use of BDM programming hardware. For this reason, a bootloader should include support for a secondary interrupt and reset vector table located just below the protected bootblock area. Each entry in the secondary interrupt table should consist of a 2-byte address mirroring the primary interrupt and reset vector table. The secondary interrupt and reset vector table is utilized by having each vector point to a single JMP instruction that uses the CPU12’s indexed-indirect program counter relative addressing mode. This form of the JMP instruction uses four bytes of...
memory and requires just six CPU clock cycles to execute. For systems operating at the maximum bus speed of 25.0 MHz, six bus cycles adds only 240 ns to the interrupt latency. In most applications, this small amount of additional time will not affect the overall performance of the system.

**Bootloader**

**S-Record Format**

The S-record object file format was designed to allow binary object code and/or data to be represented in printable ASCII hexadecimal format allowing easy transportation between computer systems and development tools. For M68HC12 Family members supporting less than 64 Kbytes of address space, S1 records, which contain a 16-bit address, are sufficient to specify the location in the device’s memory space where code and/or data are to be loaded. The load address contained in the S1 record generally corresponds directly to the address of on-chip or off-chip memory device. For M68HC12 devices that support an address space greater than 64 Kbytes, S1 records are not sufficient.

Because the M68HC12 Family is a 16-bit microcontroller with a 16-bit program counter, it cannot directly address a total of more than 64 Kbytes of memory. To enable the M68HC12 Family to address more than 64 Kbytes of program memory, a paging mechanism was designed into the architecture. Program memory space expansion provides a window of 16-Kbyte pages that are located from $8000–$BFFF. An 8-bit paging register, called the PPAGE register, provides access to a maximum of 256, 16-Kbyte pages or 4 megabytes of program memory. While there may never be any devices that contain this much on-chip memory, the MC68HC812A4 is capable of addressing this much external memory. In addition, the MC9S12DP256 contains 256 Kbytes of on-chip FLASH residing in a 1MB address space.

While many high-level debuggers are capable of directly loading linked, absolute binary object files into a target system’s memory, the bootloader does not have that ability. The bootloader is only capable of loading object files that are represented in the S-record format. Because S1 records only contain a 16-bit address, they are inadequate to specify a load address for a memory space greater than 64 Kbytes. S2 records, which contain a 24-bit load address, were originally defined for loading object files into the memory space of the M68000 Family. It would seem
that S2 records would provide the necessary load address information required for M68HC12 object files. However, as those who are familiar with the M68000 Family know, the M68000 has a linear (non-paged) address space. Thus, development tools, such as non-volatile memory device programmers, interpret the 24-bit address as a simple linear address when placing program data into memory devices.

Because the M68HC12 memory space expansion is based on 16-Kbyte pages, there is not a direct one-to-one mapping of the 24-bit linear address contained in the S2 record to the 16-Kbyte program memory expansion space. Instead of defining a new S-record type or utilizing an existing S-record type in a non-standard manner, the bootloader’s program FLASH command views the MC9S12DP256’s memory space as a simple linear array of memory that begins at an address of $C0000. This is the same format in which S-records would need to be presented to a stand alone non-volatile memory device programmer.

The MC9S12DP256 implements six bits of the PPAGE register which gives it a 1MB program memory address space that is accessed through the PPAGE window at addresses $8000–$BFFF. The lower 768-K portion of the address space, accessed with PPAGE values $00–$2F, are reserved for external memory when the part is operated in expanded mode. The upper 256 K of the address space, accessed with PPAGE values $30–$3F, is occupied by the on-chip FLASH memory. The mapping between the linear address contained in the S-record and the 16-Kbyte page viewable through the PPAGE is shown in Figure 10.

The generation of S-records that meet these requirements is the responsibility of the linker and/or S-record generation utility provided by the compiler/assembler vendor. Cosmic Software’s linker and S-record generation utility is capable of producing properly formatted S-records that can be used by the bootloader. Other vendor’s tools may or may not posses this capability. For those compilers and assemblers that produce “banked” S-records, an S-record conversion utility, SRecCvt.exe, is available on the Web that can be used to convert “banked” S-records to the linear S-record format required by the serial bootloader.

**NOTE:** The bootloader is limited to receiving S-records containing a maximum of 64 bytes in the code/data field. If an S-record containing more than 64 bytes in the code/data field is received, an error message will be displayed.
The conversion of the linear S-record load address to a PPAGE number and a PPAGE window address can be performed by the two formulas shown in Figure 11. In the first formula, PageNum is the value written to the PPAGE register, PPAGEWinSize is the size of the PPAGE window which is $4000. In the second formula, PPAGEWinAddr is the address within the PPAGE window where the S-record code/data is to be loaded. PPAGEWinStart is the beginning address of the PPAGE window which is $8000.
Using the S-Record Bootloader

The S-record bootloader presented in this application note utilizes the on-chip SCI for communications with a host computer and does not require any special programming software on the host.

The bootloader presented in this application note can be used to erase and reprogram all but the upper 4 K of on-chip FLASH memory. The bootloader program utilizes the on-chip SCI for communications and does not require any special programming software on the host computer. The only host software required is a simple terminal program that is capable of communicating at 9600 to 115,200 baud and supports XOn/XOff handshaking.

Invoking the bootloader causes the prompt shown in Figure 12 to be displayed on the host terminal’s screen. The lowercase ASCII characters a through c comprise the three valid bootloader commands. These three lowercase characters were selected, rather than the ASCII characters 1 through 3, to prevent accidental command execution. If a problem occurs while programming the FLASH, an error message is displayed, and the bootloader will redisplay its prompt and wait for a command entry from the operator. Because the host computer will continue sending the S-record file, each character of the S-record file would be interpreted as an operator command entry. Since S-records contain all of the ASCII numeric characters, it is highly likely that one of them would be understood as a valid command.

\[
\text{pageNum} = \text{SRecLoadAddr} \div \text{PPAGEWinSize};
\]

\[
\text{PPAGEWinAddr} = (\text{SRecLoadAddr} \% \text{PPAGEWinSize}) + \text{PPAGEWinStart};
\]

Figure 11. PPAGE Number and Window Address Formulas

Figure 12. Serial Bootloader Prompt

MC9S12DP256Bootloader

a) Erase Flash
b) Program Flash
c) Set Baud Rate
?

For More Information On This Product, Go to: www.freescale.com
Erase FLASH Command

Selecting the erase function by typing a lowercase \texttt{a} on the terminal will cause a bulk erase of all four 64-K FLASH arrays except for the 4-k boot block in the upper 64-K array where the S-record bootloader resides. After the erase operation is completed, a verify operation is performed to ensure that all locations were properly erased. If the erase operation is successful, the bootloader’s prompt is redisplayed.

If any locations were found to contain a value other than $FF$, an error message is displayed on the screen and the bootloader’s prompt is redisplayed. If the MC9S12DP256 device will not erase after one or two attempts, the device may be damaged.

Program FLASH Command

To increase the efficiency of the programming process, the S-record bootloader uses interrupt driven, buffered serial I/O in conjunction with XOn/XOff software handshaking to control the S-record data flow from the host computer. This allows the bootloader to continue receiving S-record data from the host computer while the data from the previously received S-record is programmed into the FLASH.

\textbf{NOTE:} The terminal program must support XOn/XOff handshaking to properly reprogram the MC9S12DP256’s FLASH memory.

Typing a lowercase \texttt{b} on the terminal causes the bootloader to enter the programming mode, waiting for S-records to be sent from the host computer. The bootloader will continue to receive and process S-records until it receives an S8 or S9 end of file record. If the object file being sent to the bootloader does not contain an S8 or S9 record, the bootloader will not return its prompt and will continue to wait for the end of file record. Pressing the system’s reset switch will cause the bootloader to return to its prompt.

If a FLASH memory location will not program properly, an error message is displayed on the terminal screen and the bootloader’s prompt is redisplayed. If the MC9S12DP256 device will not program after one or two attempts, the device may be damaged or an S-record with a load address outside the range of the available on-chip FLASH may have been received. The S-record data must have load addresses in the range $\$C0000–$FFFFF$. This address range represents the upper 256 Kbytes of the 1-MB address space of the MC9S12DP256.
Set Baud Rate Command

While the default communications rate of the bootloader is 9600 baud, this speed is much too slow if the majority of the MC9S12DP256's FLASH is to be programmed; however, it provides the best compatibility for initial communications with most terminal programs. The set baud rate command allows the bootloader communication rate to be set to one of four standard baud rates. Using a baud rate of 57,600 allows the entire 256 K of FLASH to be programmed in just under two minutes.

Typing a lowercase \texttt{c} on the terminal causes the prompt shown in Figure 13 to be displayed on the host terminal's screen. Entering a number 1 through 4 on the keyboard will select the associated baud rate and issue a secondary prompt indicating that the terminal baud rate should be changed. After changing the terminal baud rate, pressing the enter or return key will return to the main bootloader prompt. The selected baud rate will remain set until the target system is reset.

![Figure 13. Change Baud Rate Prompt](image)

Bootloader Software

The software implementing the serial FLASH bootloader, shown in Code Listing, consists of seven basic parts: startup code, bootloader control loop, programming and erase code, serial communications routines, an S-record loader and a secondary interrupt vector jump table. The code is written in a position independent manner so that the generated object code will execute properly from any address.
Startup Code

The bootloader startup code implements several setup and initialization tasks.

The first action performed by the startup code checks the state of pin 6 on port M. If a logic 1 is present, the **JMP** instruction will continue execution at the address stored in the reset vector of the secondary vector table. If a logic 0 is present at pin 6 of port M, execution continues at the label **Boot** where the COP watchdog timer is disabled.

After the watchdog timer is disabled, the bootloader copies itself into the upper 4 K of the on-chip RAM. Execution of the bootloader code from RAM is necessary so the portion of FLASH block zero not occupied by the bootloader can be erased and programmed. Notice that only the code between the labels **BootStart** and **BootLoadEnd** is copied into RAM. This does not include the secondary vector jump table or the primary interrupt vector addresses since neither is required by the bootloader. After the copy operation is complete, the RAM is relocated to overlay the upper 12 K of FLASH memory between $D000 and $FFFF. Writes to the **INITRM** register do not go into effect until one bus clock after the write cycle occurs. This means that the RAM cannot be accessed at the new address until after this one clock delay. Normally, the store instruction would simply be followed with a NOP instruction to ensure that no unintended operations occurred. However, in this case because the RAM is being moved into the same address space where the CPU is executing, a CPU free cycle must follow the write cycle.

**NOTE:** To understand why the store instruction must use extended addressing and must be aligned to an even byte boundary, it is necessary to examine the cycle-by-cycle execution detail of the store instruction.

The **STAB** instruction using extended addressing requires three clock cycles when executed from internal MCU memory. These three clock cycles consist of a **P** cycle, a **w** cycle and an **O** cycle (**PwO**). The **P** cycle is a program word access cycle where program information is fetched as an aligned 16-bit word. The **w** cycle is the 8-bit data write. Finally, the **O** cycle is an optional cycle that is used to adjust instruction alignment in the instruction queue. An **O** cycle can be a free cycle (**f**) or a program word access cycle (**P**). When the first byte of an instruction with an odd number of bytes is misaligned (at an odd address), the **O** cycle becomes
a P cycle to maintain queue order. If the first byte is aligned (at an even address), the O cycle is an f cycle. Consequently, if the first byte of the STAB instruction using extended addressing is aligned to an even byte boundary, the O cycle will be an f cycle. This will then provide the cycle of delay required while the RAM is overlaying the FLASH. Because the default address of the INITRM register is in the direct page addressing range, most assemblers will use direct rather than extended addressing. The greater than character (>) appearing as the first character in the operand field of the STAB instruction is used to force extended addressing. Note that some assemblers may not recognize this modifier character.

