

A Serial Bootloader for Reprogramming the MC68HC912B32 Flash EEPROM

By Gordon Doughman, Field Applications Engineer Software Specialist

1 Introduction

The MC68HC912B32 is a member of the M68HC12 family of 16-bit microcontrollers. It contains 32,768 bytes of bulk-erasable, byte- or word-programmable Flash EEPROM memory. Including Flash EEPROM, rather than EPROM or ROM, memory on a microcontroller has significant advantages for both the OEM and the end customer.

For the OEM, placing system firmware in Flash EEPROM memory provides numerous benefits. First, firmware development can be extended late into the product development cycle by eliminating the ROM lead times. Second, when an OEM has several products based on the same microcontroller, it can help reduce the inventory problems 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 EEPROM memory prevents the OEM from having to scrap any of the work-in-process.

The ability of Flash EEPROM memory to be electrically erased and reprogrammed also provides benefits for the OEM's end customers. The customers' products can be updated or enhanced with new features and capabilities without having to replace any components or return the product to the factory.

Unlike the the M68HC11 family, the MC68HC912B32 does not have a Bootstrap ROM containing firmware that allows initial programming of the Flash EEPROM directly through the on-chip Serial Communications Interface (SCI) port. Initial on-chip Flash EEPROM programming requires either special test and handling equipment to program the device before it is placed in the target system or a programming tool such as the SDI12 or the M68EVB912B32, available from Motorola, that is capable of programming the Flash EEPROM through the Real Time Background Debug interface.

The M68EVB912B32 on-chip Flash EEPROM, however, does contain a 2k-byte erase-protected bootblock. The bootblock may be used to contain a special bootloader program that allows erasure and programming of the remaining 30k of the on-chip Flash. In addition to implementing the Flash programming and erase algorithms, the serial bootloader firmware may contain a simple serial communications protocol that allows the use of the on-chip SCI port for obtaining the data to be programmed into the Flash.

Programming and erasing the on-chip Flash EEPROM memory of the MC68HC912B32 presents some unique challenges. Even though the on-chip Flash EEPROM memory has an erase-protected bootblock to contain the firmware implementing the programming and erase algorithms, the code cannot be run directly out of the Flash EEPROM bootblock while the remainder of the Flash array is being erased or programmed. Consequently, during the erase and reprogram process, the code must reside in other on-chip memory or in external memory. In addition, because the erase protected bootblock resides in the top 2k of the memory map (\$F800—\$FFFF), the reset and interrupt vectors cannot be changed without erasing the entire bootblock. This necessitates that a secondary reset/interrupt vector table be placed outside of the 2k bootblock.

The remainder of this application note will explore the requirements of a serial bootloader and the implementation of the programming algorithm for the MC68HC912B32 Flash EEPROM.



2 Overview of the MC68HC912B32's Flash EEPROM

The MC68HC912B32 Flash EEPROM module is arranged as a 16,384 x 16-bit module and may be read as bytes or aligned or misaligned words. Programming is accomplished only by writing bytes or aligned words. The Flash module requires an externally applied program/erase voltage (V_{FP}) to program or erase the array. The program/erase voltage is applied statically to the V_{FP} pin, however, the V_{FP} pin must always be kept at greater-than-or-equal to $V_{DD} - 0.5$ volts to prevent damage to the Flash array. To prevent the accidental erasure or programming of the Flash array, the V_{FP} should only be applied during the program/erase procedure.

Like most external Flash memory devices, the MC68HC912B32 Flash EEPROM module does not provide any automatic timing sequences during the erase or programming cycles. Programming or erasure is accomplished by a sequence of timed writes to the Flash control registers and a byte or aligned word write to the Flash array itself. The programming firmware is entirely responsible for the implementation of the erase and programming algorithms.

2.1 Erasure of the Flash EEPROM Array

Erasure of the MC68HC912B32 Flash EEPROM involves a procedure that can be divided into two parts. Erase pulses to the Flash array are applied by manipulating bits in the FEECTL register. After a pulse is applied, each location of the Flash array is checked for an erased state. When all locations in the Flash array are found to be in the erased state, or the maximum number of erase pulses have been applied, the same number of erase pulses required to erase the array are applied again. This procedure provides a 100% erase margin to the Flash array. After the margin pulses are applied, the Flash array should again be checked to ensure that it was properly erased. The simplified flowchart shown in Figure 1 describes these steps. Detailed descriptions and flowcharts, including timing requirements, describing the Flash erase procedure can be found in the *MC68HC912B32 Technical Summary* (document number MC68HC912B32TS/D).

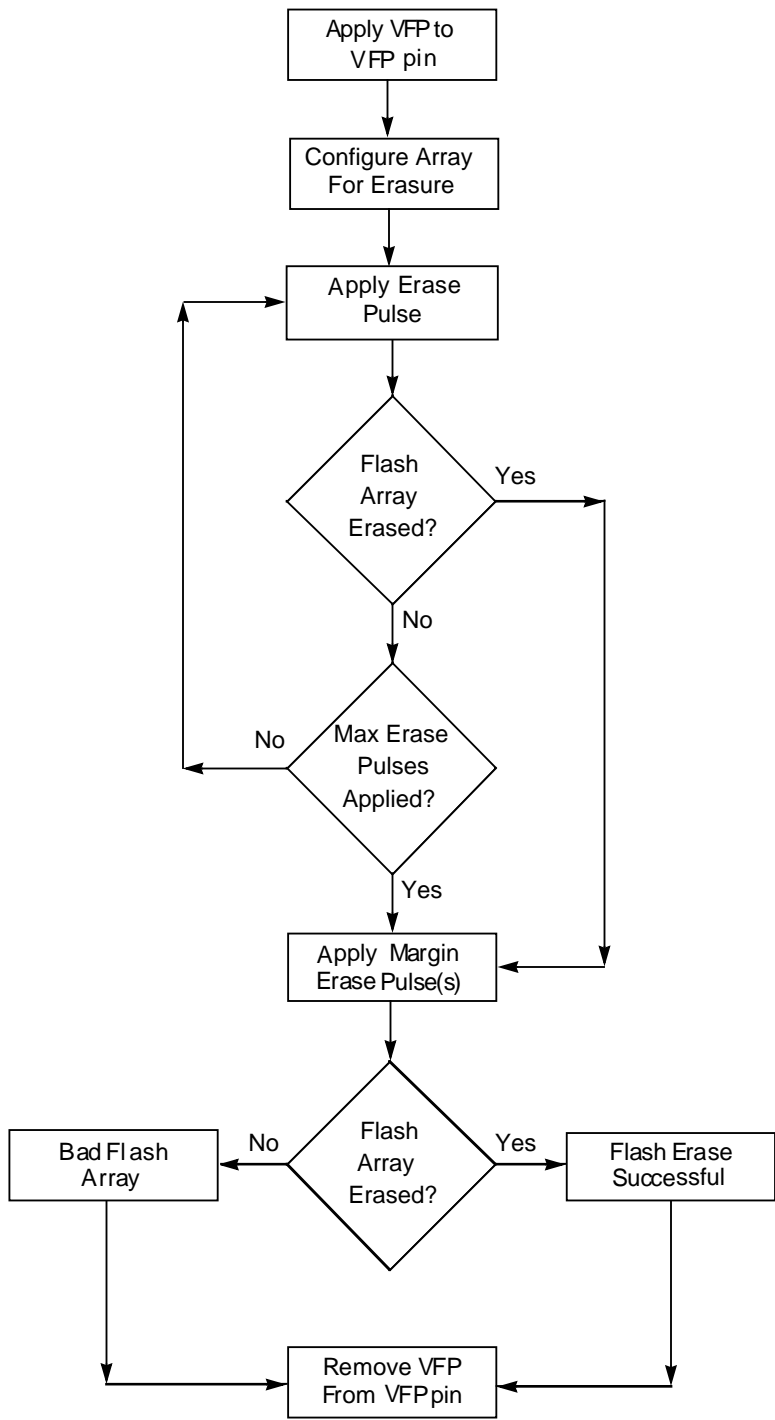


Figure 1 Simplified Flash Erase Algorithm Flowchart

2.2 Flash Array Programming

Programming the Flash array involves a procedure similar to the erase procedure. As mentioned previously, the MC68HC912B32 Flash may be programmed as either bytes or aligned words. Attempting to program a misaligned word of Flash memory will result in only the high byte (lower address) of the word being programmed into the Flash memory array. As with the erase procedure, programming the

Flash involves applying a series of programming pulses to the Flash array by manipulating bits in the FEECTL register. After each pulse is applied, the programmed location is checked to ensure that it contains the proper data. After the location reaches the proper value, or the maximum number of programming pulses have been applied, the same number of pulses required to program the array are applied again. The second set of programming pulses provides a 100% programming margin to the Flash memory location and ensures the integrity of the programmed data. The simplified flowchart shown in Figure 2 describes these steps. Detailed descriptions and flowcharts, including timing requirements, describing the Flash programming procedure can be found in the *MC68HC912B32 Technical Summary* (document number MC68HC912B32TS/D).

