

## H8S Family

### User Mode FLASH Programming Example

---

#### Introduction

One of the most useful features of microcontrollers which incorporate FLASH memory is their ability to 'self program' their FLASH memory.

With Renesas H8 & H8S microcontrollers when this FLASH programming is occurring, the application has to be executing from a memory source other than FLASH. Typically this is the internal RAM of the device. With single chip devices such as the H8S/2612, it has to be internal RAM. The processes of relocating and executing code from RAM can pose several problems, more of which will be discussed later.

Aspects associated with FLASH programming are discussed in several application notes. Examples of these are: App Notes REG05B0021-0100, REG05B0022-0100 and REG05B0023-0100.

It is recommended that these three application notes be read either in conjunction or prior to this application note.

It is the aim of this Application Note to bring together all of the concepts discussed in the earlier Application Notes into one simple example.

*The Application Note will show how an H8S/2612 can reprogram an ADC value into a user defined FLASH block in response to external interrupt.*

## Contents

USER MODE FLASH PROGRAMMING EXAMPLE .....	1
INTRODUCTION .....	1
CONTENTS .....	2
AN OVERVIEW OF FLASH PROGRAMMING .....	3
FLASH_DEMO HEW WORKSPACE & PROJECTS .....	6
'EXAMPLEAPP' MEMORY MAP .....	7
'EXAMPLEAPP' LINKER SETTINGS .....	8
'EXAMPLEAPP' HARDWARE OVERVIEW .....	9
'EXAMPLEAPP' APPLICATION OVERVIEW .....	9
'EXAMPLEAPP' CODE.....	11
POWERON RESET: .....	11
HARDWARE SETUP: .....	12
MAIN: .....	13
GLOBAL DECLARATIONS: .....	14
IRQ0 CODE: .....	14
ADC CODE: .....	17
_FERASE PROJECT .....	18
_FLASH_EARSE.C .....	18
_FERASE.C LINKER SETTINGS .....	22
_FPROGRAM PROJECT .....	23
FLASH_PROG.C .....	23
_FPROGRAM LINKER SETTINGS .....	27
SUMMARY .....	28
WEBSITE AND SUPPORT .....	28

## An Overview of FLASH Programming

It was mentioned in the introduction of the application note that when the FLASH memory of an H8S device is being erased or programmed, then the application code has to be executing from a memory other than FLASH, typically RAM.

At first the solution to this ‘problem’ seems straightforward enough. At runtime copy the program or erase routine from FLASH into RAM and call it via a function pointer. In many cases this method will work but cannot be guaranteed. The reason being that any jumps within the code or to subroutines may refer to absolute addresses. Therefore, the code may be executing correctly in RAM and then jump back into the FLASH unexpectedly. This can be avoided by using only branch statements that use offsets relative to the program counter but unfortunately with the current H8S tools there is no way to force the output of position independent code exclusively utilising branches.

The solution to this problem is to link the code that must run from RAM to the actual RAM addresses at build time. This can introduce further problems. The first is that of library routines. If a RAM based function is part of a larger project then it may happily run from RAM but may feature calls to library routines that are linked to FLASH addresses causing accesses to FLASH memory at undesirable moments during execution. Even something as innocuous as the C statement below can result in a library call.

```
i = 1 << some_variable;
```

Simply looking through the C source and avoiding calls to functions such as ‘printf’ is not enough to guarantee that there are no library calls to FLASH based routines.

The second issue concerning copying functions from FLASH to RAM is that of constant data. If the RAM routine makes reference to constant data, including items such as string literals, this can cause the FLASH memory to be accessed.

A third consideration is how to get code that is linked to RAM into FLASH for storage at build time and then back into RAM at runtime for execution.

A solution to these problems is to place the entire RAM based routines into completely separate projects with all the code, variable and constant data linked to the RAM addresses. This eliminates the problems of jumps back into FLASH for code, libraries and constant data. Getting this code from the RAM addresses into the FLASH for storage at build time can be achieved by using the ‘notice\_cl’ utility and method described in Application Note REG05B0021-0100.

This utility converts an S-record file into a constant ‘C’ array. For example, a FLASH erasing function is built as a separate project and linked to RAM. The linker is configured so that it outputs an S-record file for this project. This S-record is processed by ‘notice\_cl’ which converts it into a constant ‘C’ array which can be included into the Application project. As the array is constant data it resides in the FLASH. When the erase routine is to be called by the Application the constant array data is copied to the correct place in RAM and called by a function pointer. While the erase routine is executing only RAM is accessed for program code and data as this is all the routine knows about as it has been linked to RAM addresses in a separate project.

The above method relies on 3 things being known at runtime. These are:

1. The start address that the RAM code should be copied to from FLASH. This is achieved by storing the constant data as part of a structure which contains the start address (put there by 'notice\_cl' from the s-record) and the length of the data.
2. The size of the data to be copied to RAM so the copying routine knows how much data to move. See the explanation above for how this is known.
3. If the RAM based code contains multiple functions, e.g. erase and delay routines, the start addresses for these functions must be known so they can be correctly called via function pointers. This can be achieved by loading these addresses into a 'vector' table starting at the beginning of the RAM code area. Although the addresses of the functions may change, the location of where the value and order of these are stored does not and is known by the Application. All the Application must do is read the correct address and call the function via a pointer.

## 'ExampleApp' Software Overview

All of the software for 'ExampleApp' was written using the Renesas Integrated Debugging Environment (IDE), HEW. The version of HEW used was version 3.06. It should be noted that any HEW version could be used.

As detailed in the previous section, one of the ways to avoid the problems associated with FLASH programming is to place all of the RAM based routines into separate projects. This is the technique that was used to develop this example.

Within HEW, three projects were created under one workspace. Table 1 details the projects created within the workspace.

Name	Comment
FLASH_Demo_HEW_3	Workspace to which projects are added.
_ferase	Project that contains all code required to implement FLASH Erasing. The S-Record generated by this file is converted using notice_cl <sup>1</sup> to a constant 'C' structure called ferase_converted.c This file is added to the application project ExampleApp_HEW_3
_fprogram	Project that contains all code required to implement FLASH Programming. The S-Record generated by this file is converted using notice_cl <sup>1</sup> to a constant 'C' structure called fprogram_converted.c This file is added to the application project ExampleApp_HEW_3
ExampleApp_HEW_3	Project that contains the files required for the application. 2 addition files are added to the project: ferase_converted.c & fprogram_converted.c These 2 files contain the erasing and programming execution code pre linked to the RAM address from which they will execute. The application copies this code from FLASH to RAM when User Mode FLASH programming is required.

