Freescale Semiconductor
Application Note

Document Number:AN4440
Rev. 0, 04/10/2013

HCS08 Bootloader to Update Multiple Devices in a Field Bus Network

1 Introduction

This application note describes how to do in system reprogramming of Freescale Semiconductor's HCS08 devices using standard communication media such as SCI. Most of the codes are written in C so that make it easy to migrate to other MCUs. The subroutines for FLASH erasing and programming are written in ASM, but the calling APIs are provided in C. To make it easy of use, a GUI is also provided.

All the source codes are provided so customer can make their own bootloader applications based on them. The application can be used to upgrade single target board and multi boards connected through networks such as RS485. The bootloader application checks the availability of the nodes between the input address range, and upgrades firmware one by one of the nodes automatically.

The bootloader and user code are combined into one project so one single S19 file can be used for mass production and upgrading. When upgrading, the application software will pick out only the user code from the S19 record. For verification, the whole FLASH memory space will be compared except the nonvolatile registers. As source code is provided, customer can also revise the code to make a customized bootloader, they can define their own programming and verification memory range as needed.

Contents

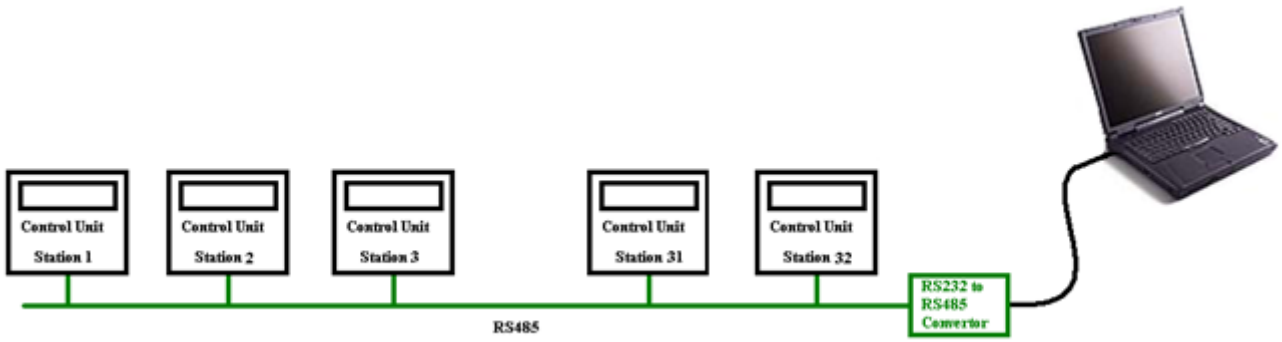

Figure 1. Typical use case

Key features of the bootloader:

- Able to update (or just verify) multiple devices in a network such as Modbus.
- Combined application code and bootloader code in a single S19 file, convenient for mass production and firmware upgrading.
- Source code available, easy for migrating.

2 Protocol Compatible with Modbus

2.1 Modbus frame

The protocol used in the bootboader application is compatible with Modbus. Bootloader Frames are used for communication between the PC and target control units. And the Bootloader Frames are encapsulated in the Modbus Frames. The frames are transmitted in half-duplex mode. If we want to use the bootloader in other networks, only the network layer needs to be changed.

Below is the format of a typical Modbus frame in RTU mode. The frame starts with a space no shorter than 3.5 characters time (showed as T1-T2-T3-T4 in the frame), followed by one byte Address, one byte Function Code, n bytes Data, two bytes CRC, and ends with a space no shorter than 3.5 characters time. The frame should be sent continuously without interrupt. A space longer than 1.5 characters will cause the next byte to be interpreted as an Address byte of the next frame. A node only processes the frames started with its node address.

Frame Start	Address	Function Code	Data	CRC	Frame End
T1-T2-T3-T4	1 Byte	1 Byte	n Bytes	2 Bytes	T1-T2-T3-T4

Figure 2. Modbus frame

Address: the address could be from 1 to 247. The Master (GUI running on PC) put a target node's address in the Address field as to communicate with it. The node also put its own node address in the Address field of the response frame, so the Master knows which node is responding. Address 0 is for broadcasting.

Function Code: could be 1 to 255. The Function Code tells the node what to do. A responding frame with the same Function Code indicates a successful command execution. The same Function Code with MSB set to 1 indicates a failed execution. For example, a Function Code 0x02 means Read Discrete Inputs. The successful response Function Code is 0x02. And if there is any error, the response Function Code will be 0x82. For Bootloader Frames, Function Codes is 43(0x2B in hex).

Data: these are data bytes sent to target nodes; values could be 0x00 to 0xFF. For the bootloader implementation, the Bootloader Frames are encapsulated in the Data field.

CRC: the two CRC bytes are used for error checking. In order to save FLASH memory, the CRC is not implemented in the bootloader. The field is always filled with 0xAA55.

