Application Note

AN2504/D
10/2003

On-Chip FLASH Programming API for CodeWarrior Software

By Mauricio Capistrán-Garza
Application Engineer
Freescale
Guadalajara, Mexico

troduction

This application note presents an easy-to-use C-language API for using FLASH[1]-programming routines that are stored in ROM in the MC68HC908GR8, MC68HC908KX8, MC68HC908JL3, MC68HC908JK3 and MC68HC908JB8 microcontrollers (MCUs)[2]. These ROM-resident routines can be used to program, erase, and verify FLASH memory as well as to communicate serially.

The CodeWarrior API for ROM-resident routines was written in C language. By using this API, the programmer takes advantage of these benefits:

- There is no need to know the absolute address of routines
- Changing from one MCU to another is easy; minimum code changes are needed
- Code is easy to understand and follow
- No in-depth knowledge of the routines is needed
- Enhancement of ROM-resident routines
- Shorter development time

When using the API, take into consideration that your code may be larger and the available RAM might decrease. Techniques can be used to avoid decreasing the available RAM. For more information, refer to the **Techniques** section.

In addition to describing how to call the API functions (parameters needed and return values), this document includes example software with typical API calls to better illustrate the procedure.

1. This product incorporates SuperFlash® technology licensed from SST.
2. These routines are accessible in both user mode and monitor mode in all listed devices except the MC68HC908GR8. This device allows access to these routines in monitor mode only.

FLASH Overview

The ROM-resident routines that the API manages are found on devices that do not have enough RAM to allow for this functionality in a RAM routine. The type of FLASH for which these routines are applicable is called split-gate FLASH because of the type of technology used.

Split-gate FLASH has significant advantages:

- Faster programming time — It takes 30 to 40 µs to program each byte, which translates to a little more than a quarter second of programming time to program an entire 8-Kbyte array.
- Better endurance — This type of FLASH is specified to withstand at least 10,000 program/erase cycles. Older technologies provided only about 100 program/erase cycles.
- Simpler programming algorithm — The programming algorithm for split-gate FLASH is a simple process of turning on high voltage, applying it to the row to be programmed, then writing values to each byte to be programmed. This differs from past technology which required an iterative process of turning on high voltage and applying it to a page, writing values to each byte in the page, checking all bytes for valid values in a “margin” read condition, and then repeating the program/verify process until all bytes are verified correctly.

  Split-gate FLASH generally is programmed on a row basis and erased on a page basis. Also, the entire array can be mass erased. A page always contains two rows, but the size of the page can vary from one device to another. A typical page size is 64 or 128 bytes. Before reprogramming a byte in one row that is currently programmed with a different value, the entire page must be erased and reprogrammed. Refer to the applicable data sheet for the proper program and erase procedure for this FLASH.

ROM-Resident Routines

The API manages four ROM-resident routines:

- GETBYTE — used to serially receive a byte
- RDVRRNG — used to read and verify a range of FLASH
- PRGRNGE — used to program a range of FLASH
- ERARNGE — used to erase a range of FLASH

The following ROM-resident routine is not managed by this API. For information about this routine, refer to *Using MC68HC908 On-Chip FLASH Programming Routines,* Freescale document order number AN1831/D.

DELNUS — a delay of *n* µs

ROM-resident routines have different numbers of parameters. Most of them use a section of RAM to receive parameters or output results. Consequently, usage of RAM must be done carefully to avoid interfering with these routines; some sections of RAM can not be used to allocate variables.

ROM-resident routines are located in different sections of FLASH depending on the MCU. **Table 1** shows the absolute address of these routines for each microcontroller discussed in this application note.

Table 1. Address of Routines

Routine	MC68HC908GR8	MC68HC908KX8	MC68HC908JL3/JK3	MC68HC908JB8
GETBYTE	0x1C00	0x1000	0xFC00	0xFC00
RDVRRNG	0x1C03	0x1003	0xFC03	0xFC03
PRGRNGE	0x1C09	0x1009	0xFC09	0xFC09
ERARNGE	0x1C06	0x1006	0xFC06	0xFC06
DELNUS	0x1C0C	0x100C	0xFC0C	0xFC0C

These routines use ROM-resident subroutines, which will not be covered in this document. For a complete understanding of these ROM-resident routines, refer to AN1831/D.

Defined Constants

Table 2 lists all constants used within the FLASH-programming API. Since the FLASH-programming API serves a variety of microcontrollers, the constants are device specific.

Table 2. Constants

Constant Name	Description	908GR8	908KX8	908JL3/JK3	908JB8
RAM	Start address of RAM	0x40	0x40	0x80	0x40
COMMPORT	Communication port	PTA0	PTA0	PTB0	PTA0
FLBPR	FLASH block protect register address	0xFF7E	0xFF7E	0xFE09	0xFE09
FLCR	FLASH control register address	0xFE08	0xFE08	0xFE08	0xFE08
Get_Bit	Address of routine to get a bit on the communication port	0xFED2	0xFECE	0xFF00	0xFF00

Virtual Registers

As mentioned before, ROM-resident routines use sections of RAM to receive parameters and output results. Extra care is needed when handling these sections. For this reason, *virtual registers* have been created. The FLASH-programming API uses this location in RAM for registers to configure ROM-resident routines. There are different ways of assuring these virtual registers work properly. Two ways of doing this are described here:

- In the [MY_PROJECT].prm file included in the project, change RAM start to address RAM + 76 bytes. (For example, in HC908JL3 RAM starts at address 0x80. Change it to 0x80 + 76 = 0xCC.) Doing this assures that the virtual registers will be untouched by RAM variables, though they are still susceptible to modification by the stack. Another problem is that the RAM size is reduced by 76 bytes.
- Place all variables manually. When declaring variables you can specify the address at which you want to allocate them. Do not overlap any variable with the virtual registers. This reduces the available RAM by 68 bytes.

