Application Note

AN2295/D
Rev. 4, 10/2003

Developer's Serial Bootloader for M68HC08

MOTOROLA intelligence everywhere™ | digital dna™

By: Pavel Lajsner
Motorola Czech System Laboratories
Roznov p. R., Czech Republic

Project Objectives

The Developer's Serial Bootloader for M68HC08 allows in-circuit reprogramming of Motorola's M68HC08 FLASH devices using standard communication media (e.g., a serial asynchronous port). Once the MCU is programmed with the bootloader, the MCU memory can be modified in-circuit. Because of its ability to modify MCU memory in-circuit, the serial bootloader is an M68HC08 MCU utility that may be useful in developing applications.

This application note is written for embedded software developers interested in alternative reprogramming tools. The Developer's Serial Bootloader is not intended to compete with existing MON08 development tools; it is more of a complementary utility for demo purposes, or for applications originally developed using MMDS and requiring minor modifications to be done in-circuit. The serial bootloader offers a zero-cost solution to applications already equipped with a serial interface and that have the SCI pins available on a connector.

This document also describes other programming techniques, including:

- FLASH reprogramming using ROM routines
- Simple software SCI
- Usage of the internal clock generator
- PLL clock programming

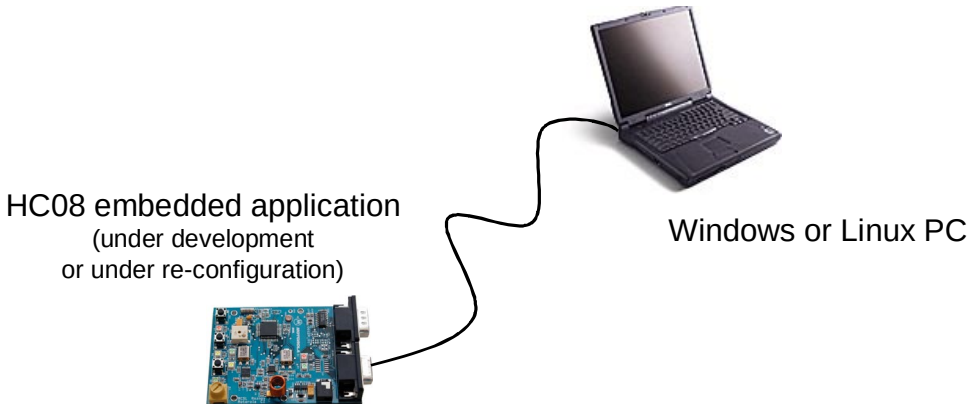


Figure 1. Top Level View

Project Goals

Motorola M68HC08 MCUs use a standard monitor mode interface for FLASH programming. Configuration of the monitor mode requires a specific clock and requires high voltage (monitor mode entry voltege $V_{tst} = V_{DD} + 2.5 = 8$ V) applied to the $\overline{IRQ}$ pin upon MCU startup. Also, establishing monitor mode communication consumes a few pins. When these requirements become obstacles, and if the application already uses a standard serial SCI interface for communication, a different code (the bootloader) can be used to communicate with the PC using the same interface that is used for reprogramming.

Note that the bootloader can be used only for reprogramming, not for in-circuit debugging. The bootloader is a low-cost, in-circuit programming solution.

Requirements

The described bootloader must fit these requirements:

- **Low occupation of memory —** Naturally, some memory will be consumed by the bootloader. The intention is keep this amount as low as possible. Other versions of bootloaders use more than 1 KB of memory, which is unacceptable on devices with 3 KB of memory available such as the MC68HC908JK3. The solution described here implements all features as simply as possible (e.g., excluding checksums, etc.). The target size is below 500 B.
- **Low pin-count** — The original intention is to use standard (already implemented) means of communication (typically SCI on boards that are primarily intended for communication). The standard SCI uses two different wires (RxD, TxD). No additional wires are used to start bootloader.
- **Transparency with respect to the user S19 file** — The complete application should be transparent to the user code S19 file. This means that no adjustments are required in the S19 file. Other known HC08 bootloader applications usually require modifications to interrupt vectors or require other tweaks to the S19 file for it to accept the bootloader.

Demo Features of Bootloader Application

This document describes several different M68HC08 bootloader implementations which vary mainly because the target M68HC08 MCUs are equipped with different features. Several features of the M68HC08 Family are also demonstrated, making this document useful to a wider audience than those who only require the bootloader. The different M68HC08 implementations also demonstrate the following features:

- Usage of built-in ROM routines for FLASH self programming (see also the application note *Using MC68HC908 On-Chip FLASH Programming Routines, ROM-Resident Routines in the MC68HC908GR8, MC68HC908KX8, MC68HC908JL3, MC68HC908JK3, and the MC68HC908JB8,* Motorola document order number AN1831/D).
- User implementation of in-circuit re-programming routines on ROM-less HC08 MCUs (e.g., the HC908GP32 family)
- Usage of different implementations of the FLASH block protection technique (M68HC908GP/GR/KX vs. M68HC908JK/JL Families)
- Implementation of software SCI on SCI-less HC08 MCUs (e.g., the M68HC908JK/JL Family)
- Usage of the internal clock generator and its trimming (M68HC908KX Family)

FC Protocol Description

This section provides a description of the protocol that is used to communicate between the PC and target MCU in order to reprogram the MCU. A non-specific MCU description is followed by an explanation of M68HC08 specific implementation features.

A simplified state diagram shows separate states of the bootloader, which are described in detail later in this document.

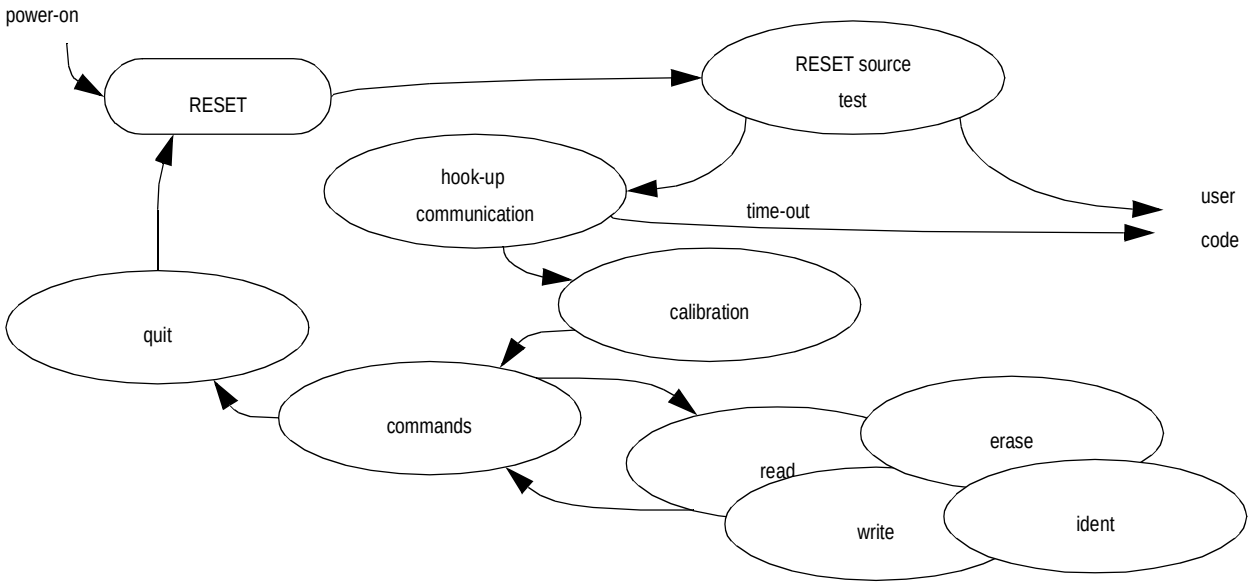


Figure 2. Simplified Flow Diagram of the Bootloader Application

FC Protocol

The requirements mentioned in the **Requirements** section dictate an implementation be as simple as possible, with low memory consumption. Because of that, the protocol running between the master PC and slave MCU is also very simple. It is called FC protocol because one significant character (the acknowledge, or ACK) $FC or 11111100b is thoroughly used. The reasons for such a specific selection are described below.

Initial Hook-Up

There are several methods to enter bootloader mode. Several other solutions use a *certain level on certain pin* method. An example of this method is: If logical 0 appears on an IRQ pin during the MCU's startup, then the bootloader code is started. Otherwise, the user code is begun.

Since another requirement for the developer's serial bootloader application is to use the lowest number of pins, a *certain character at a certain time* method is used. This means that the MCU sends out an ACK character through the serial interface and waits for an answer. If no character is received within the specified time (hook-up time-out), the process continues with the user code.

NOTE: *If this becomes a limitation for some reason, the user may modify the bootloader code to meet his/her needs (e.g., a simple IRQ pin test at startup can be easily implemented). See more in **System Limitations**.*

Here the protocol allows two scenarios, depending on whether the MCU is running on a known and exact frequency or if it uses an RC clock or internal clock (or basically a clock which is not known at compile time).

Unknown MCU Communication Speed

If the frequency is uncertain (not known at compile time), the MCU will not check that an incoming ACK character conforms only to the $FC pattern. Due to the MCU clock tolerance, several characters can be interpreted differently instead of the original $FC sent out by the PC as described below:

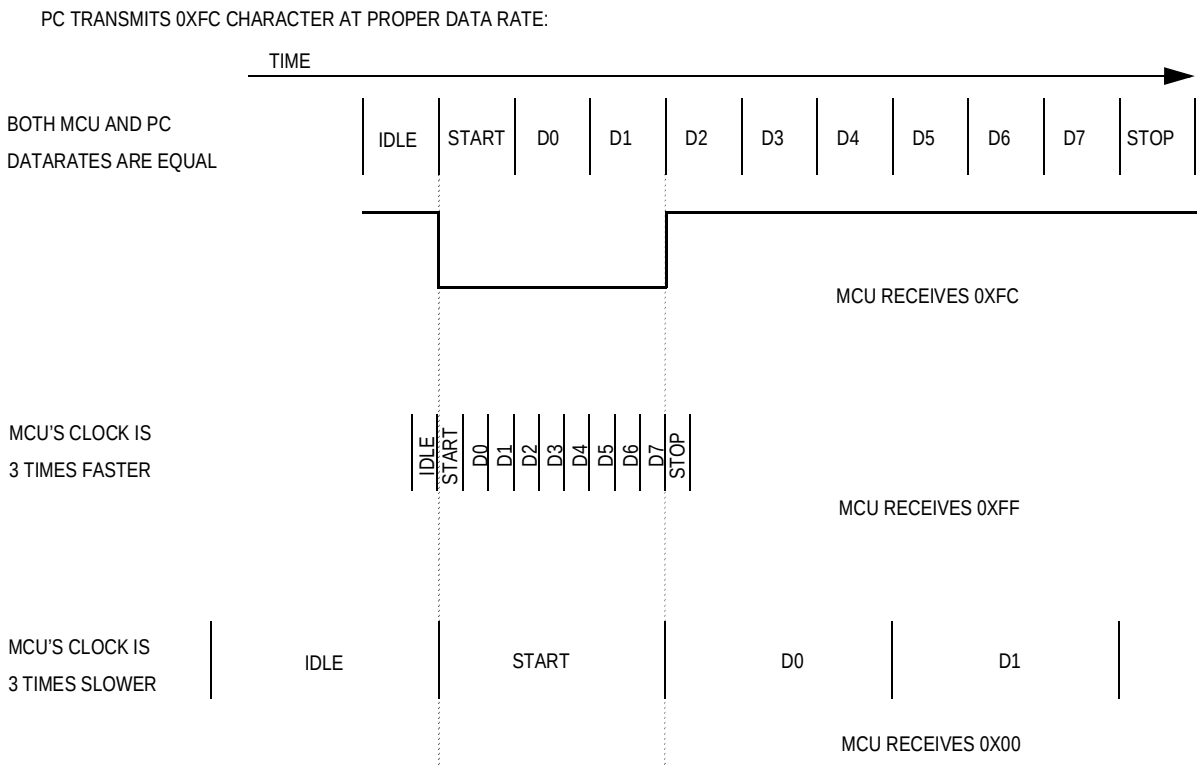


Figure 3. Matching Different Communication Speeds

Table 1 shows the characters that can still be correctly received (i.e., without framing or noise errors) if transmit and receive speeds are not equal.

Table 1. PC to MCU Transmission — Unmatched Data Rate

PC Data Rate	MCU Data Rate	Character Received in Binary	Character Received in Hex
9600	9600*1/3	11111111b	0xFF
9600	9600*2/3	11111110b	0xFE
9600	9600*3/3	11111100b	0xFC
9600	9600*4/3	11111000b	0xF8
9600	9600*5/3	11110000b	0xF0
9600	9600*6/3	11100000b	0xE0
9600	9600*7/3	11000000b	0xC0
9600	9600*8/3	10000000b	0x80
9600	9600*9/3	00000000b	0x00

NOTE: *The $FC pattern check on the MCU side can be eliminated completely, thus saving more MCU memory.*

The same situation also occurs in reverse, as in when the MCU transmits to the PC at an unmatched data rate. The PC then receives (and accepts) characters that are different from the $FC character. So, the PC accepts all characters from the mentioned set [$FF, $FE, $FC, $F8, $F0, $E0, $C0, $80, $00]. If such a character is received, an answering ACK is sent back to the MCU immediately. After the MCU recognizes this answer, it enters the next phase: **Slave Frequency Calibration**.

Known MCU Communication Speed

If the frequency is certain (known at compile time), the MCU will be configured to exactly match the communication speed of the PC. All characters are received correctly and without any distortion.

The MCU sends $FC to the PC, which immediately answers back to the MCU. After the ACK is received, the MCU also (formally) enters **Slave Frequency Calibration** phase.