<sup>1</sup>Refer to Application Note REG05B0021-0100

*Table 1.*

A screen shot taken from HEW 3 is shown in figure 1 showing the Workspace and projects.

It is mentioned in table 1 that pre linked execution codes are copied from the FLASH memory to internal RAM as and when required. To ensure that no data is corrupted when the code is copied up to RAM, it is necessary to reserve an area of RAM. This is done via the linker settings.

Figure 2 and table 2 show the 'ExampleApp' memory map and linker settings.

## FLASH\_Demo HEW Workspace & Projects

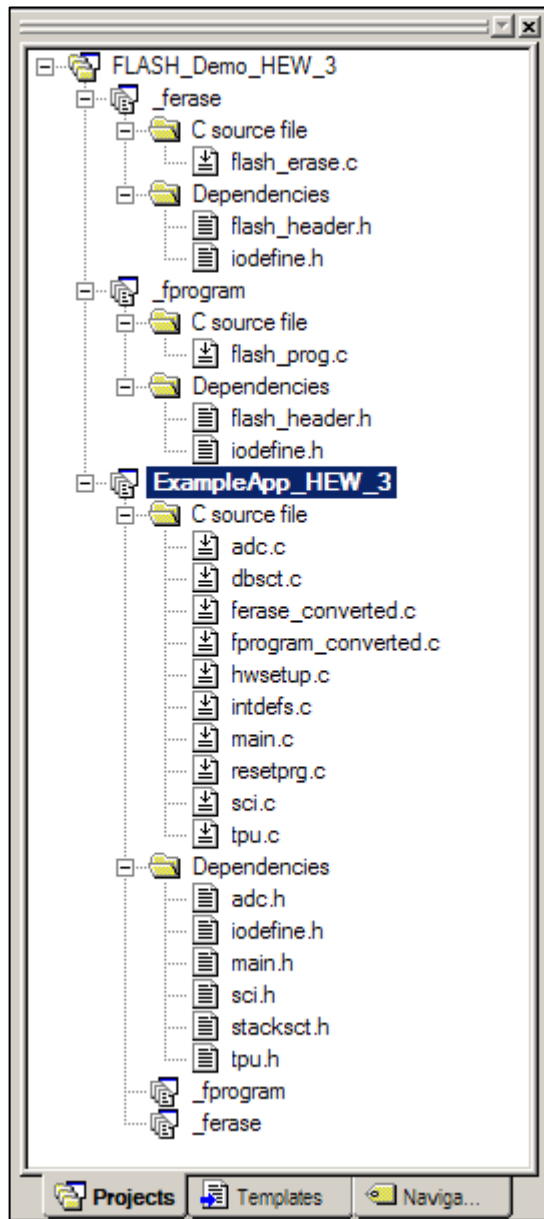


Figure 1

'ExampleApp' Memory Map

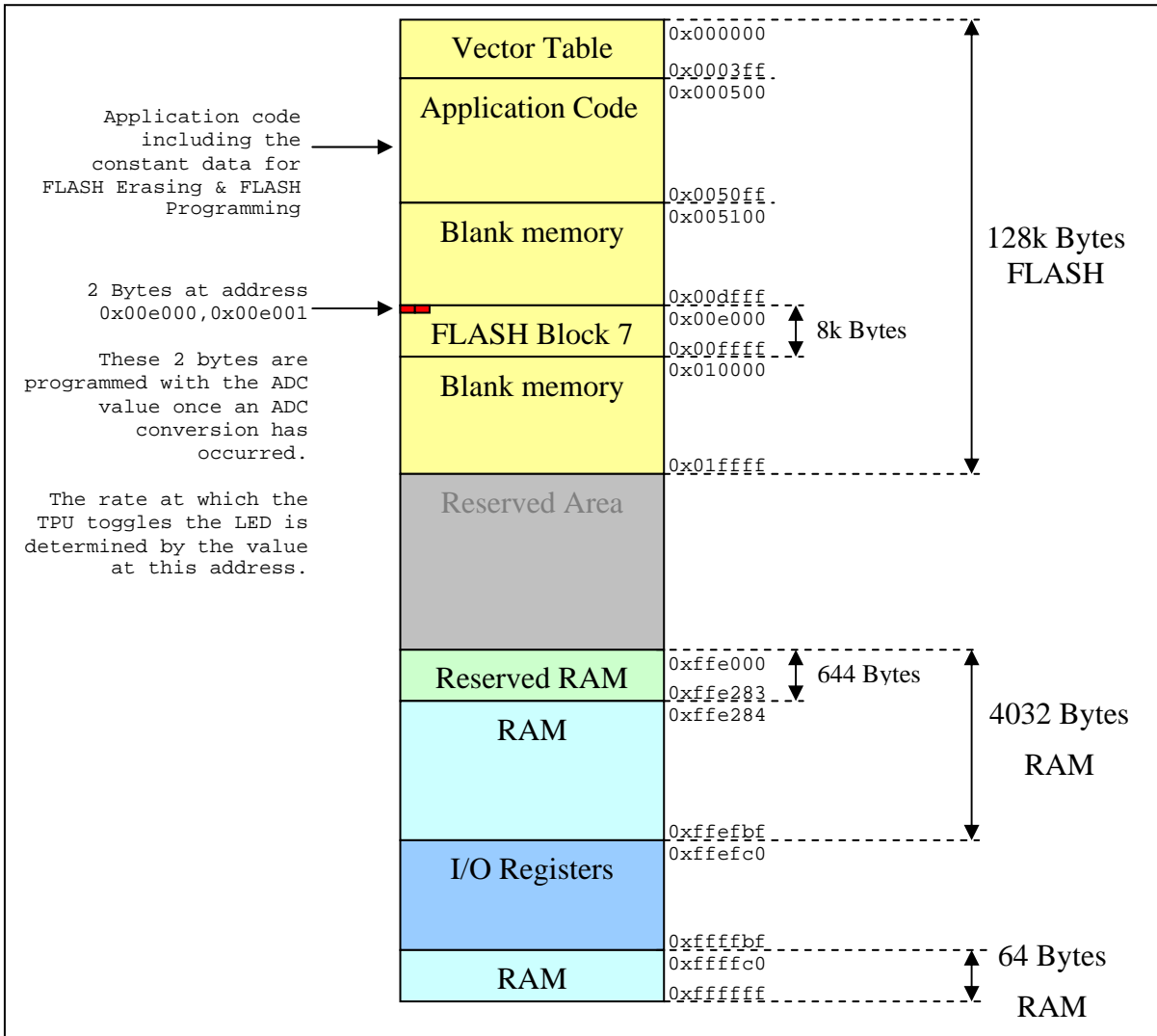


Figure 2.