2.2 Bootloader frame

A Bootloader Frame is encapsulated in the Modbus Frame as shown in below figure. The Function Code field is set to 43(0x2B in hex).

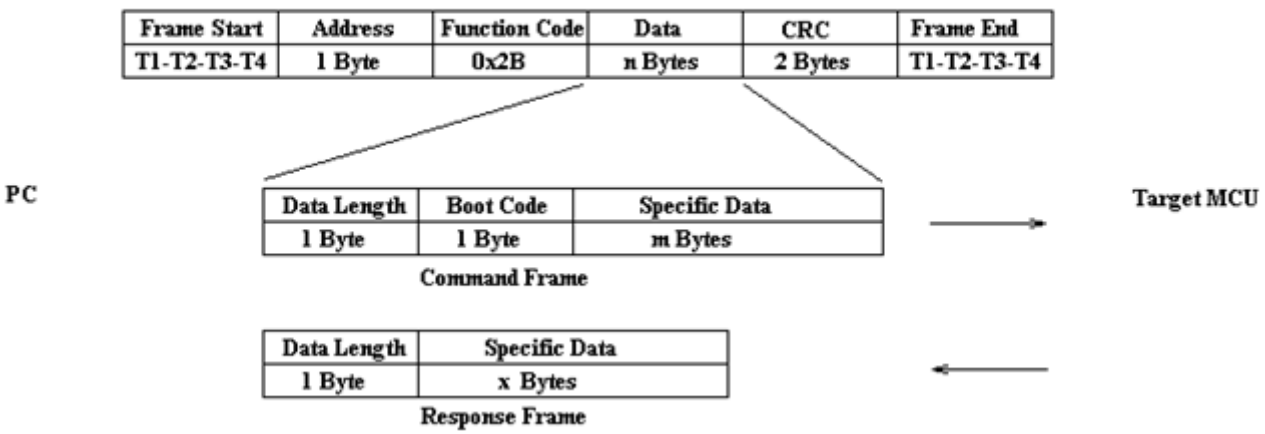

Figure 3. Bootloader frame

Bootloader Command Frame definitions:

Data Length	Boot Code	Specific Data	Function
1	B	n/a	Force target to enter Bootloader mode
1	V	n/a	Force target to enter Verification mode
1	I	n/a	Identification of target MCU
1	G	n/a	Force target to run
3	E	Addr1 Addr0	Erase specific target FLASH block
4	E	Addr2 Addr1 Addr0	Erase specific target FLASH block
4+LEN	W	Addr1 Addr0 LEN Data	Program specific target FLASH block
5+LEN	W	Addr2 Addr1 Addr0 LEN Data	Program specific target FLASH block
4	R	Addr1 Addr0 LEN	Read specific target FLASH block
5	R	Addr2 Addr1 Addr0 LEN	Read specific target FLASH block

Some of the Boot Codes are derived from FC protocol commands in AN2295, and some are added. More detailed descriptions of the Boot Codes:

‘B’ (0x42 in hex): the command forces the target to enter Bootloader mode. It does not wait for a response. A command ‘I’ is followed to check whether the target MCU is in Bootloader mode. The last unprotected FLASH page contains the remapped reset vector will be erased. So if there is a power down event during the bootload procedure, the MCU will be forced into Bootloader mode in the next power up.

‘V’ (0x56 in hex): the command forces the target to enter Verification mode. It does not wait for a response. A command ‘I’ is followed to check whether the target MCU is in Verification mode. In Verification mode, it will only compare the contents in the target MCU with specific S19 file. Nothing will be changed in the target MCU.

‘I’ (0x49 in hex): after receiving the ‘I’ command the MCU will respond a string ended with 0x00.

An example of Response Frame to command ‘I’:

Data Length	Specific Data
0x0B	0x4D 0x43 0x39 0x53 0x30 0x38 0x41 0x43 0x33 0x32 0x00

The identification string is “MC9S08AC32”.

‘G’ (0x47 in hex): after receiving the ‘G’ command the MCU will exit bootloader and force the user code to run. It does not respond a frame. The user can use application software to make sure that the target MCU is running the user code.

‘E’ (0x45 in hex): the command is used to erase a block of the FLASH in the target MCU. Based on the memory mode of the used MCU, the address may be 16 bits or 24 bits. Addr2 contains the highest bits and Addr0 contains the lowest bits. Please note the whole page including the specific address will be erased. After executing the command, it responds a zero length frame.

‘W’ (0x57 in hex): the command is used to program a block of the FLASH in the target MCU. Based on the memory mode of the used MCU, the address width may be 16 bits or 24 bits. Addr2 contains the highest bits and Addr0 contains the lowest bits. LEN is the data length to be programmed and follows the data to be programmed. After executing the command successfully, the MCU responds a zero length frame.

‘R’ (0x52 in hex): the command is used to read a block of the FLASH from the target MCU. Based on the memory mode of the used MCU, the address width may be 16 bits or 24 bits. Addr2 contains the highest bits and Addr0 contains the lowest bits. LEN is the data length to be read. After executing the command successfully, the MCU responds a Bootloader Response Frame begins with LEN (the length read data) and follows the read data.