Slave Frequency Calibration

During this phase, MCU clock calibration is accomplished. Up to now, the PC has communicated with the MCU at a speed which could be from 33% to 300% tolerance. This part must adjust the MCU communication speed to match that of the PC.

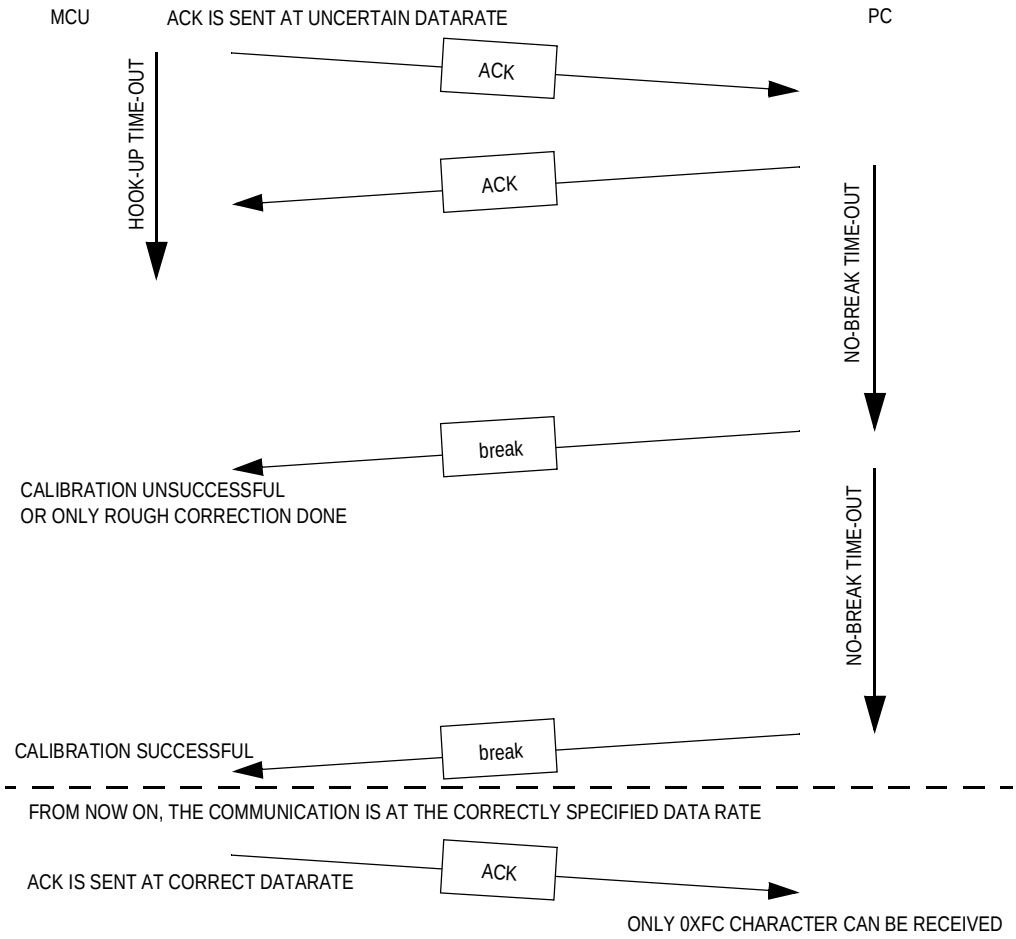


Figure 4. Start-Up Communication with Calibration

After the PC enters the calibration phase, the no-break time-out starts. If within this period a correct ACK character ($\$0xFC$) is not received, a break character is sent at the communication data rate.

NOTE: *A break character consists of 10 successive logical zeros. For example, at a 9600 baud data rate, its high-low-high pulse lasts 10 x 104* μ*s = 1.04 ms.*

The MCU then measures the length of the break character and determines, whether its clock is too fast or too slow. The MCU then makes an adjustment to its system clock (or an adjustment of receive routines, if, for example, software serial communication is employed). This can be repeated as many times as needed for the MCU to achieve the proper clock speed.

After the MCU identifies (measures) that it operates at the correct clock (or after the receive routines are calibrated), the ACK character is sent to the PC to stop sending more calibration characters. See **Figure 4**.

If the MCU knows that it operates at the correct data rate (no calibration is possible or needed; the MCU clock is crystal driven), it can send an ACK immediately, skipping the calibration phase entirely.

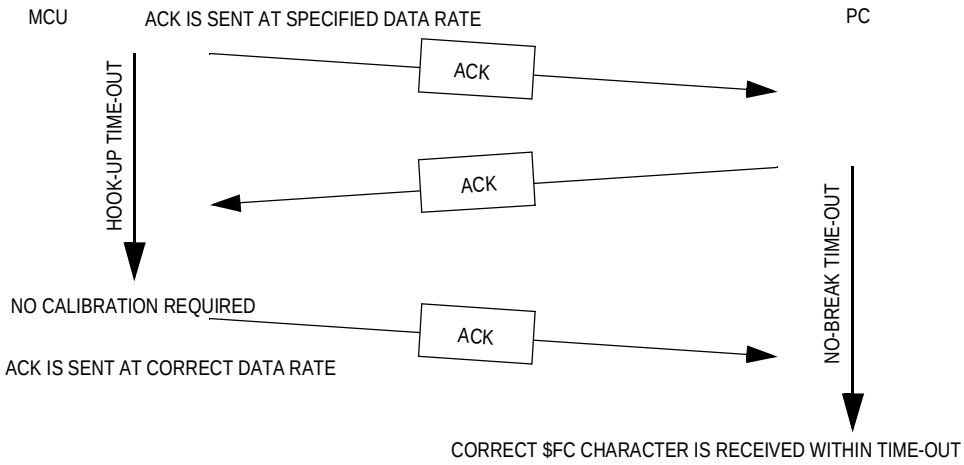


Figure 5. Start-Up Communication Without Calibration

All further communication is at the specified data rate.

Interpreting MCU Commands

After the communication between the MCU and the PC is established, the MCU enters the main command interpreter loop.The MCU executes one of a few simple commands needed to reprogram its own nonvolatile memory. The communication is conducted on a master-slave mechanism, where the PC issues the commands, the MCU executes them and acknowledges the completion of each command, either by data or by a single ACK character.

The minimal set of commands is comprised of:

- **Ident Command**
- **Quit Command**

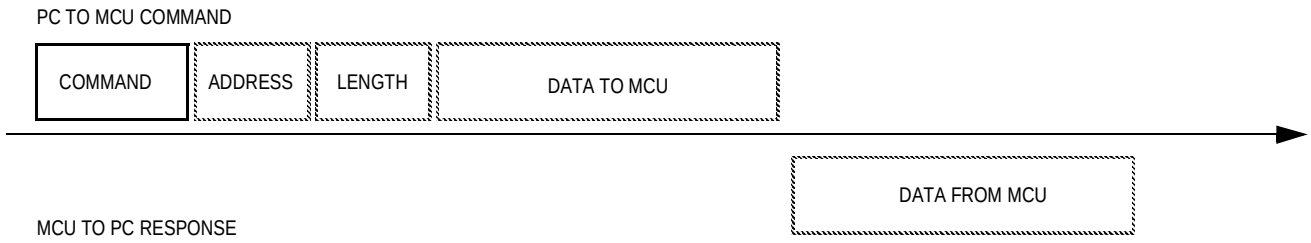
Such a subset of the commands is already functional but useless, so two more basic commands are implemented for pure reprogramming:

- **Erase Command**
- **Write Command**

If the user needs a verification feature, one additional (read) command must be compiled into the MCU code. For pure reprogramming purposes (minimal configuration), it is not required.

- **Read Command**

Generally, each command and response sequence looks like this:



* Dashed fields are not always implemented, data from the MCU may contain only an ACK character instead.

Figure 6. Typical Command and Response

Ident Command

The **Ident Command** (coded as 'I', $49) has no additional fields.

This command is immediately issued by the PC after communication is established. The purpose of the **Ident Command** is to let the PC know several basic properties of the MCU being programmed. All multi-byte fields are sent with MSB first.

- Version number and capability table — 1 byte

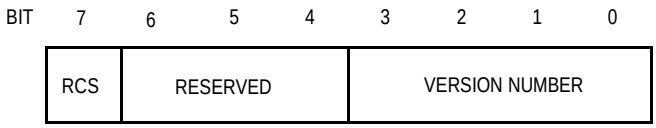


Figure 7. Version Number and Capability Table

RCS — Read Command Supported flag

The RCS flag informs the PC whether the read command is supported (implemented). If not, all calls to the read routine are simply ignored by the MCU and no response is sent back to the PC. It's advisable that PC software warns the user that no read capabilities are available.

1 = Supported
0 = Not Supported (usually due to memory constraints)

RSVD — Reserved

These bits are reserved for future use, unused, and should be set to zero.

VER — Protocol version

Current protocol version is 0x1. In version 1 of the protocol, additional fields are defined as:

- Start address of reprogrammable memory area — 2 bytes
- End address of reprogrammable memory area + 1 — 2 bytes
- Address of **Bootloader User Table** — 2 bytes
- Start address of MCU interrupt vector table — 2 bytes
- Length of MCU erase block — 2 bytes
- Length of MCU write block — 2 bytes
- Bootloader data (specific bootloader info, see implementation) — 8 bytes
- Identification string, zero terminated — <n> bytes

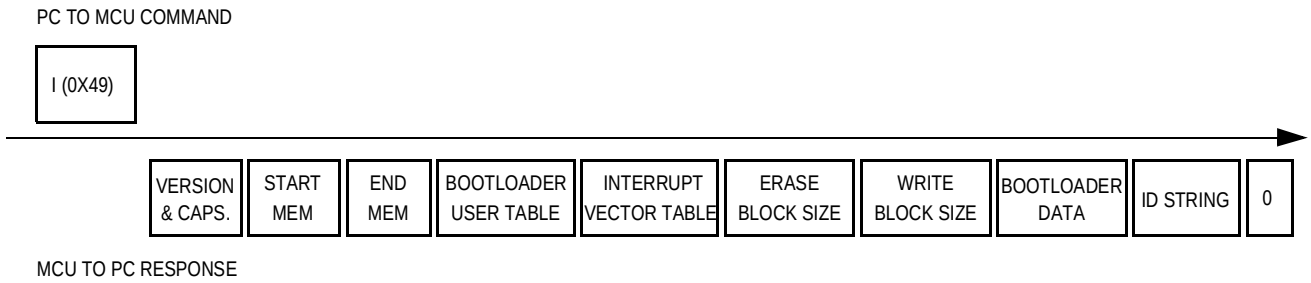


Figure 8. Ident Command

Erase Command

The erase command (coded as 'E', $45) has only an address field, no length or data fields. The start address is a two-byte field, MSB first.

The MCU erases the address block where the specified address resides. The length of block to be erased is equal to the erase block size (typically dependent on hardware).

After the MCU completes execution of the command, the ACK ($FC) character is sent back to the PC. No minimum or maximum execution times of the erase command are specified.

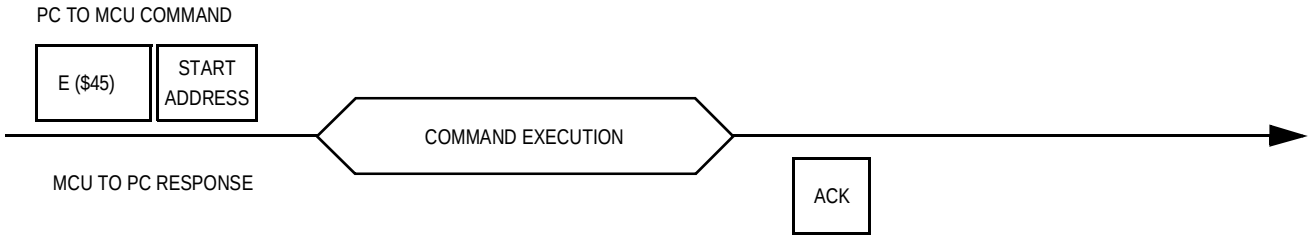


Figure 9. Erase Command

Write Command

The write command (coded as ‘W’, $57) has both address and data fields. The address contains the first address to be programmed. The first byte is the length, followed by the number of bytes to be programmed. The start address is a 2-byte field, MSB first. The length is a 1-byte field.

After the MCU completes execution of the command, the ACK ($FC) character is sent back to the PC. No minimum or maximum execution times of the write command are specified.

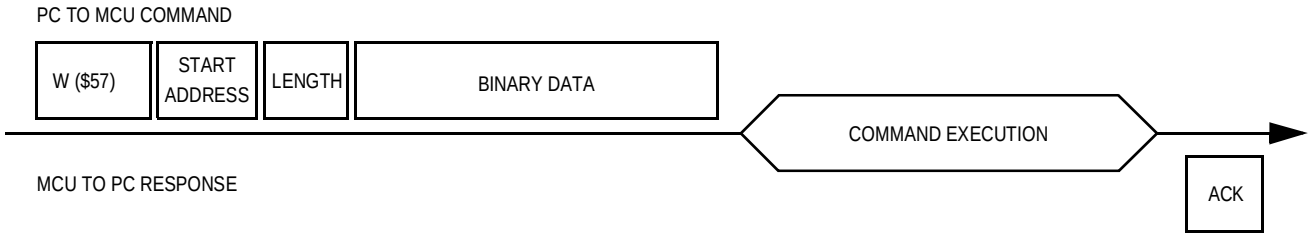


Figure 10. Write Command

Read Command

The read command (coded as ‘R’, $52) has both address and data fields. Address contains the first address to be programmed; the single byte is the length of data to be read. The start address is a two-byte field, MSB first. The length is a one-byte field.

The MCU sends this number of read bytes back to the PC.