## 'ExampleApp' Linker Settings

Address	Section	Comment
0x000000	DVECTTBL	Stores addresses of Power-On Reset Vector and Manual Reset <sup>1</sup> Vector
	DINTTBL	Stores addresses of Interrupt Vector Table
0x000500	PResetPRG	Stores PowerOnReset( ) and ManualReset( ) <sup>1</sup> code.
	PIntPRG	Stores the default execution code for all Interrupt Service Routines. The default function is the sleep( ) function.
0x000800	P	Program Code
	C	Constant Data
	C\$DSEC	Address area for Initialised data section. Stores ROM addresses, final addresses in ROM, and RAM addresses for initialised data area sections
	C\$BSEC	Address area for Non-Initialised data section. Stores addresses and final addresses for Non-Initialised data area sections
	D	Initialised Data
0xffe000	BRESERVED	A User Defined Area. This Area of RAM is reserved for the FLASH 'Erase' & 'Program' algorithms. These execute from RAM.
0xffe400	B	Non-Initialised Data
	R	Reserved area for Initialised Data. Data is copied from D to R by _INTSCT
0xffed00	S	Stack

<sup>1</sup>Not implemented on H8S/2612

*Table 2*

## 'ExampleApp' Hardware Overview

ExampleApp was developed using an EDK2612. Additional hardware requirements are a 10k pot, connected to Port 4.0 (Analogue input 0), a 10k resistor and a switch connected to Port 1.4 (IRQ0).

Figure 3 shows the hardware required for the application. The shaded components are the ones, which had to be added to EDK2612.

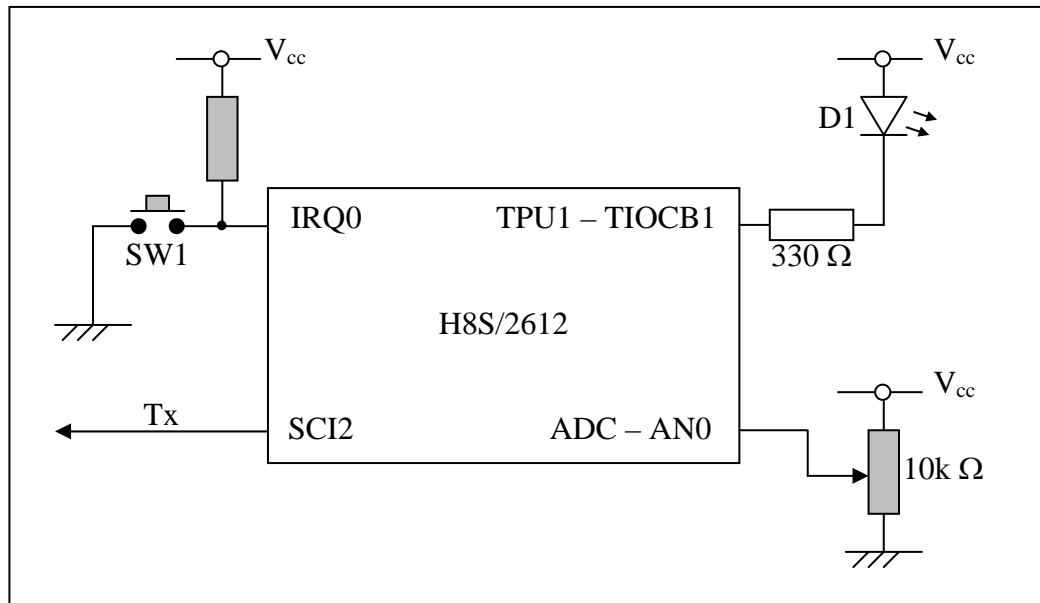


Figure 3.

## 'ExampleApp' Application Overview

TPU1 is configured to toggle the Timer I/O pin. This provides a visual indication that the application is running.

The ADC is configured to perform single AD Conversion on channel AN0.

SCI2 is configured to transmit data at 9600 Baud, 8 data bits, 1 stop bit, no parity.

IRQ0 is configured to generate an interrupt on a falling edge.

When the H8S/2612 is powered and comes out of reset, the device is initialised and TPU1 will toggle the LED D1. The application effectively now does nothing, sitting in a while(1) loop until an IRQ0 interrupt is generated by the user pressing switch SW1.

When the CPU excepts the IRQ0 interrupt, the IRQ0 ISR (Interrupt Service Routine) is executed. The IRQ0 ISR performs the following.

1. Start the ADC and when the ADC has completed, assign the ADC result (Channel AN0) to a variable.
2. Copy the FLASH Erase routine from FLASH memory to RAM.
3. Execute the FLASH Erase routine and erase the FLASH block.
4. If the FLASH erase is successful, copy the FLASH Programming routine to RAM.
5. Program the previously obtained ADC value into FLASH.
6. If the FLASH Program is successful set the TGR1B value equal to this new value. The rate at which the LED toggles is controlled by the value in the TGR1B register. This provides a visual indication that the FLASH memory has been programmed with a new value. In addition, the new value programmed into FLASH is transmitted via the serial port at 9600 Baud. If SCI2 is connected to a PC application such as Hyper terminal, the new value may be viewed.
7. If either FLASH Erase or FLASH Program is not successful, the TPU is stopped. This stops the LED toggling to indicate an error condition.