Both of these methods reduce the size of available RAM. There are other ways of ensuring the integrity of the virtual registers without compromising RAM size (refer to the **Techniques** section).

The virtual registers and their descriptions are listed in **Table 3**.

Table 3. Virtual Registers

Virtual Register Name	Function	Address
CTRLBYT	Used in erasing procedures for indicating a single PAGE erase or MASS erase	RAM + 0x08
CPUSPD	CPU speed passed as fop x 4	RAM + 0x09
LADDRH	Last address of a range (high byte)	RAM + 0x0A
LADDRL	Last address of a range (low byte)	RAM + 0x0B
DATA	Buffer that stores the data to be programmed or receives the data read	RAM + 0x0C

Virtual register DATA is of variable length (from 1 to 64 bytes long), depending on the range over which the function will actuate.

Coding Conventions

This application note follows the following coding conventions.

Table 4. Coding Conventions

Structure	Convention	Example
Macros	All macros are written in all UPPER CASE	#Define DATA_END 0xCC
Functions	The first letter of each word of the function's name is capitalized	Byte ReadByte (void)
Local variables	All local variables are in lower case letters preceded by an underscore	Byte _cancel_buttons.
Assembly labels	All assembly labels are written in all capital letters preceded by an underscore	_RECEIVE_BYTE:

Hardware Needed for Communication

The API functions that provide serial communication need external hardware to couple with RS-232 standards. The hardware is very simple, and it is the same as described in the monitor ROM section of the data sheet.

Figure 1 shows a schematic of the hardware for a MC68HC908JL3 where the communication port is PTB0.

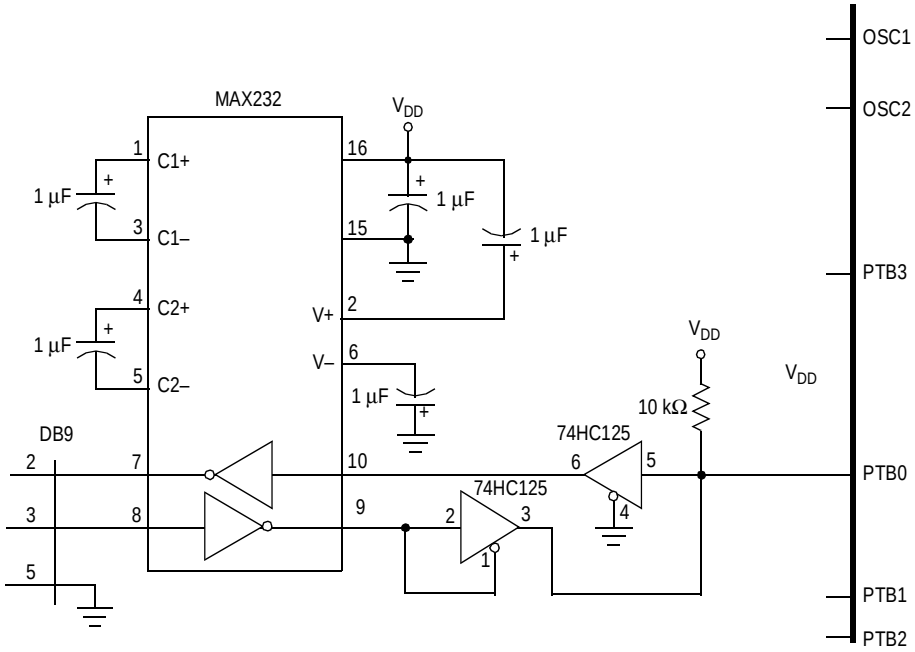

Figure 1. Communication Hardware

FLASH-Programming API

Before using API functions the user must provide the following information:

1. MCU used — The MCU used must be defined before including the API. The definition must be one of the following:

   #define MCU68HC908JL3

   #define MCU68HC908KX8

   #define MCU68HC908GR8

   #define MCU68HC908JB

If no MCU is defined, the API will use the MCU68HC908JL3 as the default.

2. Frequency of operation — The frequency of operation must be defined. This is very important because the API functions use it. If this frequency is defined incorrectly, the API functions won't work. The frequency of operation must be defined as the rounded product of four times the actual internal operating frequency. So, if the internal operating frequency is 1.2288 MHz, the value defined should be 5 ($4 \times 1.2288 = 4.9152 \approx 5$). The frequency of operation must be defined as follows:

   #define OSC 5

If no OSC is defined, the API will assign a frequency of operation equal to four as the default (OSC = 4).

ReadByte

The ReadByte function receives a byte serially through the communication port and returns it to the function caller. This routine expects the same non-return-to-zero (NRZ) communication protocol and baud rate that is used in monitor mode[1]. The monitor uses a similar routine but unlike the monitor routine, ReadByte only receives a character while the monitor routine also re-transmits each character after it is received.

Table 5. ReadByte

Prototype	Byte ReadByte (Void)
Parameters	None
Entry Conditions	None
Exit Conditions	None
Return Value	Byte read
Remarks	The program will enter an infinite loop until a byte is received. Communications at the MCU port pin use CMOS logic levels while a PC serial port requires RS-232 voltage levels. Therefore, an RS-232 level shifter device such as the MAX232 is required to interface the MCU to a PC serial port. Refer to **Hardware Needed for Communication**.

The communication port is different from one microcontroller to another. However, function ReadByte automatically uses the communication port for the microcontroller used. **Table 2** provides the communication port for each microcontroller.

1. The baud rate will be $f_{OP}/256$ for all but the MC68HC908JB8. In this device, the bit rate for this routine as well as for the monitor mode send/receive routines have been changed to accommodate a "standard" f_{OP} for this device considering it is a USB part. The bit rate for the MC68HC908JB8 is $f_{OP}/313$.