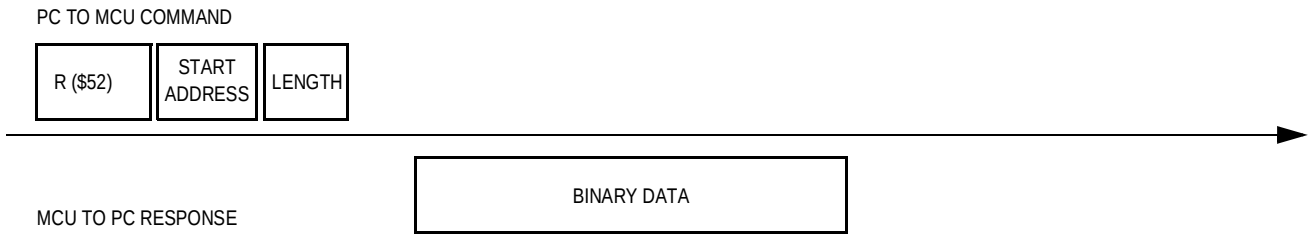


Figure 11. Read Command

Quit Command

The quit command (coded as ‘Q’, $51) has no address or data fields. Execution of bootloader code is finished immediately, and the user code is started. No ACK ($FC) character is sent back to the PC.

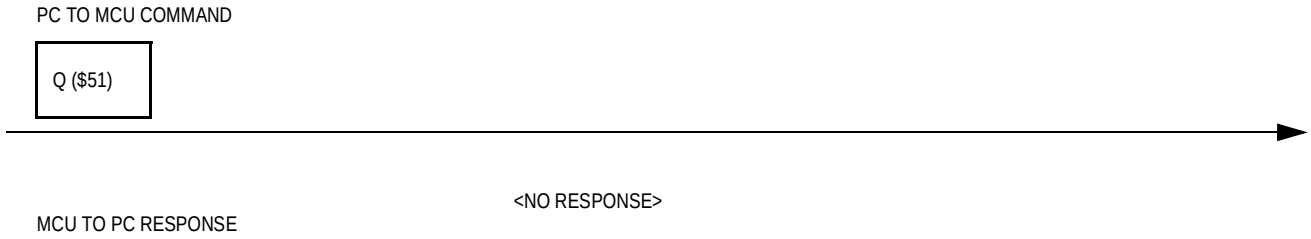


Figure 12. Quit Command

Bootloader User Table

The bootloader user table is a reprogrammable memory area intended for storage of bootloader-specific data. This memory area is not available for the user program. For memory allocation of this table refer to **FC Protocol, Version 1, M68HC908 Implementation**.

FC Protocol, Version 1, M68HC908 Implementation

This section describes all features that are specific to the M68HC908 bootloader implementation. Mainly the memory allocation is heavily MCU specific so the meaning of all variables is explained here in detail.

Figure 13 shows the memory allocation typical to the M68HC908 devices with the bootloader pre-programmed. For example, the MC68HC908KX8 device memory map includes:

- 7680 bytes of FLASH memory ($E000–$FDFF)
- 192 bytes of random-access memory (RAM) ($0040–$00FF)
- 36 bytes of user-defined vectors ($FFDC–$FFFF)

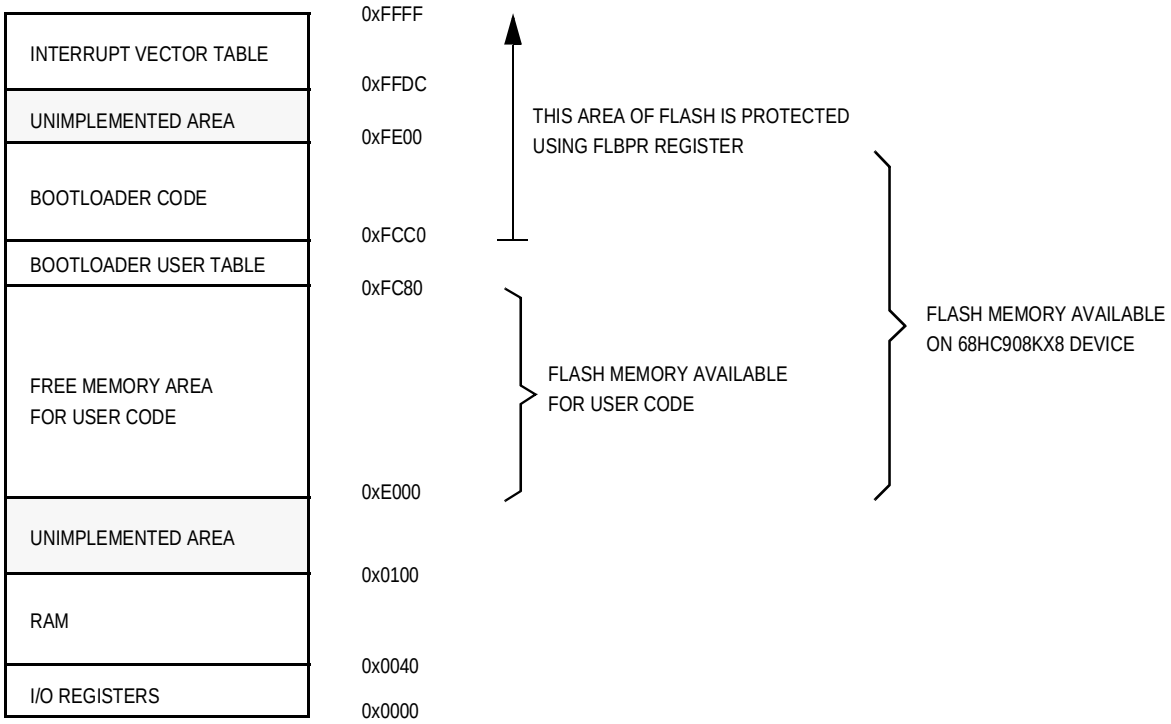


Figure 13. Simplified Example of Memory Allocation in MC68HC908KX8

The bootloader code occupies the top end of FLASH memory (the highest memory address space). This placement allows an effective use of the FLASH block protection technique (see the M68HC908 data sheet for details).

The main idea behind this is that by setting a FLBPR (FLASH block protection register), all address space above this address is protected from both intentional and unintentional erasing/re-writing. Once both bootloader and FLBPR register are programmed into memory, the bootloader code is completely protected from unintentional modification by user code.

NOTE: *Some HC908 derivatives do have an FLBPR register in RAM instead of FLASH (e.g. the HC908JK/JL family). The bootloader code sets this register properly but the user code can eventually modify FLBPR and erase/write the bootloader code.*

The following values are presented for the MC68HC908KX8 bootloader to the PC using this example:

- $01 — Version 1, Read command not implemented (bit 7)
- $E000 — Start address of re-programmable memory area
- $FC80 — End address of re-programmable memory area + 1

- $FC80 — Address of **Bootloader User Table**
- $FFDC — Start address of MCU interrupt vector table
- $0040 — Length of MCU erase block
- $0020 — Length of MCU write block
- 0,0,0,0,0,0,0,0 — Bootloader data. No strictly defined syntax; different HC08 implementations provide different values (for example, the sixth value in the MC68HC908KX8 implementation is the value of the internal clock generator trim register after calibration). All these bootloader data are then programmed back into the bootloader user table and can be retrieved during all subsequent starts (e.g., to trim the MCU's internal clock generator to the best known value before user code start).
- 'KX8-IR',0 — Identification string, zero terminated. Info to be printed on PC screen.

Interrupt Vector Table Translation

Since the FLASH block protection technique also protects the interrupt vector table from being overwritten, some method must be used to translate these vectors to the different locations. For this purpose, the bootloader user table has been implemented. It's a part of memory which is NOT protected by the FLBPR but is not available to the user program. All standard interrupt vectors are pointing to this table where JMP instructions are expected to be stored for each interrupt. The only exception is the reset vector, which points to the start of bootloader code. When an interrupt occurs, the vector is fetched from protected memory and directs execution to continue at the corresponding JMP instruction in the bootloader user table. See **Figure 14**. Note that in a standard interrupt vector table, each record is two bytes long (each vector is a 16-bit address). This is different from the bootloader user table where each record is three bytes long — a JMP opcode ($CC) plus a 16-bit address.

The bootloader requirements dictate that transparent operation with respect to the user S19 file must be programmed into the device. For that, another piece of intelligence is built into the PC master code (instead of the MCU slave). The translation works like this:

If the record in the interrupt vector table is detected in the user S19 file, the value is translated into the corresponding spot in the bootloader user table, including the JMP instruction (opcode $CC). For example, if the user S19 file contains #3 interrupt vector $E123 at address $FFE8, such a vector is translated into the sequence $CC, $E1, $23 (JMP $E123) that is programmed to the $FC81 address in the bootloader user table.

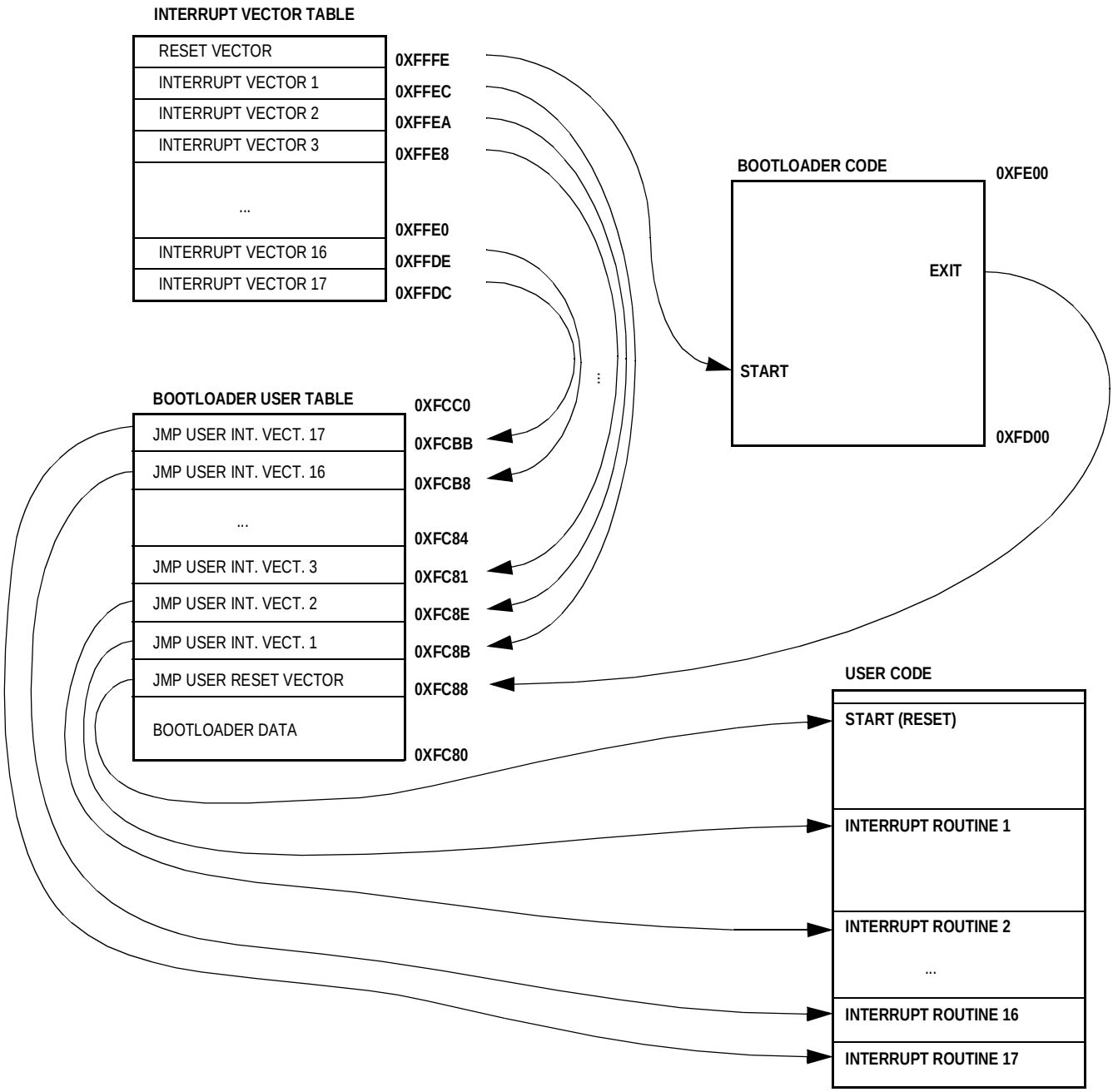


Figure 14. Interrupt Vector Table Translation Explanation

Using this method, the user S19 file does not need to be modified in any way except that the lower address of the end of FLASH memory must be considered. Also, every interrupt is delayed by the execution time of this JMP instruction (3T) as summarized in **Each Interrupt 3T Delayed**.

User Code Start

In order to provide a register setup close to the way it appears after MCU reset, the user code is started in the way that may look bizarre at first glance.

If the bootloader must quit and run user code, an illegal operation is intentionally executed (opcode $32). This causes an illegal operation reset and the MCU restarts. During bootloader startup, a SIM reset status register (SRSR) is tested. If power-on-reset is not detected, the user code is started instead of the bootloader code. This allows the transparent operation of all other resets (like illegal address, etc.) with only a short additional delay caused by testing of the SRSR register and executing associated jump instructions.

NOTE: *In some implementations, a pin reset (caused by external RESET pin) is also included as a valid source of reset for the bootloader to start. This allows remote in-circuit reprogramming in embedded applications that are able to drive the M68HC08 MCU's RESET pin.*

NOTE: *One additional test has been added to the real bootloader application — if no source of reset is detected (i.e., if the SRSR register is zero), the bootloader is also started. This may happen when reset is caused by an external pin, but the reset pulse is shorter than specified. In that case, the minimum length of reset pulse that will cause reset is shorter than the length needed for the proper propagation of the external reset flag to the SRSR register.*

Since the SRSR register is one-time readable (it clears after read), no subsequent reads of this registers provide a valid value. This is another limitation also noted in **System Limitations**.