The main reason for relocating the RAM, rather than executing the bootloader at the RAM’s default address, is to allow the SCI0 interrupt vector to be changed. Because the on-chip RAM has a higher priority in the memory decoding logic than the on-chip FLASH, overlaying the FLASH with the on-chip RAM causes the RAM to be accessed rather than the FLASH. Due to the fact that the bootloader’s communications routines utilize the SCI in a buffered, interrupt driven mode, the SCI0 interrupt vector must be initialized to point to the bootloader’s SCI interrupt service routine.

After relocating the on-chip RAM, the startup code initializes the PLL and engages it as the bus clock. The values for the REFDV and SYNR registers are calculated by the assembler based on values of the oscillator frequency (OscClk), final bus frequency (fEclock), and the desired reference frequency (RefClock). In this case, the final bus frequency is specified to be 24.0 MHz. Because this is an integer multiple of the oscillator frequency, the oscillator frequency can be used as the reference clock for the PLL. This results in a value of zero being written to the REFDV register. To obtain a bus clock of 24 MHz, the reference frequency must be multiplied by three. The value written to the SYNR register multiplies the reference clock by SYNR+1 to generate the bus clock. Therefore, a value of two is written to the SYNR register to obtain a 24-MHz bus clock. Note that the four NOP instructions following the STAB instruction work around a bug in the 0K36N mask set. This errata manifested itself in the LOCK bit not being cleared until several bus cycles after a write to the SYNR register had occurred. Also note that a 24-MHz bus clock was chosen to support a baud rate of 115,200.
The final actions performed by the startup code initialize the FCLKDIV register and call the SCIInit subroutine. The value written to the FCLKDIV register is calculated by the assembler and is based on the MC9S12DP256’s oscillator frequency, not the bus frequency. The SCIInit subroutine initializes the SCI0 hardware and associated data structures needed to support buffered, interrupt driven communications. It accepts a single parameter in the D accumulator that is used to set the initial baud rate.

Bootloader Control Loop

After the startup code has completed its task, a sign-on message is displayed and the bootloader enters its main control loop. At the start of the loop, the X index register is loaded with the address of the bootloader prompt and the subroutine PromptResp is called. The PromptResp subroutine is used to display a null terminated ($00) character string and then waits for a single character response from the operator. Upon receipt of a character, the PromptResp subroutine returns and a range check is performed on the received character to ensure it is a valid command. If the received character is not a valid command, the entry is ignored and the prompt is redisplayed.

If the received character is one of the three valid commands, its ASCII value is used as an index into a table of offsets. However, before being used as an offset, the upper four bits of the ASCII value must be removed. Next, one must be subtracted from the remaining value because the first entry in the table is at an offset of zero. The result of the subtraction must then be multiplied by two because each entry in the table consists of two bytes. Next the LEAX instruction is used in conjunction with program counter relative (PCR) indexed addressing to load the address of the command table into the X index register in a position independent manner. Because the B accumulator contains an offset to the proper entry in the command table, the LDD instruction uses B accumulator offset indexed addressing to retrieve the entry from the table.

Examining the command table at label CmdTable, it can be seen that the table does not contain the absolute address of the command to execute. Rather each table entry contains an offset from the beginning of the table to the start of the command. This offset, when added to the
base address of the table contained in the X index register, produces the absolute address of the first instruction of the requested command. Using offsets in the command table in conjunction with calculating the beginning of the table in a position independent manner, allows a computed GOTO to be performed in a position independent manner. Finally, the JSR instruction uses accumulator offset indexed addressing to calculate the address of the command and calls the command as a subroutine.

Upon return from the command, the value of the global variable ErrorFlag is examined. If it contains a value of zero, the command completed without any errors. In this case, the code branches back to the top of the command loop where the bootloader prompt is redisplayed. If, however, an error occurred during command execution, the value in ErrorFlag is used as an index into a table of offsets to null terminated error strings. Calculation of the absolute address of the error string is performed in much the same manner as the calculation of the absolute address of the command. After displaying the error message, the code branches back to the top of the command loop where the bootloader prompt is redisplayed.

Program Command Code

The firmware required to implement the FLASH programming command consists of two subroutines. The first subroutine, ProgFlash, is called through the command table. This subroutine coordinates the activities required by the ProgFBlock subroutine which performs the actual programming of the FLASH memory. The ProgFlash subroutine begins by calling the GetSRecord subroutine which is used to receive a single S-record from the host computer. Having received an valid S-record, the subroutine performs several checks to ensure that the S-record meets the programming requirements of the MC9S12DP256. Because the MC9S12DP256’s FLASH may only be programmed an align word at a time, both the code/data field length and the load address must be even numbers. If either value is odd, an error code is stored in the ErrorFlag global variable and the FLASH programming operation is terminated.
Next, the received S-record type is checked. Reception of an S8 or S9 S-record terminates the program FLASH command returning to the bootloader’s control loop where the prompt is redisplayed. S0 records, designated as header records, do not contain any program or data and are simply ignored. Because the linear S-record addresses for the MC9S12DP256 begin at $C0000 as shown in Figure 10, only S2 S-records may be used to program the on-chip FLASH. Because the GetSRecord subroutine is capable of receiving S0, S1, S2, S8 and S9 S-records, the program FLASH command is terminated and an error code is returned in the ErrorFlag global variable if an S1 record is received.

After checking the received S-record type, a range check is performed on the S-record load address to ensure it is within the range of the on-chip FLASH minus the size of the 4 K protected area containing the bootloader. When performing the range check, the load address is first checked against SRecLow, the lowest valid S-record address for the on-chip FLASH. However, when checking against the upper limit, SRecHi, the number of code/data bytes contained in the S-record must be added to the load address before the comparison is performed. This ensures that even though the initial load address is less than the upper limit, none of the S-record code/data falls outside the upper limit.

Finally, the ProgFlash subroutine uses the S-record load address to calculate the PPAGE number and PPAGE window address using the formulas in Figure 11. After initializing the PPAGE register, the PPAGE value is used to calculate a value for the block select bits. Closely examining the PPAGE values and the block numbers as shown in Figure 1, it can be determined that the block number for any of the PPAGE values corresponds to the one’s complement of bits two and three of the block’s corresponding PPAGE value. After writing the proper value to the block select bits in the FCNFG register, the ProgFBlock subroutine is called to program the received S-record data into the FLASH. If no errors occurred during the programming operation, the code branches to the label FSendPace where an ASCII asterisk character is sent to the host computer to indicate that S-record data was successfully programmed into the FLASH.
The **ProgFBlock** subroutine performs the task of programming the received S-record data into the on-chip FLASH. While the subroutine generally follows the flowchart in **Figure 9**, some operations have been rearranged to improve the efficiency of the implementation. The first two steps in the flowchart, writing the PPAGE register and block select bits, are performed in the **ProgFlash** subroutine. Note that the order of these two operations is not important. Because the value for the block select bits is derived from the PPAGE value, the **ProgFlash** subroutine writes the PPAGE register value first.

The third operation in the flowchart checks the state of the CBEIF bit to ensure that the command buffer is empty and ready to accept a new command. This check is not made at the beginning of the **ProgFBlock** subroutine because the bit is known to be set when the subroutine completes execution. This condition is inferred by the fact that the CCIF flag is set before the programmed data from the previously received S-record is verified.

The **ProgFBlock** subroutine begins by retrieving the S-record code/data field length, dividing the value by two and placing the result on the stack. The code/data field length is divided by two because the FLASH is programmed a word at a time. Next, the X and Y index registers are initialized to point to the FLASH and S-record data respectively. Note that the X index register is loaded with the value in the PPAGEAddr global variable. This value, calculated using the second formula in **Figure 11**, will always point within the PPAGE window. After initializing the pointers, the programming loop is entered at label **ProgLoop**. Note that within the programming loop there are no instructions that directly correspond to the five bus cycle delay before checking the state of the CBEIF flag after issuing the program command. Instead, the five bus cycle delay is inherent in the three instructions (LDAB, BITB, BNE) used to check the state of the ACCERR and PVIOL status bits. This loop follows the remainder of the flowchart in **Figure 9**, issuing a new programming command each time the CBEIF flag is set until all of the count in the local variable NumWords is zero.

Before verifying that all of the FLASH locations programmed properly, the firmware must wait until the CCIF flag is set, indicating that all issued programming commands have completed. Failure to observe this
constraint before performing a read operation on the FLASH will result in the setting of the ACCERR bit and any pending programming commands will be terminated. The verification process begins by reinitializing the DataBytes local variable and the X and Y index register pointers. If any of the programmed words do not match the S-record data, a “not equal” condition (Z bit in the CCR equal to 0) is returned.

Erase Command Code

The code comprising the FLASH erase command is not nearly as simple as the programming code; it consists of five subroutines. The reason for the additional complexity surrounds the method that must be used to erase a FLASH block containing protected areas. When a 64-K block has a portion of its contents protected from being erased or programmed, the FLASH’s mass erase command cannot be used. Instead, the unprotected areas must be erased one 512-byte sector at a time. Because the time required to erase a sector is 20 ms versus 100 ms for the mass erase operation, erasure of a 64-K block with protected areas requires much longer. In this case where the bootloader resides in a 4-K protected area of block zero, 120 sector erase operations must be performed. Not counting the time required to verify each sector erasure, the sector erase operations require 2.4 seconds (20 ms * 120 sectors).

The FLASH erase command begins with the subroutine EraseFlash, called through the command table. This subroutine coordinates the activities of the other four subroutines. It begins by performing a mass erase and verify on three of the 64-K FLASH blocks. After all three of the 64-K FLASH blocks have been successfully erased, the EraseBlk0 subroutine is called to perform a sector by sector erase of the unprotected portion of FLASH block zero.

The EraseBlk0 subroutine begins by allocating and initializing the local variable PPAGECnt. The initialized value of three is the number of 16-K PPAGE windows that will be completely erased a sector at a time. The PPAGE register is initialized with a value passed in the B accumulator from the EraseFlash subroutine. This value, $3C, places the lower 16 K of FLASH block zero into the PPAGE window. The block select bits are initialized to zero. After loading the X index register with the address
of the start of the PPAGE window and the B accumulator with the number of sectors to erase, the **EraseSectors** subroutine is called. In addition to erasing the requested number of sectors, the **VerfSector** subroutine is called to verify the erasure. Note that the **VerfSector** subroutine verifies the erasure a word at a time because the erase verify command built into the FLASH state machine will only operate on a 64-K block. After **EraseBlk0** performs the erasure of the lower 48 K of FLASH block zero, the lower 24 sectors ($8000–$EFFF) of the upper 16 K of block zero are erased.

### Set Baud Rate Command Code

The code comprising the set baud rate command is relatively simple. The subroutine begins by displaying the baud rate change prompt and then waiting for the operator to enter a baud rate selection. A range check is performed on the entered character; if an invalid character is entered, the prompt is redisplayed. If the selection is valid, the upper four bits are masked off, one is subtracted from the lower four bits, and the result is divided by two. The result is used as an index into the **BaudTable** to retrieve the proper **SCI0BD** register value for the selected baud rate.

Before switching to the newly selected baud rate, a message is displayed prompting the operator to change the host terminal’s baud rate. However, before the **SCI0BD** register is written with the new value, the firmware must wait until the last character of the message is shifted from the SCI0 transmit shift register. Once the last character of the message is sent, the **SCI0BD** register is written with the new value and the **getchar** subroutine is called to wait for an indication from the operator that the host terminal baud rate has been changed. Finally, a carriage return/line feed is sent to the terminal before returning to the bootloader control loop.