TransmitByte

The TransmitByte function sends a byte through the communication port. This function uses the same NRZ communication protocol and baud rate that is used in monitor mode[1].

Table 6. TransmitByte

Prototype	Void TransmitByte (Byte _data)
Parameters	_data: the byte to be transmitted
Entry Conditions	None
Exit Conditions	None
Return Value	None
Remarks	TransmitByte will try to transmit the _data with no handshake. If the receiver is not ready the transmitted _data will be lost. Communications at the MCU port pin use CMOS logic levels while a PC serial port requires RS-232 voltage levels. Therefore, an RS-232 level shifter device such as the MAX232 is required to interface the MCU to a PC serial port. Refer to **Hardware Needed for Communication**.

The communication port is different from one microcontroller to another. However, the function TransmitByte automatically chooses the communication port for the microcontroller used. **Table 2** provides the communication port for each microcontroller.

1. Techniques can be used to avoid decreasing the available RAM. Refer to the **Techniques** section.

TransmitRange

TransmitRange is a variation of **VerifyRange and ReadRange**. The same procedure is followed except instead of FLASH data overwriting RAM data, FLASH data is transmitted out through the communication port. This routine transmits with the NRZ communication protocol and baud rate that is used in monitor mode[1].

The communication port is different from one microcontroller to another. However, the function TransmitRange automatically chooses the communication port of the microcontroller. **Table 2** provides the communication port for each microcontroller.

Table 7. TransmitRange

Prototype	Byte TransmitRange (Word *_ini, Byte _num)
Parameters	_ini: the absolute address of the first location in FLASH to start transmitting _num: number of bytes to transmit
Entry Conditions	None
Exit Conditions	The checksum is stored in _ini;
Return Value	SUCCESS or FAIL
Remarks	_num must be less than or equal to 64 The communication is done in TTL values; A TTL to RS-232 converter is needed in order to interact with a PC serial port. Refer to **Hardware Needed for Communication**.

1. Techniques can be used to avoid decreasing the available RAM. Refer to the **Techniques** section.

ProgramRange

ProgramRange programs a range of FLASH memory with the contents of RAM. This routine doesn't verify whether the range to be programmed is already blank. Programming new data to a location that is not blank is not legal. The resulting value in the FLASH location after such an illegal programming attempt will be incorrect or unreliable.

Table 8. ProgramRange

Prototype	Void ProgramRange (Word *_ini, Byte _num)
Parameters	_ini: the absolute address of the first location in FLASH to be programmed _num: number of bytes to program
Entry Conditions	DATA contains the data to be programmed
Exit Conditions	None
Return value	None
Remarks	_num must be less than or equal to 64

ProgramRangeX

ProgramRangeX is the enhanced version of ProgramRange. This function programs a range of FLASH memory with the contents of RAM. Before programming, it verifies that the range to be programmed is blank. If the range is not blank, the function doesn't attempt to program it (maintaining the integrity of the range) and returns an error code.

Table 9. ProgramRangeX

Prototype	Byte ProgramRangeX (Word *_ini, Byte _num)
Parameters	_ini: the absolute address of the first location in FLASH to be programmed. _num: number of bytes to program.
Entry Conditions	DATA contains the data to be programmed.
Exit Conditions	None
Return value	SUCCESS or FAIL
Remarks	_num must be less than or equal to 64

VerifyRange and ReadRange

VerifyRange and ReadRange are names for the same function that either verifies a range of FLASH memory against data in RAM or reads a range of FLASH memory. When using this function, choose the name that most accurately describes the purpose of the function.

This function compares a range of FLASH memory against a range (the same length) of RAM. When FLASH data does not match RAM data, the RAM data is replaced with FLASH data. If all bytes compared are equal then SUCCESS is returned; otherwise FAIL is returned. This function also returns the checksum of the data read. (The checksum is the LSB of the sum of all bytes in the entire data collection.)

Table 10. VerifyRange andReadRange

Prototype	Byte VerifyRange (Word *_ini, Byte _num) Byte ReadRange (Word *_ini, Byte _num)
Parameters	_ini: the absolute address of the first location in FLASH to start reading/verifying _num: number of bytes to read/verify
Entry Conditions	DATA contains the data to be verified
Exit Conditions	DATA is overwritten with contents of FLASH The checksum is stored in _ini;
Return value	SUCCESS or FAIL
Remarks	_num must be less than or equal to 64

ErasePage

ErasePage erases a page of FLASH. The length of a page depends on the MCU. In most cases a page is 32 bytes long; however, on the MCU68HC908JL3/JK3 a page is 64 bytes long.

Table 11. ErasePage

Prototype	void ErasePage (Word *_page)
Parameters	_page: the absolute address of any of the locations within the PAGE to be erased.
Entry Conditions	None
Exit Conditions	Interrupts are disabled.
Return value	None
Remarks	All bytes within that PAGE will be driven to 0xFF.

ErasePageX

ErasePageX is the enhanced version of ErasePage. It erases a PAGE of FLASH.

Table 12. ErasePageX

Prototype	void ErasePageX (Word *_page)
Parameters	_page: the absolute address of any of the locations within the PAGE to be erased.
Entry Conditions	None
Exit Conditions	None
Return value	None
Remarks	All bytes within that PAGE will be driven to 0xFF. Interrupts are disabled during the erasing of the FLASH, but are restored to their original states before exiting the function.

EraseFlash

EraseFlash erases the entire FLASH.

Table 13. EraseFlash

Prototype	void EraseFlash (void)
Parameters	None
Entry Conditions	None
Exit Conditions	None
Return value	None
Remarks	All bytes will be driven to 0xFF. No code in FLASH will be executed after this function has been called.