3 Memory Relocation and Code Implementation

3.1 Memory and Vectors Relocation

On the left side in the figure below is a default memory map of an MCU. The example is based on MC9S08AC32. On the right side is the relocated memory and remapped vectors. In the example we can see FLASH range from 0xFC00 to 0xFFFF is protected. In this protected area, 0xFFC6 to 0xFFFF are the original vectors, the rest area is for the bootloader code.

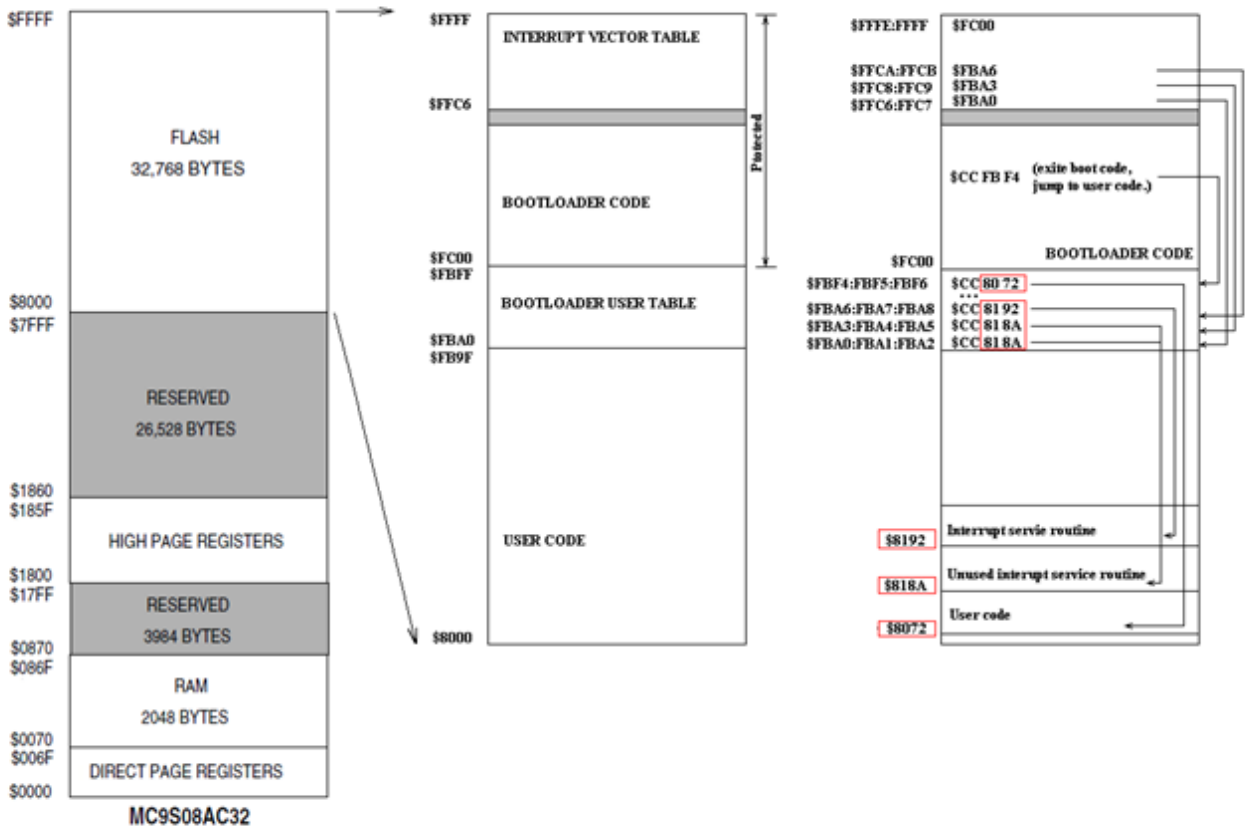

Figure 4. Memory Allocation and Vector Relocation

The unprotected FLASH from 0x8000 to 0xFBFF is for user code and can be updated in system. The Bootloader User Table ($FBA0 to $FBFF) is used for vectors remapping. All vectors except vector 0 are mapped into this jump table; three bytes are used for each vector. For example vector at $FFCA is $FBA6, at address $FBA6 it is $CC8192 (JMP $8192), $8192 is the address of interrupt service routine.

The reset vector is $FC00. It is the entry of bootloader code. When exit the bootloader code, it jumps to $FBF4. At address $FBF4 it is $CC8072 (JMP $8072), from here it jumps to the user code. Please note the addresses (in red colored squares in the figure) of the user code and interrupt service routines may be changed when the user code changed, or compiled with different options.

When bootloader code runs, it uses the whole RAM space by setting the stack pointer to the RAM end. All variables (except constant ones) in the bootloader code are local variables which are allocated on the stack. The stack will be reinitialized by the startup code when user code runs. Then all RAM memory space will be released to the user code.