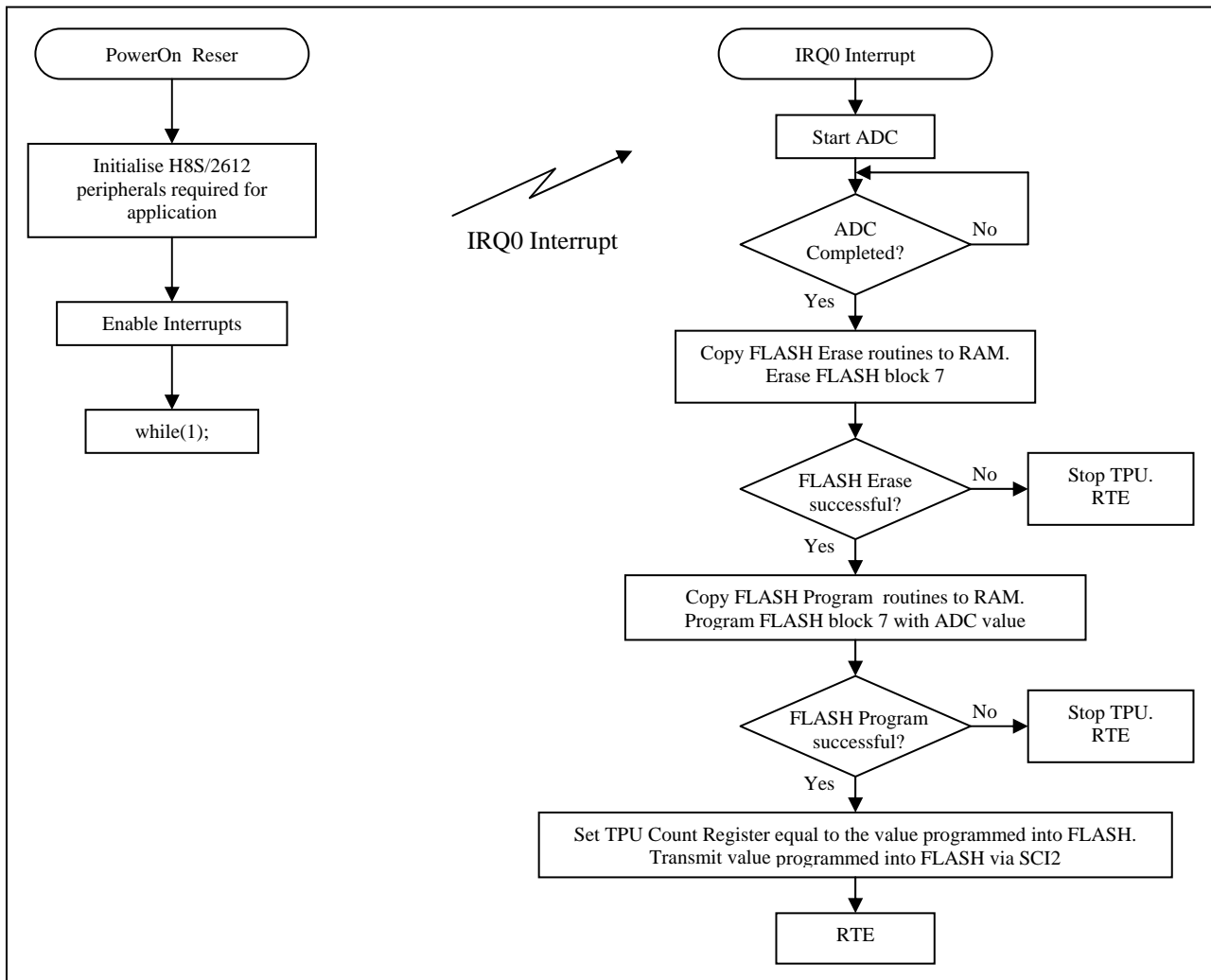


Figure 4.Basic Program Flow.

## 'ExampleApp' Code

When the H8S/2612 is powered and comes out of reset the reset vector (address 0x000000) is read. Code execution begins at this address.

After the SP has been initialised, the function PowerOn\_Reset is executed.

### **PowerOn Reset:**

```
// The compiler automatically generates code to set SP when #pragma entry is used
#pragma entry PowerON_Reset

// Ensure the Reset Code is positioned by the linker to the correct address
#pragma section ResetPRG

void PowerON_Reset(void)
{
    set_imask_ccr(1);           // Mask all interrupts - Mode 0
    set_imask_exr(7);          // Mask all interrupts - Mode 1

    HardwareSetup();           // Configure interrupt controller & I/O ports

    _INITSCT();                 // Library function
                                // Initialises & Non Initialised data Set up

    main();                     // Main application

    sleep();                    // Just incase we fall out of main();
}
```

The functions HardwareSetup() & main() are listed on pages 12 & 13 respectively.

The function \_INITSCT() is a library function.

### Hardware Setup:

```

void HardwareSetup(void)
{
    unsigned char P1DDRSshadow;

    SYSCR.BIT.MACS = 0;        // 0 - Non Saturating Calculation for MAC
                              // 1 - Saturating Calculation for MAC

    SYSCR.BIT.INTM = 0;       // 0 - Interrupt Mode 0
                              // 1 - Setting Prohibited
                              // 2 - Interrupt Mode 2
                              // 3 - Setting Prohibited

    SYSCR.BIT.NMIEG = 0;     // 0 - NMI on falling edge
                              // 1 - NMI on rising edge

    SYSCR.BIT.RAME = 1;      // 0 - On Chip RAM disabled
                              // 1 - On Chip RAM enabled

    // Set Interrupt Priorities for INT MODE 2
    // Note: Not used in this application as we are using INT MODE 0

    INTC.IPRA.BIT._IRQ0 = 1;   // IRQ0
    INTC.IPRA.BIT._IRQ1 = 1;   // IRQ1
    INTC.IPRB.BIT._IRQ23 = 1;  // IRQ2,IRQ3
    INTC.IPRB.BIT._IRQ45 = 1;  // IRQ4,IRQ5
    INTC.IPRC.BIT._DTC = 1;    // DTC
    INTC.IPRD.BIT._WDT = 1;    // WDT
    INTC.IPRE.BIT._PBC = 1;    // PBC
    INTC.IPRE.BIT._AD = 1;     // A/D
    INTC.IPRF.BIT._TPU0 = 1;   // TPU0
    INTC.IPRF.BIT._TPU1 = 1;   // TPU1
    INTC.IPRG.BIT._TPU2 = 1;   // TPU2
    INTC.IPRG.BIT._TPU3 = 1;   // TPU3
    INTC.IPRH.BIT._TPU4 = 1;   // TPU4
    INTC.IPRH.BIT._TPU5 = 1;   // TPU5
    INTC.IPRJ.BIT._SCI0 = 1;   // SCI0
    INTC.IPRK.BIT._SCI1 = 1;   // SCI1
    INTC.IPRK.BIT._SCI2 = 1;   // SCI2
    INTC.IPRM.BIT._HCAN = 1;   // HCAN
    INTC.IPRM.BIT._MMT = 1;    // MMT

    // Enable IRQ0
    INTC.IER.BIT.IRQ0E = 1;

    // Set IRQ0 sense control
    INTC.ISCR.BIT.IRQ0SC = 1;  // 00: Interrupt request generated at low level
                              // 01: Interrupt request generated at falling edge
                              // 10: Interrupt request generated at rising edge
                              // 11: Interrupt request generated at both
                              // falling and rising edges

    // It is not possible to do bit manipulation on Port 'DDR' Registers
    // Therefore, use a 'Shadow' register
    P1DDRSshadow = P1.DDR;     // Assign DDR value to Shadow
    P1DDRSshadow &= ~0x10;     // Perform bit manipulation - Set Bit 4 to 0, Input
    P1.DDR = P1DDRSshadow;     // Set DDR value equal to Shadow
}