### S-Record Loader Code

The **GetSRecord** subroutine is used to receive a single S-record from the host computer. **GetSRecord** begins by allocating space on the stack for two local variables and initializing the X index register. The **SRecBytes** variable is used to hold the converted value of the S-record length field. This value includes the number of bytes contained in the load address field, the length of the code/data field, and the length of the...
checksum field. The variable CheckSum is used to contain the calculated checksum value as the S-record is received. The X index register is initialized to point to the beginning of the 24-bit global variable, LoadAddr, where the received S-record’s address is stored. Note also that the most significant byte of LoadAddr is cleared in case an S1 record is received.

After the initializations, a search is begun for the character pairs S0, S1, S2, S8, or S9 which indicate the start of a valid S-record. Once a valid start of record is found, the number of bytes in the load address plus one is stored in the global variable DataBytes. This value is subsequently subtracted from the received S-record length byte to produce a result representing the code/data field length. Before receiving the S-record length byte, the second character of the start of record pair is stored in the global RecType. After receiving the S-record length byte, the value is saved in the local variable SRecBytes. This value is also used to initialize CheckSum which is used to calculate a checksum value as the S-record is received.

The loop beginning at the label RcvData receives the remainder of the S-record including the load address, the code/data field, and the checksum. Note that because each received byte is stored in successive memory locations, the global variables LoadAddr and SRecData must remain in the order they are declared. As each data byte and the checksum is received, it is added into the calculated checksum value. Because the received checksum is actually the one’s complement of what the calculated checksum should be, adding the two values should produce a result of $FF$. Incrementing the CheckSum variable at the end of the receive loop should produce a result of zero if the checksum and all the S-record fields were received properly. This results in an “equal” condition (CCR Z = 1) being returned if the S-record was properly received and a “not equal” condition (CCR Z = 0) being returned if a problem occurred receiving the S-record.

Operation of the GetSRecord subroutine is supported by the three additional subroutines GetHexByte, IsHex, and CvtHex. The GetHexByte subroutine retrieves two ASCII hex bytes from the serial port and converts them into a single 8-bit data byte that is returned in the B accumulator. The IsHex subroutine is used to check received byte to
ensure that it is an ASCII hexadecimal character. If the character in the B accumulator is a non-hexadecimal character, the subroutine returns a “not equal” condition (CCR Z = 0). Otherwise, an “equal” condition (CCR Z = 1) is returned. The `CvtHex` subroutine converts the ASCII hexadecimal character in the B accumulator to a binary value. The result remains in the B accumulator.

Serial Communications Code

The serial communications routines utilize SCI0 to communicate with a host computer. The routines utilize the SCI in an interrupt driven mode, allowing reception of data from the host computer while the bootloader is programming the on-chip FLASH memory. To prevent the possibility of the receive buffer overflowing, the receive routines support XOn/XOff handshaking with the host computer. Because the bootloader does not send large amounts of data to the host computer, XOn/XOff handshaking is not supported by the transmit routines.

To utilize the interrupt driven mode effectively, a circular buffer or queue must be associated with both the transmitter and receiver. The queue acts as an elastic buffer providing a software interface between the received character stream and the MC9S12DP256. In addition to the storage required by the transmit and receive queues, several other pieces of data are required for queue management. The information necessary to manage the queue consists of a way to determine the next available storage location in each queue, the next available location or piece of data in the queue, and a way to determine if a queue is full or empty. Rather than utilize 16-bit pointers to manage the queues, the communications routines employ four 1-byte variables. `RxIn`, `RxOut`, `TxIn`, and `TxOut` are used in conjunction with 8-bit accumulator offset indexed addressing to access data in the transmit and receive queues. In addition, two 1-byte variables, `RxBAvail` and `TxBAvail`, are used to keep track of the number of bytes available in each queue. When the value in each of these variables is equal to the size of the queue, the buffer is empty. When the value is zero, the queue is full. Using a byte for the index does not allow support of queue sizes greater than 255 bytes. However, this should not pose severe restrictions for most applications.
The proper queue size for an application will depend on the expected length of messages transmitted and received. If the selected transmit queue size is too small, the routines essentially will behave the same as the polled SCI example. Once the queue fills, the CPU12 will have to wait until a character is transmitted before the next character can be placed in the queue. If the receive queue is too small, there will be a risk that received characters will be lost if the queue becomes full and CPU12 does not remove some of the data before the next piece of data arrives. Conversely, picking queue sizes larger than necessary does not have a detrimental effect on program performance or loss of data. However, it will consume the valuable on-chip memory unnecessarily. If uncertain on the exact queue size for a particular application, it is best to make it larger than necessary. As shown, the transmit and receive queues do not have to be the same size, and their sizes are not required to be an even power of two.

The XOffCount and XOnCount constants are used to manage how full and how empty, respectively, the receive queue is allowed to get before the XOff and XOn control characters are sent to the host computer. The value for XOffCount should be chosen based on the number of bytes that are expected to be sent from the host after a request has been made for the TxIRQ routine to send an XOff to the host. This value, which represents the number of remaining bytes in the receive queue when an XOff should be sent, will depend on the UART characteristics of the host computer. In this case, a value of XOffCount would allow up to 10 additional characters to be sent after a request to send the XOff had been posted. This would allow for the host computer UART with an 8-byte FIFO plus the possible 2-character delay in sending the XOFF character if the transmit shift register and the transmit data register were both full.

The value for XOnCount should be selected such that the queue will never become empty as long as the host has data to send. Setting the correct value for this constant requires analysis of the rate at which data is removed from the queue by the application and the delay before the host computer begins sending data after receiving an XOn. Because the host's characteristics can vary widely, a value of the receive buffer minus eight was arbitrarily chosen. Note that the value of XOnCount represents the number of characters available in the receive queue.
The SCIInit subroutine is used to initialize the SCI hardware and the related queue data structures. The baud rate register (SCI0BD) value for the desired baud rate is passed to the subroutine in the D accumulator. The queue index values RxIn, RxOut, TxIn, TxOut, and the values for RxBAvail and TxBAvail are not specifically initialized by the subroutine because the initial values are set at the point of their declaration. This technique works in this case because the constant values were copied from the FLASH into RAM. In a situation where the variables were declared with a ds (define storage) directive each variable would have to be initialized to its proper value.

When the transmitter and receiver are enabled, notice that only the receive interrupts are enabled. Unlike the receiver interrupts, which may be enabled at all times, the transmit interrupt may be enabled only when the transmit queue contains characters to be sent. Enabling transmit interrupts at initialization would immediately cause a transmitter interrupt even though the transmit queue is empty. This is because the TDRE bit is set whenever the SCI transmitter is in an idle state. The final action performed by the SCIInit subroutine initializes the SCI0 interrupt vector to point to the SCI interrupt routine, SCIISR.

Because each SCI only has a single interrupt vector shared by the transmitter and receiver, a short dispatch routine determines the source of the interrupt and calls either the RxIRQ or TxIRQ. Note that it is not an arbitrary choice to have the dispatch routine check for receiver interrupts before transmitter interrupts. To avoid the loss of received data, an SCI interrupt dispatch routine should always check the receiver control and status flags before checking those associated with the transmitter. Failure to follow this convention will most likely result in receiver overruns when data is received during message transmissions longer than a couple of bytes.

The receive interrupt service routine, RxIRQ, has the responsibility of removing a received byte from the receive data register and placing it in the receive data queue if space is available. In addition, if space available in the queue falls below the value of XOffCount, two variables, SendXOff and XOffSent, are set to a non-zero value and transmitter interrupts are enabled. These actions cause an XOff character to be sent to the host computer the next time a transmit
interrupt is generated. \textit{XOffSent} is used by the receive interrupt service routine to ensure that only a single XOff character is sent to the host after the space available in the queue falls below the value of \textit{XOffCount}. \textit{XOffSent} is also used by the \texttt{getchar} subroutine to determine if an XOn should be sent after each character is removed from the queue. Finally, notice that if the queue becomes full, the received byte is simply discarded.

The transmit interrupt service routine, \texttt{TxIRQ}, has the responsibility of removing a byte from the transmit data queue and sending it to the host computer. Before sending a character from the transmit queue, \texttt{SendXOff} is checked. If it contains a non-zero value, an XOff character is immediately sent to the host. Sending the XOff character before sending data that may be in the transmit queue ensures data flow from the host is stopped before the receive queue overflows. Notice that if the queue becomes empty after a character is transmitted, transmitter interrupts are disabled.

The last two major routines rounding out the serial communication code are the \texttt{getchar} and \texttt{putchar} subroutines. The \texttt{getchar} subroutine’s main function is to retrieve a single character from the receive queue and return it to the calling routine in the B accumulator. Notice that if the receive queue is empty, the \texttt{getchar} subroutine will wait until a character is received from the host. Because this action may not be desirable for some applications, a utility subroutine, \texttt{SCIGetBuf}, can be called to determine if any data is in the receive queue. This small subroutine returns, in the B accumulator, a count of the number of data bytes in the receive queue. In addition to managing the receive queue variables each time a character is removed from the queue, the \texttt{getchar} subroutine checks the state of \textit{XOffSent} and the number of characters left in the receive queue to determine if an XOn character should be sent to the host computer. If an XOff character was previously sent and the number of characters left in the receive queue is less than \textit{XOnCount}, an XOn character is sent to the host by calling the \texttt{putchar} routine.
The `putchar` subroutine’s main function is to place a single character, passed in the B accumulator, into the transmit queue. Once the character is in the queue and the queue variables have been updated, the transmit interrupt enable (TIE) bit is set. If transmitter interrupts were not previously enabled and the transmit data register empty (TDRE) bit is set, setting the TIE bit will cause an SCI interrupt to occur immediately.