3.2 Code implementation

According to the memory relocation, the prm file should be revised. As shown in below figure, please note the changes (in red colored squares):

```
NAMES END

SEGMENTS
    Z_RAM                   =  READ_WRITE    0x0070 TO 0x00FF;
    RAM                     =  READ_WRITE    0x0100 TO 0x086F;
    ROM                     =  READ_ONLY     0x8000 TO 0xFB9F;
    VECTROM                 =  READ_ONLY     0xFBA0 TO 0xFBFF;
    BOOTROM                 =  READ_ONLY     0xFC00 TO 0xFFAF;
    ROM1                    =  READ_ONLY     0xFFC0 TO 0xFFC5;
 /* INTVECTS                =  READ_ONLY     0xFFC6 TO 0xFFFF;
END

PLACEMENT
    DEFAULT_RAM,
                                             INTO   RAM;

    _PRESTART,
    STARTUP,
    ROM_VAR,
    STRINGS,
    VIRTUAL_TABLE_SEGMENT,
    DEFAULT_ROM,
    COPY
                                             INTO   ROM;
    VECTOR_ROM                               INTO   VECTROM;
    BOOT_CODE,
    EE_CODE,
    BOOT_CONST                               INTO   BOOTROM;
    _DATA_ZEROPAGE,
    MY_ZEROPAGE                              INTO   Z_RAM;
END

STACKSIZE 0x80

VECTOR 0 _Boot
```

Figure 5. Revised prm file

The segment VECTROM is for the Bootloader User Table, BOOTROM is for the bootloader code, and ROM is for user code. Please note the Vector 0 is changed to _Boot instead the original _Startup.

In the code we can use #pragma to allocate code into a specific section. For example:

```
#pragma CONST_SEG BOOT_CONST
const byte const_node_addr = 0x01;                    // const_node_addr will be allocated in
section BOOT_CONST
#pragma CONST_SEG DEFAULT
```

In the example, the variable const_node_address is placed in the protected area and can't be changed anymore except reprogramming using BDM programming tools. If we want make it revisable in system, it should be placed in an unprotected reserved FLASH memory.

3.3 Steps to incorporate the bootloader

In below figure we can see an example project in which the bootloader code has been incorporated.