Techniques

ReadByte

As mentioned above, the routine ReadByte enters an infinite loop if no byte is received. There is a way to make ReadByte exit the infinite loop. The procedure is as follows:

1. Clear a flag before calling ReadByte (a flag indicating the successful reception of a byte.
2. Set up the timer to interrupt at a given time (the time to wait before exiting ReadByte).
3. Call ReadByte()
4. If a byte is received, set the flag previously cleared.
5. If a byte is not received, the Timer ISR will execute. Inside the Timer ISR verify the state of the flag. If it is cleared (no byte has been received), two actions can be taken:
   a. Change manually the PC stored in the stack (so when the ISR returns, it returns out of function ReadByte).
   b. Cause a software reset.

This technique has some disadvantages, including the time overhead needed to verify the flag on each timer ISR.

Virtual Registers

Virtual registers limit the available RAM. However these registers are only used when FLASH-programming API is called. This means that most of the time these registers are only wasting RAM space. A work-around for this problem can be achieved as follows:

1. Define global variables through all RAM space (not reserving the virtual registers bytes).
2. Define a section in FLASH memory 64 bytes long called BACKUP_RAM. In this section no code will be allocated.
3. When the need of using the FLASH-programming API arises, before doing anything do the following:
   a. Push into the stack (this must be done in assembly) the first four bytes of the virtual registers (from RAM+0x08 to RAM+0x0B).
   b. Call API function ErasePage() to erase the section BACKUP_RAM.
   c. Call API function ProgramRange() to backup the DATA virtual register (from RAM+0x0C to RAM+0x4C).
   d. Use the FLASH-programming API for whatever needed.
   e. Before ending call API function ReadFlash() to restore the data from BACKUP_RAM to RAM.
   f. Finally, pop the first four bytes of the virtual registers (from RAM+0x08 to RAM+0x0B).

Typical API Calls

The following software performs typical FLASH-programming API calls:

```
/*=================================================================*
 *
 * Copyright (c) 2002,  * Freescale Application Note
 *
 * File name    : flash_test.c
 * Author       : Mauricio Capistran-Garza
 * Department   : Guadalajara - SPS
 *
 * Description  : This is a sample program that shows how
 *                ROM-resident routines can be used through
 *                the use of a simple C-language API
 *
 * History      :
 *
 *=================================================================*/



/*******************************************************************
    INCLUDES
 *******************************************************************/
//#include "jl3.h"
#include <MC68HC908JL3.h>         // Include Peripheral declarations
#include <hidef.h>
#include <stdtypes.h>
#include "flash_api.h"

/*******************************************************************
    DEFINES
 *******************************************************************/
#define MY_INFO_ADDRESS         0xFB00
#define MY_INFO_SIZE               8
#define MY_TRANSMISION_ADDRESS  0xFB08
#define MY_TRANSMISION_SIZE        29
#define FLASH_TEST_ADDRESS      0xFB40
#define DATA_START              0x008C
#define DATA_END                0x00CC

/*******************************************************************
    TABLES
 *******************************************************************/
#pragma CONST_SEG MY_INFO
volatile const Byte TABLE1[8] = {'M','o','t','o','r','o','l','a'};

#pragma CONST_SEG MY_TRANSMISION
volatile const Byte TABLE2[29] = {
'T','r','a','n','s','m','i','t','t',' ','R','a','n','g','e',' ',
'w','a','s',' ','s','u','c','c','e','s','s','f','u','l' };

#pragma CONST_SEG DEFAULT

/*******************************************************************
    ISR
 *******************************************************************/
/* These ISR are written to show a possible work-around to
   avoid routine ReadByte from entering into an infinite loop.
 */
```

```
/*
interrupt 6 void Timer(void) {
   TASC=TASC;
    TASC &= TOF;                    // If a timer interrupt happens
    __asm swi;                      // call the Software Interrupt
}

interrupt 2 void SWI(void) {        // Software Interrupt ISR
    DisableInterrupts;              // may include user-supplied code.
    while (-1) {                    // The stack can be reset and
        TransmitByte ('X');         // the whole program started again.
    }
}
*/
/*********************************************************************
    MAIN
 *********************************************************************/
void main(void) {
    Byte temp;
    Byte size;
    Word address;