```

The function main( ) performs function calls that enable and initialise the 3 peripherals that are required in this application. These are the:

- ADC
- SCI
- TPU

**Main:**

```
void main(void)
{
    // All peripherals on H8S devices (except DMAC / DTC) are disabled by default.
    // Therefore the peripherals have to be enabled by taking them
    // out of Module Stop Mode
    Enable_ADC();
    Enable_SCI(2);
    Enable_TPU();

    // Initialise the Peripherals that will be used for the application
    Init_ADC();
    Init_SCI2( BAUD( 9600L ) );
    Init_TPU1();

    Start_TPU(1);           // Toggles LED

    set_imask_ccr(0);      // Enable interrupts

    while(1);
}
```

Once the 3 peripherals are enabled and initialised the application sits in a while(1) loop. Nothing else will happen until an IRQ0 is generated. The IRQ0 is enabled as part of the initialisation of the interrupt controller in function HardwareSetup( ).

It is response to the IRQ0 interrupt that erases and programs a single block of the FLASH memory.

When an IRQ0 occurs, the following occurs:

1. Start the ADC and when the ADC has completed, read the ADC result.
2. Copy the FLASH ‘Erase’ routine from ROM to FLASH
3. Execute the FLASH ‘Erase’ routine and erase the FLASH block
4. If the FLASH ‘Erase’ is successful, copy the FLASH ‘Program’ routine to FLASH
5. Program the previously obtained ADC value into FLASH
6. If the FLASH ‘Program’ is successful set the TGR1B value equal to this new value. The rate at which the LED toggles is controlled by the value in the TGR1B register. In addition, the new value programmed into FLASH is transmitted via the serial port at 9600 Baud.
7. If either the FLASH ‘Erase’ or FLASH ‘Program’ are not successful stop the TPU. This stops the LED toggling to indicate an error condition.

### Global declarations:

```
#pragma section RESERVED
    unsigned char ProgEraseArray[0x3ff];    // Reserved area to which
#pragma section                                // FLASH Erase & FLASH Program
                                                // code will be copied to.

extern const struct rom_data ferase;        // FLASH Erase code
extern const struct rom_data fprogram;     // FLASH Program code

//union definition that allows access to short to char data types
union {
    unsigned char c[2];    // c[1] = lsb
    unsigned short s;     // c[0] = msb
}c2s;

union char_rd_datum_union {
    unsigned char c[FLASH_LINE_SIZE];
    unsigned short s[FLASH_LINE_SIZE / 2];
}Prog_Data;

// Function Pointers
// These are used to call the functions required for
// FLASH Erasing and FLASH Programming
void (*ptr2_Init_ERASE_delay)(void);
unsigned char (*ptr2_Function_Erase)(unsigned char );

void (*ptr2_Init_PROG_delay)(void);
unsigned char (*ptr2_Function_PROG)(unsigned long, union char_rd_datum_union*);
```

### IRQ0 Code:

```
#pragma interrupt(_INT_IRQ0)
void _INT_IRQ0(void)
{
    char Buffer[4];
    unsigned char Index;
    unsigned char Erase_Status = 0;
    unsigned char Prog_Status = 0;
    unsigned short *s_ptr;
    unsigned char *c_ptr;

    // read the ADC data and assign it to the variable c2s.s
    c2s.s = Read_ADC_Value() & 0xFFC0;

    // Before the FLASH can be written to, it has to be in an erased state.
    // When the FLASH is being Erased or Programmed, code operation HAS to be
    // external to the FLASH, i.e. from internal or external RAM.
    // As the H8S/2612 is a single chip device, it is internal RAM!
    // Copy the required routines from ROM to RAM

    // memcpy( To Destination,      From Source,      Length of data      );
    memcpy( &ProgEraseArray[ 0 ], &ferase.data[ 0 ], ferase.data_length);

    // Initialise the TPU.
    // The Erase & Program routines require accurate timing pulses
    // The TPU generates these.
```

```

// Initialise the function pointer
ptr2_Init_ERASE_delay = (void*)INIT_ERASE_DT;    // Function address specified in main.h
// Function Call
ptr2_Init_ERASE_delay();

// Initialise the function pointer
ptr2_Function_Erase = (void*)ERASE_FUNC;        // Function address specified in main.h

// Function Call
// The Parameter specifies which Block is to be erased
// In this case BLOCK 7, Address 0x00E000;
Erase_Status = ptr2_Function_Erase(7);

if(Erase_Status == ERASE_PASS)
{
    // FLASH Block has erased successfully
    // The FLASH is programmed 128 bytes at a time
    // In this application we are only programming the first 2 bytes
    // Therefore fill the entire 128 bytes with 0xff
    for(Index=0; Index<128; Index++)
    {
        Prog_Data.c[Index] = 0xff;
    }

    // Fill elements 0 & 1 of the array with the required data
    Prog_Data.s[0] = c2s.s;

    // Copy the required routines from ROM to RAM
// memcpy( To Destination,      From Source,      Length of data      );
    memcpy( &ProgEraseArray[ 0 ], &fprogram.data[ 0 ], fprogram.data_length);

    // Initialise the TPU.
    // The Erase & Program routines require accurate timing pulses
    // The TPU generates these.

    // Initialise the function pointer
    ptr2_Init_PROG_delay = (void*)INIT_PROG_DT;    // Function address specified
                                                    // in main.h

    // Function Call
    ptr2_Init_PROG_delay();

    // Initialise the function pointer
    ptr2_Function_PROG = (void*)PROG_FUNC;    // Function address specified in main.h
    // Function Call
    // We pass the address to be written to, in this case 0xe000
    // and the address of data to be programmed, in this case
    // the start address of the array prog_data[128];
    Prog_Status = ptr2_Function_PROG(0xe000, &Prog_Data);
}