System Limitations

This section summarizes all limitations which must be considered when using the bootloader along with the user application.

Memory Occupied

This is a natural limitation imposed. One of the strongest requirements was to obtain the smallest code possible. Typical M68HC908 implementations are between 300 and 500 bytes including the bootloader user table. If the target M68HC08 MCU is capable of FLASH programming using internal ROM routines, then the memory consumption is near the lower limit. Bigger M68HC08 MCUs (which are not usually equipped with ROM code for FLASH programming) will require around 500 bytes of FLASH out of 32 KB (as is the case with the MC68HC908GP32).

The bootloader is placed at the upper end of FLASH memory, thus the only modification required in the user code is in the memory mapping (typically found in the linker parameter file).

The M68HC08 MCU which is to be reprogrammed also signals the actual FLASH addresses that are available. The PC master software won't allow programming if the user code overlaps with bootloader code.

Time Delay Upon Start-Up and Initial Communication

As already mentioned, another requirement dictated the minimum of pins to have specific meaning during bootloader start-up. Especially in communication systems (for example, those using a standard serial port), the "pin overhead" is zero and the recognition is done in the time domain. So, the bootloader waits a certain amount of time to receive an answer from the PC upon start-up. If none is received, the user code starts. The typical delay is the range of several hundred milliseconds.

If this start-up delay becomes an issue for the final application, the user may decide to modify the bootloader code and use an *IRQ-low upon start-up* method instead. A simple test of the voltage level on the IRQ pin (or, basically, on any other input pin) could be used to decide whether the bootloading sequence is required by the user.

Each Interrupt 3T Delayed

Every interrupt call is delayed by 3T bus clocks that are required to execute the JMP instruction stored in the bootloader user table. This interrupt vector translation (as described in **Interrupt Vector Table Translation**) has been chosen as the best solution for achieving user code transparency along with security of the bootloader code itself.

The interrupt latency is about 10 to 15T anyway (assuming that no interrupt is being executed), so this additional delay is not significant for the most applications.

FLBPR Not Usable (in Some M68HC08 Family MCUs)

The bootloader uses a FLASH block protection technique to protect itself from being overwritten (where applicable).

Some M68HC08 MCUs (such as the KX, GP and GR devices) have this FLASH block protection register stored in FLASH and it cannot be modified when in-circuit. The FLBPR itself can be erased or programmed only with an external voltage, V_{TST}, present on the IRQ pin. Since this feature is completely dedicated to bootloader code protection, it's unavailable to the user application code. If the value for FLPBR appears in the user S19 code, an error is displayed and all programming is cancelled. Such an occurrence must be omitted from user S19 code.

Some families have the FLASH block protection register stored in RAM instead (the JK/JL Families are like this). The bootloader sets the proper value at the beginning of its execution in order to protect itself. However, user code may freely modify this register and protect its own memory areas as needed. This also implies that the bootloader is not 100% protected from user code.

For both cases, see the MCU data sheets for a detailed explanation.

SRSR Register Unusable

The bootloader uses an SRSR register (as described in **User Code Start**) to recognize the source of reset to determine whether the user code shall be run. Since the SRSR register is one-time readable (i.e., it is reset after first read), the user code does not have access to the SRSR value (if the bootloader is present in the memory and makes the first read after each reset). There's no simple remedy for this situation. After the SRSR register is read by the bootloader, it's also stored in one RAM location. Unfortunately, its memory location may differ from one implementation to another. If the user desperately needs both the SRSR register and bootloader usage, he/she must redirect the SRSR reading to this specific RAM location. Its actual position in the specification can be obtained from the bootloader's MAP file.

MCU Slave Software

The following section provides a detailed description of the three typical M68HC08 bootloader implementations. All code is written in assembly language. Several selected targets and different features are described as follows:

- **MC68HC908KX Implementation — ICG, On-Chip ROM Features**
- **MC68HC908JK/JL Implementation — Soft-SCI, On-Chip ROM Features**
- **MC68HC908GP Implementation — ROM-Less Programming**
- **MC68HC908GR Implementation — On-Chip ROM Features, Known Crystal Frequency**
- **MC68HC908MR Implementation — ROM-Less Programming, PLL Circuit Usage**
- **MC68HC908GT and MC68HC908EY Implementations — Fully Compatible with MC68HC908KX**
- **MC68HC908QT/QY — Soft-SCI, On-Chip ROM features, Simpler ICG Usage, (Spare FLASH memory blocks are utilized), Single-wire Communication**
- **MC68HC908LJ Implementation — On-Chip ROM Features, Known Crystal Frequency, PLL Circuit Usage**

MC68HC908KX Implementation — ICG, On-Chip ROM Features

The M68HC908KX Family is equipped with an Internal Clock Generator (ICG) module. This allows a very effective quartz-less implementation of the bootloader, which is then also independent of any specific clock source or clock speed.

The on-chip FLASH programming routines simplify the bootloader and improve memory consumption as well.

The communication between the MCU and PC uses a standard serial channel (SCI).

The flowchart follows.

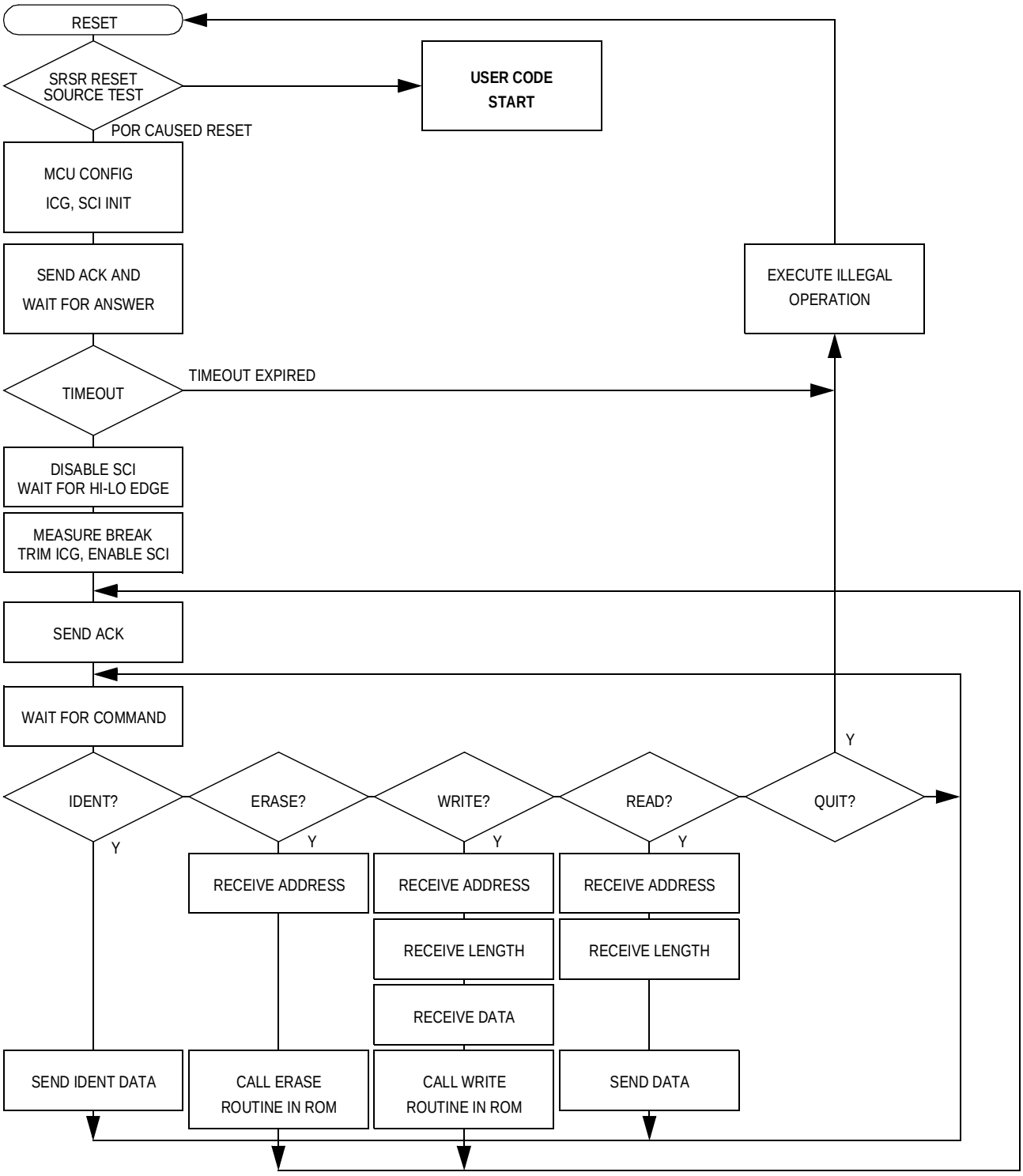


Figure 15. KX Bootloader Flowchart

Internal Clock Generator (ICG) Usage — Initialization

The initialization of the ICG is made very simple. Since the ICG is active and clock monitor is disabled after reset, the only action required is the modification of the ICG multiply register. Then, the ICGS flag (bit 2) of the ICG control register reports whether ICG is stable after the frequency change.

```
ICGMRINIT       EQU     $20

        MOV     #ICGMRINIT,ICGMR        ; set 9.8304MHz BUS clock
LOOP:   BRCLR   2,ICGCR,LOOP            ; wait until ICG stable
```

Internal Clock Generator Usage — Trimming

Even though the trimming routine is in ROM, a small bug renders this code unusable. So the source code has been taken and inserted in the bootloader code.

Although application note *Using MC68HC908 On-Chip FLASH Programming Routines, ROM-Resident Routines in the MC68HC908GR8, MC68HC908KX8, MC68HC908JL3, MC68HC908JK3, and the MC68HC908JB8)* (Motorola document order number AN1831/D) documents the procedure for calculating the trim factor out of the measured CPU speed, the code itself omits the final doubling of the number of cycles.

```
* FOLLOWING LOOP IS EXECUTED UNTIL THE END OF THE BREAK SIGNAL. THE BREAK
* SIGNAL LASTS 10 BIT TIMES. IF COMMUNICATING AT f OP /256 BPS, THEN 10 BIT
* TIMES IS 2560 CYCLES. EACH TIME THROUGH THE LOOP IS 10 CYCLES, SO WE
* EXPECT TO EXECUTE THE LOOP 256 TIMES IF THE KX8 IS IN SYNC SERIALLY WITH
* THE HOST. IF WE STAY IN THE LOOP FOR > 256 LOOP CYCLES, THEN THE KX8
* MUST BE RUNNING FASTER THAN EXPECTED, AND NEEDS TO BE SLOWED DOWN. IF WE
* STAY IN THE LOOP FOR < 256 LOOP CYCLES THEN THE KX8 MUST BE RUNNING SLOWER
* THAN EXPECTED AND NEEDS TO BE SPEEDED UP. THE AMOUNT THAT WE CHANGE THE
* CPU SPEED IS EQUAL TO THE NUMBER OF LOOP CYCLES OVER OR UNDER 256. SO IF
* WE GO THROUGH THE LOOP 240 TIMES, THEN WE ARE RUNNING
* (256-240)/256 = 6.25% FAST. EACH INCREMENTAL CHANGE WE MAKE TO THE TRIM REGISTER
* (ICGTR) WILL MAKE A 0.195% CHANGE TO THE INTERNAL CLOCK. THAT IS, INCREMENTING
* THE REGISTER BY ONE OVER THE DEFAULT VALUE OF $80 STORED THERE WILL
* DECREASE THE INTERNAL CLOCK BY 0.195%, AND VICE VERSA.
* NOW EACH EXECUTION OF THE LOOP OVER OR UNDER WHAT IS EXPECTED (256 TIMES)
* REPRESENTS AN ERROR OF 1/256 = .391% ERROR. SO WE'LL NEED TO DOUBLE THE
* NUMBER OF LOOP CYCLES AND USE THIS NUMBER TO CORRECT THE TRIM REGISTER.
* OUR PRECISION FOR TRIMMING IS THEREFORE 0.391%.
```

So the actual code adds an ASLA instruction which doubles the trim factor before the actual write to the ICG Trim Register.

```
ICGTRIM:
        CLRX
        CLRH

MONPTB4:
        BRSET   4,PTB,MONPTB4   ;WAIT FOR BREAK SIGNAL TO START
CHKPTB4:
        BRSET   4,PTB,BRKDONE   ;(5) GET OUT OF LOOP IF BREAK IS OVER
        AIX     #1              ;(2) INCREMENT THE COUNTER
        BRA     CHKPTB4         ;(3) GO BACK AND CHECK SIGNAL AGAIN
```

```
BRKDONE:
        PSHH
        PULA                    ;PUT HIGH BYTE IN ACC AND WORK WITH A:X
        TSTA                    ;IF MSB OF LOOP CYCLES = 0, THEN BREAK TAKES TOO
        TXA                     ;FEW CYCLES THAN EXPECTED, SO TRIM BY SPEEDING
        BEQ     SLOW            ;UP f OP .
FAST:   CMP     #$40            ;SEE IF BREAK IS WITHIN TOLERANCE
        BGE     OOR             ;DON'T TRIM IF OUT OF RANGE
        ASLA                    ;multiply by two to get right range
        ADD     #$80            ;BREAK LONGER THAN EXPECTED, SO SLOW DOWN f OP
        BRA     ICGDONE
SLOW:   CMP     #$C0            ;SEE IF BREAK IS WITHIN TOLERANCE
        BLT     OOR             ;DON'T TRIM IF OUT OF RANGE
        ASLA                    ;multiply by two to get right range
        SUB     #$80
ICGDONE:
        STA     ICGTR
OOR:
        RTS
```

The complete explanation of the trimming procedure can be found in the mentioned application note.