Secondary Interrupt Vector Jump Table

Because the reset and interrupt vectors reside in the protected bootblock, a secondary vector table is located just below the protected bootblock area. Each entry in the secondary interrupt table should consist of a 2-byte address mirroring the primary interrupt and reset vector table. The secondary interrupt and reset vector table is utilized by having each vector point to a single JMP instruction that uses the CPU12’s indexed-indirect program counter relative addressing mode. This form of the JMP instruction uses four bytes of memory and requires just six CPU clock cycles to execute. The table in Figure 14 associates each vector source with the secondary interrupt table address.
<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Secondary Vector Address</th>
<th>Interrupt Source</th>
<th>Secondary Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved $FF80</td>
<td>$EF80</td>
<td>I²C bus</td>
<td>$EFC0</td>
</tr>
<tr>
<td>Reserved $FF82</td>
<td>$EF82</td>
<td>DLC</td>
<td>$EFC2</td>
</tr>
<tr>
<td>Reserved $FF84</td>
<td>$EF84</td>
<td>SCME</td>
<td>$EFC4</td>
</tr>
<tr>
<td>Reserved $FF86</td>
<td>$EF86</td>
<td>CRG lock</td>
<td>$EFC6</td>
</tr>
<tr>
<td>Reserved $FF88</td>
<td>$EF88</td>
<td>Pulse accumulator B over o w</td>
<td>$EFC8</td>
</tr>
<tr>
<td>Reserved $FF8A</td>
<td>$EF8A</td>
<td>Modulus down counter under o w</td>
<td>$EFC9</td>
</tr>
<tr>
<td>PWM emergency shutdown</td>
<td>$EF8C</td>
<td>Port H interrupt</td>
<td>$EFCC</td>
</tr>
<tr>
<td>Port P interrupt</td>
<td>$EF8E</td>
<td>Port J interrupt</td>
<td>$EFCE</td>
</tr>
<tr>
<td>MSCAN 4 transmit</td>
<td>$EF90</td>
<td>ATD1</td>
<td>$EFD0</td>
</tr>
<tr>
<td>MSCAN 4 receive</td>
<td>$EF92</td>
<td>ATD0</td>
<td>$EFD2</td>
</tr>
<tr>
<td>MSCAN 4 errors</td>
<td>$EF94</td>
<td>SCI</td>
<td>$EFD4</td>
</tr>
<tr>
<td>MSCAN 4 wakeup</td>
<td>$EF96</td>
<td>SCI0</td>
<td>$EFD6</td>
</tr>
<tr>
<td>MSCAN 3 transmit</td>
<td>$EF98</td>
<td>SPI0</td>
<td>$EFD8</td>
</tr>
<tr>
<td>MSCAN 3 receive</td>
<td>$EF9A</td>
<td>Pulse accumulator A input edge</td>
<td>$EFDA</td>
</tr>
<tr>
<td>MSCAN 3 errors</td>
<td>$EF9C</td>
<td>Pulse accumulator A over o w</td>
<td>$EFD0</td>
</tr>
<tr>
<td>MSCAN 3 wakeup</td>
<td>$EF9E</td>
<td>Timer over o w</td>
<td>$EFD3</td>
</tr>
<tr>
<td>MSCAN 2 transmit</td>
<td>$EFA0</td>
<td>Timer channel 7</td>
<td>$EFE0</td>
</tr>
<tr>
<td>MSCAN 2 receive</td>
<td>$EFA2</td>
<td>Timer channel 6</td>
<td>$EFE2</td>
</tr>
<tr>
<td>MSCAN 2 errors</td>
<td>$EFA4</td>
<td>Timer channel 5</td>
<td>$EFE4</td>
</tr>
<tr>
<td>MSCAN 2 wakeup</td>
<td>$EFA6</td>
<td>Timer channel 4</td>
<td>$EFE6</td>
</tr>
<tr>
<td>MSCAN 1 transmit</td>
<td>$EFA8</td>
<td>Timer channel 3</td>
<td>$EFE8</td>
</tr>
<tr>
<td>MSCAN 1 receive</td>
<td>$EFAA</td>
<td>Timer channel 2</td>
<td>$EFEA</td>
</tr>
<tr>
<td>MSCAN 1 errors</td>
<td>$EFAC</td>
<td>Timer channel 1</td>
<td>$EFEC</td>
</tr>
<tr>
<td>MSCAN 1 wakeup</td>
<td>$EFAE</td>
<td>Timer channel 0</td>
<td>$EFED</td>
</tr>
<tr>
<td>MSCAN 0 transmit</td>
<td>$EFB0</td>
<td>Real-time interrupt</td>
<td>$EFF0</td>
</tr>
<tr>
<td>MSCAN 0 receive</td>
<td>$EFB2</td>
<td>IRQ</td>
<td>$EFF2</td>
</tr>
<tr>
<td>MSCAN 0 errors</td>
<td>$EFB4</td>
<td>XIRQ</td>
<td>$EFF4</td>
</tr>
<tr>
<td>MSCAN 0 wakeup</td>
<td>$EFB6</td>
<td>SWI</td>
<td>$EFF6</td>
</tr>
<tr>
<td>FLASH</td>
<td>$EFB8</td>
<td>Unimplemented instruction trap</td>
<td>$EFF8</td>
</tr>
<tr>
<td>EEPROM</td>
<td>$EFBA</td>
<td>COP failure reset</td>
<td>$EFFA</td>
</tr>
<tr>
<td>SPI2</td>
<td>$EFBC</td>
<td>Clock monitor fail reset</td>
<td>$EFFC</td>
</tr>
<tr>
<td>SPI1</td>
<td>$EFBE</td>
<td>Reset</td>
<td>$FFE0</td>
</tr>
</tbody>
</table>

Figure 14. Secondary Vector Table Addresses for a 4-K Bootblock
Code Listing

00000000 RegBase: equ $0000
; opt lis
;
M offset: macro opt
M PCSave: set *
M org $:0
M endm
;
M switch: macro
M ifc ".text",":0"
M org PCSave
M endif
M endm

007A1200 OscClk: equ 8000000
016E3600 fEclock: equ 24000000
007A1200 RefClk: equ 8000000
00000000 REFDVVal: equ (OscClk/RefClk)-1
00000002 SYNVal: equ (fEclock/RefClk)-1
00000000 if OscClk=12800000
else
FCLKDIVVal: equ (OscClk/20000000)+FDIV8
endif
00000028 FCLKDIVVal: equ (OscClk/2000000)

0000000D Baud115200: equ fEclock/16/115200
0000001A Baud57600: equ fEclock/16/57600
00000027 Baud38400: equ fEclock/16/38400
0000009C Baud9600: equ fEclock/16/9600
;
00000800 FlashStart: equ $8000
00001000 BootBlkSize: equ 4096
00001000 RAMStart: equ $1000
0000FF80 StackTop: equ $ff80
00003000 RAMBoot: equ $3000
;
00000200 SectorSize: equ 512
00004000 PPAGESize: equ 16384
;
000C0000 SRecLow: equ $c0000
000FF0000 SRecHi: equ $ff0000

; oscillator clock frequency.
; final E-clock frequency (PLL).
; reference clock used by the PLL.
; value for the REFDV register.
; value for the SYN register.
; value for the FCLKDIV register.
; value for the FCLKDIV register.
; baud register value for 115,200 baud.
; baud register value for 57,600 baud.
; baud register value for 38,400 baud.
; baud register value for 9,600 baud.
; start address of the flash window.
; Erase protected bootblock size.
; default RAM base address.
; stack location after RAM is moved.
; starting RAM address where the bootloader
; will be copied.
; size of a Flash Sector.
; size of the PPAGE window ($8000 - $BFFF).
; lowest S-Record load address accepted
; by the bootloader.
; highest S-Record load address + 1
; accepted by the bootloader.
Application Note

AN2153

For More Information On This Product, Go to: www.freescale.com

For More Information On This Product, Go to: www.freescale.com

00000000  S0RecType:  equ  '0'
00000001  S1RecType:  equ  '1'
00000002  S2RecType:  equ  '2'
00000003  S3RecType:  equ  '3'
00000004  S4RecType:  equ  '4'
00000005  S5RecType:  equ  '5'
00000006  S6RecType:  equ  '6'
00000007  S7RecType:  equ  '7'
00000008  S8RecType:  equ  '8'
00000009  S9RecType:  equ  '9'

;******
0000000A  ERaseError: equ 1          ; Flash failed to erase.
0000000B  SRecRngErr: equ 2         ; S-Record out of range.
0000000C  FlashPrgErr: equ 3        ; Flash programming error.
0000000D  SRecDataErr: equ 4        ; Received S-Record contained an odd number of data bytes.
0000000E  SRecAddrErr: equ 5        ; S-Record Address is odd.
0000000F  SRecLenErr: equ 6         ; S-Record is too long.

;*************************************************************************************************************

00000010  1F02514004 BootStart:  brc lr  PTIM,540,Boot ; execute the bootloader?
00000011  05FBFFFS jmp  [Reset-BootBlkSize,pcr] ; no. jump to the program pointed to by the secondary reset vector.

00000012  979003C Boot:  clr  COPCTL ; keep watchdog disabled.

00000013  CFF80 BootCopy:  lds #StackTop ; initialize the stack pointer
00000014  CEF00 BootStart:  ldx #BootStart ; point to the start of the Flash bootloader in Flash.
00000015  CD3000 RAMBoot:  ldy #RAMBoot ; point to the start of on-chip RAM.
00000016  CCF59A LoadStart:  ldd #BootLoadEnd ; calculate the size of the bootloader code.
00000017  83F000 BootEnd:  subd #BootStart
00000018  180A3070 MoveMore:  movb 1,x+,y+ ; move a byte of the bootloader into RAM.
00000019  434F9 d,Jvovtr MoreMove ; dec byte count, move till done.

0000001A  C6C1 ldab #5c0+RAMHAL ; write to the INITRM register to overlay the Flash bootblock with RAM.

0000001B  0000000 if *$0001<0 ; PC currently at an odd byte boundary?
0000001C  0000000 endif

0000001D  7B0010 stab >INITRM ; this instruction MUST use extended addressing an be aligned to an even byte boundary.

0000001E  C600 ldab #REFDVVal ; set the REFDV register.
0000001F  5B35 stab REFDV
00000020  C602 ldab #SYNVal ; set the SYNV register.
00000021  5B34 stab SYNR
00000022  A7 nop ; nops required for bug in initial silicon.
AN2153

Application Note

Code Listing

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

0000F032 A7          ; nop
0000F033 4F3708FC     ; brclr CRGFLG,#LOCK,* ; wait here till the PLL is locked.
0000F037 4C3980       ; bset CLKSEL,#PLLSel ; switch the bus clock to the PLL.
0000F03A C628        ; ldadb #FCLKDIVVal
0000F03C 7B0100       ; stab FCLKDIV
0000F03F CC009C       ; ldd #Baud9600 ; set SCI to 9600 baud.
0000F042 15FA046F     ; jsr SCIInit,pcr ; go initialize the SCI.
0000F046 10EF         

0000F048 1AFA0055     ; leax SignOn,pcr ; get the bootloader signon message
0000F04C 15FA0422     ; jsr OutStr,pcr ; send it to the terminal.
0000F050 69FA0A546    ; clr ErrorFlag,pcr ; clear the global error flag.
0000F054 1AFA0064     ; leax BIhPrompt,pcr ; get the bootloader prompt
0000F058 072A         ; bsr PromptResp ; go display the prompt & get a 1 character response.
0000F05A C160        ; cmpb $61 ; do a range check. less than 'a'?
0000F05C 25F2         ; bhi CmdLoop ; yes. just re-display the prompt.
0000F05E C163        ; cmpb $63 ; greater than 'c'?
0000F060 22EE         ; bhi CmdLoop ; yes. just re-display the prompt.
0000F062 C40F         ; andb $0f0 ; no. mask off the upper nybble.
0000F064 53           ; decb ; reduce by 1 for indexing into the command offset table.
0000F065 58           ; lab ; mul by 2 as each cmd table entry is a 2 byte address.
0000F066 1AFA0031     ; leax CmdTable,pcr ; point to the command table.
0000F06A ECE5         ; ldd b,x ; get offset from the beginning of the table to the cmd.
0000F06C 15E6         ; jsr d,x ; execute the command.
0000F06E 69FA0A528    ; lddb ErrorFlag,pcr ; error executing the command?
0000F072 27DC         ; beq CmdLoop ; no. go display the prompt, wait for entered command.
0000F074 53           ; decb ; subtract 1 from the error number for indexing.
0000F075 58           ; lab ; mul by 2 because each address in the table is 2 bytes.
0000F076 1AFA00CC     ; leax ErrorTable,pcr ; yes. point to the error table.
0000F07A ECE5         ; ldd b,x ; get offset from the start of the table to the string.
0000F07C 1AE6         ; leax d,x ; calc the address of the error string from the table.
0000F07E 15FA03F0     ; jsr OutStr,pcr ; send error message to the terminal.
0000F082 20CC         ; bra CmdLoop ; go display the prompt.

;*************************************************************************************************************

; 0000F084 15FA03EA     ; PromptResp: jsr OutStr,pcr ; send prompt to the terminal.
0000F088 15FA0488     ; jsr getchar,pcr ; get the user's choice.
0000F08C 15FA04E9     ; jsr putchar,pcr ; echo it.
0000F090 37           ; pshb ; save it.
0000F091 1AFA0060     ; leax CrlfStr,pcr ; go to the next line.
0000F095 15FA03D9     ; jsr OutStr,pcr ; restore the entered character.
0000F099 33           ; pulb
0000F09A 3D           

;*************************************************************************************************************


 CmdTable:   dc.w  EraseFlash-CmdTable  ; cmd table entry for 'Erase Flash' command.
0000F09D  01A7  dc.w  ProgFlash-CmdTable  ; cmd table entry for 'Program Flash' command.
0000F09F  0169  dc.w  SetBaud-CmdTable  ; cmd table entry for 'Set Baud Rate' command.