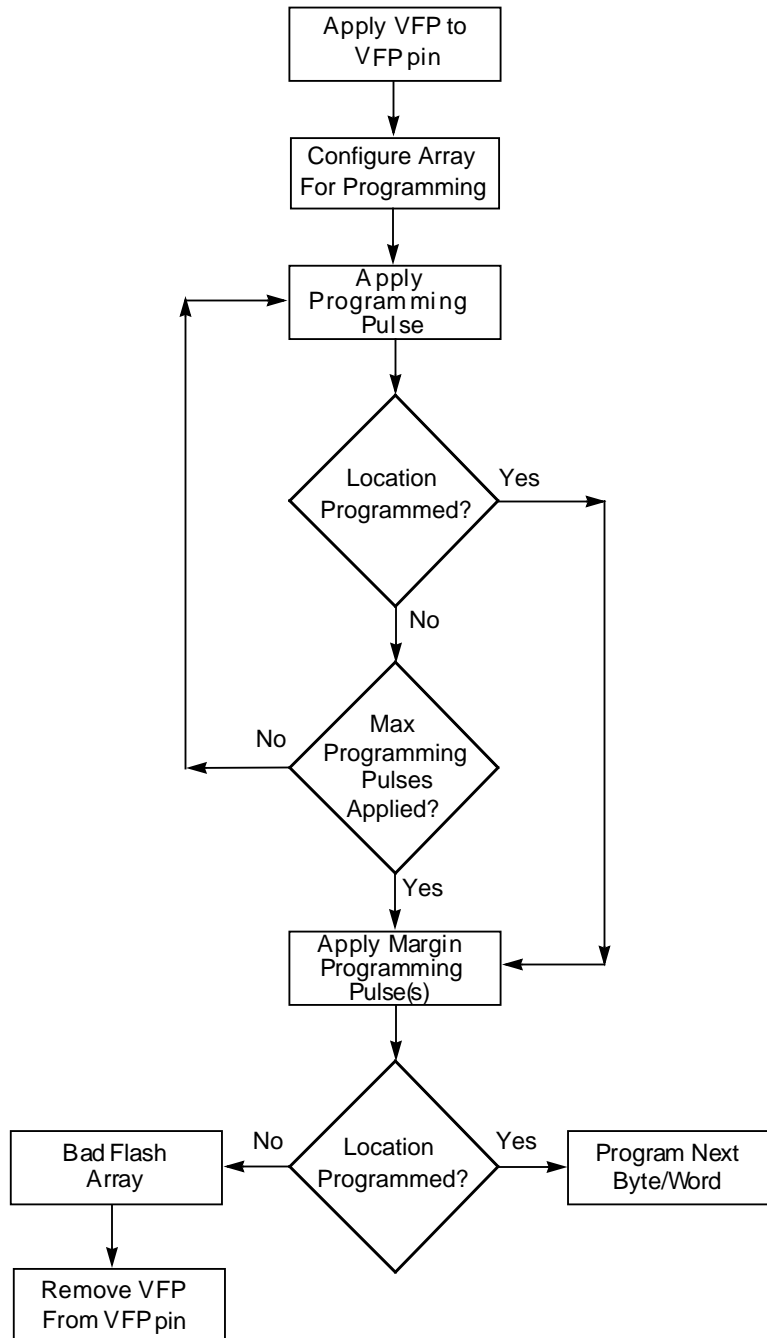


Figure 2 Simplified Flash Programming Algorithm Flowchart

3 General Flash Serial Bootloader Requirements

Two of the most important requirements for a program such as the Flash serial bootloader are that it have minimal impact on the final product's software performance and add little or nothing to the hardware costs. The Flash serial bootloader described in this application note meets both of these requirements.

Because the MC68HC912B32 includes an on-chip SCI, no additional external hardware is required to communicate with a host computer with the possible exception of an RS-232 level translator chip. In many systems, this may already be a part of the system design as the SCI is often used as a diagnostic port. If an RS-232 level translator is not included as part of the basic system design, a small adapter board could be constructed containing the level translator and RS-232 connector. This board could then be used by service personnel when updating the system firmware so that the cost of the level translator would not have to be added to each system. In addition to the SCI port, a single input pin is required to inform the serial bootloader startup code whether to execute the Flash serial bootloader code or jump to the system application program.

As mentioned previously, because the MC68HC912B32 interrupt and reset vectors reside in the 2k-byte bootblock, they cannot be changed without erasing the bootblock itself. Even though it is possible to erase and reprogram the bootblock from within the bootloader program, it is inadvisable to do so. If anything were to go wrong during the process of reprogramming the bootblock, it would be impossible to recover from the situation without the use of special programming hardware. For this reason, the serial bootloader includes a jump table that uses a secondary interrupt and reset vector table located just below the 2k bootblock. Each entry in the secondary interrupt table consists of a 2-byte address that mirrors the primary interrupt and reset vector table located in the erase-protected bootblock. Table 1 shows the correspondence between the primary and secondary interrupt vector tables.

Making use of the CPU12's indexed-indirect program counter relative addressing, each jump table entry consists of a single 4-byte JMP instruction. This form of the JMP instruction requires only six CPU clock cycles to execute, adding only 750 ns to the interrupt latency for a system operating at 8.0 MHz. In most applications this small amount of additional time will not affect the overall performance of the system.

Table 1 Primary/Secondary Interrupt Vector Addresses

Interrupt Vector Address	Interrupt Source	Secondary Vector Address
\$FFC0 – \$FFCF	Reserved	\$F7C0 – \$F7CF
\$FFD0 – \$FFD1	BDLC (J1850)	\$F7D0
\$FFD2 – \$FFD3	ATD	\$F7D2
\$FFD4 – \$FFD5	Reserved	\$F7D4
\$FFD6 – \$FFD7	SCI 0	\$F7D6
\$FFD8 – \$FFD9	SPI	\$F7D8
\$FFDA – \$FFDB	Pulse Acc. Input Edge	\$F7DA
\$FFDC – \$FFDD	Pulse Acc. Overflow	\$F7DC
\$FFDE – \$FFDF	Timer Overflow	\$F7DE
\$FFE0 – \$FFE1	Timer Channel 7	\$F7E0
\$FFE2 – \$FFE3	Timer Channel 6	\$F7E2
\$FFE4 – \$FFE5	Timer Channel 5	\$F7E4
\$FFE6 – \$FFE7	Timer Channel 4	\$F7E6
\$FFE8 – \$FFE9	Timer Channel 3	\$F7E8
\$FFEA – \$FFEB	Timer Channel 2	\$F7EA
\$FFEC – \$FFED	Timer Channel 1	\$F7EC
\$FFEE – \$FFEF	Timer Channel 0	\$F7EE
\$FFF0 – \$FFF1	Real Time Interrupt	\$F7F0
\$FFF2 – \$FFF3	IRQ	\$F7F2
\$FFF4 – \$FFF5	XIRQ	\$F7F4
\$FFF6 – \$FFF7	SWI	\$F7F6
\$FFF8 – \$FFF9	Illegal Opcode Trap	\$F7F8
\$FFFA – \$FFFB	COP Failure Reset	\$F7FA
\$FFFC – \$FFFD	Clock Mon. Fail Reset	\$F7FC
\$FFFE – \$FFFF	Reset	\$F7FE

4 Using The S-Record Bootloader

The S-Record bootloader utilizes the on-chip SCI for communications and does not require any special programming software for the host computer. The only host software required is a simple terminal program that is capable of communicating at 9600 baud and is able to wait for a prompt string before sending a line of text to the MC68HC912B32. The serial bootloader presents a simple command line interface to the user and accepts Motorola S-Record object files. The communications rate of 9600 baud was chosen simply because it is the most common baud rate available on a wide range of computing devices. However, the communication baud rate is the limiting factor in the length of time required to program the Flash. At 9600 baud, an S-record file containing 30k of object code requires approximately 90 seconds to be programmed into the Flash. If the communication rate were doubled to 19,200 baud or quadrupled it to 38,400 would cut the programming time by approximately one half or one quarter respectively.

Execution of the serial bootloader is selected by connecting port pin PDLC0 to a logic '0' level. Applying power to the target system or pressing the reset switch causes the bootloader to display the following prompt on the host terminal's screen:

```
(E)rase or (P)rogram:
```

Before selecting the Erase or Program function, V_{FP} must be applied to the V_{FP} pin of the MC68HC912B32.

4.1 Flash Erasure

Selecting the Erase function by typing an upper or lower case 'E' on the terminal will cause a bulk-erase of the Flash EEPROM array except for the 2k bootblock where the S-Record bootloader program resides. After the erase operation, a verify operation is performed to ensure that all locations are properly erased. If the erase operation was successful, the message 'Erased' is displayed on the screen and the bootloader's prompt is redisplayed.

If any locations were found to contain a value other than \$FF, the message 'Not Erased' is displayed on the terminal screen and the bootloader prompt is redisplayed. If the MC68HC912B32 device will not erase after one or two attempts, check the V_{FP} connection and measure the value of V_{FP} to ensure that it complies with the value published in the Technical Supplement *MC68HC912B32 Electrical Characteristics*. A V_{FP} voltage lower than that specified may cause the erase operation to fail. Applying a V_{FP} voltage higher than that specified may cause permanent damage to the device.

4.2 Flash Programming

The programming algorithm used for the on-chip FLASH memory is such that the time required to program each byte or word can vary from as little as 60 μ s to as long as 3.5 ms. However the programming time for each byte or word will typically take no more than 120–180 μ s. Because of this variability, the S-Record bootloader uses a software handshaking protocol to control the flow of S-Record data from the host computer. When the S-Record bootloader is ready to receive an S-Record, an ASCII asterisk character (*) is sent to the host computer. The host computer should respond by sending a single S-Record. The S-Record may include a carriage return and/or line feed character(s). Most commercial terminal programs capable of sending ASCII text files have the ability to wait for a specific character or string before sending a line of text.

Typing an upper or lower case 'P' on the terminal causes the bootloader to enter programming mode and wait for S-Records to be sent from the host computer. The host computer should begin by sending a single S-Record and then waiting for the bootloader to return an ASCII asterisk character (*) before sending subsequent S-Records.

The programming operation is terminated when the bootloader receives an 'S9' end-of-file record. If the S-Record object file being sent to the bootloader does not contain an 'S9' record, the bootloader will not return its prompt and will continue to wait for the end of file record. Pressing the target's reset switch, will cause the bootloader to return to its prompt.

If a Flash memory location will not program properly, the message 'Not Programmed' is displayed on the terminal screen and the bootloader's prompt is redisplayed. If problems are encountered when programming the Flash memory, check the V_{FP} connection to the target MCU and measure the value of V_{FP} to ensure that it complies with the value published in the MC68HC912B32 data sheet. A V_{FP} voltage lower than that specified may cause the programming operation to fail. Applying a V_{FP} voltage higher than that specified may cause permanent damage to the device.

If the V_{FP} connection is okay and V_{FP} is within the specified range, the problem may be caused by an S-Record containing data that is outside the range of the available on-chip Flash. The S-Record data must be within the range \$8000—\$F800.