MC68HC908JK/JL Implementation — Soft-SCI, On-Chip ROM Features

The MC68HC908JK/JL devices are among the least expensive in the M68HC08 Family. Due to the cost, no hardware SCI is present. Therefore, its software counterpart must be implemented. This also allows the unrestricted selection of which pins are used for serial communication (the provisions are made in the code that also an $\overline{\text{IRQ}}$ pin can be used as an input serial line too).

The JK/JL Family also exists in an RC version (an RC oscillator is used instead of quartz). Thanks to the existence of the bootloader's calibration, any variation in speed is compensated for. If the desired clock frequency is out of the specified range covered by the calibration system, the code must be modified.

The JK/JL Family is also equipped with on-chip FLASH programming routines, so FLASH programming is performed by these, saving some additional memory.

The main program flowchart follows and is very similar to the previous case.

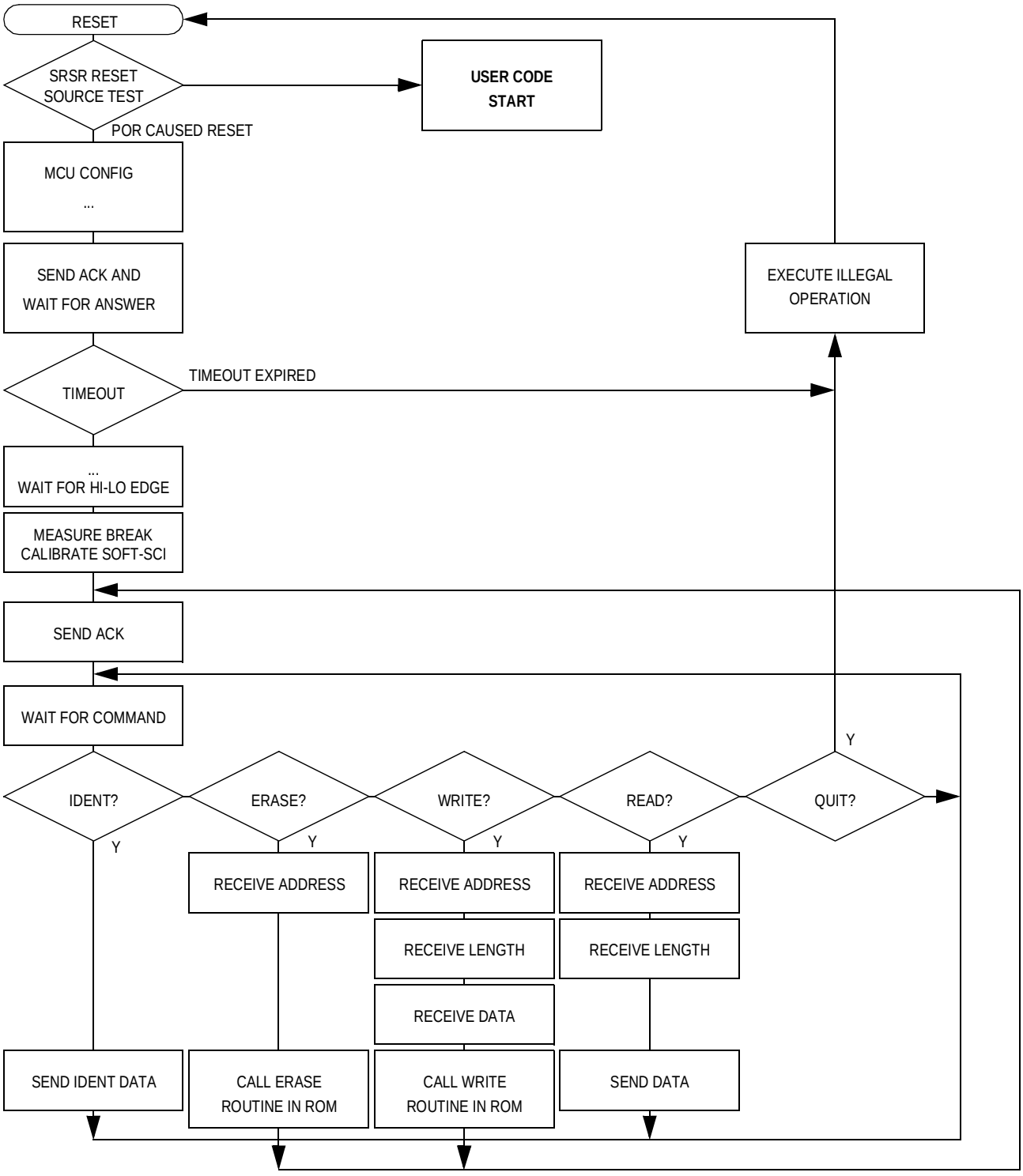


Figure 16. JK/JL Bootloader Flowchart

Software-SCI Transmit Char Routine

Since the simple software-SCI routines are of some interest to this application, a more detailed description of the software-SCI transmit and receive sub-routines follows. They both are based on a 16-bit timer where the output-compare event is polled within the background loop.

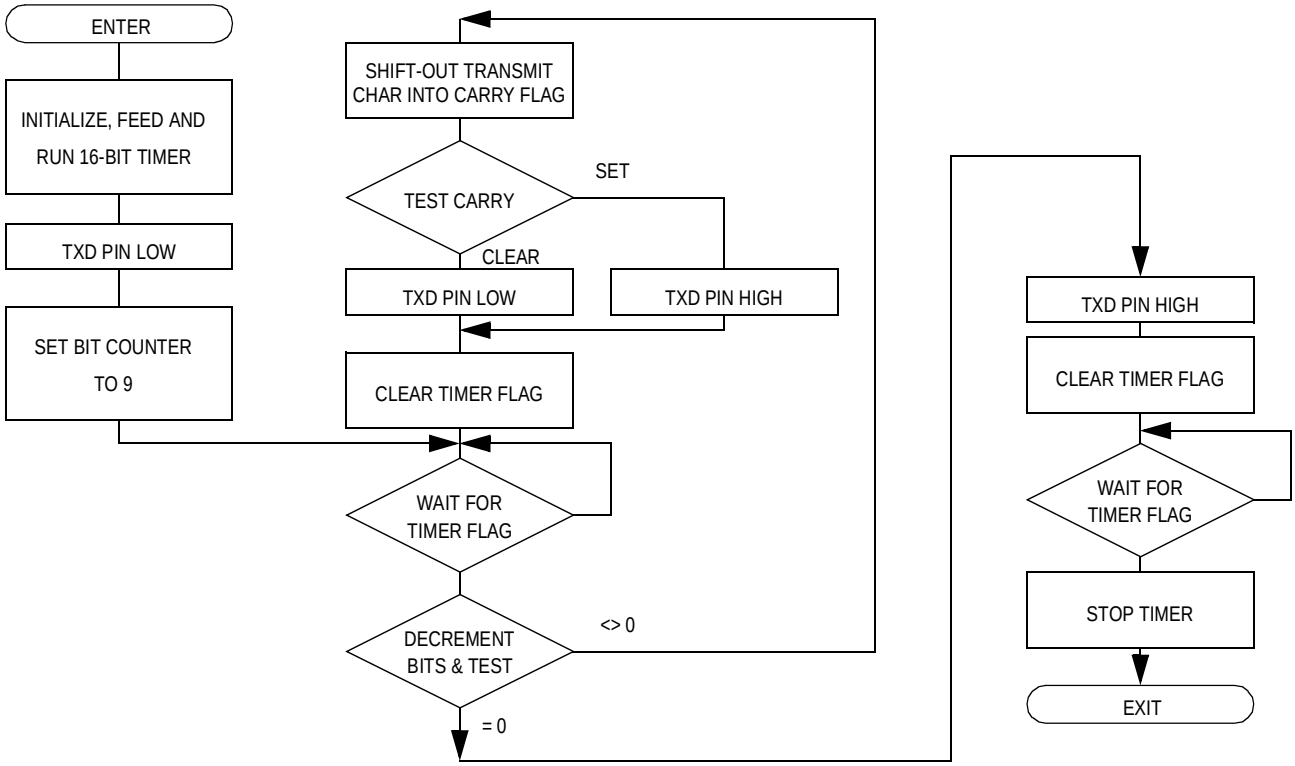


Figure 17. Soft-SCI Transmit Char Routine Flowchart

The source code for the two routines is also listed here. Besides a few counters, a 16-bit `ONEBIT` variable is used. It contains the actual length of one bit at the current communication speed in 16-bit timer ticks. This variable is initialized during the calibration phase.

Software-SCI Transmit Char Routine Source Code

```
;*******************************************************************************************
SCITX:
        PSHH
        PSHX

        BCLR    7,TSC           ; and clear TOF
        LDHX    ONEBIT
        STHX    TMOD
        BSET    4,TSC           ; clear timer
        BCLR    5,TSC           ; run timer

        TXDCLR

        MOV     #9,BITS         ; number of bits + 1
        BRA     SCITX1          ; jump to loop

SCITX2:
        LSRA                    ; shift out lowest bit
        BCC     DATALOW

        TXDSET
        SKIP2                   ; skip next two bytes
DATALOW:
        TXDCLR

        BCLR    7,TSC           ; and clear TOF
SCITX1: BRCLR   7,TSC,SCITX1    ; wait for TOF

        DBNZ    BITS,SCITX2     ; and loop for next bit

SCISTOP:
        TXDSET

        BCLR    7,TSC           ; and clear TOF
SCITX3: BRCLR   7,TSC,SCITX3    ; wait for TOF
EPILOG:
        BSET    5,TSC           ; stop timer

        PULX
        PULH
        RTS
```

Software-SCI Receive Char Routine

The software-SCI receive routine works in a similar way. When the 16-bit output-compare event is polled, the value of the receive pin is scanned. No provisions are made for stop-bit checking, framing check, noise detection, etc., mainly due to memory restrictions. **Figure 18** shows the software-SCI receive routine flowchart.

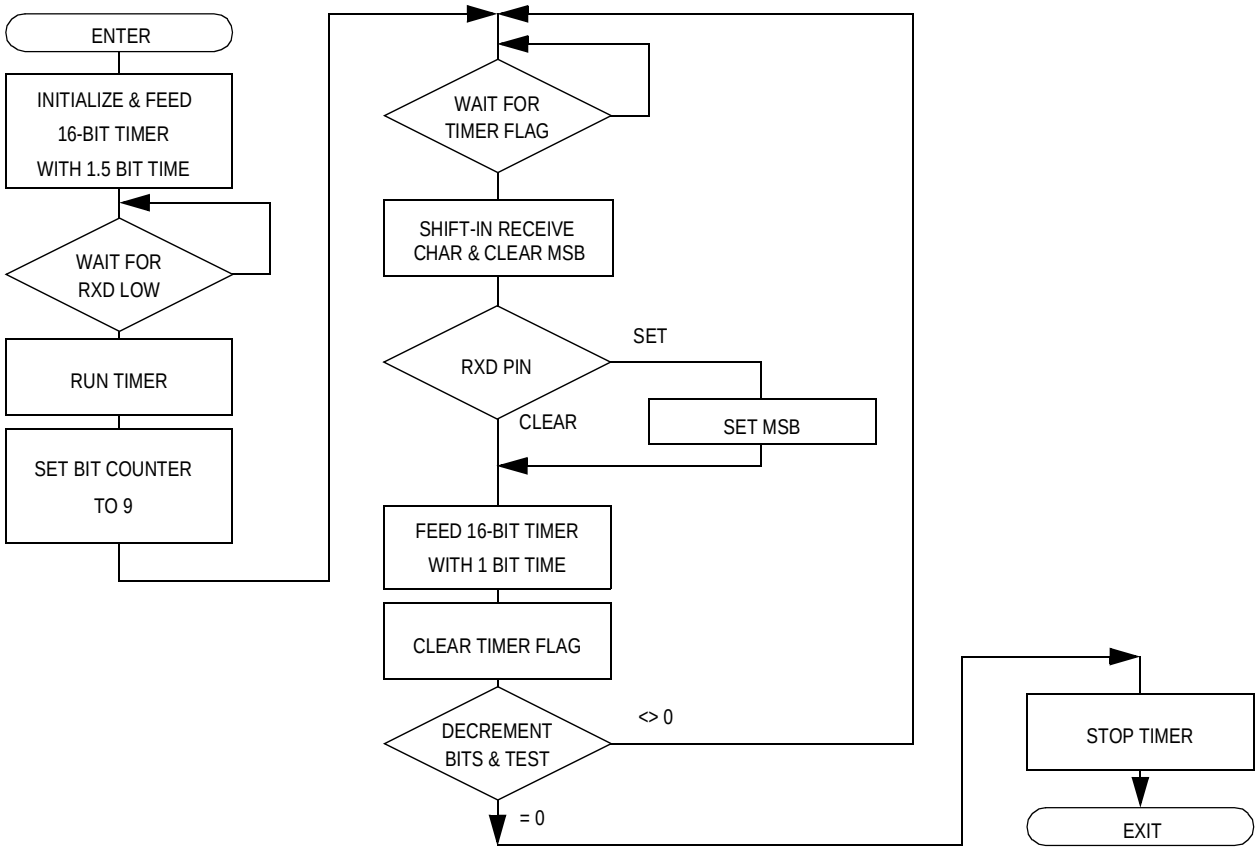


Figure 18. Software-SCI Receive Char Routine Flowchart

Software-SCI Receive Char Routine Source Code

Here, the source code for software-SCI receive routine is listed.

```
;*************************************************************************************
SCIRX:
        BRRXDLO SCIRX           ; loop until RXD high (idle)

SCIRXNOEDGE:
        PSHH
        PSHX
        BCLR    7,TSC           ; and clear TOF

        LDX     ONEBIT
        LDA     ONEBIT+1
        LSRX
        RORA
        STX     TMODH
        STA     TMODL

        BSET    4,TSC           ; clear timer

SCIRX1:
        BRRXDHI SCIRX1          ; loop until RXD low (wait for start bit)

        BCLR    5,TSC           ; run timer
        MOV     #9,BITS         ; number of bits + 1

SCIRX2: BRCLR   7,TSC,SCIRX2    ; wait for TOF

        LSRA                    ; shift data right (highest bit cleared)
        BRRXDLO RXDLOW          ; skip if RXD low
        ORA     #$80            ; set highest bit if RXD high

RXDLOW: LDHX    ONEBIT
        STHX    TMOD

        BCLR    7,TSC           ; and clear TOF
        DBNZ    BITS,SCIRX2     ; and loop for next bit

        BRA     EPILOG
```