; 0000F0A1  0DA4D33953  SignOn:  dc.b  $0d,$0a,"MC9S12DP256 Bootloader",$0d,$0a,0

; 0000F0BC  0DDA61292045  BIPrompt:  dc.b  $0d,$0a,"a) Erase Flash",$0d,$0a
0000F0CE  62292050726F  dc.b  "b) Program Flash",$0d,$0a
0000F0E0  632920536574  dc.b  "c) Set Baud Rate",$0d,$0a
0000F0F2  3F2000  dc.b  ",",0

; 0000F0F5  0DDA00  CrlfStr:  dc.b  $0d,$0a,0

; 0000F0F8  0DDA31292039  BaudPrompt:  dc.b  $0d,$0a,"1) 9600",$0d,$0a
0000F0F9  322920333834  dc.b  "2) 38400",$0d,$0a
0000F100  332920353736  dc.b  "3) 57600",$0d,$0a
0000F117  342920313135  dc.b  "4) 115200",$0d,$0a
0000F122  3F2000  dc.b  ",",0

; 0000F125  4368616E6765  BaudChgPrompt:  dc.b  "Change Terminal BR, Press Return",0

; 0000F146  000C  ErrorTable:  dc.w  FNotErasedStr ErrorTable
0000F148  0021  dc.w  SRecRngStr ErrorTable
0000F14A  003B  dc.w  FlashPrgErrStr ErrorTable
0000F14C  0057  dc.w  SRecDataErrStr ErrorTable
0000F14E  007C  dc.w  SRecAddrErrStr ErrorTable
0000F150  0090  dc.w  SRecLenErrStr ErrorTable

; 0000F152  0DDA466C6173  FNotErasedStr:  dc.b  $0d,$0a,"Flash Not Erased",$0d,$0a,0
0000F167  0DDA532D5265  SRecRngStr:  dc.b  $0d,$0a,"S-Record out of Range",$0d,$0a,0
0000F181  0DDA466C6173  FlashPrgErrStr:  dc.b  $0d,$0a,"Flash Programming Error",$0d,$0a,0
0000F190  0DDA532D5265  SRecDataErrStr:  dc.b  $0d,$0a,"S-Record code/data length is odd",$0d,$0a,0
0000F1C2  0DDA532D5265  SRecAddrErrStr:  dc.b  $0d,$0a,"S-Record Address is odd",$0d,$0a,0
0000F1DE  0DDA532D5265  SRecLenErrStr:  dc.b  $0d,$0a,"S-Record Code/Data Field Too Long",$0d,$0a,0

;******************************************************************************

; 0000F204  01FAFEF0  SetBaud:  equ  *  
0000F204  1FAFEF78  leax  BaudPrompt, pcr  ; get the baud rate change prompt
0000F208  15FAFE78  jar  PromptResp, pcr  ; go display the prompt & get a 1 character response.
0000F20C  C31  cmpb "$31"  ; do a range check. less than '1'?
0000F20E  25F4  blo  SetBaud  ; yes. just re-display the prompt.
0000F210  C314  cmpb "$34"  ; greater than '4'?
0000F212  22F0  bih  SetBaud  ; yes. just re-display the prompt.
0000F214  C40F  andb "$0f"  ; no. mask off the upper nibble.
0000F216  53  debc  "$0f"  ; subtract 1 for table indexing.
0000F217  58  lslb  "$0f"  ; multiply by 2 because each table entry is 2 bytes.
0000F218  1FA001E  leax  BaudTable, pcr  ; point to the start of the table.
0000F21C  EC5  ldd  b, x  ; get the SCI0BD value from the table.
0000F21E  3B  psphd  ; save the value.
leax BaudChgPrompt, pcr ; prompt the user to change the terminal baud rate.
jsr OutStr, pcr ; send it to the terminal.
brclr SCI0SR1, #TC,* ; wait until the last character is sent until we change the baud rate.
puld ; restore the SCI0BD value from the stack.
std SCI0BD ; change the baud rate.
getchar, pcr ; go wait for the user to change the baud rate.
leax CrIfStr, pcr ; go to the next line.
jsr OutStr, pcr ; return.
dc.w Baud9600 ; SCI0BD value for 9600 baud.
dc.w Baud38400 ; SCI0BD value for 38400 baud.
dc.w Baud57600 ; SCI0BD value for 57600 baud.
dc.w Baud115200 ; SCI0BD value for 115200 baud.

lea #PVIOL+ACCERR ; if either the PVIOL or ACCERR bit is set from a previous error, reset them so we can program the Flash.
ldab #$001F+ACCERR
bra FSkipFirst ; don't send the progress character the first time.
ldab #$2C
bra #FSkipFirst ; the ascii asterisk is the progress character.
jsr putchar, pcr ; let the user know we've processed an S-Record.
jsr GetSRecord, pcr ; go get an S-Record.
ldab LoadAddr, pcr ; was an S2 record. Get high byte of the 24-bit address.
cmp #SRecRngErr ; less than $C0000 - $FEFFF
bhs ChkHiLimit ; allowed.
ldab #SRecAddrErr
bra ProgDone ; no. report the error.
ldab #SRecAddrErr
bra ProgDone ; stop programming.
ldab #SRecRngErr
bra ProgDone ; no. report the error.
ldab #SRecAddrErr
bra ProgDone ; stop programming.
ldab RecType, pcr ; check the record type.
cmpb #S1RecType
bne ChckNext ; S1 record received?
cmpb #S2RecType
bne ChckNext ; no. check for S0, S2, S8 & S9 records.
cmpb #S8RecType
bne ChckNext ; yes. only S2 records w/ load addresses $C0000 - $FEFFF allowed.
cmpb #SRecRngErr ; was it an S9 record?
beq ProgRtn ; yes. we're done.
beq ProgRtn ; yes. we're done.
beq ProgRtn ; no. was it an S8 record?
beq FSendPace ; yes. just ignore it.
ldab LoadAddr, pcr ; was an S2 record. Get high byte of the 24-bit address.
cmpb #SRecLow>>16 ; less than $C0000?
bhs ChkHiLimit ; no. check the upper limit.
0000F289 8602  BadSRecRng: ldaa #$RecRngErr ; yes. S-Record out of range.
0000F28B 2045  bra  ProgDone   ; save the error code & return.

0000F28D E6FA030B ChkHiLimit:  ldab  DataBytes,pcr ; get the number of bytes in the S-Record.
0000F291 B76 clra  ; zero extend it.
0000F292 E6FA030A addd  LoadAddr+1,pcr ; add in the lower 16-bits of the 24-bit address.
0000F296 B745 tfr  d,x ; save the result in X.
0000F298 E6FA0303 ldab  LoadAddr,pcr ; get the upper 8-bits of the 24-bit address.
0000F29C C900 adcb  #$000 ; add in possible carry from lower 16-bits.
0000F29E C10F cmpb  #$RecL1 >> 16 ; greater than 0Fxxxx?  
0000F2A0 2505 blo  AddrOK ; no. S-Record within range.
0000F2A2 8EF000 cpx  #$RecL1 & 00000 ; yes. check the lower 16- bits. Out of range?
0000F2A5 22E2 bhi  BadSRecRng ; yes. S-Record out of range.

0000F2A7 E6FA02F4 AddrOK:  ldab  LoadAddr,pcr ; get upper 8-bits of 24-bit load address.
0000F2AB B796 exc  b,y ; zero extend b into y for the 32-bit divide
0000F2AD E6FA02EF ldd  LoadAddr+1,pcr ; get the lower 16-bits of 24-bit load address.
0000F2B1 CE4000 ldx  #PPAGESize ; divide the load address by the PPAGE window size.
0000F2B4 11  ediv ; if we fall through, we automatically return a non-zero condition.

0000F2B5 C38000 addd  #FlashStart ; add the PPAGE window start address to the remainder (this gives the PPAGE window load address).
0000F2B7 E6FA02F4 AddrOK:  ldab  LoadAddr,pcr ; get upper 8-bits of 24-bit load address.
0000F2B8 B7C6 exc  d,y ; lower byte of the quotient is the PPAGE value.
0000F2BA 5B30 stab PPAGE ; calculate the value of the block select bits based on bits 3:2 of the PPAGE register value.
0000F2BC 54 lsr b ; select the block to erase.
0000F2BD 54 lsr b ; select the block to erase.
0000F2BE 51 comb ; select the block to erase.
0000F2BF C403 andb  #$03 ; mask off all but the lower 2 bits.
0000F2C1 7B0103 stab FCNF ; select the block to erase.
0000F2C4 6DFA02D5 sty  PPAGEWAddr,pcr ; save the PPAGE window address.
0000F2C8 15FA000B jsr  ProgFBlock,pcr ; go program the data into Flash.
0000F2CC 1827FF79 lbeq FSendError ; zero condition means all went ok.
0000F2D0 8603 ldd  #FlashProgErr ; put error code where pod can access it.
0000F2D2 E6FA02F4 ProgDone: staa ErrorFlag,pcr ; if we fall through, we automatically return a non-zero condition.
0000F2D6 3D ProgRtn:  rts ; exit program.

0000F2D7 offset 0
0000F2D7 PCSave:  set *
0000F2D7 org $0 ;
0000F2D7 NumWords: ds 1
0000F2D7 LocalSize: set *

0000F2D7 switch .text
0000F2D7 ifc '.text', '.text'
0000F2D7 org PCSave
0000F2D7 endif

0000F2E7 E6FA02F4 ProgFBlock: ldab  DataBytes,pcr ; get the block size.
0000F2DB  54   lsrdb ; divide the byte count by 2 since we program a word
  0000F2DC  37   pshb ; at a time.
  0000F2DD  EEFA02BC ldx PPAGEWAddr, pcr ; allocate the local.
  0000F2EE  19FA02BD leay SRecData, pcr ; get the PPAGE window Flash address.
  0000F2E5  EC71 ProgLoop: ldd 2, y+ ; point to the received S-Record data.
  0000F2E7  6C31 std 2, x+ ; get a word from the buffer.
  0000F2E9  C620 ldb #PROG ; latch the address & data into the Flash
  0000F2E8  7B0106 ladb FCMD ; program/erase buffers.
  0000F2EE  C680 ldb #CBEIF ; start the command by writing a 1 to CBEIF.
  0000F2F0  7B0105 ladb FSTAT ; check to see if there was a problem executing
  0000F2F3  F60105 ldab FSTAT ; the command.
  0000F2F6  C530 bitb #PRIOL+ACCERR ; if either the PVIOL or ACCERR bit is set,
  0000F2F9  2627 bne Return ; return.
  0000F2FA  1F010580FB brcir FSTAT, #CBEIF,* ; wait here till the command buffer is empty.
  0000F2FF  6380 dec NumWords, sp ; any more words to program?
  0000F301  26E2 bne ProgLoop ; yes, continue until done.
  0000F303  1F010540FB brcir FSTAT, #CCIF,* ; no, wait until all commands complete.
  0000F308  E6FA0290 ladb DataBytes, pcr ; get the block size.
  0000F30C  54   lsrdb ; divide the byte count by 2 since we verify a
  0000F30D  6B80 stab NumWords, sp ; word at a time.
  0000F30F  EEFA028A ldx PPAGEWAddr, pcr ; get the PPAGE window Flash address.
  0000F313  19FA028B leay SRecData, pcr ; point to the received S-Record data.
  0000F317  EC71 VerifLoop: ldd 2, y+ ; get a word from the buffer.
  0000F319  AC31 cpd 2, x+ ; same as the word in Flash?
  0000F31B  26B4 bne Return ; no, return w/ an error (!= condition).
  0000F31D  6380 dec NumWords, sp ; yes, done comparing all words?
  0000F31F  26F6 bne VerifLoop ; no, compare some more.
  0000F321  33   Return: pulb ; deallocates the local.
  0000F322  3D   rts ; return.