/* System configuration */

    CONFIG2 = 0x80;    /* 10000000b;
                          ||||||||_________ Reserved
                          |||||||__________ LVIT
                          ||| _____________ Reserved
                          |_________________ IRQ Internal Pull-Up     */

    CONFIG1 = 0x11;    /* 00010001b;
                          ||||||||_________ COP disabled
                          |||||||__________ STOP as illegal opcode
                          ||||||___________ Short Stop Recovery Bit
                          |||||____________ Reserved
                          ||||_____________ LVI enabled
                          |||______________ Reserved
                          ||_______________ COP reset period         */

/*  Configure the timer if wanted */

/*
    TMODH = 0x4C;      // Interrupt every 16 ms
    TMODL = 0xCD;

    TSC   = 0x40;      // 01000000b;
                       // ||||||||_________ Timer Prescaled by 1
                       // |||||            Int Bus Clk = 1.2288MHz
                       // |||||            1229 counts,
                       // |||||            error = 0.58 sec/hour
                       // ||||_____________ Unimplemented
                       // |||______________ Reset Bit = 0 -> No effect
                       // ||_______________ Stop Bit: Start Counter
                       // |________________ TIM Overflow ints enabled
                       // _________________ TIM Overflow Flag
*/
    temp = TABLE1[0];
    size = TABLE2[0];
```

```
/*
 * Receive one byte and echo it. This shows a typical usage
 * of API functions GetByte and Transmit Byte.
 * This functions can not be debugged with the In-Circuit
 * Debugger (ICD) since the Communication Port is used by the ICD.
 */
    temp = ReadByte();
    TransmitByte(temp);
    DATA(0) = temp;           // this is a way of accessing DATA,
                              // however it is very space-consuming
                              // when it gets translated into assembly

    size = 1;                 // One byte is ready to be programmed

/*
 * Erase a PAGE of flash named FLASH_TEST. Notice how Interrupts
 * will be automatically disabled.
 */
     address = FLASH_TEST_ADDRESS;
     EnableInterrupts;
     ErasePage(&address);

/*
 * Program FLASH_TEST with the received Byte.
 */
    ProgramRange(&address, size);

/*
 * Verify that the programming was successful.
 */
    temp = VerifyRange(&address, size); // Now temp will be set to
                                        // SUCCESS or FAIL, and address
                                        // will be set with the checksum
                                        // the data verified
    if (temp == SUCCESS) {
        temp = temp + 1;          // Do anything wanted
    }

/*
 * Read a section in FLASH named MY_INFO. In this section
 * it is stored the message "Freescale". Notice how
 * the DATA (RAM_START + 0x0C) will be replaced with FLASH data.
*/
    address = MY_INFO_ADDRESS;
    size = MY_INFO_SIZE;
    temp = ReadRange(&address, size);

/*
 * Transmit a range of FLASH named MY_TRANSMISION. In this
 * sections it's stored the message "TransmitRange was successful".
 * Notice how the DATA (RAM_START + 0x0C) will NOT be replaced
 * with the data transmitted.
 */
    address = MY_TRANSMISION_ADDRESS;
    size = MY_TRANSMISION_SIZE;
    temp = TransmitRange(&address, size);

/*
 * Fill out manually 60 bytes of DATA (RAM_START + 0x0C).
 */

/*  for (i=0; i<64; i++) {        // This is a way of filling out DATA
        DATA_STR(i) = 'm';        // but it is very space-consuming
    }                             // Next, another way to do it:
*/
```

```
    __asm            LDHX #DATA_START; // Where DATA begins
    __asm            LDA #122;
    __asm _FILL_RAM: STA ,X;           // save the received byte
    __asm           INCX;             // point to next location in RAM
    __asm            DECA;
    __asm           CPX #DATA_END;    // if DATA_END reached exit loop
    __asm            BNE _FILL_RAM;

/*
 * Try to program again the section FLASH_TEST with ProgramRangeX
 * It is going to fail since this section is not blank.
 */
    address = FLASH_TEST_ADDRESS;
    size = 10;
    temp = ProgramRangeX(&address, size);
    if (temp == FAIL) {
        temp = temp + 1;          // Do anything wanted
    }

/*
 * Erase section FLASH_TEST. Notice how interrupts won't be disabled.
 */
    EnableInterrupts;
    ErasePageX(&address);

/*
 * Program section FLASH_TEST with ProgramRangeX.
 * This time ProgramRangeX is going to succeed.
 */
    temp = ProgramRangeX(&address, size);
    if (temp == FAIL) {
        temp = temp + 1;          // Do anything wanted
    }

/*
 * Finally Erase the entire FLASH.
 */
    EraseFlash();
    for ( ;; );
}
```