Software-SCI Macros Source Code

Several macros are defined across the two pieces of code. They improve the readability or memory consumption.

```
SKIP1           MACRO
                DC.B    $21             ; BRANCH NEVER (saves memory)
                ENDM

SKIP2           MACRO
                DC.B    $65             ; CPHX (saves memory)
                ENDM

BRRXDLO         MACRO

    IFNE    RXDISIRQ
      IFNE    SCIRXINV
        BIH     \1       ; branch if RXD low
      ELSE
        BIL     \1       ; branch if RXD low
      ENDIF
    ELSE    ; RXD uses normal I/O pin
      IFNE    SCIRXINV
        BRSET   RXDPIN,RXDPORT,\1    ; branch if RXD low
      ELSE
        BRCLR   RXDPIN,RXDPORT,\1    ; branch if RXD low
      ENDIF
    ENDIF

        ENDM

BRRXDHI         MACRO

    IFNE    RXDISIRQ
      IFNE    SCIRXINV
        BIL     \1       ; branch if RXD hi
      ELSE
        BIH     \1       ; branch if RXD hi
      ENDIF
    ELSE    ; RXD uses normal I/O pin
      IFNE    SCIRXINV
        BRCLR   RXDPIN,RXDPORT,\1    ; branch if RXD hi
      ELSE
        BRSET   RXDPIN,RXDPORT,\1    ; branch if RXD hi
      ENDIF
    ENDIF

        ENDM

TXDCLR          MACRO

      IFNE    SCITXINV
        BSET    TXDPIN,TXDPORT  ; clr bit
      ELSE
```

```
        BCLR    TXDPIN,TXDPORT  ; clr bit
      ENDIF

        ENDM

TXDSET          MACRO

      IFNE    SCITXINV
        BCLR    TXDPIN,TXDPORT  ; set bit
      ELSE
        BSET    TXDPIN,TXDPORT  ; set bit
      ENDIF

        ENDM
```

MC68HC908GP Implementation — ROM-Less Programming

The GP devices belong to the M68HC08 Family. The GP devices have no on-chip FLASH programming routines available. Therefore, all FLASH programming must be done by the bootloader itself and is demonstrated in this particular implementation.

The GP devices are primarily targeted for use with a low-cost watch 32.768 kHz crystal. Since the value of the crystal is known, no calibration is conducted, thus saving some additional MCU memory. A simpler **Known MCU Communication Speed** scenario is demonstrated within this code.

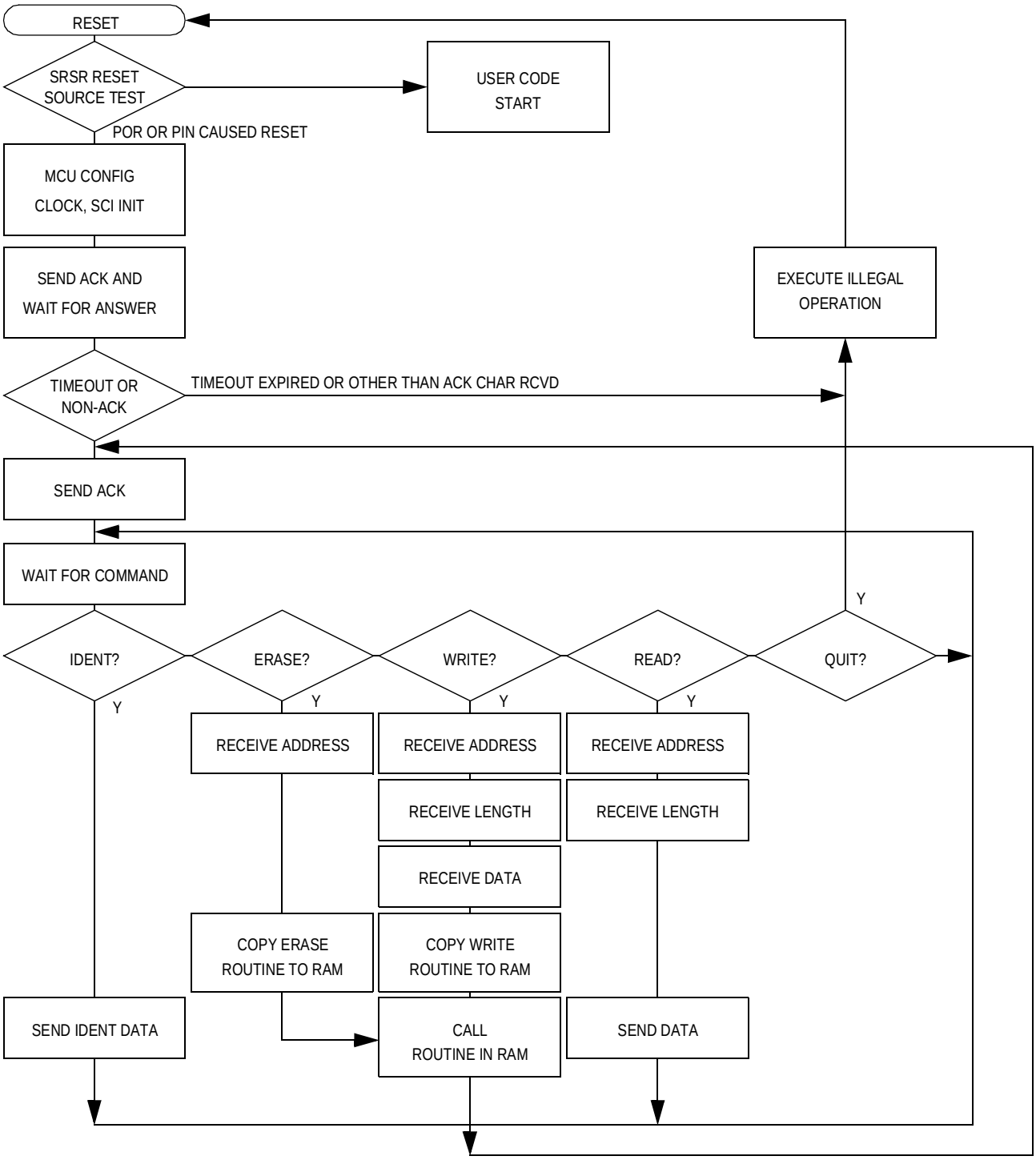


Figure 19. GP Bootloader Flowchart

FLASH Programming Routines

The main code is similar to the previous implementation with the calibration phase omitted. The FLASH programming by the bootloader is demonstrated below. Three main sub-routines are defined:

- CPY_PRG — copies the selected routine into RAM
- ERASE_ALG — whole FLASH erase routine
- WR_ALG — whole WRITE erase routine

Since the flow is straight-forward, no flowchart is provided. Basically the sequence of events is executed as per FLASH erasing/programming specifications.

For improved readability, two timing macros (D_US and D_MS) are used within the code.

FLASH Programming Routines Source Code

```
;*******************************************************************************************
CPY_PRG:
        TSX                     ;
        STHX    STACK           ; copy stack for later re-call

        LDHX    SOURCE          ; LOAD WRITE ALGORITHM TO RAM
        TXS
        LDHX    #PRG
CPY_PRG_L1:
        PULA
        STA     X
        AIX     #1
        DBNZ    STAT,CPY_PRG_L1

        LDHX    STACK
        TXS                     ; restore stack
        RTS
;*******************************************************************************************
ERASE_ALG:

        LDA     #%00000010
        STA     FLCR            ; ERASE bit on
        LDA     FLBPR           ; dummy read FLBPR

        LDHX    ADRS            ; write anything
        STA     X               ; to desired range
        D_US    #T10US          ; wait 10us

        LDA     #%00001010
        STA     FLCR            ; set HVEN, keep ERASE
        D_MS    #T1MS           ; wait 1ms

        LDA     #%00001000
        STA     FLCR            ; keep HVEN, ERASE off
        D_US    #T5US           ; wait 5us
```

```
        CLRA
        STA     FLCR            ; HVEN off
        D_US    #T1US           ; wait 1us

        JMP     SUCC            ; finish with ACK
ERASE_ALG_END:
;*******************************************************************************************
WR_ALG:
        LDA     #%00000001
        STA     FLCR            ; PGM bit on
        LDA     FLBPR           ; dummy read FLBPR

        LDHX    ADRS            ; prepare addrs'
        STA     X               ; and write to desired range
        D_US    #T10US          ; wait 10us

        LDA     #%00001001
        STA     FLCR            ; set HVEN, keep PGM
        D_US    #T5US           ; wait 5us

        LDHX    #DAT            ; prepare addrs'
        TXS
        LDHX    ADRS
        MOV     LEN,POM
WR_ALG_L1:
        PULA
        STA     X
        AIX     #1
        D_US    #T30US          ; wait 30us
        DBNZ    POM,WR_ALG_L1   ; copy desired block of data

        LDA     #%00001000
        STA     FLCR            ; keep HVEN, PGM off
        D_US    #T5US           ; wait 5us

        CLRA
        STA     FLCR            ; HVEN off
        D_US    #T1US           ; wait 1us

        JMP     RETWR           ; finish with ACK (& restore STACK before)
WR_ALG_END:
END
```

FLASH Programming Macros Source Code

```
;*******************************************************************************************
D_MS:   MACRO
        LDA     \1          ; [2] ||
\@L2:   CLRX                ; [1] ||
\@L1:   NOP                 ; [1] |
        DBNZX   \@L1        ; [3] |    256*4 = 1024T
        DBNZA   \@L2        ; [3] || (1024+4)*(arg-1) + 2 T
        ENDM

D_US:   MACRO
        LDA     \1          ; [2]
\@L1:   NOP                 ; [1]
        DBNZA   \@L1        ; [3] 4*(arg-1) + 2 T
        ENDM
```

MC68HC908GR Implementation — On-Chip ROM Features, Known Crystal Frequency

The GR devices also belong to the M68HC08 Family. The GR devices are smaller derivatives of the GP family and in addition are equipped with ROM memory **with on-chip FLASH programming routines available** also in the user mode.

Both the GP and GR devices are primarily targeted for use with a low-cost watch 32.768 kHz crystal. Since the value of the crystal is known, no calibration is conducted, thus saving some additional MCU memory. A simpler **Known MCU Communication Speed** scenario is also demonstrated within this code.

MC68HC908MR Implementation — ROM-Less Programming, PLL Circuit Usage

The MR devices are motor-control oriented members of the M68HC08 Family. The MR devices also have no on-chip FLASH programming routines available. Therefore, all FLASH programming must be done by the bootloader itself.

The MR family is equipped with a PLL circuit that can multiply the crystal frequency. Typically, a 4-MHz XTAL is used as the reference frequency. This implementation demonstrates how the PLL circuit is initialized for 8 times the crystal frequency. The source PLL frequency is then 32 MHz and the bus frequency is 8 MHz.

Again, when the value of the crystal is known, no calibration is conducted. A simpler **Known MCU Communication Speed** scenario is also demonstrated.

MC68HC908GT and MC68HC908EY Implementations — Fully Compatible with MC68HC908KX

The code for GT and EY devices is identical to the KX code, except for the memory mappings and ROM routines location. The one minor difference is that GT family can't use CGMXCLK clock as a SCI module source. Thus baud rate selection is possible only from the bus clock.

MC68HC908QT/QY — Soft-SCI, On-Chip ROM features, Simpler ICG Usage, (Spare FLASH memory blocks are utilized), Single-wire Communication

The QT/QY devices are the smallest members of the M68HC08 Family. They are equipped with a simple ICG module (running on fixed frequency 12.8 MHz ±25%). Also, ROM routines are available.

There are several spare FLASH locations (mainly among unused interrupt vectors) that are also used for storing the bootloader code.

Single-Wire Communication

Due to the small number of pins on QT devices, the single-wire SCI version has been developed to keep the number of pins that are occupied by communication to an absolute minimum. **Figure 20** illustrates an example single-wire RS-232 interface. The single-wire option has been backported to JK/JL bootloader because it uses a software SCI also.

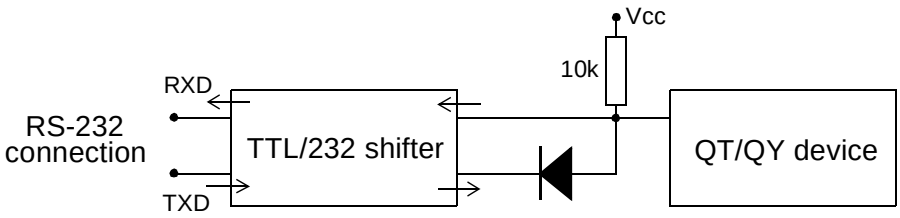


Figure 20. Example Single-Wire Schematic

SCI Application Program Interface (SCIAPI)

Software SCI communication is implemented on QT/QY (and also on JK/JL) devices to reduce cost and enable the user code to call the SCI send and receive routines (under certain limitations). The bootloader code for QT/QY and JK/JL Families now implements SCIAPI. SCIAPI is the defined way to call the SCI send and receive routines.