;*************************************************************************************************************
;0000F323                              offset     0
  0000F323                              ifc .text, .text
  0000F323                              org PCSave
endif

; EraseFlash:
ldab #$03
push
ldab #$30
stab PPAGE
; write the PPAGE register to allow writes to the
; proper Flash block.

ldab #$03
; calculate the value of the block select bits based
; on bits 3:2 of the PPAGE register.

com
andb #$03
; mask off all but the lower 2 bits.

ldx #FlashStart
ldab #ERASE+MASS
bset FSTAT,#BLANK,Erased
; latch address for erase command
; perform a bulk erase.

bset FSTAT,#BLANK,Erased
; flag a not erased error if the BLANK bit did not set.

bra SaveError
ldab #BLANK
; clear the BLANK status bit.

ldab PPAGE
; get the current PPAGE value.

addb #$04
; add 4 to select the next 64K Flash block.

dec BlockCnt.sp
bset FSTAT,#BLANK,Erased
; done with 3 of the 64K blocks?

bne EraseLoop
; block 0 must be erased separately because it
; contains the bootblock.

sta ErrorFlag,pcr
; put error code where pod can access it.

push
bra ErrorFlag,pcr
; return.

; EraseBlk0 erases Flash block 0 a sector (512 bytes) at a time because the bootblock is protected.

1F010540FB VerfCmdOK:
bra FSTAT,#CCIF,*
; wait until the command has completed.

1E01050404 brset FSTAT,#BLANK,Erased
; flag a not erased error if the BLANK bit did not set.

8601 ldaa #FEraseError

200F bra SaveError

C604 Erased:
ldab #BLANK

7B0105 stab FSTAT

D630 ldab PPAGE

C604 addb #$04

6380 dec BlockCnt.sp

26CE bne EraseLoop

706 bsr EraseBlk0

6AF0A023A SaveError:
sta ErrorFlag,pcr

33 FEEDone:
pulb

3D rts

0000F362 offset 0
0000F362 PCSave: set *
0000F362 org $0

0000000 PPAGECnt: ds.b 1
; number of 16K PPAGE windows that will be
; completely erased.

0000001 LocalSize: set *

0000001 switch .text
0000001 ifc '".text',".text"
0000F362 org PCSave
endif

For More Information On This Product, Go to: www.freescale.com
AN2153

Freescale Semiconductor, Inc.

0000F362 1808AP03  EraseBlk0:  movb  #3,1,-sp                   ; 3 16K PPAGE windows will be completely erased.
0000F366  5B30       stab  PPAGE                  ; PPAGE for first 16K page of block 0
0000F369  790103     clr   FCNFG                   ; set block select bits to 0.
0000F36B  C80000     EraseBlk0Loop:  ldab  #FlashStart  ; point to the start of the PPAGE window.
0000F36D  C620       ldab  #PPAGESize/SectorSize ; number of sectors in a PPAGE window.
0000F370  0712       bsr  EraseSectors          ; go erase the PPAGE window a sector at a time.
0000F372  260E       bne  BadBlk0                 ; non-zero value returned in A indicates a sector
0000F374  720030     inc   PPAGE                  ; didn't erase.
0000F377  6380       dec   PPAGECnt,sp           ; go to the next PPAGE.
0000F379  2F6F       bne  EraseBlk0Loop         ; done with all full PPAGE blocks?
0000F37B  C68000     ldab  #FlashStart  ; yes. point to the start of the PPAGE window.
0000F37D  C618       ldab  (#PPAGESize-BootBlkSize)/SectorSize ; number of sectors in PPAGE $3F
0000F383  3D         rts

; Erases 'b' (accumulator) sectors beginning at address 'x' (index register)
0000F384  B796       exg   b,y                   ; put the sector count in y.
0000F386  C640       EraseSectLoop:  ldab  #ERASE   ; perform a sector erase.
0000F388  070F       bsr  EraseCmd
0000F38A  2701       breq  DoEraseVerf          ; if no problem with the erase command, do a verify.
0000F38C  3D         Rtn:  rts
0000F38E  0723       DoEraseVerf:  bsr  VerfSector ; if problem, return with an error code in a.
0000F38F  C6FB       bne  Rtn                   ; problem, return with an error code in a.
0000F391  1AE2000     leax  SectorSize,x       ; point to the next sector.
0000F395  0436EE     dbne  y,EraseSectLoop   ; continue to erase remaining sectors.
0000F398  3D         rts

; Erases a block or sector of Flash
0000F399  6C00       EraseCmd:  std   0,x                   ; latch address for erase command.
0000F39B  7B0106     stab  FCMD
0000F39E  C680       ldab  #CBEIF
0000F3A0  7B0105     stab  FSTAT           ; initiate the erase command.
0000F3A3  1F0103003   brcir  FSTAT,#PVIO+ACCERR,EraseCmdOK ; continue if the privilege violation &
0000F3A8  8601       ldaa  #FEraseError ; Access error flags are clear.
0000F3AA  3D         rts
0000F3AB  1F010540FB  EraseCmdOK:  brcir  FSTAT,#CCIF,* wait until the command has completed.
0000F3B0  87         clra
0000F3B1  3D         rts

; Verify that a sector was properly erased
; Must verify a word at a time because the built in verify command only works on a block (64K)


save the base address of the sector.

; save the sector count.

; get a byte from the sector.

; get a byte from the sector.

; yes, dec the sector word count.

; restore the sector address of the sector.

; restore the base address of the sector.

; return.

;*************************************************************************************************************

; allocate stack space for variables.

; point to the code/data buffer.

; get a character from the receiver.

; get another record character.

; get another record character.

; found an S9 (end) record? (16-bit load address)

; go check for an S9 record.

; go check for an S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go check for another record.

; adjust the storage pointer to compensate for

; found an S9 record? (16-bit load address)

; go check for an S9 record.

; go receive the S9 record.

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.

; found an S9 record? (16-bit load address)

; go receive the S9 record.

; found an S1 record? (16-bit load address)

; go receive the S9 record.
0000F3EC 8603       ldaa #3             ; 2 byte load address.
0000F3EE 6AFOA1AA   staa DataBytes,pcr  ; 2 address bytes plus the checksum.
0000F3F2 2010       bra SaveRecType    ; go receive the S9 record.

0000F3F4 C132       ChkForS2:        cmpb #$8RecType   ; S2 record? (24-bit load address)
0000F3F6 2602       bne ChkForS8     ; S2 record? (24-bit transfer address)
0000F3F8 2004       bra Addr24      ; no. s8 record? (24-bit transfer address)

0000F3FA C138       ChkForS8:       cmpb #$8RecType   ; no. s8 record? (24-bit transfer address)
0000F3FC 26D1       bne LookForSor     ; no. go look for next Start of Record.
0000F3FE 8604       Addr24:        ldda #4          ; 3 address bytes plus the checksum.
0000F400 6AFOA198    staa DataBytes,pcr  ; 3 address bytes plus the checksum.
0000F404 6BFA0193    SaveRecType:    stab RecType,pcr    ; yes. save the record type.

0000F408 15FA003E    RcvSRec:       jsr GetHexByte,pcr  ; get the S-Record length byte.
0000F40C 2626       bne BadSRec     ; return if there was an error.
0000F40E 6B80       stab SRecBytes,sp  ; save the total number of S-Record bytes we
                                                      ; are to receive.
0000F410 6B81       stab CheckSum,sp  ; initialize the checksum calculation with the
                                                      ; data byte count
0000F412 E0FA0186    subb DataBytes,pcr  ; subtract the load address & checksum field
                                                      ; length from the data field count.
0000F416 6BFA0182    stab DataBytes,pcr  ; subtract the load address & checksum field
                                                      ; length from the data field count.
0000F41A C140       cmpb #$64         ; is the code/data field size.
0000F41C 2304       bls RcvData       ; yes. it can be received.
0000F41E 8606       ldda #$SRecLenErr  ; no. the code/data field is limited to 64 bytes.
0000F420 0012       bra BadSRec       ; return with the error code in a.
0000F422 15FA0024    RcvData:       jsr GetHexByte,pcr  ; get an S-Record data byte.
0000F426 260C       bne BadSRec     ; return if there was an error.
0000F428 6B30       stab 1,x+        ; save the byte in the data buffer.
0000F42A 6B81       addb CheckSum,sp  ; add the byte into the checksum.
0000F42C 6B81       stab CheckSum,sp  ; add the byte into the checksum.
0000F42E 6380       dec SRecBytes,sp  ; received all the S-Record bytes?
0000F430 26F0       bne RcvData      ; no. go get some more.
0000F432 6281       inc CheckSum,sp  ; if checksum was ok, the result will be zero.
0000F434 1B82       BadSRec:       leas LocalSize,sp  ;
                                                      ;
0000F436 3D         rts

*************************************************************************************************************

0000F437 C130       cmpb #$0'         ; less than ascii hex zero?
0000F439 250E       blo NotHex       ; yes. character is not hex. return a non-zero
                                                      ; ccr indication.
0000F43B C139       cmpb #$9'         ; less than or equal to ascii hex nine?
0000F43D 2308       bls IsHex1       ; yes. character is hex. return a zero ccr indication.
0000F43F C141       cmpb #$A'         ; less than ascii hex 'A'?
0000F441 2506       blo NotHex       ; yes. character is not hex. return a non-zero
                                                      ; ccr indication.
cmpb #$'F', ; less than or equal to ascii hex 'F'?  
bhi NotHex; yes. character is hex. return a non-zero  
ccr indication.

orcc #$'F', ; no. return a zero ccr indication.

;*************************************************************************************************************

;*************************************************************************************************************

jsr getchar,pcr; get the upper nybble from the SCI.

bsr IsHex; valid hex character?

beq OK1; yes. go convert it to binary.

rts; no. return with a non-zero ccr indication.

bar CvtHex; convert the ascii-hex character to binary.

lda $16; shift it to the upper 4-bits.

mul; save it on the stack.

jsr getchar,pcr; get the lower nybble from the SCI.

bsr IsHex; valid hex character?

beq OK2; yes. go convert it to binary.

rts; no. return with a non-zero ccr indication.

bar CvtHex; convert the ascii-hex character to binary.

addb 1,sp+; add it to the upper nybble.

cira; simple way to set the Z ccr bit.

rts; return.

subb #$'0',; subtract ascii '0' from the hex character.

cmpb #$09; was it a decimal digit?

bss CvtHexRtn; yes. ok as is.

subb #$07; no. it was an ascii hex letter ('A' - 'F').

rts;

;*************************************************************************************************************

;*************************************************************************************************************

jsr putchar,pcr; no. send it out the SCI.

bra OutStr; go get the next character.

;*************************************************************************************************************

;*************************************************************************************************************

equ *; send a null terminated string to the display.

lda $1,*; get a character, advance pointer, null?

beq OutStrDone; yes. return.

jsr putchar,pcr; no. send it out the SCI.

bra OutStr; go get the next character.

;*************************************************************************************************************

equ 32; receive queue size.

equ 16; transmit queue size.
00000018  XOnCount:   equ    RxBufSize-8            ; number of bytes avail. in the Rx queue before an XOn can be sent.
0000000A  XOffCount:  equ    10                         ; number of bytes remaining in the Rx queue when an XOff is sent.