```

```

        if(Prog_Status == PROG_PASS)
        {
            // Read the contents of the address that has just been programmed
            // Adjust the TGR of TPU controlling the LED

            // initialise the ptr to the correct address
            s_ptr = (unsigned short *)0xe000;
            // Timer TGR = the contents of address
            TPU1.TGRB = *s_ptr;

            // Convert the contents of the address to a string
            // Send the string out of the serial port, SCI2
            c_ptr = (unsigned char *)0xe000;
            sprintf(Buffer, "%d\n\r", *c_ptr++, *c_ptr);
            Send_String(Buffer);
        }
        else
        {
            Stop_TPU(1);
        }
    }
    else
    {
        Stop_TPU(1);
    }

    INTC.ISR.BIT.IRQ0F = 0;           // Clear Interrupt Flag
}

```

**ADC Code:**

```

void Enable_ADC(void)
{
    MSTP.CRA.BIT._AD = 0;           // Enable ADC - Clear BIT to 0
}

void Disable_ADC(void)
{
    MSTP.CRA.BIT._AD = 1;           // Disable ADC - Set BIT to 1
}

void Init_ADC(void)
{
    AD.ADCSR.BIT.ADST = 0;           // 0 - Stop ADC
                                     // 1 - Start ADC

    AD.ADCSR.BIT.ADIE = 0;           // 0 - ADC End Interrupt Disabled
                                     // 1 - ADC End Interrupt Enabled

    AD.ADCSR.BIT.SCAN = 0;           // 0 - Scan Mode Disabled
                                     // 1 - Scan Mode Enabled

    AD.ADCSR.BIT.CH = 0;             // Channel 0 selected;

    AD.ADCR.BIT.TRGS = 0;            // 00: A/D conversion start by software is enabled
                                     // 01: A/D conversion start by TPU conversion start
                                     //      trigger is enabled
                                     // 10: Setting prohibited
                                     // 11: A/D conversion start by external trigger pin
                                     //      (ADTRG) is enabled

    AD.ADCR.BIT.CKS = 2;             // 134 State conversion time
}

unsigned short Read_ADC_Value(void)
{
    AD.ADCSR.BIT.ADST = 1;           // Start ADC

    while(AD.ADCSR.BIT.ADF != 1);    // Wait for end of ADC flag

    AD.ADCSR.BIT.ADF = 0;            // Clear ADC flag

    return(AD.ADDRA);
}

```

## \_ferase Project

The code for the FLASH Erasing is developed as a separate project. The code and linker settings are shown over the next couple of pages.

### flash\_erase.c

```
#include "flash_header.h"

// prototypes
unsigned char  init_erase_delay_timer ( unsigned char );
void erase_delay (unsigned short d);
unsigned char erase_block_035_um (unsigned char block_num);

#pragma section CONSTANTS
const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] = {
    0x00000000L,
    0x00000400L,
    0x00000800L,
    0x00000c00L,
    0x00001000L,
    0x00008000L,
    0x0000c000L,
    0x0000e000L,
    0x00010000L,
    0x00018000L,
    0x00020000L}; /* max flash address + 1 */

const unsigned char EraseBlocks[ 8 ] = {
    0x01,
    0x02,
    0x04,
    0x08,
    0x10,
    0x20,
    0x40,
    0x80
};
#pragma section
```

```

#pragma section INIT_ERASE_DT
unsigned char init_erase_delay_timer ( unsigned char x )
{
    // enable TPU in module top register
    MSTP.CRA.BIT._TPU = 0;

    FLASH_DELAY_TIMER_CH_TCR.BIT.CCLR = 1;           // TCNT cleared by TGRA C/M, I/C
    FLASH_DELAY_TIMER_CH_TCR.BIT.CKEG = 0;           // Count at rising edge
    FLASH_DELAY_TIMER_CH_TCR.BIT.TPSC = 2;           // Timer pre-scaler = clk / 16

    FLASH_DELAY_TIMER_CH_TMDR.BIT.MD = 0;            // Normal operation

    FLASH_DELAY_TIMER_CH_TIOR.BIT.IOB = 0;           // Output disabled
    FLASH_DELAY_TIMER_CH_TIOR.BIT.IOA = 0;           // Output disabled

    FLASH_DELAY_TIMER_CH_TIER.BIT.TTGE = 0;          // ADC start request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TCIEU = 0;         // Underflow interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TCIEV = 0;         // Overflow interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TGIEB = 0;         // TGRB interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TGIEA = 0;         // TGRA interrupt request enabled

    FLASH_DELAY_TIMER_CH_TCNT = 0;
    FLASH_DELAY_TIMER_CH_TGRA = 0;
    FLASH_DELAY_TIMER_CH_TGRA = 0;
    FLASH_DELAY_TIMER_CH_TGRB = 0;
    FLASH_DELAY_TIMER_CH_TGRB = 0;

    return 0;
}

```

```

#pragma section
#pragma section ERASE_FUNC
unsigned char erase_block_035_um (unsigned char block_num)
{
    unsigned char erase;           // flag showing erase status - BLANK or NOT_BLANK
    unsigned char ax;             // loop counter
    unsigned long attempts;       // loop counter for erase attempts (0->MAX_ERASE_ATTEMPTS)
    read_datum *ul_v_read;       // pointer for reading verify data
    unsigned char *uc_v_write;    // pointer for writing to verify data area

    // check that block is not already erased
    erase = BLANK;
    for (attempts=eb_block_addr[block_num]; attempts<eb_block_addr[block_num + 1]; attempts++)
    {
        if ( *(unsigned char *) attempts != 0xff)
            erase = NOT_BLANK;
    }

    if (erase == BLANK)
        return ERASE_PASS;
    else
    {
        // block needs erasing
        //
        // enable flash writes
        FLASH_SWE = 1;

        // wait tSSWE
        erase_delay (ONE_USEC);

        // set the correct EB bit in correct EBR register
        // this is usually device specific
        FLASH_EBR1 = 0;
        FLASH_EBR2 = 0;

        if ( block_num < 8 )
        {
            FLASH_EBR1 = EraseBlocks[ block_num ];
        }
        else
        {
            FLASH_EBR2 = EraseBlocks[ block_num - 8 ];
        }

        // initialise the attempts counter
        attempts = 0;
        erase = NOT_BLANK;
        while ( (attempts < MAX_ERASE_ATTEMPTS) && (erase == NOT_BLANK) )
        {
            // increment the attempts counter
            attempts++;

            // enter erase mode
            FLASH_ESU = 1;

            // wait tSESU (100 us)
            erase_delay (ONE_HUNDRED_USEC);

            // start erasing
            FLASH_E = 1;

            // wait tSE
            erase_delay (TEN_MSEC);

            // stop erasing
            FLASH_E = 0;

            // wait tCE
            erase_delay (TEN_USEC);
        }
    }
}