The details, implementation notes, and limitations are provided in the `sci.h` file (of the QTQY folder). This file is the only resource that must be included in the user ‘C’ code. The calling convention and overall usage is also mentioned there. The main limiting factor for most applications will be that the SCI receive routine is a blocking one. This means that routines will not return until an SCI character is received. The 16-bit timer registers are manipulated also. Some applications will use this code without problems.

The master side of the bootloader must be informed that the single-wire communication is used. This can be done by calling the hc08sprg.exe software. Use the following extended calling convention:

```
hc08sprg 1:S filename.s19
```

where 1 specifies which COM port is used for communication, and S stands for single-wire. Original (old) format:

```
hc08sprg 1 filename.s19
```

now defaults to:

```
hc08sprg 1:D filename.s19
```

where D stands for dual-wire mode. The bootloader master can also detect the presence of a single-wire interface if called:

```
hc08sprg 1:? filename.s19
```

The detection is only possible if the complete serial interface (mainly the level shifter) is powered up and working BEFORE the bootloading process starts. Because this is not usually the case, always specify the bootloading mode by including either a ":S" or a ":D" in the parameter.

MC68HC908LJ Implementation — On-Chip ROM Features, Known Crystal Frequency, PLL Circuit Usage

The LJ devices are members of the M68HC08 Family used to drive LCD displays. The LJ devices have the ROM on-chip FLASH programming routines available. The calling convention is slightly different from other HC08s (see LJ datasheet, chapter Monitor ROM).

The LJ devices are primarily targeted for use with a low-cost watch 32.768 kHz crystal. Since the value of the crystal is known, no calibration is conducted, which saves additional MCU memory. A simpler **Known MCU Communication Speed** scenario is also demonstrated within this code.

Target Implementation Comparison

Following table shows some attributes of different HC08 bootloader implementations.

Table 2 Target implementation Comparison

HC08 Family	FLASH Memory Consumption	Clock Source	ROM Routines Usage	Calibration Conducted	SCI	FLASH Erase Page Size	FLASH Program Page Size
GP32	512B	32768Hz Xtal or external clk.	no	no	hardware	128B	64B
GR4/GR8	320B	32768Hz Xtal or external clk.	yes	no	hardware	64B	32B
JK1/JL1/ JK3/JL3 *(JK8/JL8 excluded)	384B	Xtal, RC oscillator or ext. source	yes	yes	software	64B	32B
KX2/KX8	384B	ICG	yes	yes	hardware	64B	32B
GT8/GT16	384B	ICG	yes	yes	hardware	64B	32B
EY16	384B	ICG	yes	yes	hardware	64B	32B
MR8	512B	PLL with Xtal (4MHz)	no	no	hardware	64B	32B
MR16/MR32	512B	PLL with Xtal (4MHz)	no	no	hardware	128B	64B
QT1/QT4/ QY1/QY4	320B	simpler ICG	yes	yes	software, single-wire possible	64B	32B
LJ12/LJ24	384B	32768Hz Xtal or external clk.	yes, different version	no	hardware	128B	64B

Master PC Software

This section provides a detailed description of the bootloader host computer master software. All code is written in C language with provisions made allowing compilation both for Linux® and Win32® platforms.

The bootloader specifications dictate that, as much as possible, intelligence is executed in the host computer instead of by the MCU, minimizing MCU memory consumption. Only primitive functions are implemented in the MCU.

The host computer master software design is straight-forward and is a sequence of several steps:

1. Opening serial port
2. Opening source S19 file
3. Waiting for reset of MCU
4. Calibrating MCU
5. Reading MCU info
6. Remapping MCU interrupt vectors
7. Checking whether source S19 data fits into physical MCU memory
8. Erasing and programming MCU
9. Cleaning-up, exiting program

This sequence is shown in **Figure 21**.

Linux® is a trademark of Linus Torvalds
Windows® and Win32® are registered trademarks of Microsoft Corporation in the U.S. and other countries.

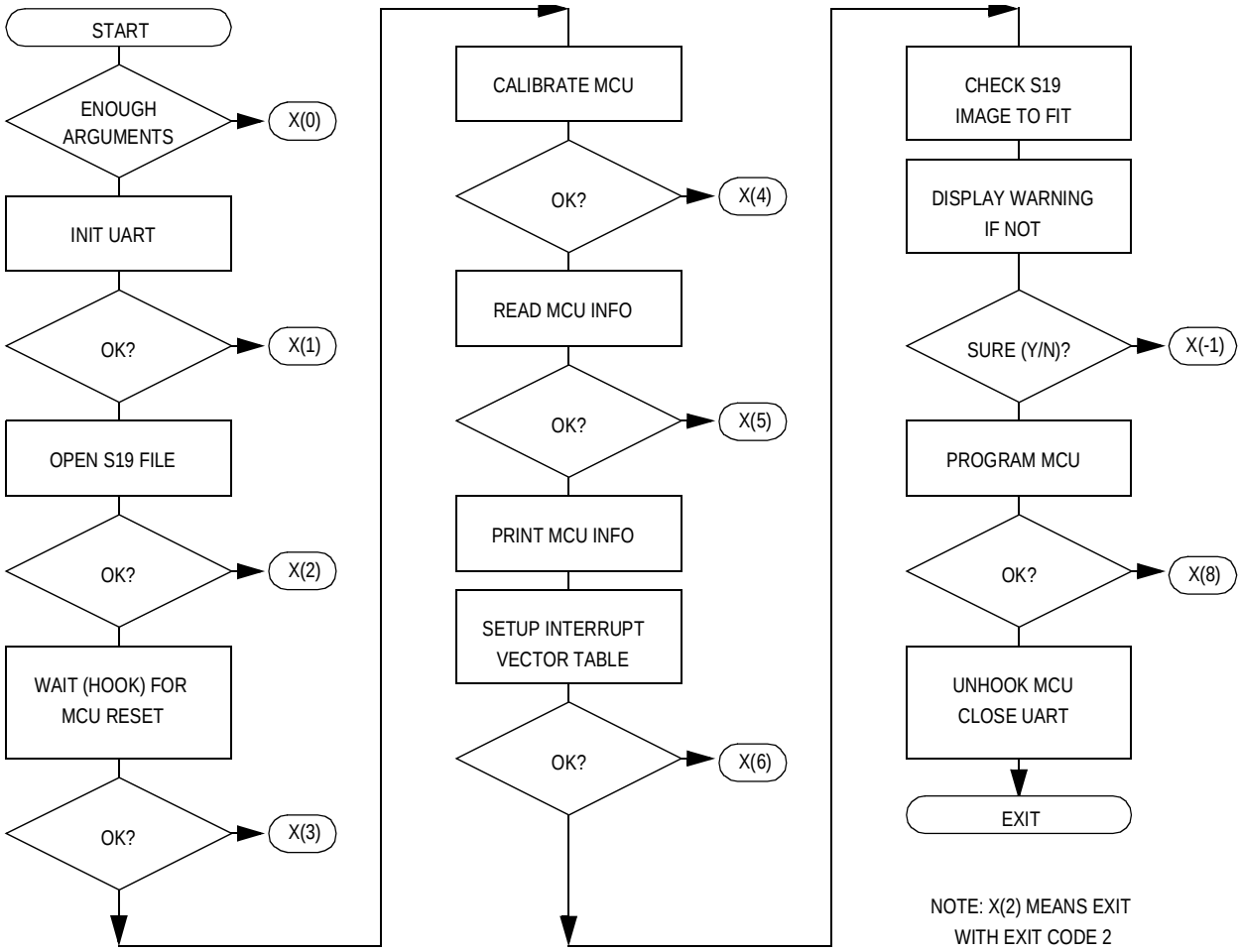


Figure 21. Bootloader Master Flowchart

In the following text, some sections of the master bootloader code will be described in more detail. All actions required for reprogramming the M68HC08 device are fully described in the slave implementation and protocol sections of this document. Mainly the specific master characteristics are emphasized.

File Structure

The following file structure is set up:

- **M68HC08 Image Operations**:

```
s19.c
```

- **UART Manipulations**:

```
serial.h
seriallinux.c (serialw32.c)
```

- **System Platform Dependent Files**:

```
sysdep.h
- sysdeplinux.h
```

- sysdepw32.h
  - **Generic and Main Program Files**:

hc08sprg.h
main.c

  - **M68HC08 Specific Programming Files**:

prog.c

M68HC08 Image Operations

In order to perform the necessary operations with the M68HC08 code, the master software keeps a binary image of the M68HC08 memory. Also, the information about whether an actual byte is to be programmed into the MCU is stored. This is realized through following structure:

```
typedef struct {
        BYTE d[0x10000];     // data
        BYTE f[0x10000];     // valid flag 0=empty; 1=usercode; 2=systemcode
} BOARD_MEM;
```

where `image` is the actual variable defined as follows:

```
BOARD_MEM  image;
```

After the source S19 files are read, this array contains the actual data to be programmed into the MCU irrespective of its original order in the S19 file. The function `int read_s19(char *fn)` defined in `s19.c` implements the S19 file opening, reading, and translation from S19 hex-format into this binary array.

Interrupt Vector Table Translation

After the ident information is read out of the MCU, the following operations within the image are carried out:

- The code is scanned if any interrupt vectors are present between the MCU interrupt vector table address and `0xFFFF` (the last existing physical address of the M68HC08 MCU). *It doesn't make sense to translate data in S19 files without interrupt vectors, such as the S19 configuration data.*
- If interrupt vectors are present, translation of these vectors is done exactly as described in **Interrupt Vector Table Translation**. Then, the original address spaces in the interrupt vector table are marked as unused (thus not being reprogrammed).

These operations are executed in the function `int setup_vect_tbl(void)` defined in `prog.c` file.

Checking Memory Boundaries

The last check carried out before the code is actually programmed into the MCU is whether the code from the S19 file lies in the proper memory locations (between the memory boundaries that are reported by the MCU in the ident table).

If any value outside the range of addresses between the *Start address of reprogrammable memory area* and the *End address of reprogrammable memory area* is found, a warning is generated.

This check is done within `int check_image(void)` also defined in the `prog.c` file.

UART Manipulations

In files `seriallinux.c` or `serialw32.c`, the following UART manipulation functions are defined:

```
int init_uart(char* nm);
int close_uart(void);
int send_break10(void);
int flush_uart(int out, int in);
int wb(const void* data, unsigned len);
int rb(void* dest, unsigned len);
```

The pair `int init_uart(char* nm)` and `int close_uart(void)` manage opening (initialization) and closing of the specified UART port.

Another pair of functions `int wb(const void* data, unsigned len)` and `int rb(void* dest, unsigned len)` is used for writing and reading blocks of data into/out of UART.

There are two additional functions required for the bootloader to work: `int send_break10(void)` and `int flush_uart(int out, int in)`. The first one sends a BREAK character to the UART, the latter one cleans up both directions (in/out) of the UART buffers.

System Platform Dependent Files

The header file `sysdep.h` includes either `sysdeplinux.h` or `sysdepw32.h` depending on which platform software is being compiled. The platform-specific declarations are then used.

Generic and Main Program Files

The header file `hc08sprg.h` contains the rest of the platform non-specific declarations needed to compile the application. The tiny `main.c` containing the main program as shown at the beginning of this chapter (see **Figure 21**).

M68HC08 Specific Programming Files

The last, but most important part of the PC master bootloader software, is contained in the `prog.c` file. It implements most of the intelligence of the PC bootloader software as mentioned in previous chapters.

Numerous routines are implemented in the `prog.c` file:

```
int hook_reset(void)
int could_be_ack(unsigned b)
int calibrate_speed(void)
int read_mcu_info(void)
```

```
int setup_vect_tbl(void)
int check_image()
int read_blk(unsigned adr, int len, BYTE *dest)
int erase_blk(unsigned a)
int prg_blk(unsigned a, int len)
int prg_area(unsigned start, unsigned end)
int prg_mem(void)
int unhook(void)
```

Initial Hook (Waiting for MCU Reset)

Right after all initializations are done in the PC, a loop, in which any communication from the MCU is expected, is started. The `int hook_reset(void)` routine implements all necessary steps to establish initial communication with the MCU. The following routine is also an essential part.

Checking ACK

A routine, `int could_be_ack(unsigned b)`, checks whether a received character fits the possible set of characters that can be received due to a communication speed mismatch (See **Unknown MCU Communication Speed**).

Speed Calibration

A speed calibration loop, implemented in the `int calibrate_speed(void)` routine, follows the scenario fully described in the section entitled **Slave Frequency Calibration**. If no ACK is received from the MCU, another break character is sent.

MCU Info Reading

Right after the calibration is successfully finished, the PC requests the **Ident Command** so the MCU sends out the information about itself. This is achieved in the `int read_mcu_info(void)` routine.

Image Manipulations

The two functions `int setup_vect_tbl(void)` and `int check_image()` are described above (see **M68HC08 Image Operations**).

Block Operations

Three main data exchange operations are performed:

- Erase block
- Read block
- Write (program) block

These basic operations are implemented in the functions:

```
int erase_blk(unsigned a)
int read_blk(unsigned adr, int len, BYTE *dest)
int prg_blk(unsigned a, int len)
```

The actual implementation is very straight forward and follows the rules described in respective sections in **Interpreting MCU Commands**.

Main Programming Loop

The core of the bootloader's programming capabilities is implemented in the function `int` **`prg_area`**`(unsigned start, unsigned end)`. The task of this routine is to read data from an image, split it into blocks of appropriate size (minimum erase/write block sizes). Then the erase block and write block routines are called, respectively.