00000011  XOn:        equ    $11                          ; ASCII DC1
00000013  XOff:       equ    $13                          ; ASCII DC3

0000F4D7  000000000000 RxBuff:   dcb    RxBufSize,0        ; receive queue.
0000F4D9  000000000000 TxBuff:   dcb    TxBufSize,0        ; transmit queue.
0000F4DA  00          RxIn:    dc.b    0                     ; next available location in the Rx queue.
0000F4AE  00          RxOut:   dc.b    0                     ; next character to be removed from the Rx queue.
0000F4AF  00          TxIn:    dc.b    0                     ; next available location in the Tx queue.
0000F4B0  00          TxOut:   dc.b    0                     ; next character to be sent from the Tx queue.
0000F4B1  20          RxBAvail: dc.b    RxBufSize          ; number of bytes left in the Rx queue.
0000F4B2  10          TxBAvail: dc.b    TxBufSize          ; number of bytes left in the Tx queue.
0000F4B3  00          XOffSent: dc.b    0                     ; if != 0, an XOff has been sent.
0000F4B4  00          SendXOff: dc.b    0                     ; request to TX ISR to send an XOff to the host if != 0.

0000F4BB  5CC8     SCIInit:    std     SCI0BD                 ; initialize the baud rate register.
0000F4B7  C62C     SCIInit:    ldab    #TE+RxE+RIE          ; get bit mask for Tx, Rx & Rx interrupt.
0000F4B9  58CB     SCIInit:    stab    SCI0CR2               ; enable Tx & Rx & Rx interrupts.
0000F4BB  1AFA0004 leax    SCIISR,pcr               ; setup SCI0 interrupt vector to point to the
0000F4BF  7EFF6D   stx     SCI0                      ; bootloader's SCI interrupt service routine.
0000F4C2  3D       scall    SCIISR                   ; done.

0000F4C3  4FCB2004 SCIISR:     brc     SCI0CR2,#RIE,ChkRxInts  ; Rx interrupts enabled?
0000F4C7  4EEC2009 SCIISR:     brcr    SCI0SR1,#RDF,RxIRQ   ; yes. if RDRF flag set, service Rx interrupt.
0000F4CB  4FCB8004 ChkRxInts:  brcr    SCI0CR2,#RIE,SCIISR    ; Tx interrupts enabled?
0000F4CF  4EEC8035 ChkRxInts:  brcr    SCI0SR1,#TDRE,TxIRQ    ; Yes. if TDRE is set, service Tx interrupt
0000F4D3  0B       NoSCIISR:  rti                                 ; return w/o any action.

0000F4D4  E7F9DC   RxIRQ:     tst     XOffSent,pcr             ; was an XOff previously sent to the host?
0000F4D7  2610     bne     AlreadySent                       ; yes. go place the received char in the Rx queue.
0000F4D9  A6F9D5   ldaa    RxBAvail,pcr                       ; no. get the number of bytes available in the Rx queue.
0000F4DC  810A     cmpa    #XOffCount                        ; more than enough space to receive a FIFO full
0000F4DE  2209     bhi     AlreadySent                       ; of data from the host?
0000F4E0  72F4B4   inc     SendXOff                         ; set flag so that XOff will be sent by the Tx ISR.
0000F4E3  4CC8B8   bset    SCI0CR2,#RIE                      ; enable transmitter interrupts.
0000F4E6  62F9CA   inc     XOffSent,pcr                      ; set the 'XOff Sent' flag
0000F4E9 E7F9C5       AlreadySent:    tst     RxBAvail,pcr  ; any room left in the Rx queue?
0000F4EC 2717       beq     Bufull       ; no. just throw the character away.
0000F4EE 63F9C0       dec     RxBAvail,pcr  ; yes. there'll be one less now.
0000F4F1 1AF998       leax    RxBuff,pcr  ; point to the physical start of the Rx queue.
0000F4FA A6F9B6       ldaa    RxIn,pcr     ; get the index for the next available queue location.
0000F4FD D6CF       ldab    SCIORDL  ; get the received character.
0000F4F9 6BE4       stab    a,x   ; place it in the queue.
0000F4FB 42       inc     ; next available queue location.
0000F4FC 8120       cmpa    #RxBufSize  ; wrap around to start of queue?
0000F4FE 2501       blo     NoRxWrap  ; no. just update the index.
0000F500 87       cira    ; yes. start at beginning of queue.
0000F501 6AF9A9       NoRxWrap:    staa    RxIn,pcr     ; update the next available queue location index.
0000F504 0B       rti       ; return from the SCI Rx interrupt.
0000F505 D6CF       Buffull:    ldab    SCIORDL  ; the queue was full. get character & throw it away.
0000F507 0B       rti       ; return.

;*************************************************************************************************************

0000F508 F7F4B4       TxIRQ:    tst     SendXOff      ; request to send an XOff.
0000F50B 2712       beq     NoSendXOff     ; no. go send a character from the Tx queue.
0000F50D 69F9A4       clr     SendXOff,pcr  ; yes. clear the request flag.
0000F510 C613       ldaa    #XOff      ; get the XOff character.
0000F512 5B2B       stab    SCIORDL     ; send it.
0000F514 E6F99B       ldaa    TxBAvail,pcr  ; any other characters in the Tx queue?
0000F517 C110       cmpb    #TxBufSize  ; yes. just return & let the next interrupt
0000F519 2622       bne     TxRTI       ; send the character.
0000F51B 4DCC80       bclr    SCIOCR2,#TIE     ; no. disable Tx interrupts.
0000F51E 0B       rti       ; return.

0000F520 1AF97B       NoSendXOff:    leax    TxBuff,pcr  ; point to the physical start of the Tx queue.
0000F522 A6F9B8       ldaa    TxOut,pcr  ; get the index for the next character to send.
0000F525 E6E4       ldaa    a,x   ; get the data.
0000F527 5B2B       stab    SCIORDL     ; send it.
0000F529 42       clca    ; advance to next character to send.
0000F52A 8110       cmpa    #TxBufSize  ; reached the end of the queue?
0000F52C 2501       blo     NoTxWrap     ; no.
0000F52E 87       clra    ; yes. wrap to the start.
0000F52F 6AF97E       NoTxWrap:    staa    TxOut,pcr  ; update the queue index.
0000F530 2501       blo     TxBVail,pcr  ; one more byte available in the queue.
0000F533 A1F977       cmpa    TxIn,pcr     ; TxIn = TxOut?
0000F535 2603       bne     TxRTI       ; no. more characters to send.
0000F537 4DCC80       bclr    SCIOCR2,#TIE     ; yes. queue is empty turn off TDRE interrupts.
0000F539 0B       rti       ; return.

;*************************************************************************************************************

0000F53A C620       SCIGetBuf:    ldab    #RxBufSize  ; are there any characters in the Rx queue?
0000F540 E0F96E       subb    RxBAvail,pcr
getchar:        pshx                              ; save the registers we'll use.
           ldab #XBufSize                      ; any characters available?
           subb RxBAvail, pcr                 ; no. just wait until some are.
           beq RxChk                          ; point to the physical start of the Rx queue.
           lea RxBuff, pcr                     ; get the index to the next available character
           ldaa RxOut, pcr                     ; in the Rx queue.
           ldab a, x                          ; get the character.
           inca                               ; point to the next location in the queue.
           cmpa #XBufSize                      ; reached the end of the queue?
           blo NocWrap                         ; no.
           clra                               ; yes. wrap to the start.
           staa RxOut, pcr                     ; update the queue index.
           inc RxBAvail, pcr                   ; we removed a character from the queue, there's
                                                  ; 1 more available.
                                              ;*************************************************************************************************************
putchar:        pshx                              ; save the registers we'll use.
           ldab #XOn                           ; send an XOn character to the host.
           cmpa #XOnCount                     ; enough space available to receive more?
           bha gcReturn                       ; no. just return.
           ldaa RxBAvail, pcr                 ; yes. get the number of bytes available in the Rx queue.
           clra                               ; yes. save the character we retrieved from the Rx queue.
           clra                               ; send an XOn character to the host.
           staa XOffSent, pcr                 ; was an XOff character previously sent by the RX ISR?
           pulx                              ; restore what we saved.
                                  ;*************************************************************************************************************
putchar:        pshx                              ; save the registers we'll use.
           ldab #TxBufSize                    ; Any room left in the Tx queue?
           beq TxBuff, pcr                    ; no. just wait here till it is.
           lea TxBuff, pcr                    ; point to the physical start of the Tx queue.
           ldaa TxBail, pcr                   ; get the index to the next available spot.
           stb a, x                          ; put the character in.
           inca                               ; point to the next available spot.
           cmpa #TxBufSize                    ; go past the end of the queue?
           blo NocWrap                         ; no.
           clra                               ; yes. wrap around to the start.
           staa TxBail, pcr                   ; update the queue index
           dec TxBail, pcr                    ; one less byte available in the Tx queue.
           bset SCI0CR2,#TIE                  ; enable transmitter interrupts.
0000F597 32  pula                        ; restore what we saved.
0000F598 30  pulx
0000F599 3D  rts                        ; return

0000F59A  BootLoadEnd: equ *

; Global Variable declarations

0000F59A  ErrorFlag: ds.b 1            ; error code stored by various routines.
0000F59B  RecType:   ds.b 1            ; received record type. '0' = SO; '1' = SI; '2' = S2;
                                           ; '8' = SB; '9' = S9
0000F59C  DataBytes: ds.b 1            ; number of data bytes in the S-Record.
0000F59D  PAGEAddr: ds.b 2             ; PPAGE window address ($B000 - $BFFF)
0000F59E  LoadAddr: ds.b 3             ; load address of the S-Record.
0000F5A2  SRecData: ds.b 65            ; S-Record data storage. (handle 64-byte S-Records
                                           ; + received checksum)

;*************************************************************************************************************
;                      ;       This is the jump table that is used to access the secondary interrupt vector table. Each one
;       of the actual interrupt vectors, beginning at $f8c, points to an entry in this table. Each jump
;       instruction uses indexed indirect program counter relative (pcr) addressing to access the
;       secondary interrupt vector table that is located just below the bootblock.
;*************************************************************************************************************

;-----------------------------------------------------------------------------------------------------------------
;  ;
;  ; This is the jump table that is used to access the secondary interrupt vector table. Each one
;  ; of the actual interrupt vectors, beginning at $f8c, points to an entry in this table. Each jmp
;  ; instruction uses indexed indirect program counter relative (pcr) addressing to access the
;  ; secondary interrupt vector table that is located just below the bootblock.
;-----------------------------------------------------------------------------------------------------------------

0000F5E3  05FBF9A5  JPWMEShutdown: jmp  [PWMEShutdown-BootBlkSize,pcr]
0000F5E7  05FBF9A3  JPortPInt:  jmp  [PortPInt-BootBlkSize,pcr]
0000F5EB  05FBF9A1  JMSCAN4Tx: jmp  [MSCAN4Tx-BootBlkSize,pcr]
0000F5EF  05FBF99F  JMSCAN4Rx: jmp  [MSCAN4Rx-BootBlkSize,pcr]
0000F5F3  05FBF99D  JMSCAN4Errs: jmp  [MSCAN4Errs-BootBlkSize,pcr]
0000F5F7  05FBF99B  JMSCAN4WakeUp: jmp  [MSCAN4WakeUp-BootBlkSize,pcr]
0000F5FB  05FBF999  JMSCAN3Tx: jmp  [MSCAN3Tx-BootBlkSize,pcr]
0000F5FF  05FBF997  JMSCAN3Rx: jmp  [MSCAN3Rx-BootBlkSize,pcr]
0000F603  05FBF995  JMSCAN3Errs: jmp  [MSCAN3Errs-BootBlkSize,pcr]
0000F607  05FBF993  JMSCAN3WakeUp: jmp  [MSCAN3WakeUp-BootBlkSize,pcr]
0000F60B  05FBF991  JMSCAN2Tx: jmp  [MSCAN2Tx-BootBlkSize,pcr]
0000F60F  05FBF98F  JMSCAN2Rx: jmp  [MSCAN2Rx-BootBlkSize,pcr]
0000F613  05FBF98D  JMSCAN2Errs: jmp  [MSCAN2Errs-BootBlkSize,pcr]
0000F617  05FBF98B  JMSCAN2WakeUp: jmp  [MSCAN2WakeUp-BootBlkSize,pcr]
0000F61B  05FBF989  JMSCAN1Tx: jmp  [MSCAN1Tx-BootBlkSize,pcr]
0000F61F  05FBF987  JMSCAN1Rx: jmp  [MSCAN1Rx-BootBlkSize,pcr]
0000F623  05FBF985  JMSCAN1Errs: jmp  [MSCAN1Errs-BootBlkSize,pcr]
0000F627  05FBF983  JMSCAN1WakeUp: jmp  [MSCAN1WakeUp-BootBlkSize,pcr]
0000F62B  05FBF981  JMSCAN0Tx: jmp  [MSCAN0Tx-BootBlkSize,pcr]
AN2153

55

Application Note

Code Listing

0000F62F 05FB9F7F  JMSCAN0Rx:  jmp  [MSCAN0Rx-BootBlkSize,pcr]
0000F633 05FB9F7D  JMSCAN0Errs: jmp  [MSCAN0Errs-BootBlkSize,pcr]
0000F637 05FB9F7B  JMSCAN0WakeUp: jmp  [MSCAN0WakeUp-BootBlkSize,pcr]
0000F63B 05FB9F79  JFlash: jmp  [Flash-BootBlkSize,pcr]
0000F63F 05FB9F77  JEEPROM: jmp  [EEPROM-BootBlkSize,pcr]
0000F643 05FB9F75  JSPI2: jmp  [SPI2-BootBlkSize,pcr]
0000F647 05FB9F73  JSPI1: jmp  [SPI1-BootBlkSize,pcr]
0000F64B 05FB9F71  JICBus: jmp  [IICBus-BootBlkSize,pcr]
0000F64F 05FB9F6F  JDLC: jmp  [DLC-BootBlkSize,pcr]
0000F653 05FB9F6D  JSCEME: jmp  [SCMEVect-BootBlkSize,pcr]
0000F657 05FB9F6B  JCRGLock: jmp  [CRGLock-BootBlkSize,pcr]
0000F65B 05FB9F69  JPACCAv: jmp  [PACCAv-BootBlkSize,pcr]
0000F65F 05FB9F67  JModDnCtr: jmp  [ModDnCtr-BootBlkSize,pcr]
0000F663 05FB9F65  JPortHInt: jmp  [PortHInt-BootBlkSize,pcr]
0000F667 05FB9F63  JPortJInt: jmp  [PortJInt-BootBlkSize,pcr]
0000F66B 05FB9F61  JATD1: jmp  [ATD1-BootBlkSize,pcr]
0000F66F 05FB9F5F  JATD0: jmp  [ATD0-BootBlkSize,pcr]
0000F673 05FB9F5D  JSCI1: jmp  [SCI1-BootBlkSize,pcr]
0000F677 05FB9F5B  JSCI0: jmp  [SCI0-BootBlkSize,pcr]
0000F67B 05FB9F59  JSPI0: jmp  [SPI0-BootBlkSize,pcr]
0000F67F 05FB9F57  JPACCAEdge: jmp  [PACCAEdge-BootBlkSize,pcr]
0000F683 05FB9F55  JPACCAv: jmp  [PACCAv-BootBlkSize,pcr]
0000F687 05FB9F53  JTimerOv: jmp  [TimerOv-BootBlkSize,pcr]
0000F68B 05FB9F51  JTimerCh7: jmp  [TimerCh7-BootBlkSize,pcr]
0000F68F 05FB9F4F  JTimerCh6: jmp  [TimerCh6-BootBlkSize,pcr]
0000F693 05FB9F4D  JTimerCh5: jmp  [TimerCh5-BootBlkSize,pcr]
0000F697 05FB9F4B  JTimerCh4: jmp  [TimerCh4-BootBlkSize,pcr]
0000F69B 05FB9F49  JTimerCh3: jmp  [TimerCh3-BootBlkSize,pcr]
0000F69F 05FB9F47  JTimerCh2: jmp  [TimerCh2-BootBlkSize,pcr]
0000F6A3 05FB9F45  JTimerCh1: jmp  [TimerCh1-BootBlkSize,pcr]
0000F6A7 05FB9F43  JTimerCh0: jmp  [TimerCh0-BootBlkSize,pcr]
0000F6AB 05FB9F41  JRTI: jmp  [RTI-BootBlkSize,pcr]
0000F6AF 05FB9F3F  JIRQ: jmp  [IRQ-BootBlkSize,pcr]
0000F6B3 05FB9F3D  JXIRQ: jmp  [XIRQ-BootBlkSize,pcr]
0000F6B7 05FB9F3B  JSWI: jmp  [SWI-BootBlkSize,pcr]
0000F6BB 05FB9F39  JIllop: jmp  [Illop-BootBlkSize,pcr]
0000F6BF 05FB9F37  JCOPFail: jmp  [COPFail-BootBlkSize,pcr]
0000F6C3 05FB9F35  JClockFail: jmp  [ClockFail-BootBlkSize,pcr]

; 0000FF0D  org $ff0d
;  0000FF0D  CF    dc.b $cf            ; setup a 4K bootblock in Flash block 0.
;  0000FF0F  org $ff0f            ; location of security byte.
;  0000FF0F  FE    dc.b $fe            ; value of security byte for unsecured state.
;  0000FF8C  org $ff8c
;  0000FF8C F5E3  PWMESShutdown: dc.w JPWMESShutdown
<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000FF8E</td>
<td>PortPInt</td>
<td>dc.w</td>
<td>JPortPInt</td>
</tr>
<tr>
<td>0000FF90</td>
<td>MSCAN1Tx</td>
<td>dc.w</td>
<td>JMSCAN1Tx</td>
</tr>
<tr>
<td>0000FF92</td>
<td>MSCAN4Rx</td>
<td>dc.w</td>
<td>JMSCAN4Rx</td>
</tr>
<tr>
<td>0000FF94</td>
<td>MSCAN4Errs</td>
<td>dc.w</td>
<td>JMSCAN4Errs</td>
</tr>
<tr>
<td>0000FF96</td>
<td>MSCAN4WakeUp</td>
<td>dc.w</td>
<td>JMSCAN4WakeUp</td>
</tr>
<tr>
<td>0000FF98</td>
<td>MSCAN3Tx</td>
<td>dc.w</td>
<td>JMSCAN3Tx</td>
</tr>
<tr>
<td>0000FF9A</td>
<td>MSCAN3Rx</td>
<td>dc.w</td>
<td>JMSCAN3Rx</td>
</tr>
<tr>
<td>0000FF9C</td>
<td>MSCAN3Errs</td>
<td>dc.w</td>
<td>JMSCAN3Errs</td>
</tr>
<tr>
<td>0000FFE9</td>
<td>PortPInt</td>
<td>dc.w</td>
<td>JPortPInt</td>
</tr>
<tr>
<td>0000FFE9</td>
<td>MSCAN4Rx</td>
<td>dc.w</td>
<td>JMSCAN4Rx</td>
</tr>
<tr>
<td>0000FFEA</td>
<td>MSCAN4Errs</td>
<td>dc.w</td>
<td>JMSCAN4Errs</td>
</tr>
<tr>
<td>0000FFEB</td>
<td>MSCAN4WakeUp</td>
<td>dc.w</td>
<td>JMSCAN4WakeUp</td>
</tr>
<tr>
<td>0000FFEC</td>
<td>MSCAN3Tx</td>
<td>dc.w</td>
<td>JMSCAN3Tx</td>
</tr>
<tr>
<td>0000FFEE</td>
<td>MSCAN3Rx</td>
<td>dc.w</td>
<td>JMSCAN3Rx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JPortPInt</td>
<td>dc.w</td>
<td>JPortPInt</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN4Rx</td>
<td>dc.w</td>
<td>JMSCAN4Rx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN4Errs</td>
<td>dc.w</td>
<td>JMSCAN4Errs</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN4WakeUp</td>
<td>dc.w</td>
<td>JMSCAN4WakeUp</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN1Tx</td>
<td>dc.w</td>
<td>JMSCAN1Tx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN1Rx</td>
<td>dc.w</td>
<td>JMSCAN1Rx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN1Errs</td>
<td>dc.w</td>
<td>JMSCAN1Errs</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN1WakeUp</td>
<td>dc.w</td>
<td>JMSCAN1WakeUp</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN0Tx</td>
<td>dc.w</td>
<td>JMSCAN0Tx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN0Rx</td>
<td>dc.w</td>
<td>JMSCAN0Rx</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN0Errs</td>
<td>dc.w</td>
<td>JMSCAN0Errs</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JMSCAN0WakeUp</td>
<td>dc.w</td>
<td>JMSCAN0WakeUp</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JFlash</td>
<td>dc.w</td>
<td>JFlash</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>EEPROM</td>
<td>dc.w</td>
<td>JEEPROM</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>SPI2</td>
<td>dc.w</td>
<td>JSPI2</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>SPI1</td>
<td>dc.w</td>
<td>JSPI1</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>IICBus</td>
<td>dc.w</td>
<td>JIICBus</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JDLC</td>
<td>dc.w</td>
<td>JDLG</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>SCMEVect</td>
<td>dc.w</td>
<td>JSCMVE</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>CRGLock</td>
<td>dc.w</td>
<td>JCRLock</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>PACCBov</td>
<td>dc.w</td>
<td>JPACCBov</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>ModDnCtr</td>
<td>dc.w</td>
<td>JModDnCtr</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>PortHInt</td>
<td>dc.w</td>
<td>JPortHInt</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>PortJInt</td>
<td>dc.w</td>
<td>JPortJInt</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JATD1</td>
<td>dc.w</td>
<td>JATD1</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JATD0</td>
<td>dc.w</td>
<td>JATD0</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JSCI1</td>
<td>dc.w</td>
<td>JSCI1</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JSCI0</td>
<td>dc.w</td>
<td>JSCI0</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JSPI10</td>
<td>dc.w</td>
<td>JSPI10</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>PACCAEdge</td>
<td>dc.w</td>
<td>JPACCAEdge</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>PACCAov</td>
<td>dc.w</td>
<td>JPACCAov</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerOv</td>
<td>dc.w</td>
<td>JTimerOv</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh7</td>
<td>dc.w</td>
<td>JTimerCh7</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh6</td>
<td>dc.w</td>
<td>JTimerCh6</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh5</td>
<td>dc.w</td>
<td>JTimerCh5</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh4</td>
<td>dc.w</td>
<td>JTimerCh4</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh3</td>
<td>dc.w</td>
<td>JTimerCh3</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh2</td>
<td>dc.w</td>
<td>JTimerCh2</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh1</td>
<td>dc.w</td>
<td>JTimerCh1</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>TimerCh0</td>
<td>dc.w</td>
<td>JTimerCh0</td>
</tr>
<tr>
<td>0000FFF0</td>
<td>JRTI</td>
<td>dc.w</td>
<td>JRTI</td>
</tr>
</tbody>
</table>

For More Information On This Product, go to: www.freescale.com
0000FFF2 F6AF  IRQ: dc.w JIRQ
0000FFF4 F6B3  XIRQ: dc.w JXIRQ
0000FFF6 F6B7  SWI: dc.w JSWI
0000FFF8 F6BB  IIOp: dc.w JJIOp
0000FFFF F6BF  COPFail: dc.w JCOPFail
0000FFFC F6C3  ClockFail: dc.w JClockFail
0000FFE F000  Reset: dc.w BootStart

Errors: None
Labels: 472
Last Program Address: $0000FFFF
Last Storage Address: $FFFFFFFF
Program Bytes: $000006F4  1780
Storage Bytes: $000004E  78