Note: The S-Record bootloader should not be used with S-Records containing a code/data field longer than 64 bytes (S-Record length field greater than 67 (0 x 43) bytes). Sending an S-Record with a code/data field longer than 64 bytes (S-Record length field greater than 67 (0 x 43) bytes) will cause the bootloader to crash and/or program incorrect data into the Flash.

5 Bootloader Software

The software implementing the serial Flash bootloader, shown in Listing 1, consists of five basic parts: Startup code, secondary interrupt vector jump table, bootloader control loop, programming code and erase code.

5.1 Startup Code

At power up or reset, CPU control is transferred to the routine beginning at the label `BootStart`. This routine checks the state of PORTDLC bit number 6. If PORTDLC bit number 6 is equal to a logic '0', the code between the labels `BootLoad` and `BootLoadEnd` are copied from Flash into the on-chip RAM and CPU control is passed to the bootloader code in RAM. If PORTDLC bit number 6 is equal to a logic '1', CPU control is transferred to the program defined by the address in the secondary reset vector.

5.2 Bootloader Control Loop

The bootloader control loop begins by initializing the SCI and timer system. The SCI is initialized to 9600 baud, 8 data bits, 1 start bit, 1 stop bit and no parity. The timer system is enabled with the fast flag clear option and configures channel 0 for use as an output compare. The output compare function is used to produce accurate timing delays for both the programming and erase routines. Enabling the fast flag clear option allows the timer interrupt flag bit for channel 0 to be cleared simply by writing a new value to the channel 0 timer register.

After initialization of the hardware, the bootloader displays its prompt and waits for the erase or program command to be entered. If a letter other than 'E' or 'P' is entered, the bootloader prompt is simply redisplayed on the next line. After returning from the execution of either the erase or program command, a message is displayed indicating either success or failure. Execution is then transferred back to the top of the control loop where the command prompt is redisplayed.

5.3 Erase Command Code

The code implementing the erase command consists of a single subroutine beginning at the label `FErase` in Listing 1. The subroutine implements the Flash erase algorithm as described in the *MC68HC912B32 Technical Summary* (document number MC68HC912B32TS/D). Basically the algorithm involves applying successive 100 ms erase pulses to the Flash array until the array is erased or a maximum of five erase pulses have been applied. Once the array is erased, the same number of erase pulses that were required to erase the Flash array is applied once again to provide a 100% erase margin. Figure 3 contains a detailed flow chart of the *FErase* subroutine.

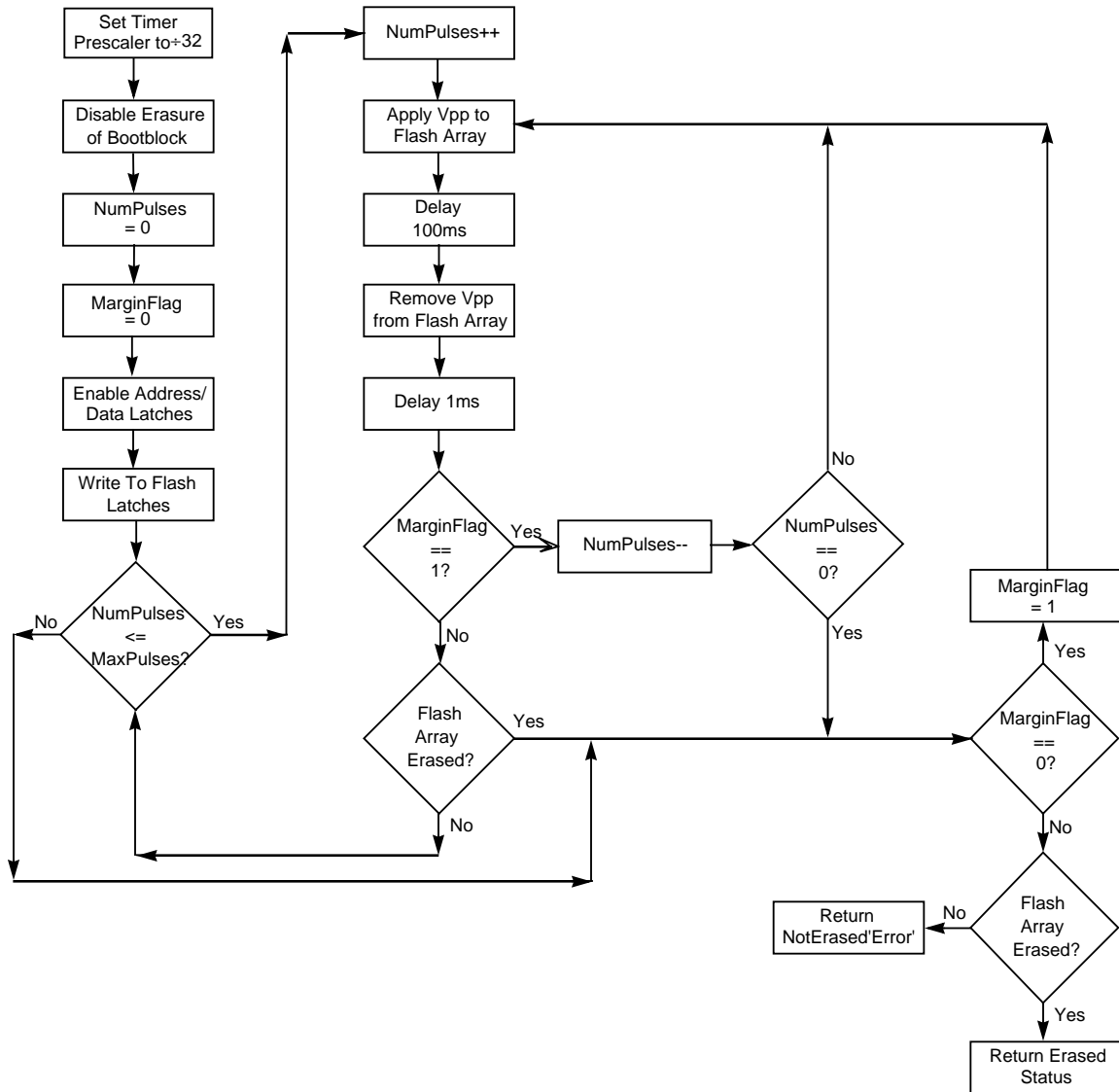


Figure 3 FErase Subroutine Flowchart

5.4 Program Command Code

The software required to implement the program command is more complex than the Flash erase routine and requires three major subroutines and several simple supporting subroutines. The main subroutine implementing the program command begins at the label `FProg` in Listing 1. This small subroutine, shown in the flowchart in Figure 4, simply coordinates the reception of S-Records, the programming of the S-Record data into the Flash and sending the 'pace' character to the host computer requesting that the host send the next S-Record. In addition, it checks the type of each S-Record received from the host, ignoring 'S0' records and terminating the command when an 'S9' record is received. Each time a valid 'S1' record is received, the `ProgFlashBlock` subroutine is called to program the received data into Flash. If an error occurs during the reception of an S-Record or during the Flash programming process, the program command is terminated.

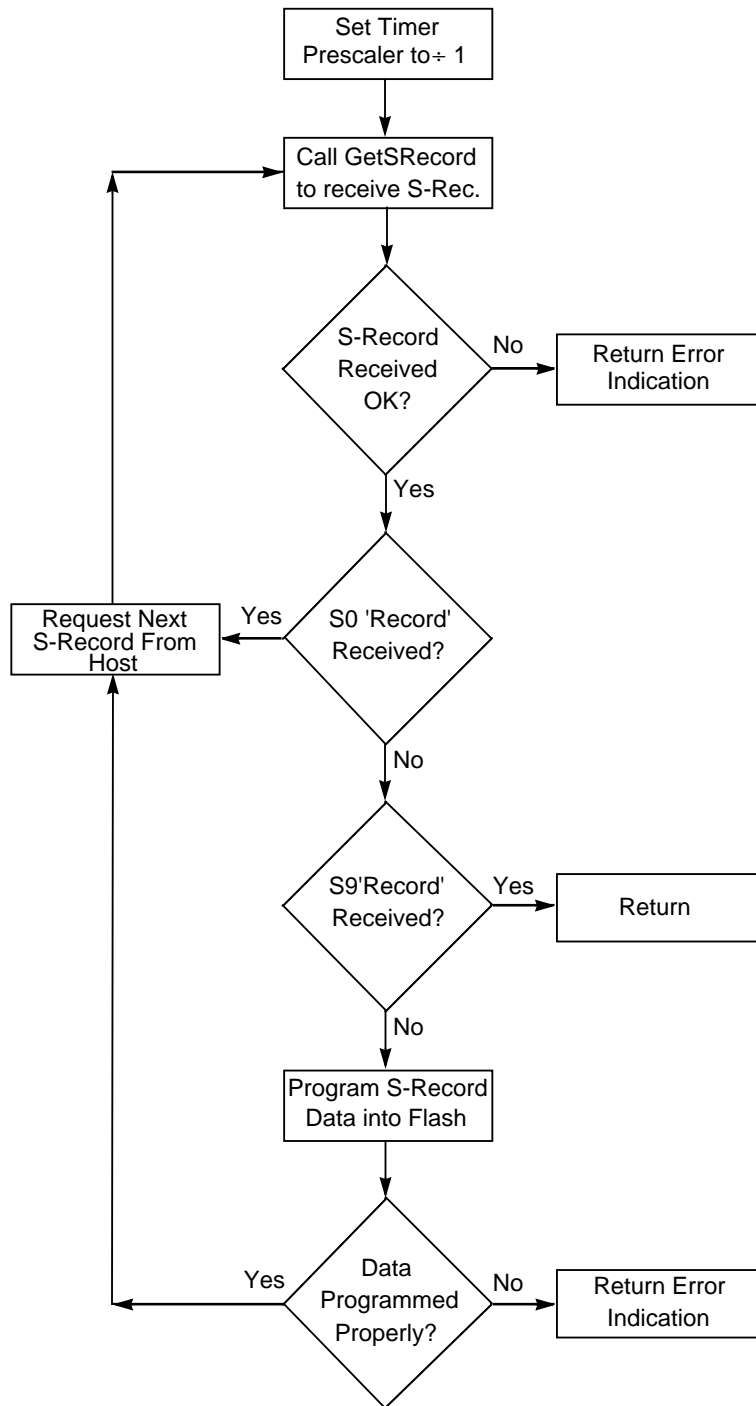


Figure 4 Program Command Flowchart

5.5 GetSRecord Subroutine

The `GetSRecord` subroutine is called by `FProg` to receive a single S-Record from the host computer. `GetSRecord` begins by allocating space on the stack for two local variables, `SRecBytes` and `CheckSum`. 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 code/data field and the checksum field. The variable `CheckSum` is used to contain the calculated checksum value as the

S-Record is received.