The routine also prints the progress information to the standard IO (for example, block boundary addresses, progress indicator).

One additional auxiliary function is included `int` **`prg_mem`**`(void)`. It retrieves the actual lowest and highest memory addresses that must be programmed and that are used for calling the `int` **`prg_area`**`(unsigned start, unsigned end)` function finally.

Final Unhook

Function `int` **`unhook`**`(void)` sends out the **Quit Command**.

Bootloading Procedure Demonstration

The bootloader binary code (S19 file) is loaded in the MCU like any other regular HC08 code (e.g. using MON08 serial programmer or other). Then the MCU is soldered (or socketed) in the application.

From now on, the user can download the HC08 user application code via SCI interface using the bootloader utility.

Bootloading Operation

Open a command prompt in Linux or Windows in the directory where the copy of `hc08sprg` executable and S19 files reside.

Assuming the serial board is connected (but not yet powered on) to first serial port (COM1, /dev/ttyS0) - invoke the bootloader using following sequence:

```
hc08sprg 1:D test.s19
```

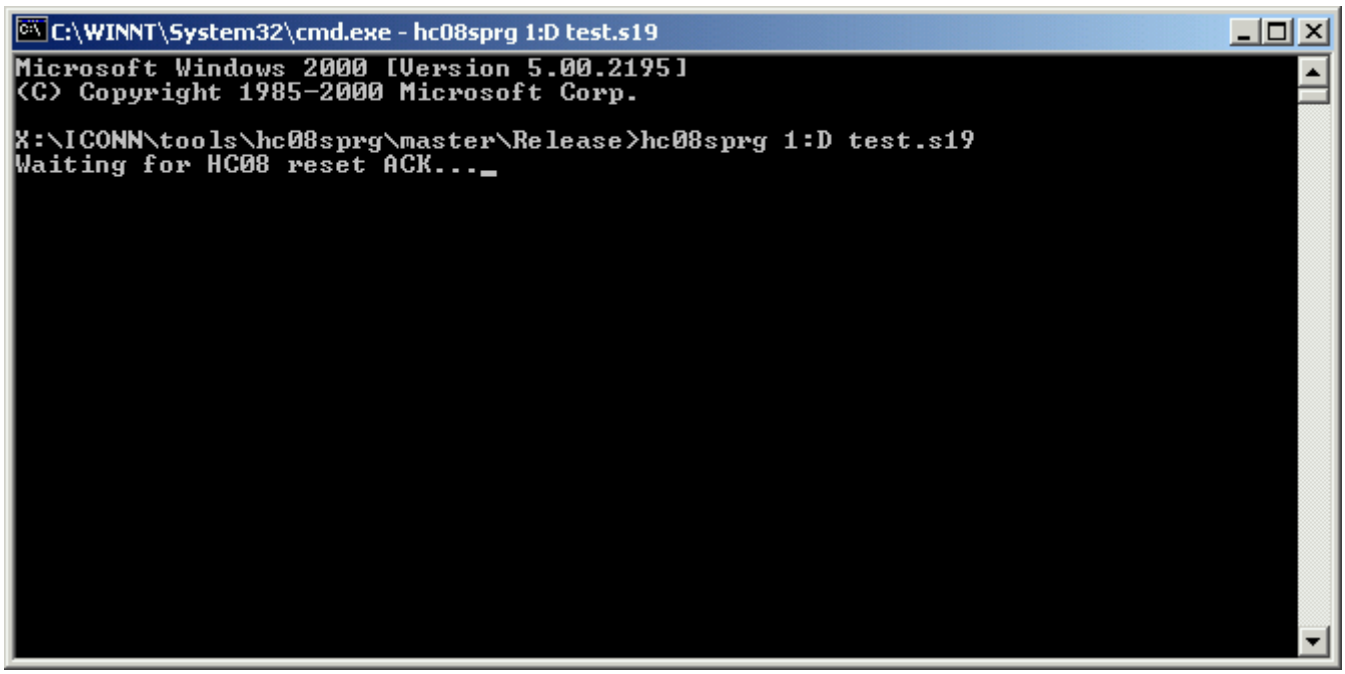


Figure 22. Bootloader Invocation

The bootloader now expects the ACK command to be received from the MCU bootloader enabled application. So now turn the power on for serial board and if all connections are OK, the MCU now communicates with the PC. The calibration procedure occurs (the bootloader version with unknown communication speed is used), followed by IDENT command. The information that are aquired from the MCU are then displayed on the screen as shown in **Figure 23**:

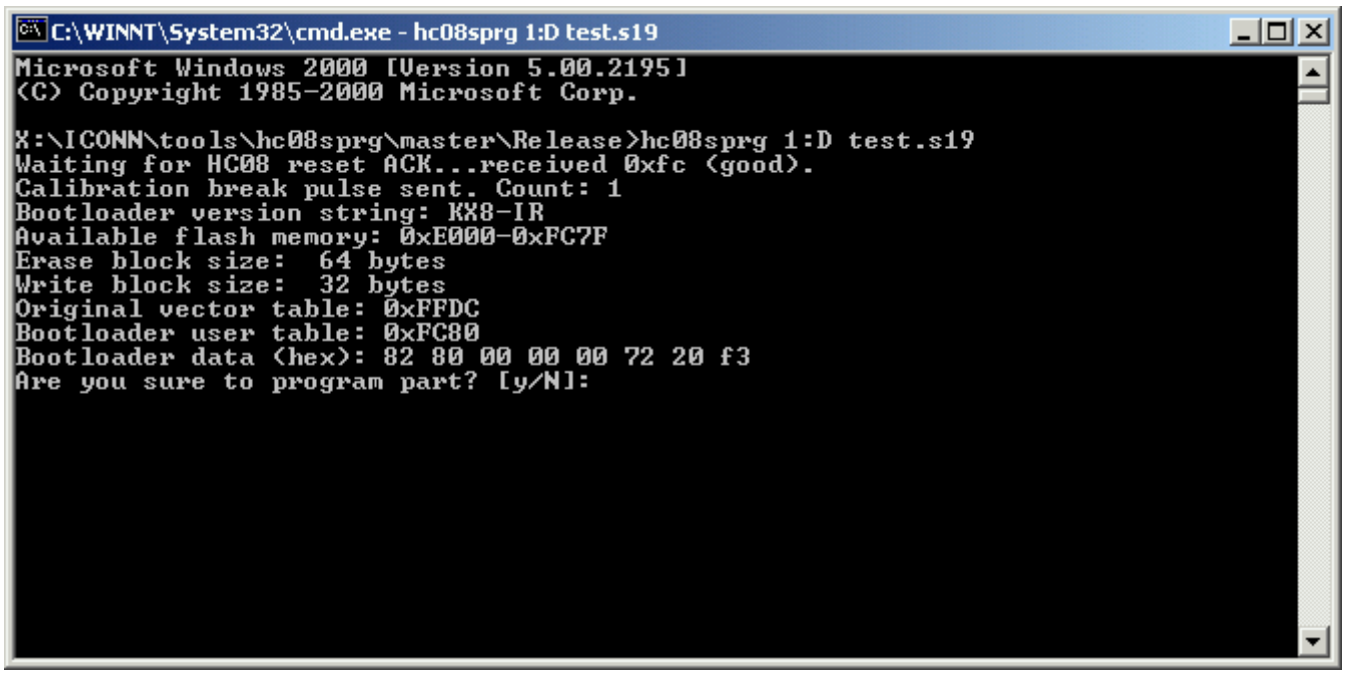


Figure 23. First Stage of Bootloading

Confirm by pressing ‘y’ and the bootloading (FLASH reprogramming) will proceed. The user application will be then started.

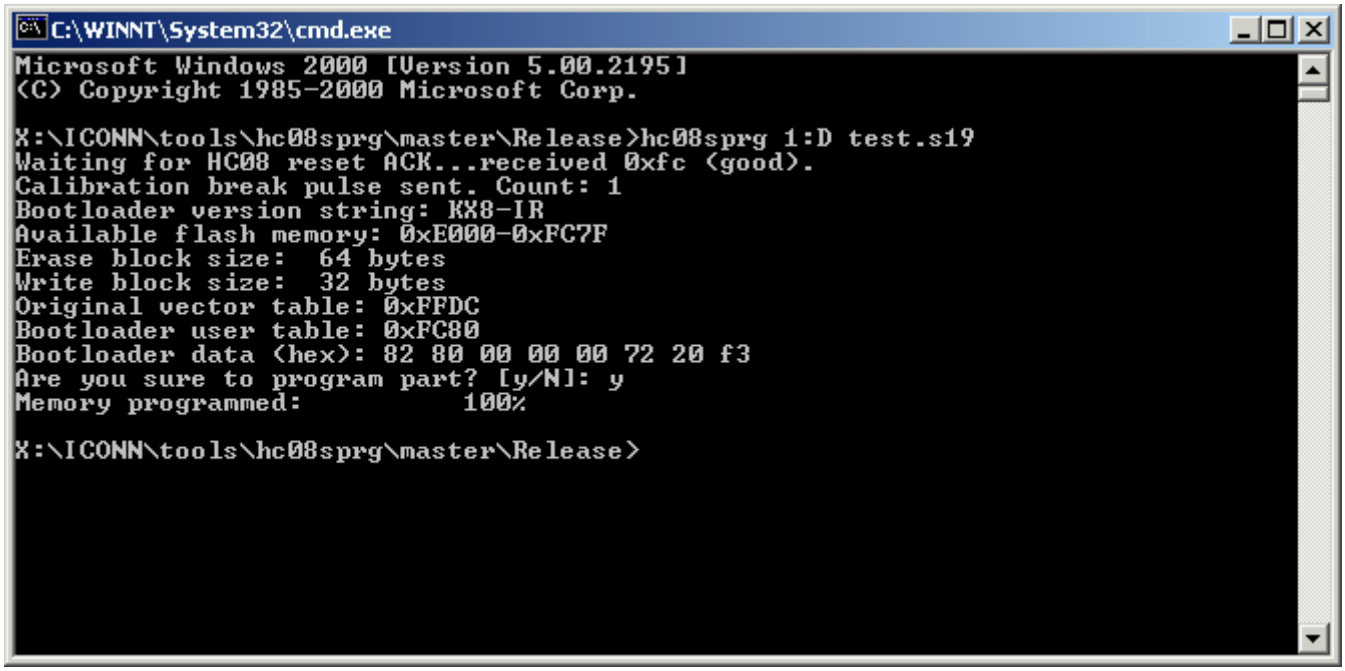


Figure 24. Bootloading Completed

Memory Boundary Overlap Example

If the user tries to bootload an application that will not fit in the actual MCU memory, a warning is displayed. The user may decide to continue, but very probably not all memory locations would be properly programmed (the user code is either out of available FLASH memory or it overlaps with the bootloader code).

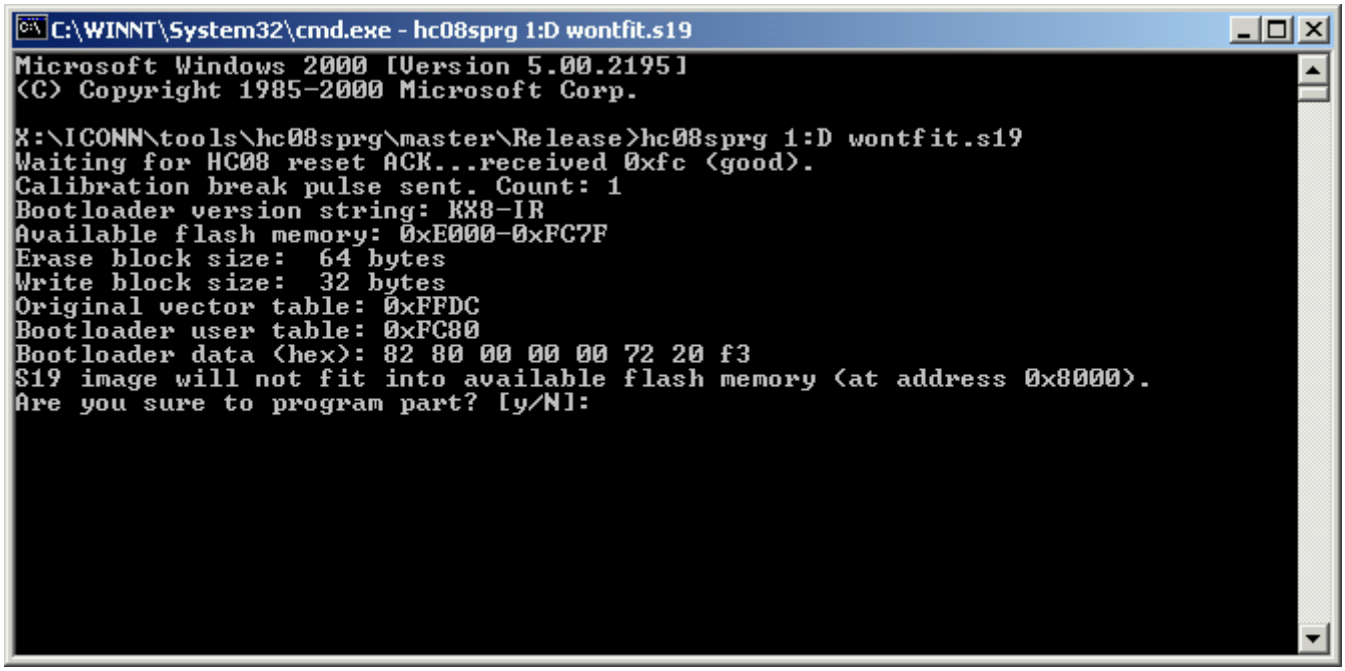


Figure 25. Memory Boundary Overlap Example

This page is intentionally left blank.

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

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

ASIA/PACIFIC:

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

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors