Developer’s Serial Bootloader for M68HC08 and HCS08 MCUs

By: Pavel Lajsner
Freescale Czech System Center
Roznov p. R., Czech Republic

Project Objectives

The developer’s serial bootloader for M68HC08 and HCS08 microcontroller units (MCUs) allows in-circuit reprogramming of Freescale Semiconductor’s M68HC08 and HCS08 FLASH devices using standard communication media, such as a serial asynchronous port. As soon as 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 utility that may be useful in developing applications.

This application note is 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 a complementary utility for either demo purposes or applications that were originally developed using MMDS and require minor modifications to be done in-circuit. The serial bootloader offers a zero-cost solution to applications already equipped with a serial interface and have SCI pins available on a connector.

This document also describes other programming techniques, including:

- FLASH reprogramming using ROM routines
- Simple software SCI
- Use of the internal clock generator
- PLL clock programming
- EEPROM programming (AS/AZ HC08 families)

This product incorporates SuperFlash® technology licensed from SST.
© Freescale Semiconductor, Inc., 2005. All rights reserved.
Project Objectives

Project Goals

Freescale Semiconductor M68HC08 MCUs use a standard monitor mode interface for FLASH programming. Configuration of monitor mode requires a specific clock and high voltage (monitor mode entry voltage $V_{TST} = V_{DD} + 2.5 = 8\, \text{V}$) applied to the IRQ pin upon MCU startup. Also, establishing monitor mode communication uses a few pins. 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.

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

Requirements

This bootloader application requires:

- **Low memory use** — The bootloader must use as little memory 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 in this document implements all features as simply as possible, excluding checksums, etc. The target size is less than 500 B.

- **Low pin-count** — This bootloader uses standard (already implemented) means of communication (typically SCI on boards that are primarily intended for communication). The standard SCI uses two 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 M68HC08 and HCS08 bootloader applications require modification to interrupt vectors or other modifications to the S19 file for it to accept the bootloader.
Demo Features of Bootloader Application

This document describes several different M68HC(S)08 bootloader implementations which vary mainly because the target M68HC(S)08 MCUs have different features. Several features of the M68HC(S)08 Family are also demonstrated, making this document useful to a wider audience than those who require only the bootloader. The different M68HC(S)08 implementations also demonstrate the following features:

- Use of built-in ROM routines for FLASH self-programming (see also AN1831/D in References).
- User implementation of in-circuit reprogramming routines on ROM-less MCUs, such as the MC68HC908GP32 Family or MC9S08GB/GT Family
- Use of different implementations of the FLASH block protection technique (MC68HC908GP/GR/KX vs. MC68HC908JK/JL Families)
- Implementation of software SCI on SCI-less MCUs, such as the MC68HC908JK/JL Family
- Use of the internal clock generator and its trimming (for the MC68HC908KX Family)
- EEPROM programming (for the MC68HC908AS/AZ Family)

FC Protocol Description

As described in Requirements, an implementation must be as simple as possible and use as little memory as possible. Therefore, 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 used.

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

Figure 2 is a simplified state diagram that shows separate states of the bootloader, which are described in this document.
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 logic 0 appears on an IRQ pin during MCU startup, the bootloader code starts. Otherwise, the user code starts.

Because the developer’s serial bootloader application must 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.

If this becomes a limitation for any reason, the user may modify the bootloader code to meet the application needs; for example, an additional simple IRQ pin test at startup can be implemented). See more in M68HC08 System Limitations.

Clock Source

FC protocol allows two scenarios, depending on whether the MCU runs on a known and exact frequency or uses an RC (resistor, capacitor) clock or internal clock (or any 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 whether an incoming ACK character conforms only to the $FC pattern. Because of the MCU clock tolerance, several characters can be interpreted differently instead of the original $FC sent out by the PC as shown in Figure 3. The $FC pattern check on the MCU side can be eliminated completely, which saves MCU memory.
Table 1 shows the characters that can 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**

<table>
<thead>
<tr>
<th>PC Data Rate</th>
<th>MCU Data Rate</th>
<th>Character Received in Binary</th>
<th>Character Received in Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>9600</td>
<td>9600*1/3</td>
<td>11111111b</td>
<td>$FF</td>
</tr>
<tr>
<td>9600</td>
<td>9600*2/3</td>
<td>11111110b</td>
<td>$FE</td>
</tr>
<tr>
<td>9600</td>
<td>9600*3/3</td>
<td>11111100b</td>
<td>$FC</td>
</tr>
<tr>
<td>9600</td>
<td>9600*4/3</td>
<td>11111000b</td>
<td>$F8</td>
</tr>
<tr>
<td>9600</td>
<td>9600*5/3</td>
<td>11110000b</td>
<td>$F0</td>
</tr>
<tr>
<td>9600</td>
<td>9600*6/3</td>
<td>11000000b</td>
<td>$E0</td>
</tr>
<tr>
<td>9600</td>
<td>9600*7/3</td>
<td>10000000b</td>
<td>$C0</td>
</tr>
<tr>
<td>9600</td>
<td>9600*8/3</td>
<td>00000000b</td>
<td>$80</td>
</tr>
<tr>
<td>9600</td>
<td>9600*9/3</td>
<td>00000000b</td>
<td>$00</td>
</tr>
</tbody>
</table>
FC Protocol Description

If the MCU transmits to the PC at an unmatched data rate, the PC receives (and accepts) characters that are different from the $FC character. The PC accepts all characters from the mentioned set ($FF, $FE, $FC, $F8, $F0, $E0, $C0, $80, $00). If a character is received, an 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 distortion.

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

Slave Frequency Calibration

During this phase, the MCU clock is calibrated. Up to now, the PC has communicated with the MCU at a speed which could be from 33% to 300% tolerance. During this phase, the MCU communication speed must be adjusted to match the PC communication speed.

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

A break character consists of 10 consecutive logical zeros. For example, at a 9600 baud data rate, its high-low-high pulse lasts $10 \times 10^4 \mu s = 1.04 \text{ ms}$.

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 used). This can be repeated as many times as needed for the MCU to achieve the proper clock speed.

After the MCU is calibrated to the correct clock (or after the receive routines are calibrated), the ACK character is sent to the PC to stop sending calibration characters. See Figure 4.
If the MCU is operating at the correct data rate (no calibration is possible or needed and the MCU clock is crystal driven), the PC can send an ACK immediately, skipping the calibration phase entirely. See Figure 5.

Figure 4. Start-Up Communication with Calibration
Interpreting MCU Commands

After communication between the MCU and the PC is established, the MCU enters the main command interpreter loop. The MCU executes simple commands 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

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

Figure 6 illustrates a typical command and response.
Ident Command

The indent 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 indent 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

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 ignored by the MCU and no response is sent back to the PC. The 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 0.

VER — Protocol Version
FC Protocol Description

**FC Protocol Version 1 (M68HC08)**

Version 1 of the protocol is for M68HC08 MCUs. In version 1, 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 device-specific implementation; compared in Table 2) — 8 bytes
- Identification string, zero terminated — <n> bytes

**Figure 8. Ident Command (FC Protocol Version 1, M68HC08)**

**FC Protocol Version 2 (HCS08) and FC Protocol Version 3 (large M68HC08)**

Version 2 of the protocol is for HCS08 MCUs, version 3 is for large M68HC08 (HC08 with two or more FLASH memory banks). In both versions, additional fields are defined as:

- System device Identification register content — 2 bytes (unused in protocol version 3, coded as $FFFF)
- Number of reprogrammable memory areas (N) — 1 byte
- Start address of reprogrammable memory area #1 — 2 bytes
- End address of reprogrammable memory area #1 + 1 — 2 bytes
- Start address of reprogrammable memory area #2 — 2 bytes
- End address of reprogrammable memory area #2 + 1 — 2 bytes
- ...
- Start address of reprogrammable memory area #N — 2 bytes
- End address of reprogrammable memory area #N + 1 — 2 bytes
- Address of relocated interrupt vector 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
- Identification string, zero terminated — <n> bytes
Erase Command

The erase command (coded as ‘E’, $45) has only an address field, no length or data fields. The start address is a 2-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.

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.
FC Protocol Description

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 2-byte field, MSB first. The length is a 1-byte field.

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

![Figure 12. Read Command](image)

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 13. Quit Command](image)

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

Figure 14 shows the typical memory allocation for M68HC908 MCUs with the bootloader pre-programmed. For example, the MC68HC908KX8 MCU 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 14. Simplified Example of Memory Allocation in MC68HC908KX8]
Memory Allocation

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 specific MCU data sheet for details).

FLASH Block Protection Register (FLBPR)

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

NOTE

Some M68HC908 MCUs have an FLBPR register in RAM instead of FLASH (e.g., the MC68HC908JK/JL Families). The bootloader code sets this register properly but the user code can eventually modify FLBPR and erase/write the bootloader code. See FLBPR Not Usable (in Some M68HC08 Family MCUs).

For example, the MC68HC908KX8 bootloader to the PC memory allocation is:

- $01 — Version 1, read command not implemented (bit 7)
- $E000 — Start address of reprogrammable memory area
- $FC80 — End address of reprogrammable 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 M68HC08 implementations provide different values (for example, the sixth value in the MC68HC908KX8 implementation is the value of the internal clock generator (ICG) 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 ICG to the best known value before user code start).
- ‘KX8-IR’,0 — Identification string, zero terminated. Information to be displayed on the PC screen.

Interrupt Vector Table Relocation

Because the FLASH block protection technique also protects the interrupt vector table from being overwritten, some method must be used to relocate these vectors to the different locations. To do this, the bootloader user table is used. It is a part of memory that is not protected by the FLBPR, but it 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.
Figure 15 shows interrupt vector table relocation for M68HC08 MCUs. Note that in a standard interrupt vector table, each record is 2 bytes long (each vector is a 16-bit address). This is different from the bootloader user table, for which each record is 3 bytes long — a JMP opcode ($CC) plus a 16-bit address.

**Figure 15. Interrupt Vector Table Relocation (M68HC08 MCUs)**
FC Protocol, Version 1, M68HC908 Implementation

S19 File

Because bootloader operation must be transparent to the user S19 file, another piece of intelligence is built into the PC master code (instead of the MCU slave). The relocation works like this:

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

Using this method, the user S19 file does not need to be modified, but 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 explained in Each Interrupt 3T Delayed.

User Code Start

To provide a register setup similar to the way it appears after MCU reset, the user code is started in an unusual way.

Software Reset

If the bootloader must quit and run user code, an illegal operation is intentionally executed (M68HC08 illegal opcode $32). This causes an illegal operation reset and the MCU restarts. During bootloader startup, the system integration module (SIM) reset status register (SRSR) is tested. If a 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 (such as illegal address, etc.) with only a short additional delay caused by testing of the SRSR register and executing associated jump instructions.

Hardware Reset

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 reset pin.

Another test has been added to the real bootloader application: If no source of reset is detected (i.e., if the SRSR register is 0), the bootloader is selected by default. 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.

Because the SRSR register is one-time readable (it clears after read), no subsequent reads of this register provide a valid value. See M68HC08 System Limitations for details.

M68HC08 System Limitations

This section summarizes limitations that must be considered when using the bootloader with the user application.
Memory Occupied

One of the most important requirements is to use 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, the memory consumption is near the lower limit. Larger M68HC08 MCUs (which are not usually equipped with ROM code for FLASH programming) will require approximately 500 bytes of FLASH of the total 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 signals the actual FLASH addresses that are available. The PC Bootloader software will not allow programming if the user code overlaps with bootloader code.

Time Delay Upon Start-Up and Initial Communication

The number of pins that have specific meaning during bootloader start-up must be as small as possible. Especially in communication systems (for example, those using a standard serial port), pin overhead is zero and a “certain level character at a certain time” method is used. So, the bootloader waits a certain amount of time to receive an answer from the PC at startup. If none is received, the user code starts. The typical delay is in the range of several hundred milliseconds.

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

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 relocation (as described in Interrupt Vector Table Relocation) has been chosen as the best solution for achieving user code transparency and security of the bootloader code.

The interrupt latency is about 10 to 15T (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; see FLASH Block Protection Register (FLBPR) for details).

Some M68HC08 MCUs (such as the KX, GP, and GR devices) have this FLASH block protection register stored in FLASH, so it cannot be modified in user mode. The FLBPR can be erased or programmed only with an external voltage, $V_{TST}$, present on the IRQ pin (normal monitor mode). Because this feature is completely dedicated to bootloader code protection, it is unavailable to the user application code. If the value for FLPBR appears in the user S19 code, a warning is displayed. Such an occurrence should be omitted from user S19 code.
Some families have the FLASH block protection register stored in RAM instead (the MC68HC908JK/JL Families are like this). The bootloader sets the correct value at the beginning of its execution to protect itself. However, user code can modify this register and protect its own memory areas as needed. This also implies that the bootloader is not 100% protected from user code.

See the specific MCU data sheet 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 will run. Because 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 is stored in one RAM location. Unfortunately, its memory location may differ from one implementation to another. If the application requires the SRSR register and bootloader, the user must redirect the SRSR reading to this specific RAM location. This location can be obtained from the bootloader’s MAP file.
This section describes features that are specific to the HC9S08 bootloader implementation. The memory allocation is heavily MCU specific so the meaning of variables is explained in this section.

Figure 16 shows the memory allocation typical to the HC9S08 devices with the bootloader pre-programmed. For example, the MC9S08GB/GT60 device memory map includes:

- 60 Kbytes of FLASH memory ($1080–$17FF, $182C–$FFAF)
- 4 Kbytes of random-access memory (RAM) ($0080–$107F)
- 16 bytes of nonvolatile registers ($FFB0–$FFBF)
- 64 bytes of user-defined vectors ($FFC0–$FFFFFF)

**Figure 16. Simplified Example of Memory Allocation in MC9S08GB/GT60**

### Memory Allocation

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

By setting a FLASH protection register, all address space above this address is protected from both intentional and unintentional erasing/re-writing. After the bootloader and the FLASH protection register are programmed into memory, the bootloader code is protected from unintentional modification by user code.

NOTE

See FLASH Protection Technique Not Usable for limitations.

Example Memory Allocation

For example, the MC9S08GB/GT60 bootloader to the PC memory allocation is:

- $82 — Version 2, read command implemented (bit 7)
- $r002 — System device identification register (SDIDR) content ($002 for GB/GT Family, r (four top bits) is chip revision number reflecting current silicon level
- $02 — Number of reprogrammable memory areas
- $1080 — Start address of reprogrammable memory area #1
- $1800 — End address of reprogrammable memory area #1 + 1
- $182C — Start address of reprogrammable memory area #2
- $FDC0 — End address of reprogrammable memory area #2 + 1
- $FDC0 — Address of relocated interrupt vector table
- $FFC0 — Start address of MCU interrupt vector table
- $0200 — Length of MCU erase block
- $0040 — Length of MCU write block
- ‘GB/GT60’,0 — Identification string, zero terminated. Information to be displayed on PC screen

Interrupt Vector Table Relocation

If FLASH protection is enabled, the reset and interrupt vectors would be protected. Vector redirection (HCS08 hardware feature) allows the user to modify memory allocation of interrupt vector information.

Vector redirection is enabled by programming the NVOPT (nonvolatile option) register. For redirection to occur, at least some portion—but not all—of the FLASH memory must be block protected by programming the NVPROT (nonvolatile protection) register. All of the interrupt vectors (memory locations $FFC0–$FFFD) are redirected, but the reset vector ($FFFE:FFFF) is not.

For example, if 512 bytes of FLASH are protected, the protected address region is from $FE00 through $FFFF. The interrupt vectors ($FFC0–$FFFD) are redirected to the locations $FDC0–$FDFD.

If an SPI interrupt is taken—for example—the values in the locations $FDE0:FDE1 are used for the vector instead of the values in the locations $FFE0:FFE1. This allows the user to reprogram the unprotected portion of the FLASH with new program code including new interrupt vector values while leaving the protected area, which includes the default vector locations, unchanged.
**S19 File**

Because bootloader operation must be transparent to the user S19 file, another piece of intelligence is built into the PC master code (instead of the MCU slave). If the record in the interrupt vector table is detected in the user S19 file, the vector is relocated into the corresponding area in the relocated interrupt vector table. For example, if the user S19 file contains #2 interrupt vector at address $FFEA, such a vector is relocated to the $FDEA address in the relocated interrupt vector table.

Using this method, the user S19 file **does not need to be modified**, but the lower address of the end of FLASH memory must be considered.

Figure 17 illustrates HC9S08 interrupt vector table relocation.
**User Code Start**

To provide a register setup similar to the way it appears after MCU reset, the user code is started in an unusual way.

**Software Reset**

If the bootloader must quit and run user code, an illegal operation is intentionally executed (HCS08 illegal opcode $8D). This causes an illegal operation reset and the MCU restarts. During bootloader startup, the
system reset status register (SRS) is tested. If a 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 (such as illegal address, etc.) with only a short additional delay caused by testing of the SRSR register and executing associated jump instructions.

**Hardware Reset**

In some implementations, a pin reset (caused by external reset pin) is 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 HCS08 MCU reset pin.

**HCS08 System Limitations**

This section summarizes limitations that must be considered when using the bootloader with the user application.

**Memory Occupied**

One of the strongest requirements is to use the smallest code possible. Typical HC9S08 implementations are 432 bytes (minimal memory size that can be protected) plus another 64 bytes page for relocated interrupt vector table.

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 HCS08 MCU signals the actual FLASH addresses that are available. The PC Bootloader software will warn before programming if the user code overlaps with bootloader code.

**Time Delay Upon Start-Up and Initial Communication**

The number of pins that have specific meaning during bootloader start-up must be as small as possible. Especially in communication systems (for example, those using a standard serial port), pin overhead is zero and a *certain character at a certain time method* is used. So, the bootloader waits a certain amount of time to receive an answer from the PC at startup. 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 modify the bootloader code and use a *certain level on certain pin* method instead. A simple test of the voltage level on the IRQ pin (or any other input pin) could be used to decide whether the bootloading sequence is required.

**FLASH Protection Technique Not Usable**

The bootloader uses a FLASH block protection technique to protect itself from being overwritten, therefore this feature is not available for the user code. This includes FLASH memory security related registers (namely NVPROT, NVOPT, and NVBACKKEY), which are used for protection and interrupt vector relocation by bootloader.
FC Protocol, Version 3, large M68HC08 Implementation

This section describes features that are specific to the protocol version 3 of bootloader. It is intended for large HC08s (with two or more FLASH memory banks, or more precisely with two or more separated FLASH memory areas). The format of Ident Command from version 2 is used, the rest remains the same as with protocol version 1 (HC08) — namely Interrupt Vector Table Relocation.
MCU Slave Software

This section provides a detailed description of the three typical M68HC(S)08 bootloader implementations. All code is written in assembly language. Several selected targets and different features are described as shown in Table 2.

Table 2. Target Implementation Comparison

<table>
<thead>
<tr>
<th>MCU Family</th>
<th>FLASH Memory Use (in Bytes)</th>
<th>Clock Source</th>
<th>ROM Routines Usage</th>
<th>Calibration Conducted</th>
<th>SCI</th>
<th>FLASH Erase Page Size (in Bytes)</th>
<th>FLASH Program Page Size (in Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC68HC908AP AP8/AP16/ AP32/AP64</td>
<td>592</td>
<td>32768 Hz XTAL or external clock.</td>
<td>Yes, different version</td>
<td>No</td>
<td>Hardware</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>MC68HC908AB/AS/AZ AB32/AS32/AZ32 AS60/AZ60</td>
<td>640</td>
<td>4.9152MHz XTAL</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>MC68HC908EYEY16</td>
<td>384</td>
<td>ICG</td>
<td>Yes</td>
<td>Yes</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908GP GP32</td>
<td>512</td>
<td>32768 Hz XTAL or external clock.</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>MC68HC908GR GR4/GR8/GR16 GR8A/GR16A</td>
<td>320</td>
<td>32768 Hz XTAL or external clock; 8MHz XTAL (A Family)</td>
<td>Yes</td>
<td>No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908GT GT8/GT16</td>
<td>384</td>
<td>ICG</td>
<td>Yes</td>
<td>Yes</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908GZ GZ8/GZ16</td>
<td>512</td>
<td>8 MHz XTAL</td>
<td>Yes</td>
<td>No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908GZ GZ60</td>
<td>512</td>
<td>8 MHz XTAL</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>MC68HC908JK/JL JK1/JL1/ JK3/JL3</td>
<td>395</td>
<td>XTL, RC oscillator or ext. source</td>
<td>Yes</td>
<td>Yes</td>
<td>Software, single-wire possible</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908JK/JL JK8/JL8</td>
<td>384</td>
<td>4.9152MHz XTAL</td>
<td>Yes, different version</td>
<td>No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908LB LB8</td>
<td>384</td>
<td>ICG</td>
<td>Yes</td>
<td>Yes</td>
<td>Software, single-wire possible</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908LJ LJ12/ LJ/LK24</td>
<td>324</td>
<td>32768 Hz XTAL or external clock.</td>
<td>Yes, different version</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
</tbody>
</table>
### Table 2. Target Implementation Comparison

<table>
<thead>
<tr>
<th>MCU Family</th>
<th>FLASH Memory Use (in Bytes)</th>
<th>Clock Source</th>
<th>ROM Routines Usage</th>
<th>Calibration Conducted</th>
<th>SCI</th>
<th>FLASH Erase Page Size (in Bytes)</th>
<th>FLASH Program Page Size (in Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC68HC908KX KX2/KX8</td>
<td>384</td>
<td>ICG</td>
<td>Yes</td>
<td>Yes</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908MR MR8</td>
<td>461</td>
<td>PLL with XTAL (4 MHz)</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908MR MR16/MR32</td>
<td>461</td>
<td>PLL with XTAL (4 MHz)</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>MC68HC908QB QB4/QB8</td>
<td>362/302</td>
<td>QB/QC ICG</td>
<td>Yes</td>
<td>Yes/No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908QC QC8/QC16</td>
<td>387/323</td>
<td>QB/QC ICG</td>
<td>Yes</td>
<td>Yes/No</td>
<td>Hardware</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908QT/QY QT1/QT4/QY1/QY4</td>
<td>320</td>
<td>Simpler ICG</td>
<td>Yes</td>
<td>Yes</td>
<td>Software, single-wire possible</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>MC68HC908SR SR12</td>
<td>512</td>
<td>32768 Hz XTAL</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>MC9S08GB/GT HCS08GB/GT32 HCS08GB/GT60</td>
<td>576</td>
<td>HCS08 ICG</td>
<td>No</td>
<td>Yes</td>
<td>Hardware</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>MC9S08Rx HCS08RD/RG/RE8 HCS08RD/RG/RE16 HCS08RD/RG/RE32 HCS08RD/RG/RE60</td>
<td>335</td>
<td>16MHz XTAL</td>
<td>No</td>
<td>No</td>
<td>Hardware</td>
<td>512</td>
<td>64</td>
</tr>
</tbody>
</table>
MC68HC908KX

The M68HC908KX Family has an internal clock generator (ICG) module. This allows a very effective implementation of the bootloader without a crystal.

The on-chip FLASH programming routines simplify the bootloader and improve memory use. The communication between the MCU and PC uses a standard serial channel (SCI).
Figure 18. MC68HC908KX Bootloader Flowchart
**Internal Clock Generator (ICG) — Initialization**

The ICG is simple to initialize. Because the ICG is active and the 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 indicates 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 — 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 AN1831/D provides the procedure for calculating the trim factor from 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 KK8 IS IN SYNC SERIALLY WITH
* THE HOST. IF WE STAY IN THE LOOP FOR > 256 LOOP CYCLES, THEN THE KK8
* MUST BE RUNNING FASTER THAN EXPECTED, AND NEEDS TO BE SLOWED DOWN. IF WE
* STAY IN THE LOOP FOR < 256 LOOP CYCLES THEN THE KK8 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%.

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
```
MCU Slave Software

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 AN1831/D see References.

MC68HC908JK/JL

MC68HC908JK/JL MCUs are among the least expensive in the M68HC08 Family, and they no hardware SCI. Therefore a software SCI must be implemented. This allows the unrestricted selection of which pins are used for serial communication (the provisions are made in the code that also an IRQ pin can be used as an input serial line, too).

The MC68HC908JK/JL Family has an RC version (an RC oscillator is used instead of a crystal). The bootloader’s calibration compensates for any variation in speed. If the desired clock frequency is outside of the specified range covered by the calibration system, the code must be modified.

The MC68HC908JK/JL Family has on-chip FLASH programming routines. Using FLASH programming saves memory.

The main program flowchart (shown in Figure 19) is very similar to the previous case.
Figure 19. MC68HC908JK/JL Bootloader
**Software-SCI Transmit Char Routine**

A detailed description of the software-SCI transmit and receive subroutines is provided in this section. They both are based on a 16-bit timer and the output-compare event is polled in the background loop.

The source code for the two routines is shown in Figure 21. Other than a few counters, a 16-bit `ONEBIT` variable is used. It contains the actual length of 1 bit at the current communication speed in 16-bit timer clock cycles. This variable is initialized during the calibration phase (see Slave Frequency Calibration).
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

Figure 21. Software-SCI Transmit Char Routine Source Code
Software-SCI Receive Char Routine

The software-SCI receive routine is similar to software-SCI transmit. 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 because of memory restrictions. Figure 22 shows the software-SCI receive routine flowchart, and the source code is provided in Figure 23.

Diagram:

Figure 22. Software-SCI Receive Char Routine
,**--------------------------------------------------------------------------------**
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
           ; shift data right (highest bit cleared)
  LSRA
  BRRXDLO RXDLOW          ; skip if RXD low
  ORA     #$80            ; set highest bit if RXD high
  RXDLOW: LDHX    ONEBIT
           ; and clear TOF
  STHX
  STDNZ  BITS,SCIRX2
           ; and loop for next bit
  BRA     EPILOG

Figure 23. Software-SCI Receive Char Routine Source Code

Macros

Several macros are defined across the two codelistings. They improve the readability or memory consumption. They are provided in Figure 24.
Figure 24. Software-SCI Macros Source Code
MC68HC908GP

MC68HC908GP MCUs have no on-chip FLASH programming routines available. Therefore, all FLASH programming must be done by the bootloader and demonstrated in this section.

MC68HC908GP MCUs are primarily targeted for use with a low-cost 32.768 kHz crystal. Because the frequency of the crystal is known, no calibration is performed, which saves MCU memory. Therefore, this MCU uses the Known MCU Communication Speed method.

Figure 25 is a flowchart of the MC68HC908GP bootloader process.
Figure 25. MC68HC908GP 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 shown in Figure 26. Three main subroutines are defined:

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

Because the flow is straight-forward, no flowchart is provided. Basically the sequence of events is executed according to FLASH erasing/programming specifications.

```
;*******************************************************************************************
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
AI X #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:
;*******************************************************************************************
```
MCU Slave Software

WR_ALG:

LDA   #00000001            ; PGM bit on
STA   FLCR
LDA   FLBPR                ; dummy read FLBPR
LDHX  ADRS                  ; prepare addresses
STA   X                      ; and write to desired range
D_US  #T10US                  ; wait 10us
LDA   #00001001              ; set HVEN, keep PGM
D_US  #T5US                   ; wait 5us
LDHX  #DAT                    ; prepare addresses
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              ; 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

Figure 26. FLASH Programming Routines Source Code
For improved readability, two timing macros (\texttt{D\_US} and \texttt{D\_MS}) are used in the code (shown in Figure 27).

\begin{verbatim}
;*******************************************************************************************
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

Figure 27. FLASH Programming Macros Source Code
\end{verbatim}

**MC68HC908GR**

MC68HC908GR MCUs are smaller members of the MC68HC908GP Family that are equipped with ROM memory with on-chip FLASH programming routines available in the user mode.

MC68HC908GP and MC68HC908GR MCUs are primarily targeted for use with a low-cost 32.768 kHz crystal. Because the frequency of the crystal is known, no calibration is performed, which saves MCU memory. Therefore, these MCUs use the Known MCU Communication Speed method.

**MC68HC908MR**

MC68HC908MR MCUs are motor-control oriented members of the M68HC08 Family. The MC68HC908MR MCUs have no on-chip FLASH programming routines available. Therefore, all FLASH programming must be done by the bootloader itself.

The MC68HC908MR Family has a PLL (phase-locked loop) 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 therefore 32 MHz and the bus frequency is 8 MHz.

Because the frequency of the crystal is known, no calibration is performed, which saves MCU memory. Therefore, these MCUs use the Known MCU Communication Speed method.
MC68HC908GT
MC68HC908EY

The code for MC68HC908GT and MC68HC908EY MCUs is similar to the MC68HC908KX code, except for the memory maps and ROM routine locations. One minor difference is that MC68HC908GT Family cannot use the CGMXCLK clock as the SCI module source. Thus the bus clock is the only possible clock source.

MC68HC908QT/QY

MC68HC908QT/QY MCUs are the smallest members of the M68HC08 Family. They have a simple ICG module (running on fixed frequency 12.8 MHz ±25%). ROM routines are available.

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

SCI Application Program Interface (SCI-API)

Software SCI communication is implemented on MC68HC908QT/QY, MC68HC908JK/JL and MC68HC908LB MCUs to reduce cost and enable the user code to call the SCI send and receive routines (with certain limitations). The bootloader code now implements so-called SCI-API which 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 described in this file too. 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.

Single-Wire Communication

Because of the small number of pins on MC68HC908QT devices, the single-wire SCI version has been developed to keep the number of pins that are occupied by communication to a minimum. Figure 28 illustrates an example single-wire RS-232 interface. The single-wire option has been ported to MC68HC908JK/JL and MC68HC908LB bootloader because they use a software SCI also.

![Figure 28. Example Single-Wire Schematic](image-url)
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.exe 1:S filename.s19
```

where 1 specifies which COM port is used for communication, and S stands for single-wire.

**Original (old) format:**
```
hc08sprg.exe 1 filename.s19
```

**Now defaults to:**
```
hc08sprg.exe 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.exe 1:? filename.s19
```

The detection is only possible if the 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**

MC68HC908LJ MCUs are members of the M68HC08 Family used to drive LCD displays. MC68HC908LJ MCUs have the ROM on-chip FLASH programming routines available. The calling convention is slightly different from other M68HC08s (see MC68HC908LJ data sheet, monitor ROM section).

MC68HC908LJ MCUs are primarily targeted for use with a low-cost 32.768 kHz crystal. Because the frequency of the crystal is known, no calibration is performed, which saves MCU memory. Therefore, these MCUs use the Known MCU Communication Speed method.

**MC68HC908AP**

MC68HC908AP devices are members of the M68HC08 Family that have two SCIs (the SCI channel must be selected at compile time). MC68HC908AP MCUs have ROM on-chip FLASH programming routines available. The calling convention is slightly different from other M68HC08s (same as MC68HC908LJ devices).

Because of the simplicity of the internal oscillator, it does not have the accuracy and stability of the RC oscillator or the XTAL oscillator. Therefore, the internal oscillator is not suitable if an accurate bus clock is required and it should not be used as source for bus clock.

A low-cost 32.768 kHz crystal was selected as default source clock for the bootloader and user application. Because the frequency of the crystal is known, no calibration is performed, which saves MCU memory. Therefore, these MCUs use the Known MCU Communication Speed method.

**MC68HC908AB/AS/AZ**

MC68HC908AB/AS/AZ devices are members of the M68HC08 Family that have (in addition) EEPROM memory. This code also demonstrates the way how to program these EEPROM cells using AUTO (automatic clear of EEPGM) mode.
MCU Slave Software

Since the memory map is not continuous, FC protocol version 3 needs to be used (it allows the “holes” in the memory map, i.e., several separate memory blocks).

MC9S08GB/GT

MC9S08GB/GT devices are the first members of the HCS08 Family. Because of different hardware features and FLASH memory allocation, another version of the protocol was required. The protocol is detected automatically by the latest `hc08sprg.exe` PC Bootloader software and becomes invisible to the user.

MC9S08GB/GT MCUs have two SCIs (the SCI channel must be selected at compile time).

These MCUs have no on-chip FLASH programming routines. Therefore, all FLASH programming must be done by the bootloader and is demonstrated in this implementation (it has been entirely adopted from HCS08 Family Reference Manual Volume 1 (Freescale Semiconductor order number HCS08RMv1/D; see References).
PC Bootloader Master Software

This section provides a detailed description of the bootloader host computer master software, which is downloadable as a zip file from the Freescale Semiconductor website, http://www.freescale.com. All code is written in C language and is compatible with 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.

In this section, portions of the master bootloader code will be described in more detail. All actions required for reprogramming the M68HC(S)08 device are fully described in the slave implementation and protocol sections of this document. The specific master characteristics are emphasized.

The host computer master software design is straightforward and is a sequence of several steps (which are shown in Figure 29):

1. Opening serial port
2. Opening source S19 file
3. Waiting for reset of MCU
4. Calibrating MCU
5. Reading MCU information
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
Figure 29. Bootloader Master Flowchart

File Structure

The following file structure is set up:

- 8-Bit MCU 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
- M68HC(S)08 Specific Programming Files:
  - prog.c

NOTE: X(2) MEANS EXIT WITH EXIT CODE 2
8-Bit MCU Image Operations

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

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

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

```c
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 relocation from S19 hexadecimal format into this binary array.

**Interrupt Vector Table Relocation**

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

- The code is scanned to determine whether any interrupt vectors are present between the MCU interrupt vector table address and `0xFFFF` (the last existing physical address of the M68HC(S)08 MCU).
- If interrupt vectors are present, relocation of these vectors is done as described in Interrupt Vector Table Relocation. 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 performed before the code is actually programmed into the MCU is to determine whether the code from the S19 file is in the correct 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 in `int check_image(void)` also defined in the `prog.c` file.

**UART Manipulations**

In `seriallinux.c` or `serialw32.c`, depending on the platform used, the following UART manipulation functions are defined:
PC Bootloader Master Software

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.

The pair 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.

Two additional functions are required for the bootloader to work: int send_break10(void) and int flush_uart(int out, int in). The first sends a BREAK character to the UART, the second 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 generic declarations needed to compile the application. The file main.c contains the main program and is shown at the beginning of this section (see Figure 29).

M68HC(S)08 Specific Programming Files

The most important part of the PC Bootloader software is contained in the file prog.c implements most of the intelligence of the PC bootloader software as mentioned in previous sections.

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)

Immediately after all initializations are done in the PC, a loop is started to wait for communication from the MCU. The `int hook_reset(void)` routine implements all necessary steps to establish initial communication with the MCU.

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 described in Slave Frequency Calibration. If no ACK is received from the MCU, another break character is sent.

MCU Info Reading

Immediately after the calibration is successfully completed, the PC requests the Ident Command, to which the MCU responds with 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 in 8-Bit MCU 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 straight-forward and follows the rules described 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 and split the data into blocks of appropriate size (minimum erase/write block sizes). Then the erase block and write block routines are called, in that order.

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

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

**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 8-bit MCU (using MON08 serial programmer or other, for HCS08 using BDM interface). Then the MCU is soldered (or socketed) in the application.

Using the bootloader that was pre-programmed into the MCU, the user can download the 8-bit MCU user application code via SCI interface using the bootloader utility.

**Bootloading Operation**

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

Assuming the serial board is connected to—for example—second serial port (COM2, /dev/ttyS1) and is not yet powered on, invoke the bootloader using following sequence: `hc08sprg.exe 2:D test.s19`
Figure 30. Bootloader Invocation
Bootloading Procedure Demonstration

The bootloader now expects the ACK command to be received from the MCU bootloader-enabled application. Then turn the power on for serial board and if all connections are OK, the MCU begins communication with the PC. The calibration procedure does not occur (the bootloader version with known communication speed is used), followed by IDENT command. The information that is acquired from the MCU is then displayed on the screen as shown in Figure 31.

![Figure 31. First Stage of Bootloading](image)

Developer’s Serial Bootloader for M68HC08 and HCS08 MCUs, Rev. 7
Confirm by pressing ‘y’ and the bootloading (FLASH reprogramming) will continue. The user application will be then started.

![Figure 32. Bootloading Completed](image)

**Developer’s Serial Bootloader for M68HC08 and HCS08 MCUs, Rev. 7**
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 some memory locations would likely be programmed incorrectly (the user code is either out of available FLASH memory or it overlaps with the bootloader code).

Figure 33. Memory Boundary Overlap Example
References

For additional information, refer to these documents from the Freescale Semiconductor website, http://www.freescale.com

1. **AN2295SW**: Contains all of the software files for this application note in a zip file.
2. HCS08RMv1/D: *HCS08 Family Reference Manual Volume 1*
3. AN1831/D: *Using MC68HC908 On-Chip FLASH Programming Routines*
4. AN2140/D: *Serial Monitor for MC9S08GB/GT*
5. AN2498/D: *Initial trimming of the MC68HC908 ICG*
6. AN2504/D: *On-Chip FLASH Programming API for CodeWarrior Software*
7. AN2508/D: *Generating Clocks for HC908 MCU Families*
8. AN2545/D: *On-Chip FLASH Programming Routines for MC68HC908GR/GZ*
9. AN2637/D: *Software SCI MC68HC908QT/QY MCU*

**NOTE**

With the exception of mask set errata documents, if any other Freescale Semiconductor document contains information that conflicts with the information in the device data sheet, the data sheet should be considered to have the most current and correct data.