```

```

        // exit erase mode
        FLASH_ESU = 0;

        // wait tCESU
        erase_delay (TEN_USEC);

        // enter erase verify mode
        FLASH_EV = 1;

        // wait tSEV
        erase_delay (TWENTY_USEC);

        // verify flash has been erased
        ul_v_read = (read_datum *) eb_block_addr [block_num];
        uc_v_write = (unsigned char *) eb_block_addr [block_num];

        erase = BLANK;
        while ( (erase == BLANK) && ( ul_v_read < (read_datum *) eb_block_addr
[block_num + 1] ) )
        {
            // this loop will exit either when one long word is not erased
            // or all addresses have been read as erased
            //
            // dummy write
            *uc_v_write = 0xff;

            // wait tSEVR
            erase_delay (TWO_USEC);

            if (*ul_v_read != BLANK_VALUE)
            {
                // this word is not erased yet
                erase = NOT_BLANK;
            }
            else
            {
                // advance to the next byte write address
                for (ax=0; ax<sizeof(read_datum); ax++)
                    uc_v_write++;

                // advance to the next verify read address
                ul_v_read++;
            }
        }

        // exit erase verify mode
        FLASH_EV = 0;

        // wait tCEV
        erase_delay (FOUR_USEC);
    } // end of outer while loop

```

```

        // end either of erase attempts or block has been erased ok
        //
        // disable flash writes
        FLASH_SWE = 0;

        // wait tCSWE
        erase_delay (ONE_HUNDRED_USEC);

        // check if block has been erased ok
        if (erase == BLANK)
        {
            // successfully erased
            return ERASE_PASS;
        }
        else
        {
            // failed to erase this block
            return ERASE_FAIL;
        }
    }
}
#pragma section

#pragma section ERASE_DT
void erase_delay (unsigned short d)
{
    FLASH_DELAY_TIMER_CH_TSR.BIT.TGFA = 0;
    FLASH_DELAY_TIMER_CH_TGRA = d;           // set compare value
    FLASH_DELAY_TIMER_CH_TCNT = 0;         // clear TCNT to 0
    SET_FLASH_DELAY_TIMER_CH_CST;         // start timer
    while(FLASH_DELAY_TIMER_CH_TSR.BIT.TGFA == 0); // wait until compare value is met
    CLEAR_FLASH_DELAY_TIMER_CH_CST;       // stop timer
}
#pragma section

```

### ferase.c Linker Settings

Address	Section	Comment
0xFFE000	CCONSTANTS	Constant data array containing the address of the FLASH blocks
0xFFE400	PINIT_ERASE_DT	Initialisation code for TPU
0xFFE0A0	PERASE_DT	TPU routines
0xFFE0D0	PERASE_FUNC	FLASH Erasing routine

## \_fprogram Project

The code for the FLASH Programming is developed as a separate project. The code and linker settings are shown over the next couple of pages.

### flash\_prog.c

```
#include "flash_header.h"
//#include "command.h"

// function prototypes
unsigned char init_prog_delay_timer ( unsigned long, union char_rd_datum_union * );
void prog_delay (unsigned short d);
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data);

#pragma section INIT_PROG_DT
unsigned char init_prog_delay_timer ( unsigned long x, union char_rd_datum_union * y )
{
    // enable TPU in module stop register
    MSTP.CRA.BIT._TPU = 0;

    FLASH_DELAY_TIMER_CH_TCR.BIT.CCLR = 1;           // TCNT cleared by TGRA C/M, I/C
    FLASH_DELAY_TIMER_CH_TCR.BIT.CKEG = 0;           // Count at rising edge
    FLASH_DELAY_TIMER_CH_TCR.BIT.TPSC = 2;           // Timer pre-scaler = clk / 16

    FLASH_DELAY_TIMER_CH_TMDR.BIT.MD = 0;           // Normal operation

    FLASH_DELAY_TIMER_CH_TIOR.BIT.IOB = 0;           // Output disabled
    FLASH_DELAY_TIMER_CH_TIOR.BIT.IOA = 0;           // Output disabled

    FLASH_DELAY_TIMER_CH_TIER.BIT.TTGE = 0;          // ADC start request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TCIEU = 0;         // Underflow interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TCIEV = 0;         // Overflow interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TGIEB = 0;         // TGRB interrupt request disabled
    FLASH_DELAY_TIMER_CH_TIER.BIT.TGIEA = 0;         // TGRA interrupt request enabled

    FLASH_DELAY_TIMER_CH_TCNT = 0;
    FLASH_DELAY_TIMER_CH_TGRA = 0;
    FLASH_DELAY_TIMER_CH_TGRB = 0;

    return 0;
}
#pragma section

#pragma section PROG_DT
void prog_delay (unsigned short d)
{
    FLASH_DELAY_TIMER_CH_TSR.BIT.TGFA = 0;
    FLASH_DELAY_TIMER_CH_TGRA = d;                   // set compare value
    FLASH_DELAY_TIMER_CH_TCNT = 0;                   // clear TCNT to 0
    SET_FLASH_DELAY_TIMER_CH_CST;                    // start timer
    while(FLASH_DELAY_TIMER_CH_TSR.BIT.TGFA == 0);  // wait until compare value is met
    CLEAR_FLASH_DELAY_TIMER_CH_CST;                  // stop timer
}
#pragma section
#pragma section PROG_FUNC
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data)
{
    // function to program one 128 byte flash line
    // t_address is the start address for the flash line to be programmed
    // data to be prgrammed should be passed to this function in the form of a
    // 'char_rd_datum_union' union pointer
    // data must be written to the flash in byte units