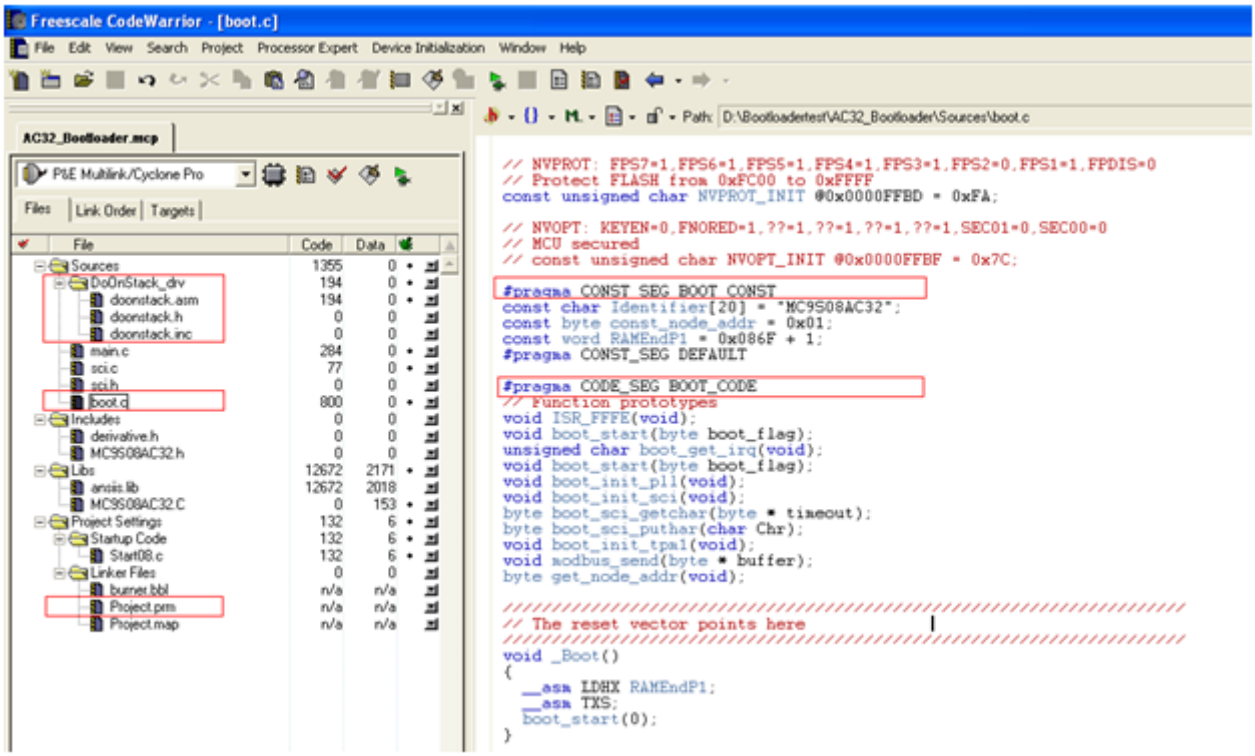

Figure 6. Example Project

In folder Sources->DoOnStack_drv is the driver for FLASH erasing and programming. The code is allocated in EE_CODE section. The bootloader code called the driver for FLASH operation. User code can also call the driver, remember to configure register FCDIV before calling. The APIs are provided in doonstack.h. More detailed information can be found in application note AN3942 (Flash Programming Routines for the HCS08 and the ColdFire (V1) Devices).

The file boot.c is the bootloader code. The code is allocated in section BOOT_CODE. Vectors are remapped to jump table in the section VECTOR_ROM. The constant variables are allocated in the section BOOT_CONST.

The parameter file Project.prm should be revised. Refer to Figure 5. Revised prm file.

The file main.c, sci.c and sci.h are user codes. Because an illegal opcode is issued to cause a reset after firmware been upgraded, all resources (including the TPM and SCI module used by the bootloader code) will be released to the user code. The Modbus protocol is implemented in the example user code as to receive commands ‘B’ and ‘V’ to force the application to enter Bootloader mode or Verification mode. But this is not necessary if we use an external pin like IRQ to force a Bootloader entry.

All unused interrupts are mapped to unused interrupt service routine:

```
void interrupt Unsued_service(void)
{
  asm nop;
}
```

If we want to use an interrupt, we should enable it in the user code, define the interrupt service routine in the user code and replace the default service routine (Unused_service( )) in boot.c.

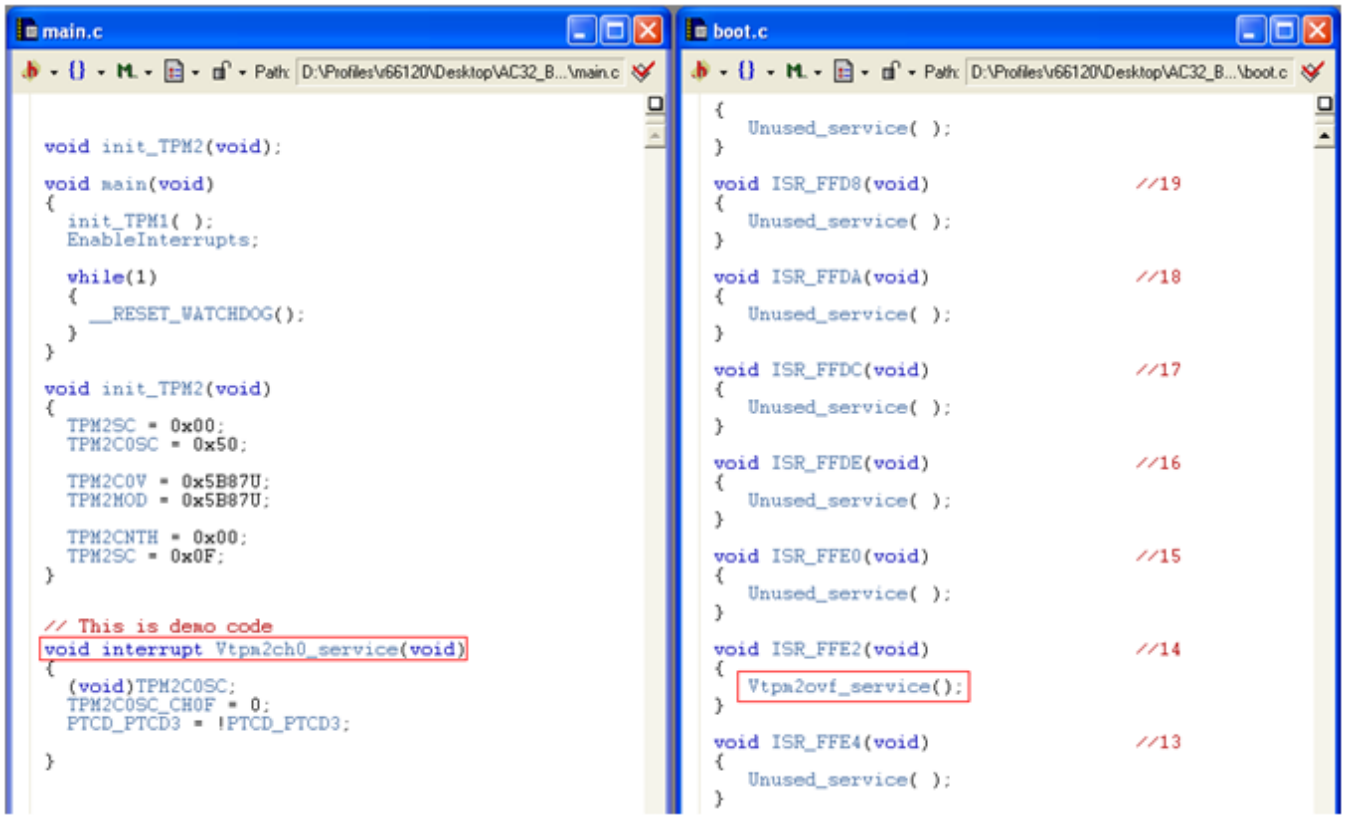

Figure 7. Vector Remap

Step 1: Copy the folder DoOnStack_drv, files main.c, boot.c, sci.c and sci.h to [project dir]\Sources and add them to the current project. Just drag the folder and files into the project panel. The original main.c will be replaced.

Step 2: Revise the prm file according to the MCU been used. Refer to Figure 5. Revised prm file.

Step 3: In sci.h and boot.c define the SCI port used for bootloader.

```
// Only one of the SCI port can be defined to 1
#define SCI1  1                                 // using SCI1
#define SCI2  0                                 // using SCI2
```

Step 4: In boot.c change the const variables according to the used MCU.

```
#pragma CONST_SEG BOOT_CONST
const char Identifier[20] = "MC9S08AC32";       // String ended with 0
const byte const_node_addr = 0x01;
const word RAMEndP1 = 0x086F + 1;               // The end address of RAM plus 1
#pragma CONST_SEG DEFAULT
```

Step 5: If any interrupt is used, remap the interrupt service routine in boot.c. Refer to Figure 7. Vector Remap.

4 Bootloader GUI

4.1 GUI codes

The graphics user interface on PC side is written using Visual C# 2008 Express Edition. It is a free edition which can be downloaded from Microsoft's website.

The main files are:

Form1.cs is the human interface code.

S19.cs provides the functions for s19 record decoding.

prog.cs provides functions for programming and verification.

GlobalVars.cs in the file there are the variables shared in all the modules and routines to configure them.

In prog.cs, function enter_boot(int mode); is used to force the target MCU into Bootloader mode or Verification mode. It issues a 'B' or 'V' command followed an 'I' command. The responded string will be assigned to MyVar.ident.targ_name. Then My.Var.Config() is called to initialize the parameters such as the erase block and program block size of the target MCU. In the current version MC9S08AC16 and MC9S08AC32 are implemented and tested. For other MCUs, we can add them in the function Config() in class MyVar(in file GlobalVars.cs):

```
public static void Config()
{
    switch (ident.targ_name)
    {
        case "MC9S08AC16":
        case "MC9S08AC32":
            {
                ident.bl_version = BL_HCS08;
                ident.bl_rcs = true;
                ident.erblk = 512;
                ident.wrblk = 64;
                ident.addr_limit = 0xFC00;
                ident.verify_addr_limit = 0x10000;
                ident.dontcare_addrl = 0xFFB0;
                ident.dontcare_addrh = 0xFFBF;

                break;
            }
        default:
            break;

    }
}
```

Figure 8. Parameters Config

Ident.bl_version:

Bootloader version. For S19 files with only 16 bit address, set it to BL_HCS08; For S19 files with 24 bit address, set it to BL_HCS08_LARGE (FLASH size larger than 64k) or BL_HCS08_LONG (chips with EEPROM).

Ident.bl_rcs:

Read support. Please set to true.

ident.erblk:

Erase block size. For HCS08 it is the page size of the FLASH memory.

ident.wrblk:

Write block size. For HCS08 it is the row size of the FLASH memory.

ident.addr_limit:

It is the start address of the bootloader.

ident.verify_addr_limit:

When making verification between selected S19 file and the target MCU, it defines the address upper limit. By default it is the FLASH end address + 1. If we do not want to verify all the FLASH memory, we can define it to a smaller value.

ident.dontcare_addrl:

ident.dontcare_addrh:

The contents between address ident.dontcare_addrl and ident.dontcare_addrh will not be compared when making verification. The purpose is to define an area which may be different from chip to chip, for example the nonvolatile register stores trim value.