Next, the subroutine begins to receive characters from the host searching for the character pairs 'S0', 'S1' or 'S9' which indicate the start of a valid S-Record. Once a start of record is found, the S-Record length byte is received and saved in the local variable `SRecBytes`. Three is subtracted from the S-Record length byte and saved in the global variable `DataBytes`. This value represents the length of the code/data field and is used by the `ProgFBlock` subroutine when programming the S-Record data into the Flash. Finally, the load address, code/data field and the checksum field are received and placed in a global data buffer.

During the process of receiving the load address, code/data field and the checksum field each received byte is added to the `Checksum` local variable. Because the received checksum is actually the ones compliment of what the calculated checksum should be, adding the two values should produce a result of \$FF. The increment of the variable `Checksum` 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 will result in a 'equal' condition being returned if the S-Record was properly received and a 'not equal' condition being returned if there was a problem receiving the S-Record. Figure 5 contains the flowchart for the `GetSRecord` subroutine.

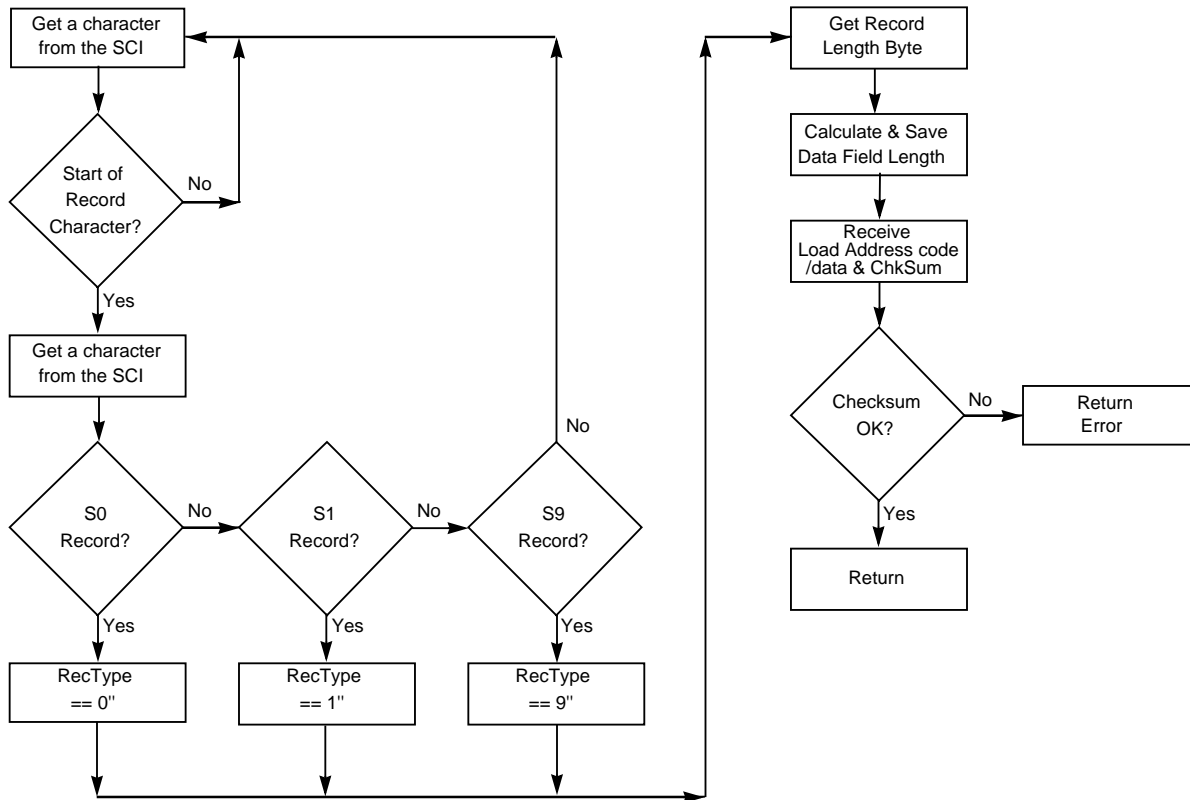


Figure 5 GetSRecord Subroutine Flowchart

5.6 ProgFBlock Subroutine

The `ProgFBlock` subroutine programs the data received by the `GetSRecord` subroutine into the on-chip Flash. The subroutine implements the Flash programming algorithm as described in the *MC68HC912B32 Technical Summary* (document number MC68HC912B32TS/D). Essentially the algo-

rithm involves applying successive 20–25 μs programming pulses to a byte or aligned word until the memory location is properly programmed or a maximum of 50 programming pulses. Once the memory location is programmed, the same number of pulses that were required to program the location are applied again to provide a 100% programming margin.

To simplify the implementation of the programming algorithm and to keep the bootloader code as small as possible, the `ProgFBlock` routine only programs a single byte of the Flash at a time. This may seem to impose a severe time penalty when programming 30k of Flash. However, the actual time saved would be extremely small in relation to the amount of time required to send an S-Record file containing 30k of object code. Consider, for example, that most Flash locations are able to be programmed with the application of three programming and three margin pulses. Therefore using a total time of 33 μs per byte, 22 μs programming time and 11 μs read/recovery time, would require 33 $\mu\text{s} \times 6 \times 30720$ or approximately 6.1 seconds to program 30K bytes a byte at a time. If words were programmed instead, the time would be cut approximately in half.

As mentioned previously, the communication baud rate is the limiting factor in the length of time required to program the Flash. Consider an S-Record file containing 30k of object code. If each S-Record contained 32 bytes in the code/data field, each S-Record would be comprised of 74 ASCII characters and the file would contain 960 S-Records for a total file size of 71040 bytes not counting carriage return and/or line feeds. Just transmitting this much ASCII data at 9600 baud would require approximately 74 seconds. This is more than an order of magnitude greater than the three seconds that would be saved by programming a word at a time. Even at a baud rate of 38,400 it would require approximately 19 seconds to transmit 71040 bytes.

The `ProgFBlock` routine begins by allocating space on the stack for two variables, `ProgPulses` and `PMarginFlag`. During programming, the `ProgPulses` variable is used to maintain a count of the number of programming pulses applied to each programmed byte. When applying the margin pulses, this value is decremented until it reaches zero. The `PMarginFlag` variable is used as a boolean flag to indicate that the programming margin pulses are being applied. When set to non-zero, it modifies the program flow so that the contents of the Flash memory is not compared to the S-Record data after the application of each margin pulse.

Like the `FErase` subroutine, channel 0 of the on-chip timer is used to produce the timing delays required for the programming pulses and the read/recovery period. However, because of the need to produce short, accurate time delays, the timer is used in a slightly different manner. Before each program and read/recovery cycle begins, the timer subsystem is disabled by clearing the Timer ENable (TEN) bit in the Timer Status and Control Register (TSCR). When the timer is disabled, the contents of all timer registers, including the value of the Timer CouNTER Register (TCNT), are maintained. This allows the software to read the static value of the TCNT register, add to it a value that will produce a delay of 22 μs and write the resulting value to the TC0 register without having to compensate for the intervening instruction execution time. The programming voltage is then applied to the array by setting the ENable Programming/Erase bit (ENPE) in the Flash EEPROM ConTroL register (FEECTL) and the timer system enabled. When the programming time period has expired, the programming voltage is removed from the Flash array. The timer is then setup to produce a delay of approximately 11 μs for the read/recovery period. Because this delay does not have to be as accurate as the programming pulse, the specification states a minimum of 10 μs , the timer system is not disabled when setting up the output compare register.

At the end of each program and read/recovery cycle, the Flash data is compared to the received S-Record data. If the two do not match, the program and read/recovery cycle is repeated until the data matches or the maximum number of programming pulses have been applied. Next, an equal number of program and read/recovery cycles are once again applied to the Flash memory location to provide a 100% programming margin. Finally, the Flash data is once again compared to the received S-Record data. If the two do not match, the `ProgFBlock` routine terminates returning a 'not equal' condition indicating that the programming operation failed.

Figure 6 contains a detailed flow chart of the ProgFlash subroutine.