API Source Code

```
/*====================================================================*
 *
 * Copyright (c) 2002,
 * Freescale Application Note
 *
 * File name    : MCU_constants.h
 * Author       : Mauricio Capistran-Garza
 * Department   : Guadalajara - SPS
 *
 * Description  : It contains the defines needed for all constants
 *                used in flash_api.c for the following MCUs:
 *                                                    MC68HC908GR8,
 *                                                    MC68HC908KX8,
 *                                                    MC68HC908JL3,
 *                                                    MC68HC908JK3,
 *                                                    MC68HC908JB8.
 *
 * History      :
 *
 *====================================================================*/
```

```
#ifndef __MCU_CONSTANTS_H__
#define __MCU_CONSTANTS_H__

#include <MC68HC908JL3.h>

/* API Configuration */

#define MASSBIT            0x06   // CTRLBYT MASS ERASE BIT = 6
#ifndef FLCR
    #define FLCR           0xFE08 // FLASH CONTROL REGISTER
#endif

#ifdef MC68HC908GR8
    /* Communication Port Constants */
    #define COMMPORT       PTA
    #define COMMPORT_DIR   DDRA
    #define COMMPORT_ADDR  0x00

    /* FLASH Constants */
//  #define FLASH_START    0xEC00
    #define PAGE_SIZE      32

    /* RAM Constants */
    #define RAMSTART       0x40
    #define CTRLBYT        (*(volatile unsigned char*)(0x48))
    #define CPUSPD         (*(volatile unsigned char*)(0x49))
    #define LADDRH         (*(volatile unsigned char*)(0x4A))
    #define LADDRL         (*(volatile unsigned char*)(0x4B))
    #define DATA(X)        (*(volatile unsigned char*)(0x4C + X))
    /* ROM-resident Routines Constants */
    #define GETBYTE()      {__asm jsr 0x1C00;}
    #define RDVRRNG()      {__asm jsr 0x1C03;}
    #define ERARNGE()      {__asm jsr 0x1D06;}
    #define PRGRNGE()      {__asm jsr 0x1C09;}
    #define DELNUS()       {__asm jsr 0x1D0C;}
    #define GET_BIT()      {__asm jsr 0xFED2;}
    #define PUT_BYTE()     {__asm jsr 0xFEAE;}
#endif  // MC68HC908GR8

#ifdef MC68HC908KX8
    /* Communication Port Constants */
    #define COMMPORT       PTA
    #define COMMPORT_DIR   DDRA
    #define COMMPORT_ADDR  0x00

    /* FLASH Constants */
//  #define FLASH_START    0xEC00
    #define PAGE_SIZE      32

    /* RAM Constants */
    #define RAMSTART       0x40
    #define CTRLBYT        (*(volatile unsigned char*)(0x48))
    #define CPUSPD         (*(volatile unsigned char*)(0x49))
    #define LADDRH         (*(volatile unsigned char*)(0x4A))
    #define LADDRL         (*(volatile unsigned char*)(0x4B))
    #define DATA(X)        (*(volatile unsigned char*)(0x4C + X))
```

```
    /* ROM-resident Routines Constants */
    #define GETBYTE()       {__asm jsr 0x1000;}
    #define RDVRRNG()       {__asm jsr 0x1003;}
    #define ERARNGE()       {__asm jsr 0x1006;}
    #define PRGRNGE()       {__asm jsr 0x1009;}
    #define DELNUS()        {__asm jsr 0x100C;}
    #define GET_BIT()       {__asm jsr 0xFECE;}
    #define PUT_BYTE()      {__asm jsr 0xFEAA;}
#endif  // MC68HC908KX8

#ifdef MC68HC908JL3   //  || MC68HC908JK3
    /* Communication Port Constants */
    #define COMMPORT        PTB
    #define COMMPORT_DIR    DDRB
    #define COMMPORT_ADDR   0x01

    /* FLASH Constants */
    #define FLASH_START     0xEC00
    #define PAGE_SIZE       32

    /* RAM Constants */
    #define RAMSTART        0x80
    #define CTRLBYT         (*(volatile unsigned char*)(0x88))
    #define CPUSPD          (*(volatile unsigned char*)(0x89))
    #define LADDRH          (*(volatile unsigned char*)(0x8A))
    #define LADDRL          (*(volatile unsigned char*)(0x8B))
    #define DATA(X)         (*(volatile unsigned char*)(0x8C + X))

    /* ROM-resident Routines Constants */
    #define GETBYTE()       {__asm jsr 0xFC00;}
    #define RDVRRNG()       {__asm jsr 0xFC03;}
    #define ERARNGE()       {__asm jsr 0xFC06;}
    #define PRGRNGE()       {__asm jsr 0xFC09;}
    #define DELNUS()        {__asm jsr 0xFC0C;}
    #define GET_BIT()       {__asm jsr 0xFF00;}
    #define PUT_BYTE()      {__asm jsr 0xFED0;}
#endif  // MC68HC908JL3 || MC68HC908JK3

#ifdef MC68HC908JB8
    /* Communication Port Constants */
    #define COMMPORT        PTA
    #define COMMPORT_DIR    DDRA
    #define COMMPORT_ADDR   0x00

    /* FLASH Constants */
//  #define FLASH_START     0xEC00
    #define PAGE_SIZE       64

    /* RAM Constants */
    #define RAMSTART        0x40
    #define CTRLBYT         (*(volatile unsigned char*)(0x48))
    #define CPUSPD          (*(volatile unsigned char*)(0x49))
    #define LADDRH          (*(volatile unsigned char*)(0x4A))
    #define LADDRL          (*(volatile unsigned char*)(0x4B))
    #define DATA(X)         (*(volatile unsigned char*)(0x4C + X))

    /* ROM-resident Routines Constants */
    #define GETBYTE()       {__asm jsr 0xFC00;}
    #define RDVRRNG()       {__asm jsr 0xFC03;}
    #define ERARNGE()       {__asm jsr 0xFC06;}
    #define PRGRNGE()       {__asm jsr 0xFC09;}
    #define DELNUS()        {__asm jsr 0xFC0C;}
    #define GET_BIT()       {__asm jsr 0xFF00;}
    #define PUT_BYTE()      {__asm jsr 0xFED5;}
#endif  // MC68HC908JB8

#endif  //__MCU_CONSTANTS_H__
```

```
/*====================================================================*
 *
 * Copyright (c) 2002,
 * Freescale Application Note
 *
 * File name    : flash_api.h
 * Author       : Mauricio Capistran-Garza
 * Department   : Guadalajara - SPS
 *
 * Description  : It contains all API function declarations.
 *
 * History      :
 *
 *====================================================================*/

#ifndef __FLASH_API_H__
#define __FLASH_API_H__


/**********************************************************************
    DEFINES
 **********************************************************************/
/* MCU used */
#ifndef MC68HC908JL3
  #ifndef MC68HC908GR8
    #ifndef MC68HC908KX8
      #ifndef MC68HC908JB8
        #define MC68HC908JL3   /* Default MCU used */
      #endif
    #endif
  #endif
#endif

/* Frequency of operation */
#ifndef OSC
    #define OSC      0x04         /* Default freqnency op = 1Mhz */
#endif


#define ReadRange VerifyRange

#define FAIL      0x00
#define SUCCESS   0x01

#include "MCU_constants.h"   // This file contains the defines
                             // for all the MCUHC908 used
                             // in this API.

/**********************************************************************
    FUNCTION PROTOTYPES
 **********************************************************************/

/**********************************************************************
 * ReadByte: It reads a byte from the communication port
 *           and returns it.
 *
 * Parameters:       None.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           The byte received.
 *
 * Remarks:          The function will not exit until a byte
 *                   is received.
 */

Byte ReadByte(void);
```

```
/*****************************************************************