```

```

unsigned short n_prog_count;    // loop counter for programming attempts
                                //(0->MAX_PROG_COUNT)
unsigned short d;              // variable used for various loop counts
unsigned char m;               // flag to indicate if re-programming required
                                //(1=yes 0=no)
unsigned char ax;              // loop counter for incrementing 'uc_v_write_address' ptr
unsigned char *dest_address;   // pointer for writing to flash
unsigned char *uc_v_write_address; // pointer for writing to address to be verified
read_datum *ul_v_read_address; // pointer for reading verify address
union char_rd_datum_union additional_prog_data, re_program_data; // storage on stack

// validate address
if ( ( t_address % FLASH_LINE_SIZE ) != 0 )
{
    return( PROG_FAIL );
}

// if ( ( t_address > ( LAST_USER_FLASH_ADDR - FLASH_LINE_SIZE - 1 ) ) )
// {
//     return( PROG_FAIL );
// }

// enable flash writes
FLASH_SWE = 1;

// wait tSSWE
prog_delay(ONE_USEC);

// copy data from program data area to reprogram data area
for (d=0; d<FLASH_LINE_SIZE; d++)
{
    re_program_data.c[d] = p_data->c[d];
}

// program the data in FLASH_LINE_SIZE byte chunks
for (n_prog_count=0; n_prog_count<MAX_PROG_COUNT; n_prog_count++)
{
    // clear reprogram required flag
    m = 0;

    // copy data from reprogram data area into the flash with byte access
    dest_address = (unsigned char *) t_address;
    for (d=0; d<FLASH_LINE_SIZE; d++)
    {
        *dest_address++ = re_program_data.c[d];
    }

    // apply the write pulse
    // note that this is specified as a sub-routine call in the hw manual
    // flowchart but is part of this single function here
    //
    // if code size is a problem then placing this code in a sub-routine may be
    // beneficial
    //

```

```

// enter program setup
FLASH_PSU = 1;

// wait tSPSU
prog_delay (FIFTY_USEC);

// start programming pulse
FLASH_P = 1;

if (n_prog_count < 6)
    prog_delay (THIRTY_USEC);
else
    prog_delay (TWO_HUNDRED_USEC);

// stop programming
FLASH_P = 0;

// wait tCP
prog_delay (FIVE_USEC);

// exit program setup
FLASH_PSU = 0;

// wait tCPSU
prog_delay (FIVE_USEC);

// verify the data via read_datum size reads
uc_v_write_address = (unsigned char *) t_address;
ul_v_read_address = (read_datum *) t_address;

// enter program verify mode
FLASH_PV = 1;

// wait tSPV
prog_delay (FOUR_USEC);

// read data in read_datum size chunks
// verify loop
for (d=0; d<(FLASH_LINE_SIZE / sizeof(read_datum)); d++)
{
    // dummy write of H'FF to verify address
    *uc_v_write_address = 0xff;

    // wait tSPVR
    prog_delay (TWO_USEC);

    // increment this pointer to get to next verify address
    for (ax=0; ax<sizeof(read_datum); ax++)
        uc_v_write_address++;

    // read verify data
    // check with the original data
    if (*ul_v_read_address != p_data->u[d])
    {
        // 1 or more bits failed to program
        //
        // set the reprogram required flag
        m = 1;
    }
}

```

```

        // check if we need to calculate additional programming data
        if (n_prog_count < 6)
        {
            // calculate additional programming data
            // simple ORing of the reprog and verify data
            additional_prog_data.u[d] = re_program_data.u[d] |
*ul_v_read_address;
        }

        // calculate reprog data
        re_program_data.u[d] = p_data->u[d] | ~(p_data->u[d] | *ul_v_read_address);

        // increment the verify read pointer
        ul_v_read_address++;
    } // end of verify loop

    // exit program verify mode
    FLASH_PV = 0;

    // wait tCPV
    prog_delay (TWO_USEC);

    // check if additional programming is required
    if (n_prog_count < 6)
    {
        // perform additional programming
        //
        // copy data from additional programming area to flash memory
        dest_address = (unsigned char *) t_address;
        for (d=0; d<FLASH_LINE_SIZE; d++)
        {
            *dest_address++ = additional_prog_data.c[d];
        }

        // enter program setup
        FLASH_PSU = 1;

        // wait SPSU
        prog_delay (FIFTY_USEC);

        // start programming pulse
        FLASH_P = 1;

        // wait tSP
        prog_delay (TEN_USEC);

        // stop programming
        FLASH_P = 0;

        // wait
        prog_delay (FIVE_USEC);

        // exit program setup
        FLASH_PSU = 0;

        // wait tCPSU
        prog_delay (FIVE_USEC);
    }

```

```

        // check if flash line has successfully been programmed
        if (m == 0)
        {
            // program verified ok
            //
            // disable flash writes
            FLASH_SWE = 0;

            // wait tCSWE
            prog_delay (ONE_HUNDRED_USEC);

            // end of successful programming
            return (PROG_PASS);
        }
    } // end of for loop (n<MAX_PROG_COUNT) at this point we have made MAX_PROG_COUNT prog
attempts

    // failed to program after MAX_PROG_COUNT attempts
    // disable flash writes
    FLASH_SWE = 0;

    // wait tCSWE
    prog_delay (ONE_HUNDRED_USEC);

    // end of failed programming
    return (PROG_FAIL);
}
#pragma section

```

### fprogram Linker Settings

Address	Section	Comment
0xFFE000	PINIT_PROG_DT	Initialisation code for TPU
0xFFE05A	PPROG_DT	TPU routines
0xFFE080	PPROG_FUNC	FLASH Programming routine

## Summary

It has been the aim of this application note to demonstrate via a simple example how it is possible to implement User Mode Flash Programming on a Renesas H8 microcontroller.

Even though the example was written for the H8S/2612, all of the code and concepts can be easily ported to other members of the H8 family.

Accompanying this application note there are 3 file downloads, each containing a HEW workspace. Please choose the correct download for the version of HEW you are using.

Please note that HEW workspaces are only upwardly compatible, i.e. a HEW 1.3 workspace can be opened (and updated) to a HEW 2 or HEW 3 workspace, but a HEW 3 workspace can not be opened in HEW 1.3 or HEW 2

## Website and Support

Renesas Technology Website  
<http://www.renesas.com/>

Inquiries  
<http://www.renesas.com/inquiry>  
[csc@renesas.com](mailto:csc@renesas.com)

All trademarks and registered trademarks are the property of their respective owners.

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (<http://www.renesas.com>)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
  - (1) artificial life support devices or systems
  - (2) surgical implantations
  - (3) healthcare intervention (e.g., excision, administration of medication, etc.)
  - (4) any other purposes that pose a direct threat to human life
 Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.

© 2008. Renesas Technology Corp., All rights reserved.