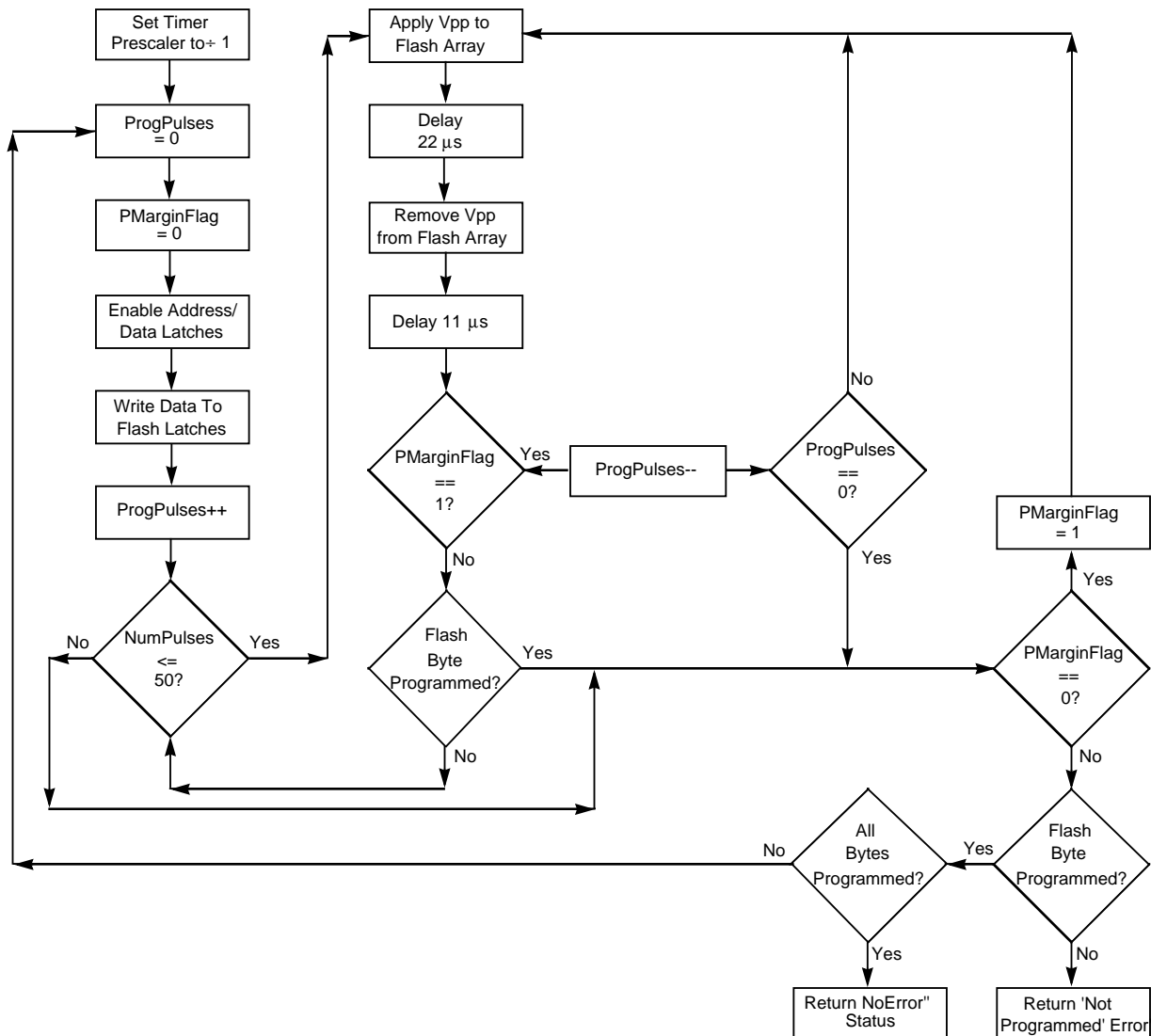


Figure 6 ProgFlash Subroutine Flowchart

5.7 Support Routines

Several additional support subroutines are required by the program and erase functions of the bootloader. The `getchar` and `putchar` subroutines provide SCI character I/O. The `GetHexByte`, `CvtHex`, and `IsHex` subroutines provide ASCII hexadecimal-to-binary conversion. The `OutStr` subroutine is used to send a null (0) terminated ASCII string to the on-chip SCI. It is called by the bootloader main loop to display the its prompt, error messages and command results. Because of the simplicity of these subroutines, no flowcharts are provided.

5.8 Secondary Reset/Interrupt Table

As noted previously, the bootloader supports a secondary reset/interrupt vector table that resides just below the 2k erase-protected bootblock. The jump table, located near the beginning of Listing 1, utilizes a form of indexed addressing that may not be supported by all assemblers. This addressing mode is a form of indexed indirect addressing that uses the program counter as an index register. The `pcr mne-`

monic used in place of an index register name stands for *Program Counter Relative* addressing. In reality, the CPU12 does not support an addressing mode known as Program Counter Relative or `pcr`. Instead, the CPU supports constant offsets from the value of the PC at the first byte of the next instruction. The PCR mnemonic is used to instruct the assembler to calculate an offset to the address specified by the expression preceding the `,pcr` index specification. The offset is calculated by subtracting the value of the PC at the address of the first object code byte of the next instruction from the value supplied in the index offset field. When the JMP instruction is executed, just the opposite occurs. The CPU12 adds the value of the PC at the first object code byte of the next instruction to the offset embedded in the instruction object code. The indirect addressing, indicated by the square brackets, specifies that the address calculated as the sum of the index register (in this case the PC) and the 16-bit offset contains a pointer to the destination of the JMP.

If an assembler does not support Program Counter Relative addressing the following substitution may be made. Replace the text between the square brackets of each JMP instruction with:

```
[(<InterruptVectorName> - $800) - (* + 4),pc]
```

Where `<InterruptVectorName>` represents the name of the interrupt vector as shown in each JMP instruction, the `(* + 4)` represents the value of the program counter at the beginning of the next instruction and `$800` is the offset from real interrupt vector to the secondary interrupt vector. This entire expression allows the assembler to calculate the proper offset to the interrupt relative to the value of the program counter.

5.9 Stack Space Allocation

Several of the subroutines in Listing 1 allocate storage space on the stack for temporary variables. These variables are accessed using indexed addressing with the stack pointer as the index register. The offsets to these variables are calculated using the facilities of the assembler and may not be available in all assemblers. As an example, the assembler source sequence that appears just before the `FProgBlock` subroutine is shown below.

```
CurrentPC    set    *           ; save the current value of the PC
             org    0           ; set PC to zero so we can use assembler to
                                 ; generate an offset into the stack.
;
ProgPulses:  ds     1           ; local variable to hold the number of
                                 ; programming pulses.
PMarginFlag: ds     1           ; local variable to indicate we're applying the
                                 ; margin pulses
;
             org    CurrentPC   ; restore the original value of the PC
```

In this example the `set` assembler directive is used to assign a value to the label `CurrentPC`. In this case it is assigning or saving the current value of the Program Counter. In this regards the `set` directive is similar to the `equ` directive. However, the `set` directive may be used to reassign a new value to a label. So the label `CurrentPC` may be used to save the current value of the program counter each time labels are declared for accessing local storage. Next, the program counter is then set to zero with the use of the `org` directive. The `ds` directive, normally used to reserve global variable storage, is simply used to advance the program counter, assigning 'offset' values for the labels `ProgPulses` and `PMarginFlag` that may be used to access the actual variables on the stack. Finally, the assembler's program counter is restored to its previous value through the use of the `org` assembler directive.