 * TransmitByte: It sends a byte out the communication port.
 *
 * Parameters:       _data: the byte to be sent.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           The byte received.
 *
 * Remarks:          The function will not exit until a byte
 *                   is received.
 */

void TransmitByte(Byte _data);

/*****************************************************************
 * TransmitRange: It reads a range of FLASH memory and sends
 *                it out the communication port
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num: number of bytes to transmit.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  The checksum is stored in _ini;
 *
 * Return:           SUCCESS or FAIL
 *
 * Remarks:          _num must be less to or equal to 64
 */

Byte TransmitRange(Word *_ini, Byte  _num);


/*****************************************************************
 * ProgramRange: Programs a range of FLASH.
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num: length of the range.
 *
 * Entry Conditions: DATA contains the data to be programmed
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          _num must be less to or equal to 64
 */

void ProgramRange(Word *_ini, Byte  _num);
```

```
/*****************************************************************
 * ProgramRangeX: Programs a range of FLASH after verifying
 *                the range is blank. If it isn't blank it
 *                doesn't programs and returns FAIL.
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num: length of the range.
 *
 * Entry Conditions: DATA contains the data to be programmed
 *
 * Exit Conditions:  None.
 *
 * Return:           SUCCESS or FAIL.
 *
 * Remarks:          _num must be less to or equal to 64
 */

 Byte ProgramRangeX(Word *_ini, Byte  _num);

/*****************************************************************
 * VerifyRange: Verifies a range of FLASH against the data
 *              contained in DATA. It can also be used to
 *              read a range of FLASH into RAM.
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num: length of the range.
 *
 * Entry Conditions: DATA contains the data to be verified.
 *
 * Exit Conditions:  DATA is overwritten with contents of FLASH.
 *                   The checksum is stored in _ini;
 *
 * Return:           The byte received.
 *
 * Remarks:          _num must be less to or equal to 64
 */

Byte VerifyRange(Word *_ini, Byte  _num);

/*****************************************************************
 * ErasePage: It erases a PAGE of FLASH
 *
 * Parameters:       *_page: pointer to any address within
 * *                           the PAGE to be erased.
 * Entry Conditions: None.
 *
 * Exit Conditions:  Interrupts are disabled.
 *
 * Return:           None.
 *
 * Remarks:          All bytes within that PAGE will be driven
 *                   to 0xFF
 */

void ErasePage(Word *_page);
```

```
/*********************************************************************
 * ErasePageX: It erases a PAGE of FLASH but leaves the state
 *             of the interrupts as it was before calling it.
 *
 * Parameters:       *_page: pointer to any address within
 *                           the PAGE to be erased.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          All bytes within that PAGE will be driven
 *                   to 0xFF.
 *                   Interrupts are disabled during the erasing
 *                   of the flash, but are restored to its
 *                   original state before exiting the function.
 */

void ErasePageX (Word *_page);

/*********************************************************************
 * EraseFlash: It erases the entire FLASH.
 *
 * Parameters:       None.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          All bytes will be driven to 0xFF.
 *                   No code in FLASH will be executed after
 *                   this function has been called.
 */

void EraseFlash(void);

#endif   // __FLASH_API_H__

/*====================================================================*
 *
 * Copyright (c) 2002,
 * Freescale Application Note
 *
 * File name    : flash_api.c
 * Author       : Mauricio Capistran-Garza
 * Department   : Guadalajara - SPS
 *
 * Description  : This files contains the API functions definition.
 *
 * History      :
 *
 *====================================================================*/

#include <stdtypes.h>
#include "flash_api.h"
```

```
/*****************************************************************
 * ReadByte: It reads a byte from the communication port
 *           and returns it.
 *
 * Parameters:       None.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           The byte received.
 *
 * Remarks:          The function will not exit until a byte
 *                   is received.
 */

Byte ReadByte(void) {
    Byte _backup1, _backup2;
    Byte _data;

    _backup1 = COMMPORT;          // Backup port values.
    _backup2 = COMMPORT_DIR;
    COMMPORT_DIR &= 0xFE;         // Configure COMMPORT as input.
    COMMPORT &= 0xFE;
    GETBYTE();                    // Call ROM-resident routine.
    __asm sta _data;
    COMMPORT = _backup1;          // Restore port values.
    COMMPORT_DIR = _backup2;
    return _data;
}

/*****************************************************************
 * TransmitByte: It sends a byte out the communication port.
 *
 * Parameters:       _data: the byte to be sent.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           The byte received.
 *
 * Remarks:          The function will not exit until a byte
 *                   is received.
 */

void TransmitByte(Byte _data) {
    Byte _backup1, _backup2;

    _backup1 = COMMPORT;          // Backup port values.
    _backup2 = COMMPORT_DIR;
    COMMPORT_DIR &= 0xFE;         // Configure COMMPORT as input.
    COMMPORT &= 0xFE;
    __asm LDA _data;
    PUT_BYTE();                   // Call ROM-resident routine.
    COMMPORT = _backup1;          // Restore port values.
    COMMPORT_DIR = _backup2;
    return;
}
```

```
/*********************************************************************
 * TransmitRange: It reads a range of FLASH memory and sends
 *                it out the communication port
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num:  number of bytes to transmit.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  The checksum is stored in _ini;
 *
 * Return:           SUCCESS or FAIL
 *
 * Remarks:          _num must be less to or equal to 64
 */

Byte TransmitRange(Word *_ini, Byte  _num) {
    Byte _backup1, _backup2, _backup3;
    Byte _status = 0;
    Word _first;

    _first = *_ini;
    _backup1 = COMMPORT;          // Backup port values.
    _backup2 = COMMPORT_DIR;
    COMMPORT &= 0xFE;             // Configure COMMPORT as input.
    COMMPORT_DIR &= 0xFE;

                                  // Define Last Address High & Low
    LADDRH = ((_first + _num - 1) & 0xFF00) >> 8;
    LADDRL = ((_first + _num - 1) & 0x00FF);

    __asm  ldhx _first;           // Define first address.