4.1.1 Run the demo

The GUI looks like this:

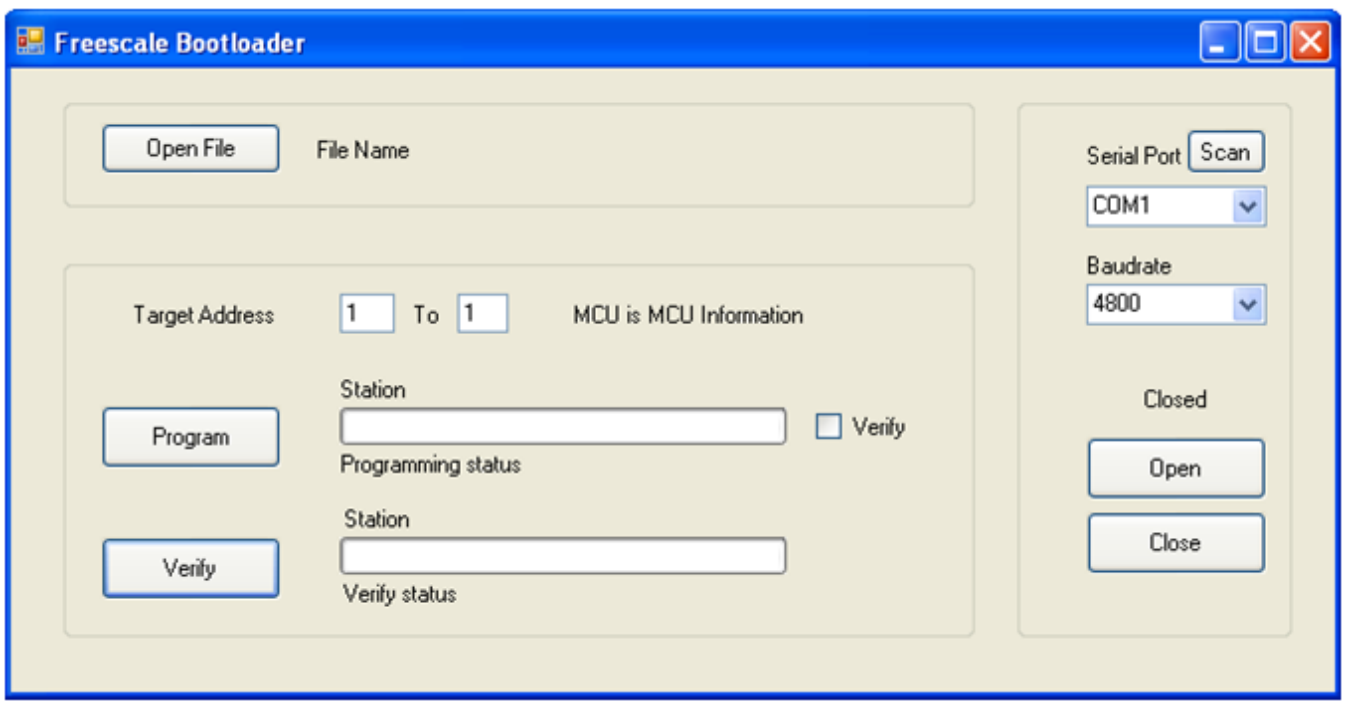

Figure 9. Graphics User Interface

When click button Program, the opened file will be programmed to the target nodes one by one as defined in the Target Address. In the implementation, Target Address is limited to 1 to 32. If we check the check box Verify, after programming one block of FLASH, the contents will be read back and compared with the S19 file. While in programming process, we can terminate the process anytime.

The button Verify is used to compare a selected S19 file with target nodes one by one. It will not change the contents in target MCUs.

Please note that both Program and Verify commands will stop the target node from running. A ‘G’ command will be issued automatically after the process finished successfully which force the target node to run.

Remember to open the S19 file and open the COM port first.

A running GUI:

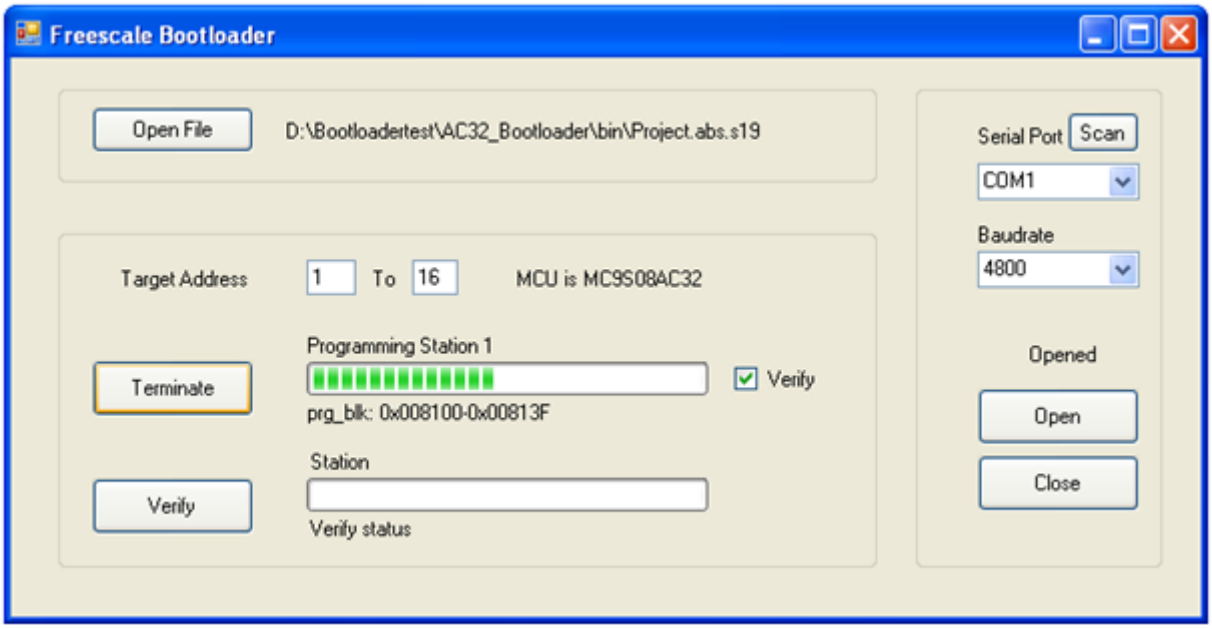

Figure 10. Running GUI

A log file named boot log.txt will be created in the same folder where the opened s19 file is stored. The operating history will be recorded with time stamps.

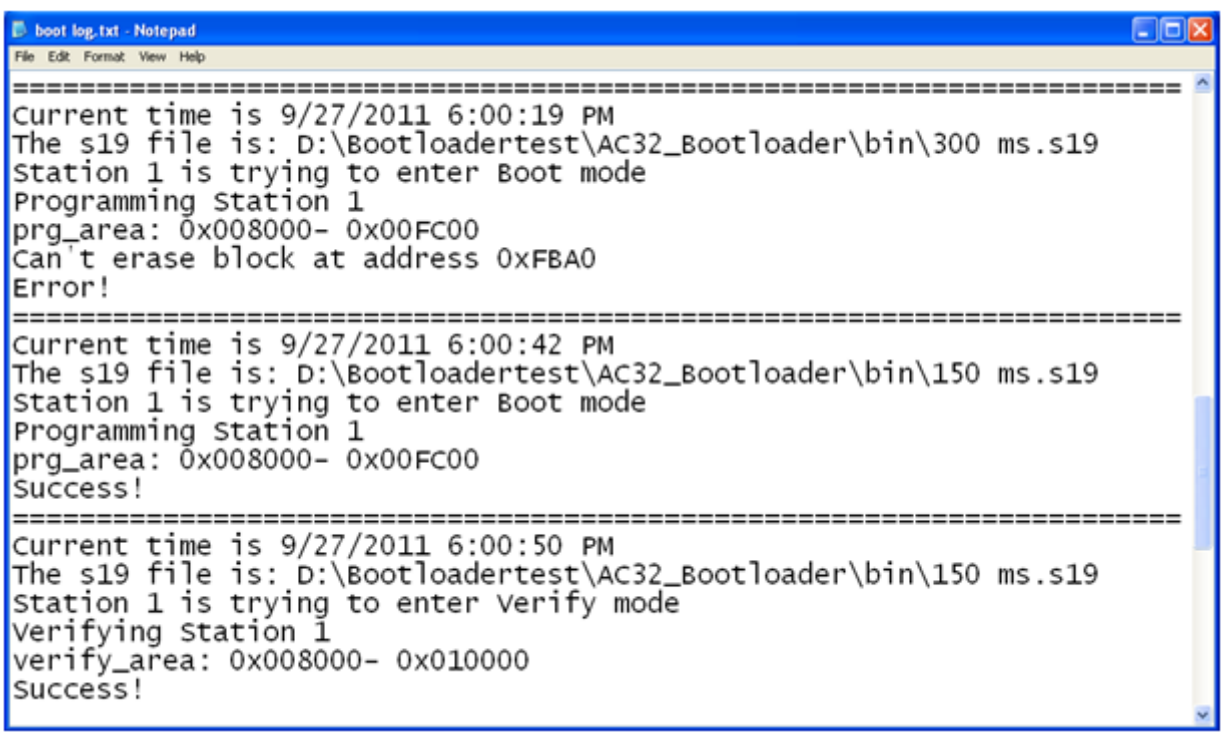

Figure 11. Boot Log File

5 Conclusion

This application note makes a detailed description on how to implement a bootloader in C. The bootloader code and application code are combined in one project. So a single S19 record file can be used for both mass production and in system firmware upgrading. The application can be used for single control unit or control units connected by networks. The firmware and GUI are tested on customized boards using MC9S08AC32. And the codes can be migrated to other chips easily.

6 References

- Application note titled *Developer's Serial Bootloader for M68HC08 and HCS08 MCUs* (document number AN2295).
- Application note titled *Flash Programming Routines for the HCS08 and the ColdFire (V1) Devices* (document number AN3942).
- Datasheet titled *MC9S08AC60 MC9S08AC48 MC9S08AC32 Data Sheet HCS08 Microcontrollers* (document number MC9S08AC60). http://www.freescale.com
- http://msdn.microsoft.com/en-us/vcsharp
- Modbus Application Protocol. http://www.modbus.org

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4440
Rev. 0, 04/10/2013