6 Program Listings

6.1 Listing 1 — Serial Flash Bootloader

-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 1

```
1          ;
2          ;
3 00FE      PORTDLC:      equ    $00fe          ; BDLC Port data register
4          ;
5 0016      COPCTL:      equ    $0016          ; COP timer control register.
6          ;
7 00C4      SRL:         equ    $00c4          ; SCIO status register #1.
8 00C7      DRL:         equ    $00c7          ; SCIO data register (low byte).
9 00C0      Baud:        equ    $00c0          ; SCIO baud rate register (16-bits).
10 00C3     CR2:         equ    $00c3          ; SCIO control register.
11         ;
12 0086     TSCR:        equ    $0086          ; timer status & control register.
13 0080     TIOS:        equ    $0080          ;
14 0084     TCNT:        equ    $0084          ; timer/counter register (16-bits).
15 008D     TMSK2:       equ    $008d          ; timer interrupt mask/prescaler control register.
16 008E     TPLG1:       equ    $008e          ; timer interrupt flag register.
17 0090     TCO:         equ    $0090          ; timer capture/compare register (16-bits).
18         ;
19 00F4     FEELCK:      equ    $00f4          ; Flash bootblock lock register.
20 00F5     FEEMCR:      equ    $00f5          ; Flash module configuration register.
21 00F7     FEECTL:      equ    $00f7          ; Flash erase/programming control register.
22         ;
23 0080     TDRE:        equ    $80            ; transmit data register empty bit.
24 0020     RDRF:        equ    $20            ; receive data register full bit.
25         ;
26 0010     FEESWAI:     equ    $10            ; Disable FLASH array in WAIT mode bit in FEECTL register
27 0008     SVFP:        equ    $08            ; Flash programming voltage present bit in FEECTL register
28 0004     ERAS:        equ    $04            ; Flash Erase bit in FEECTL register
29 0002     LAT:         equ    $02            ; Address/Data latch enable bit in FEECTL register
30 0001     ENPE:        equ    $01            ; Flash programming voltage enable bit in FEECTL register
31         ;
32 0080     TEN:         equ    $80            ; Timer enable bit.
33         ;
34         ;
;*****
35         ;
36         ;Constants
37         ;
38 1200     EClock:      equ    800000         ; E-clock frequency in Hz.
39 0034     Baud9600:    equ    800000/16/9600 ; value for baud register, based on clock frequency.
40 61A8     mS100:      equ    EClock/320     ; timer delay constant for 100 mS delay based on /32 prescaler.
41 00FA     mS1:        equ    EClock/32000   ; timer delay constant for 1 mS delay based on /32 prescaler.
42 00B0     us22:       equ    ((EClock/10000)*22)/100 ; timer delay constant for 22 uS delay based on /1 prescaler.
43 0058     uS11:       equ    ((EClock/10000)*11)/100 ; timer delay constant for 11 uS delay based on /1 prescaler.
44         ;
45 8000     FlashStart: equ    $8000          ; Flash EEPROM start address (Single chip).
46 8000     FlashSize:  equ    32768         ; Flash size for 912B32.
47 0800     BootBlkSize: equ    2048         ; Erase protected bootblock size.
48 0032     MaxProgPulses: equ    50         ; maximum number of programming pulses.
49 0005     MaxErasePulses: equ    5         ; maximum number of erase pulses.
50         ;
51 0800     RAMStart:   equ    $800           ; start address of on-chip RAM.
52 0400     RAMSize:   equ    $400           ; size of on-chip RAM.
53 0C00     StackTop:  equ    RAMStart+RAMSize ; address to initialize the stack pointer.
54         ;
55 0030     S0RecType:  equ    '0'           ; ASCII '0' used as S0 record type indicator.
56 0031     S1RecType:  equ    '1'           ; ASCII '1' used as S1 record type indicator.
57 0039     S9RecType:  equ    '9'           ; ASCII '9' used as S9 record type indicator.
58         ;
59         ;
60         ;
61         ;
;*****
62         ;
63         ;
64 FC00     org          $fc00
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 2

65         ;
66         ;
67 FC00 CF0C00      BootStart:      lds    #StackTop          ; initialize the stack pointer
68 FC03 4FFE4004    brclr  PORTDLC,$40,BootCopy ; PortDLC bit #6 == 0?
69 FC07 05FBFBF3    jmp    [Reset-$800,pcr] ; no. jump to the users program pointed to by the the secondary
70         ;
71         ;
72 FC0B 790016      BootCopy:      clr    COPCTL          ; disable watchdog
73 FC0E CEF7C       ldx    #BootLoad      ; point to the start of the Flash bootloader in Flash.
74 FC11 CD0800      ldy    #RAMStart       ; point to the start of on-chip RAM.
75 FC14 CCFECE      ldd    #BootLoadEnd    ; calculate the size of the bootloader code.
76 FC17 83FC7C      subd   #BootLoad
77 FC1A 180A3070    MoveMore:      movb  1,x+,1,y+      ; move a byte of the bootloader into RAM.
78 FC1E 0434F9      dbne  d,MoveMore     ; dec byte count, move till done.
79 FC21 060800      jmp    RAMStart      ; execute the bootloader code.
80         ;
81         ;
```

```

;*****
82 ;
83 ;
84 ; This is the jump table that is used to access the secondary interrupt vector table. Each one
85 ; of the actual interrupt vectors, beginning at $ffd0, points to an entry in this table. Each jmp
86 ; instruction uses indexed indirect program counter relative (pcr) addressing to access the
87 ; secondary interrupt vector table that is located just below the 2k bootblock.
88 ;
;*****
89 ;
90 ;
91 FC24 05FBFB88 JBDLC: jmp [BDLC-$800,pcr]
92 FC28 05FBFB86 JATD: jmp [ATD-$800,pcr]
93 FC2C 05FBFB84 JSCIO: jmp [SCIO-$800,pcr]
94 FC30 05FBFB82 JSPI: jmp [SPI-$800,pcr]
95 FC34 05FBFB80 JPACCIE: jmp [PACCIE-$800,pcr]
96 FC38 05FBFB7E JPACCOV: jmp [PACCOV-$800,pcr]
97 FC3C 05FBFB7C JTimerOv: jmp [TimerOv-$800,pcr]
98 FC40 05FBFB7A JTimerCh7: jmp [TimerCh7-$800,pcr]
99 FC44 05FBFB78 JTimerCh6: jmp [TimerCh6-$800,pcr]
100 FC48 05FBFB76 JTimerCh5: jmp [TimerCh5-$800,pcr]
101 FC4C 05FBFB74 JTimerCh4: jmp [TimerCh4-$800,pcr]
102 FC50 05FBFB72 JTimerCh3: jmp [TimerCh3-$800,pcr]
103 FC54 05FBFB70 JTimerCh2: jmp [TimerCh2-$800,pcr]
104 FC58 05FBFB6E JTimerCh1: jmp [TimerCh1-$800,pcr]
105 FC5C 05FBFB6C JTimerCh0: jmp [TimerCh0-$800,pcr]
106 FC60 05FBFB6A JRTI: jmp [RTI-$800,pcr]
107 FC64 05FBFB68 JIRQ: jmp [IRQ-$800,pcr]
108 FC68 05FBFB66 JXIRQ: jmp [XIRQ-$800,pcr]
109 FC6C 05FBFB64 JSWI: jmp [SWI-$800,pcr]
110 FC70 05FBFB62 JIlop: jmp [Ilop-$800,pcr]
111 FC74 05FBFB60 JCOPFail: jmp [COPFail-$800,pcr]
112 FC78 05FBFB5E JClockFail: jmp [ClockFail-$800,pcr]
113 ;
114 ;
115 ;
;*****
116 ;
117 ; The code residing between the labels BootLoad and BootLoadEnd comprises the bootloader code
118 ; that is copied into RAM. The bootloader must execute from the on-chip RAM because the Flash
119 ; array is not accessible while it is being programmed or erased. The bootloader code was
120 ; written in a position independent manner so that it will execute properly when copied into RAM.
121 ;
122 ;
;*****
123 ;
124 ;
125 FC7C BootLoad: equ *
126 FC7C CC0034 ldd #Baud9600 ; set SCI to 9600 baud @ 8.0 MHz
127 FC7F 5CC0 std Baud
128 FC81 C60C ldab #$0c ; enable the transmitter & receiver.
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 3

129 FC83 5BC3 stab CR2
130 FC85 C601 ldab #$01 ; disable the erasure or programming of the 2k bootblock.
131 FC87 5BF4 stab FEELCK
132 FC89 C6B0 ldab #$b0 ; enable the timer system. set for fast flag clears.
133 FC8B 5B86 stab TSCR
134 FC8D C601 ldab #$01 ; enable timer channel 0 as an output compare.
135 FC8F 5B80 stab TIOS
136 FC91 1AFA006C BLLoop: leax BLPrompt,pcr ; point to the bootloader prompt.
137 FC95 15FA022C jsr OutStr,pcr ; display it.
138 FC99 15FA01F6 jsr getchar,pcr ; get the command from the user.
139 FC9D 15FA01F9 jsr putchar,pcr ; echo it.
140 FCA1 37 pshb ; save it.
141 FCA2 1AFA0058 leax CrLfStr,pcr ; go to the next line.
142 FCA6 15FA021B jsr OutStr,pcr
143 FCAA 33 pulb ; restore the entered character.
144 FCAB C4DF andb #$df ; simple convert to upper case (only works for alpha characters).
145
146 FCAD C145 CheckFErase: cmpb #'E' ; erase command entered?
147 FCAF 261A bne ChkProg ; no. go check for the program command.
148 FCB1 15FA003A jsr CheckVfp,pcr ; yes. check for Vfp present.
149 FCB5 26DA bne BLLoop ; go print prompt if not present.
150 FCB7 15FA0118 jsr FErase,pcr ; yes. go erase the Flash.
151 FCBB 1AFA005A leax ENot,pcr ; point to the 'not erased' string.
152 FCBF 2604 bne BadErase ; branch if it didn't erase properly.
153 FCC1 1AFA0058 leax Erased,pcr ; if it did, point to the 'erased' string
154 FCC5 15FA01FC BadErase: jsr OutStr,pcr
155 FCC9 20C6 bra BLLoop ; go back & print the prompt again.
156 ;
157 FCCB C150 ChkProg: cmpb #'P' ; program command entered?
158 FCCD 26C2 bne BLLoop ; no. go redisplay the command prompt.
159 FCCF 15FA001C jsr CheckVfp,pcr ; yes. check for Vfp present.
160 FCD3 26BC bne BLLoop ; go print prompt if not present.
161 FCD5 15FA006C jsr FProg,pcr ; yes. go program the Flash.
162 FCD9 39 EEProgStat: pshc ; save the returned success/fail condition.
163 FCDA 1AFA0020 leax CrLfStr,pcr ; go to the next line.
164 FCDE 15FA01E3 jsr OutStr,pcr
165 FCE2 38 pulc ; restore the returned success/fail condition.
166 FCE3 1AFA003D leax FNot,pcr ; point to the 'not programmed' string.
167 FCE7 26DC bne BadErase ; go display the string if programming failed.
168 FCE9 1AFA003B leax Programmed,pcr ; otherwise, point to the 'programmed' string.
169 FCED 20D6 BadProg: bra BadErase ; go display the prompt again.

```

```

170 ;
171 ;
172 ;
;*****
173 ;
174 ; The CheckVfp subroutine checks the SVFP bit in the FEECTL register to see if Vfp has been applied
175 ; to the Vfp pin. If Vfp is present, a zero or equal condition is returned. If Vfp is not present,
176 ; a not zero or not equal condition is returned.
177 ;
178 ;
;*****
179 ;
180 FCEF 87 CheckVfp: clra ; assume that Vfp is present (set Z == 1).
181 FCF0 4EF70809 brset FEECTL,SVFP,VfpOK ; programming voltage present?
182 FCF4 1AFA003B leax NoVfpError,pcr ; no. inform the user.
183 FCF8 15FA01C9 jsr OutStr,pcr
184 FCFC 42 inca ; return Z == 0 (not zero condition)
185 FCFD 3D VfpOK: rts
186 ;
187 ;
188 ;
;*****
189 ;
190 FCFE 0D0A00 CrLfStr: fcb $0d,$0a,0
191 FD01 0D0A28452972 BLPrompt: fcb $0d,$0a,"(E)rase or (P)rogram:",0
192 FD19 4E6F7420 ENot: fcb "Not "
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 4

193 FD1D 457261736564 Erased: fcb "Erased",0
194 FD24 4E6F7420 PNot: fcb "Not "
195 FD28 50726F677261 Programmed: fcb "Programmed",0
196 FD33 0D0A56667020 NoVfpError: fcb $0d,$0a,"Vfp Not Present",0
197 ;
198 ;
199 ;
;*****
200 ;
201 FD45 FProg: equ *
202 FD45 C600 ldab #$00 ; set the prescaler to /1.
203 FD47 5B8D stab TMSK2
204 FD49 2006 bra FSkipFirst ; don't send the 'pace' character the first time.
205 FD4B C62A FSendPace: ldab #'*' ; the ascii asterisk is the pace character.
206 FD4D 15FA0149 jsr putchar,pcr ; tell the host it's ok to send the next S-Record.
207 FD51 15FA00E4 FSkipFirst: jsr GetSRecord,pcr ; go get the S-Record.
208 FD55 2612 bne ProgDone ; non-zero condition means there was an error
209 FD57 E6FA0174 ldab RecType,pcr ; check the record type.
210 FD5B C139 cmpb #S9RecType ; was it an S9 record?
211 FD5D 270A beq ProgDone ; yes. we're done.
212 FD5F C130 cmpb #S0RecType ; no. was it an S0 record?
213 FD61 27E8 beq FSendPace ; yes. just ignore it.
214 FD63 15FA0003 jsr ProgFBlock,pcr ; no. that means it was an S1 record. go program the data into Flash.
215 FD67 27E2 beq FSendPace ; zero condition means all went ok.
216 FD69 3D ProgDone: rts ; if we fall through, we automatically return a non-zero condition.
; if we get here after detecting an S9 record, we'll return a zero condition.
217 ;
218 ;
219 ;
220 ;
;*****
221 ;
222 ;
223 FD6A CurrentPC set * ; save the current value of the PC
224 0000 org 0 ; set PC to zero so we can use assembler to generate an
offset into the stack.
225 ;
226 0000 ProgPulses: ds 1 ; local variable to hold the number of programming pulses.
227 0001 PMarginFlag: ds 1 ; local variable to indicate we're applying the margin pulses
228 ;
229 FD6A org CurrentPC
230 ;
231 FD6A ProgFBlock: equ *
232 FD6A 3B pshd ; easy way to allocate 2 bytes on the stack.
233 FD6B EEFA0162 ldx LoadAddr,pcr ; get the S-Record (Flash) load address.
234 FD6F 19FA0160 leay SRecData,pcr ; point to the received S-Record data.
235 FD73 6980 ProgLoop: clr ProgPulses,sp ; initialize the ProgPulses local variable.
236 FD75 6981 clr PMarginFlag,sp ; initialize the PMarginFlag local variable.
237 FD77 4CF702 bset FEECTL,LAT ; turn on the Flash address/data latches.
238 FD7A 180A4000 movb 0,y,0,x ; put the data into the latches.
239 FD7E 4D8680 PPulseLoop: bclr TSCR,TEN ; stop the timer so we can produce accurate time delays.
240 FD81 6280 inc ProgPulses,sp ; add 1 to the number of programming pulses we've applied.
241 FD83 E680 ldab ProgPulses,sp ; get the new value.
242 FD85 C132 cmpb #MaxProgPulses ; have we applied the maximum allowable programming pulses?
243 FD87 2304 bls PMarginLoop ; no. go apply a programming pulse.
244 FD89 18088101 movb #1,PMarginFlag,sp ; yes. now try applying 'MaxProgPulses' of margin.
245 FD8D CC00B0 PMarginLoop: ldd #us22 ; get the constant for a 22 uS delay.
246 FD90 D384 addd TCNT ; add it to the current value of the timer counter register.
247 FD92 5C90 std TC0 ; initialize the output compare register with the delay value.
248 FD94 4CF701 bset FEECTL,ENPE ; turn on Vfp
249 FD97 4C8680 bset TSCR,TEN ; turn on the timer.
250 FD9A 4F8E01FC brclr TFLG1,$01,* ; wait here until Vfp has been applied for 22 uS.
251 FD9E 4DF701 bclr FEECTL,ENPE ; turn off Vfp.
252 FDA1 CC0058 ldd #us11 ; get the constant for a 11 uS delay.
253 FDA4 D384 addd TCNT ; add it to the current value of the timer counter register.
254 FDA6 5C90 std TC0 ; initialize the output compare register with the delay value.
255 FDA8 4F8E01FC brclr TFLG1,$01,* ; wait here until Vfp has been removed for 11 uS.

```

```

256 FDAC E781          tst      PMarginFlag,sp      ; are we applying the programming margin pulses?
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 5

257 FDAE 2706          beq      CmpData              ; no. go see if the data programmed properly.
258 FDB0 6380          dec      ProgPulses,sp       ; yes. have we applied margin pulses equal to the number
of programming pulses?
259 FDB2 26D9          bne      PMarginLoop         ; no. go apply more margin pulses.
260 FDB4 200C          bra      PMarginDone         ; yes. go check the data again.
261                    ;
262 FDB6 E600          CmpData: ldab      0,x          ; get the data from the Flash memory.
263 FDB8 E140          cmpb     0,y                  ; same as the S-Record data?
264 FDBA 26C2          bne      PPulseLoop          ; no. go apply some more programming pulses.
265 FDBC 18088101      movb     #1,PMarginFlag,sp    ; yes. set the programming margin flag.
266 FDC0 20CB          bra      PMarginLoop         ; go apply the margin programming pulses.
267                    ;
268 FDC2 4DF702        PMarginDone: bclr     FEECTL,LAT      ; turn off the Flash address/data latches to prepare for
programming the next location.
269 FDC5 E630          ldab     1,x+                 ; get the data from the Flash memory for a final compare.
270 FDC7 E170          cmpb     1,y+                 ; same as the S-Record data?
271 FDC9 2606          bne      PDone                ; no. bad Flash memory (or Vfp not applied).
272 FDCB 63FA0101      dec      DataBytes,pcr        ; done with all the S-Record bytes?
273 FDCF 26A2          bne      ProgLoop            ; no. program the next location.
274 FDD1 3A           PDone:    puld                ; deallocate the locals.
275 FDD2 3D           rts                          ; return.
276                    ;
277                    ;
;*****
278                    ;
279 FDD3          CurrentPC      set      *          ; save the current value of the PC
280 0000          org      0          ; set PC to zero so we can use assembler to generate an
offset into the stack.
281                    ;
282 0000          NumPulses:  ds      1          ; local variable to hold the number of erase pulses.
283 0001          EMarginFlag: ds      1      ; local variable to indicate we're applying margin erase pulses.
284 0002          NotErasedFlag: ds    1      ; local variable to indicate that the Flash array is not erased.
285                    ;
286 FDD3          org      CurrentPC
287                    ;
288 FDD3          FErase:    equ      *
289 FDD3 1B9D      leas     -3,sp              ; allocate stack space for locals.
290 FDD5 C605      ldab     #$05                ; set the prescaler to /32.
291 FDD7 5B8D      stab     TMSK2
292 FDD9 6981      clr      EMarginFlag,sp     ; clear the margin pulse flag.
293 FDEB 6980      clr      NumPulses,sp       ; clear the erase pulse count.
294 FDDD 4CF706    bset     FEECTL,LAT+ERAS     ; turn on the address/data latches & erase bit.
295 FDE0 7C8000    std      FlashStart         ; write to any Flash address (data doesn't matter).
296                    ;
297 FDE3 E680      EraseLoop: ldab     NumPulses,sp ; get the 'pulse' count
298 FDE5 C105      cmpb     #MaxErasePulses    ; applied the maximum number of erase pulses?
299 FDE7 2738      beq      DoEMargin          ; yes. go apply the erase margin pulse.
300 FDE9 6280      inc      NumPulses,sp       ; add 1 to the number of 100 mS 'pulses' to apply
301 FDEB 4CF701    PulseLoop: bset     FEECTL,ENPE        ; turn on Vfp.
302 FDEE CC61A8    ldd      #mS100              ; timer constant to produce a 100 mS delay
303 FDF1 D384      addd     TCNT                ; add it to the current value of the timer.
304 FDF3 5C90      std      TC0                 ; initialize the output compare register.
305 FDF5 4F8E01FC brclr    TFLG1,$01,*         ; check for the output compare flag to be set.
306 FDF9 4DF701    bclr     FEECTL,ENPE        ; no turn off Vfp
307 FDFC CC00FA    ldd      #mS1                ; timer constant to produce a 1 mS delay
308 FDFE D384      addd     TCNT                ; add it to the current value of the timer.
309 FE01 5C90      std      TC0                 ; initialize the output compare register.
310 FE03 4F8E01FC brclr    TFLG1,$01,*         ; check for the output compare flag to be set.
311 FE07 E781      tst      EMarginFlag,sp     ; are we applying margin erase pulses?
312 FE09 2704      beq      CheckErase         ; no. go check to see if the last pulse erased the array.
313 FE0B 6380      dec      NumPulses,sp       ; yes. have we applied enough margin pulses?
314 FE0D 26DC      bne      PulseLoop          ; no. go apply some more.
315                    ;
316 FE0F 6982      CheckErase: clr      NotErasedFlag,sp ; clear the erased flag
317 FE11 CE8000    ldx      #FlashStart         ; point to the start of the flash block.
318 FE14 CD3C00    ldy      #(FlashSize-BootBlkSize)/2 ; get a count of the number of words we're going to check.
319 FE17 CFFFFF    ldd      #$FFFF              ; the value of an erased word.
320 FE1A AC31      EraseChkLoop: cpd      2,x+    ; this word erased?
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 6

321 FE1C 260B          bne      NotErased           ; no. go set flag & apply another erase pulse.
322 FE1E 0436F9        dbne     y,EraseChkLoop     ; yes. decrement word count & go check the next word.
323                    ;
324 FE21 E781          DoEMargin: tst      EMarginFlag,sp     ; have we already applied the margin pulse?
325 FE23 260C          bne      EraseDone          ; yes. we're done. the result of the erase function is in
the NotErasedFlag.
326 FE25 6281          inc      EMarginFlag,sp     ; no. set the 'margin pulse applied' flag.
327 FE27 20C2          bra      PulseLoop          ; go apply the margin erase pulse.
328                    ;
329 FE29 6282          NotErased: inc      NotErasedFlag,sp ; array was not erased. flag the condition.
330 FE2B E781          tst      EMarginFlag,sp     ; have we already applied the margin pulse?
331 FE2D 2602          bne      EraseDone          ; yes. we're done. the Flash is bad.
332 FE2F 20B2          bra      EraseLoop          ; haven't yet applied the margin pulse. go apply another erase pulse.
333                    ;
334 FE31 7900F7        EraseDone: clr      FEECTL          ; make sure that the LAT & ERAS bit is clear.
335 FE34 E682          ldab     NotErasedFlag,sp   ; get the erase result.
336 FE36 1B83          leas     3,sp                ; get rid of the locals.
337 FE38 3D           rts                          ; return.
338                    ;
339                    ;

```

```

340
;*****
341 ;
342 FE39 CurrentPC set * ; save the current value of the PC
343 0000 org 0 ; set PC to zero so we can use assembler to generate an
offset into the stack.
344 ;
345 0000 SRecBytes: ds 1 ; holds the number of bytes in the received S-Record.
346 0001 CheckSum: ds 1 ; used for calculated checksum.
347 ;
348 FE39 org CurrentPC
349 ;
350 FE39 GetSRecord: equ *
351 FE39 1B9E leas -2,sp ; allocate stack space for variables.
352 FE3B 15FA0054 LookForSOR: jsr getChar,pcr ; get a character from the receiver.
353 FE3F C153 cmpb #'S' ; start-of-record character?
354 FE41 26F8 bne LookForSOR ; no. go back & get another character.
355 FE43 15FA004C jsr getChar,pcr ; yes. we found the start-of-record character (ASCII 'S')
356 FE47 C130 cmpb #S0RecType ; found an S0 (header) record?
357 FE49 2708 beq SaveRecType ; no. go check for an S9 record.
358 ;
359 FE4B C139 CheckForS9: cmpb #S9RecType ; found an S9 (end) record?
360 FE4D 2704 beq SaveRecType ; no. go check for an S1 record.
361 ;
362 FE4F C131 ChkForS1: cmpb #S1RecType ; found an S1 (code/data) record?
363 FE51 26E8 bne LookForSOR ; no. false start-of-record character received. go check for another.
364 FE53 6BFA0078 SaveRecType: stab RecType,pcr ; yes. set the record type to '1'
365 FE57 15FA0046 jsr GetHexByte,pcr ; get the S-Record length byte.
366 FE5B 2620 bne BadSRec ; return if there was an error.
367 FE5D 6B80 stab SRecBytes,sp ; save the total number of S-Record bytes we are to receive.
368 FE5F 6B81 stab CheckSum,sp ; initialize the checksum calculation with the data byte count
369 FE61 C003 subb #3 ; subtract the load address & checksum field from the data field count.
370 FE63 6BFA0069 stab DataBytes,pcr ; save the code/data field size.
371 FE67 1AFA0066 leax LoadAddr,pcr ; point to the load address/code/data/checksum buffer.
372 FE6B 15FA0032 RcvData: jsr GetHexByte,pcr ; get an S-Record data byte.
373 FE6F 260C bne BadSRec ; return if there was an error.
374 FE71 6B30 stab 1,x+ ; save the byte in the data buffer.
375 FE73 EB81 addb CheckSum,sp ; add the byte into the checksum.
376 FE75 6B81 stab CheckSum,sp ; save the result.
377 FE77 6380 dec SRecBytes,sp ; received all the S-Record bytes?
378 FE79 26F0 bne RcvData ; no. go get some more.
379 FE7B 6281 inc CheckSum,sp ; if checksum was ok, the result will be zero.
380 FE7D 1B82 BadSRec: leas 2,sp
381 FE7F 3D rts
382 ;
383;*****
384 ;
-- Micro Dialects, Inc. uASM-HC12 Assembler Tue, Mar 18, 1997 3:10 PM -- Page 7

385 FE80 IsHex: equ *
386 FE80 C130 cmpb #'0' ; less than ascii hex zero?
387 FE82 250E blo NotHex ; yes. character is not hex. return a non-zero ccr indication.
388 FE84 C139 cmpb #'9' ; less than or equal to ascii hex nine?
389 FE86 2308 bls IsHex1 ; yes. character is hex. return a zero ccr indication.
390 FE88 C141 cmpb #'A' ; less than ascii hex 'A'?
391 FE8A 250E blo NotHex ; yes. character is not hex. return a non-zero ccr indication.
392 FE8C C146 cmpb #'F' ; less than or equal to ascii hex 'F'?
393 FE8E 2202 bhi NotHex ; yes. character is hex. return a non-zero ccr indication.
394 FE90 1404 IsHex1: orcc #S04 ; no. return a zero ccr indication.
395 FE92 3D NotHex: rts
396 ;
397 ;
;*****
398 ;
399 FE93 getChar: equ *
400 FE93 4FC420FC brclr SR1,RDRF,* ; loop waiting for the RDRF bit to be set.
401 FE97 D6C7 ldab DRL ; retrieve the character.
402 FE99 3D rts ; return.
403 ;
404 ;
;*****
405 ;
406 FE9A putChar: equ *
407 FE9A 4FC480FC brclr SR1,TDRE,* ; loop waiting for the TDRE bit to be set.
408 FE9E 5BC7 stab DRL ; send the character.
409 FEA0 3D rts ; return.
410 ;
411 ;
;*****
412 ;
413 FEA1 GetHexByte: equ *
414 FEA1 07F0 bsr getChar ; get the upper nybble from the SCI.
415 FEA3 07DB bsr IsHex ; valid hex character?
416 FEA5 2701 beq OK1 ; yes. go convert it to binary.
417 FEA7 3D rts ; no. return with a non-zero ccr indication.
418 FEA8 0712 OK1: bsr CvtHex ; convert the ascii-hex character to binary.
419 FEAA 8610 ldaa #16 ; shift it to the upper 4-bits.
420 FEAC 12 mul
421 FEAD 37 pshb ; save it on the stack.
422 FEAE 07E3 bsr getChar ; get the lower nybble from the SCI.
423 FEB0 07CE bsr IsHex ; valid hex character?
424 FEB2 2702 beq OK2 ; yes. go convert it to binary.
425 FEB4 33 pulb ; remove saved upper byte from the stack.
426 FEB5 3D rts ; no. return with a non-zero ccr indication.

```

```

427 FEB6 0704      OK2:      bsr      CvtHex      ; convert the ascii-hex character to binary.
428 FEB8 EBB0      ;          addb     1,sp+      ; add it to the upper nybble.
429 FEBA 87        ;          clra      ; simple way to set the Z ccr bit.
430 FEBB 3D        ;          rts       ; return.
431 ;
432 ;
;*****
433 ;
434 FEBC C030      CvtHex:      subb     #'0'      ; subtract ascii '0' from the hex character.
435 FEBE C109      ;          cmpb     #$09      ; was it a decimal digit?
436 FEC0 2302      ;          bls      CvtHexRtn  ; yes. ok as is.
437 FEC2 C007      ;          subb     #$07      ; no. it was an ascii hex letter ('A' - 'F').
438 FEC4 3D        ;          CvtHexRtn:  rts
439 ;
440 ;
;*****
441 ;
442 FEC5          OutStr:      equ      *          ; send a null terminated string to the display.
443 FEC5 E630      ;          ldab     1,x+      ; get a character, advance pointer, null?
444 FEC7 2705      ;          beq      OutStrDone ; yes. return.
445 FEC9 15F9CE    ;          jsr      putchar,pcr ; no. send it out the SCI.
446 FECC 20P7      ;          bra      OutStr     ; go get the next character.
447 FECE 3D        ;          OutStrDone:  rts
448 ;
-- Micro Dialects, Inc. uASM-HC12 Assembler   Tue, Mar 18, 1997 3:10 PM -- Page 8

449 ;
450 ;
;*****
451 ;
452 FECF          BootLoadEnd: equ      *
453 ;
454 ;
455 ;Global Variable declarations
456 ;
457 ;
458 FECF          RecType:      ds      1          ; received record type. ascii '0' = S0; ascii '1' = S1; ascii '9' = S9
459 FED0          DataBytes:    ds      1          ; number of data bytes in the S-Record.
460 FED1          LoadAddr:     ds      2          ; load address of the S-Record.
461 FED3          SRecData:     ds      65         ; S-Record data storage. (handle 64-byte S-Records + received checksum)
462 ;
463 ;
;*****
464 ;
465 ;
466 ;
467 FFD0          ;          org      $ff00
468 ;
469 FFD0 FC24      BDLC:        dw      JB DLC
470 FFD2 FC28      ATD:         dw      JATD
471 FFD4 FFFF      ;          dw      $ffff
472 FFD6 FC2C      SCIO:        dw      JSCIO
473 FFD8 FC30      SPI:         dw      JSPI
474 FFDA FC34      PACCCIE:     dw      JPACCIE
475 FFDC FC38      PACCOv:     dw      JPACCOv
476 FFDE FC3C      TimerOv:     dw      JTimerOv
477 FFE0 FC40      TimerCh7:    dw      JTimerCh7
478 FFE2 FC44      TimerCh6:    dw      JTimerCh6
479 FFE4 FC48      TimerCh5:    dw      JTimerCh5
480 FFE6 FC4C      TimerCh4:    dw      JTimerCh4
481 FFE8 FC50      TimerCh3:    dw      JTimerCh3
482 FFEA FC54      TimerCh2:    dw      JTimerCh2
483 FFEC FC58      TimerCh1:    dw      JTimerCh1
484 FFEE FC5C      TimerCh0:    dw      JTimerCh0
485 FFF0 FC60      RTI:         dw      JRTI
486 FFF2 FC64      IRQ:         dw      JIRQ
487 FFF4 FC68      XIRQ:        dw      JXIRQ
488 FFF6 FC6C      SWI:         dw      JSWI
489 FFF8 FC70      Illop:       dw      JIloop
490 FFFA FC74      COPFail:     dw      JCOPFail
491 FFFC FC78      ClockFail:   dw      JClockFail
492 FFFE FC00      Reset:        dw      BootStart
493 ;
494 0000          ;          end

```

```

Errors: None
Labels: 155
Last Program Address: $FFFF
Last Storage Address: $FFFF
Program Bytes: $02FF 767
Storage Bytes: $004C 76

```


Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

MCUinit, MCUasm, MCUdebug, and RTEK are trademarks of Motorola, Inc. MOTOROLA and M are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;

P.O. Box 5405, Denver Colorado 80217. 1-800-441-2447, (303) 675-2140

Mfax™: RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609, U.S. and Canada Only 1-800-774-1848

INTERNET: <http://Design-NET.com>

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC,

6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 81-3-3521-8315

ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,

51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

Mfax is a trademark of Motorola, Inc.



MOTOROLA

AN1718/D