    __asm  lda  #0x00;            // Configure RDVRRNG() to transmit.
    RDVRRNG();                    // Call ROM-resident routine.
    __asm  sta _backup3;          // Store checksum.
    __asm  clra;
    __asm  adc #0;
    __asm  sta _status;           // Store status.
    *_ini = _backup3;
    COMMPORT = _backup1;          // Restore port values.
    COMMPORT_DIR = _backup2;
    return _status;
}
```

```
/*****************************************************************
 * ProgramRange: Programs a range of FLASH.
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num: length of the range.
 *
 * Entry Conditions: DATA contains the data to be programmed
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          _num must be less to or equal to 64
 */

void ProgramRange(Word *_ini, Byte  _num) {
    Word _first;

    _first = *_ini;
    FLBPR = 0xFF;                 // Disables write protection.
    CPUSPD = OSC;                 // Set Clock Bus Operation speed.

                                  // Define Last Address High & Low
    LADDRH = ((_first + _num - 1) & 0xFF00) >> 8;
    LADDRL = ((_first + _num - 1) & 0x00FF);

    __asm  ldhx _first;           // Define first address.
    PRGRNGE();                    // Call ROM-resident routine.
    FLBPR = 0x00;                 // Enables write protection.
    return;
}
```

```
/*********************************************************************
 * ProgramRangeX: Programs a range of FLASH after verifying
 *                the range is blank. If it isn't blank it
 *                doesn't programs and returns FAIL.
 *
 * Parameters:        _*ini: pointer to the starting address
 *                           of the range.
 *                    _num:  length of the range.
 *
 * Entry Conditions: DATA contains the data to be programmed
 *
 * Exit Conditions:  None.
 *
 * Return:           SUCCESS or FAIL.
 *
 * Remarks:          _num must be less to or equal to 64
 */

Byte ProgramRangeX(Word *_ini, Byte  _num) {
    Byte _test;
    Byte _i;
    Word _first;

    for (_i=0; _i < _num; _i++) {
        _test =  (*(Byte*)(*_ini + _i));
        if (_test != 0xFF) {
            return FAIL;
        }
    }
    _first = *_ini;
    FLBPR = 0xFF;                   // Disables write protection.
    CPUSPD = OSC;                   // Set Clock Bus Operation speed.

                                    // Define Last Address High & Low
    LADDRH = ((_first + _num - 1) & 0xFF00) >> 8;
    LADDRL = ((_first + _num - 1) & 0x00FF);

    __asm  ldhx _first;             // Define first address.
    PRGRNGE();                      // Call ROM-resident routine.
    FLBPR = 0x00;                   // Enables write protection.
    return SUCCESS;
}
```

```
/*********************************************************************
 * VerifyRange: Verifies a range of FLASH against the data
 *              contained in DATA. It can also be used to
 *              read a range of FLASH into RAM.
 *
 * Parameters:       _*ini: pointer to the starting address
 *                          of the range.
 *                   _num:  length of the range.
 *
 * Entry Conditions: DATA contains the data to be verified.
 *
 * Exit Conditions:  DATA is overwritten with contents of FLASH.
 *                   The checksum is stored in _ini;
 *
 * Return:           The byte received.
 *
 * Remarks:          _num must be less to or equal to 64
 */

Byte VerifyRange(Word *_ini, Byte  _num) {
    Byte _backup1;
    Byte _status = 0;
    Word _first;

    _first = *_ini;

                                    // Define Last Address High & Low
    LADDRH = ((_first + _num - 1) & 0xFF00) >> 8;
    LADDRL = ((_first + _num - 1) & 0x00FF);

    __asm  ldhx _first;           // Define first address.
    __asm  lda  #0x01;            // Config RDVRRNG() to store in RAM.
    RDVRRNG();                    // Call ROM-resident routine.
    __asm  sta _backup1;          // Store checksum.
    __asm  clra;
    __asm  adc #0;
    __asm  sta _status;           // Store status.
    *_ini = _backup1;
    return _status;
}
```

```
/*********************************************************************
 * ErasePage: It erases a PAGE of FLASH
 *
 * Parameters:       *_page: pointer to any address within
 *                           the PAGE to be erased.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  Interrupts are disabled.
 *
 * Return:           None.
 *
 * Remarks:          All bytes within that PAGE will be driven
 *                   to 0xFF
 */

void ErasePage(Word *_page) {
    Word _address;

    _address = *_page;
    FLBPR = 0xFF;                  // Enables erase/write protection.
    CPUSPD = OSC;                  // Set Clock Bus Operation speed.
    CTRLBYT &= 0xBF;               // Clear bit 6 to page erase mode.
    __asm ldhx _address;           // Set the page to be erased.
    ERARNGE();                     // Call ROM-resident routine.
    return;
}

/*********************************************************************
 * ErasePageX: It erases a PAGE of FLASH but leaves the state
 *             of the interrupts as it was before calling it.
 *
 * Parameters:       *_page: pointer to any address within
 *                           the PAGE to be erased.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          All bytes within that PAGE will be driven
 *                   to 0xFF.
 *                   Interrupts are disabled during the erasing
 *                   of the flash, but are restored to its
 *                   original state before exiting the function.
 */

void ErasePageX (Word *_page) {
    Byte _backup1;
    Word _address;

    __asm  tpa;
    __asm  sta _backup1;           // Backup Condition Code Register
    _address = *_page;
    FLBPR = 0xFF;                  // Enables erase/write protection.
    CPUSPD = OSC;                  // Set Clock Bus Operation speed.
    CTRLBYT &= 0xBF;               // Clear bit 6 to page erase mode.
    __asm ldhx _address;           // Set the page to be erased.
    ERARNGE();                     // Call ROM-resident routine.
    if ((_backup1 & 0x80) != 0x80) { // Restore interrupts state.
        __asm CLI;
    }
    return;
}
```

```
/*****************************************************************
 * EraseFlash: It erases the entire FLASH.
 *
 * Parameters:       None.
 *
 * Entry Conditions: None.
 *
 * Exit Conditions:  None.
 *
 * Return:           None.
 *
 * Remarks:          All bytes will be driven to 0xFF.
 *                   No code in FLASH will be executed after
 *                   this function has been called.
 */

void EraseFlash(void) {
    FLBPR = 0xFF;                // Enables erase/write protection.
    CPUSPD = OSC;                // Set Clock Bus Operation speed.
    CTRLBYT |= 0x40;             // Set bit 6 to flash erase mode.
    ERARNGE();                   // Call ROM-resident routine.
}
```

[redacted]

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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)
support@freescale.com

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor