# HCS08 Peripheral Module Quick Reference

## A Compilation of Demonstration Software for HCS08 Modules

This collection of code examples, useful tips, and quick reference material has been created to help users speed the development of their applications. Each section within this document contains an example that may be modified to work with HCS08 MCU Family members. When you're developing your application, consult your device data sheet for part-specific information, such as which versions of the peripheral modules are on your device.

This book begins with a section about device initialization, and then explores the different peripheral modules found in the HCS08 Family of MCUs. It concludes with two sections on implementing interrupt subroutines and making memory usage assignments in an embedded C environment with CodeWarrior.

Each section of this users guide contains:

- Programmer's model register figure for quick reference
- Example code
- Supplemental information supporting the code

All code is available inside a CodeWarrior project, or from Freescale's Web site in HCS08QGUGSW.zip.

In-depth material about using the HCS08 modules is also available in Freescale's application notes. See the Freescale Web site: http://freescale.com

**Topic Reference**





freescale™ semiconductor

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://freescale.com/

## Revision History

| Date | Revision Level | Description | Page Number(s) |
|---|---|---|---|
| 11/2005 | 0 | Initial release | N/A |
| 2/2006 | 1 | Changing SCI1S1 line of code on page 65.<br>Replacement code page 83, 84, and 89. | 65, 83, 84, 89 |

**Freescale Semiconductor**
Users Guide

# Using the Device Initialization for the HCS08 Family Microcontrollers

By Gonzalo Delgado
RTAC Americas
México 2005

## 1 Overview

This document is a quick reference to the CodeWarrior Device Initialization tool for the HCS08 Family microcontrollers (MCUs). Basic information about the functional description and configuration are provided. The example may be modified to suit the specific needs for your application — refer to the data sheet for your device.

The Device Initialization (DI) tool is a user-friendly application integrated into the CodeWarrior version 5.0 that contains a powerful code generator used to create startup and initialization code that includes the configuration of registers to allow the use of specific modules in the MCU.

This time-saving application will help the user in the generation of code (relocatable ASM or C) to configure the registers of the MCU modules. With the DI, the user can migrate the initialization code from one family to another in an easier way.

This friendly graphical interface presents the MCU's pins, modules, and packages. When the user rolls the

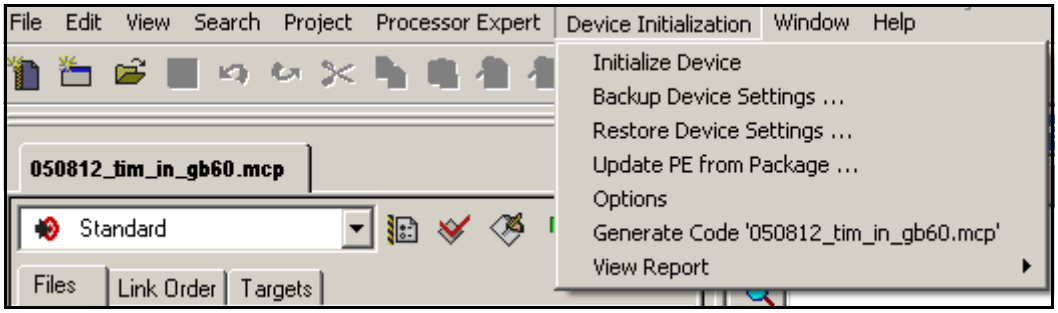
**Table of Contents**

mouse over the modules their pins are highlighted and a brief explanation of the device appears. Warnings appear when a value or configuration can't be defined. The DI has the ability to suggest or guide the user in the configuration of modules. There is a section of the registers concerned in each module and a brief description of each bit; these registers can be configured clicking bit by bit or with a predefined value.

The Device Initialization includes the following initialization modules, or beans[1]:

- Init_ACMP_HCS08
- Init_ADC_HC08
- Init_ADC_HCS08
- Init_AnalogModule_HC08
- Init_CMT_HCS08
- Init_FLASH_HCS08
- Init_IIC_HCS08
- Init_RTI_HCS08
- Init_SCI_HCS08
- Init_SPI_HCS08
- Init_TPM_HCS08

# 2 Device Initialization Main Menu (Integrated into CW Main Menu)

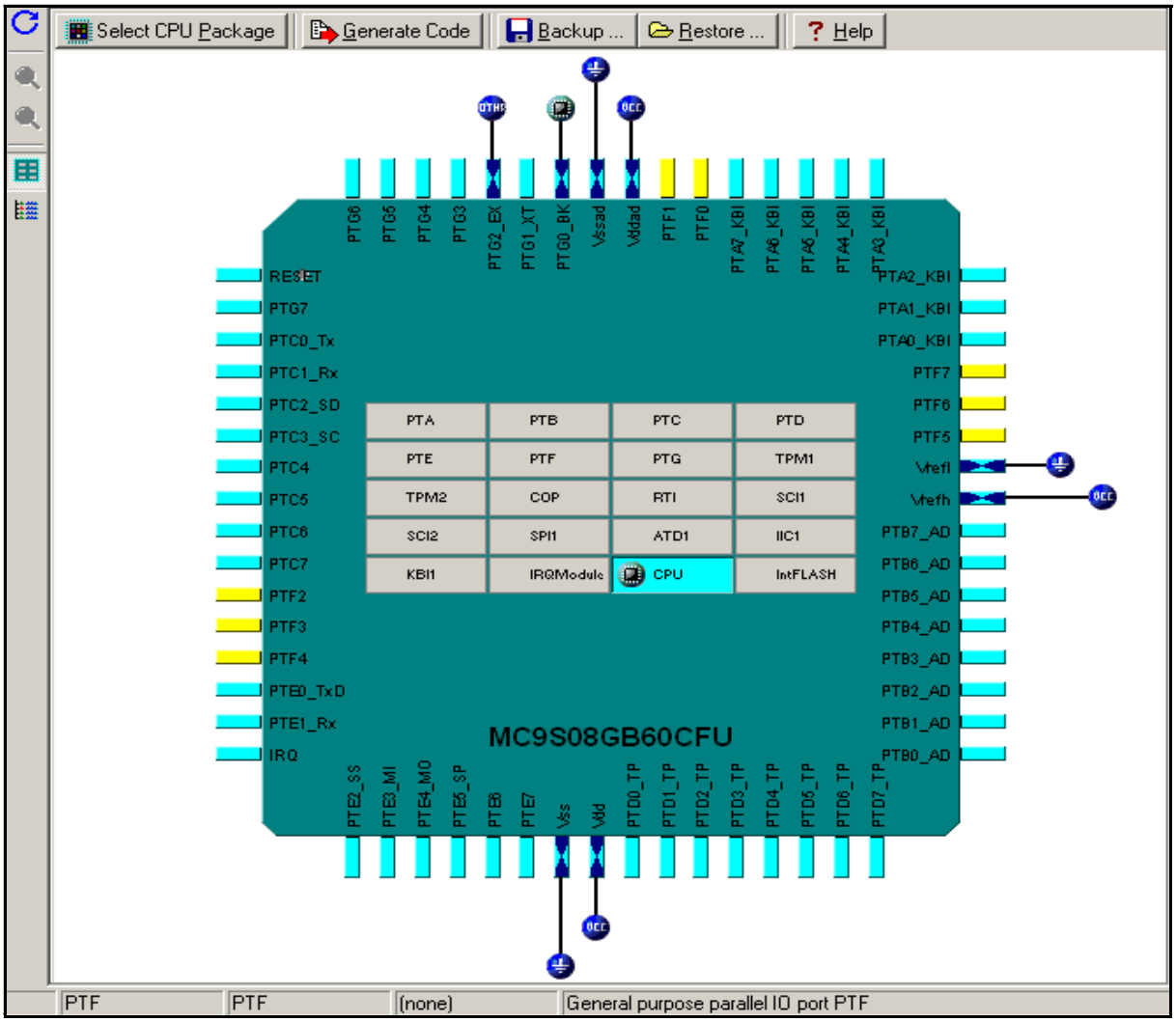



- Initialize Device – This command opens Target CPU window.
- Backup Device Settings – This command stores complete design into single configuration file. Directory and file name will be same as CW project. Previous version of the settings will be automatically stored in the same directory in the following way:
  - ProjectName.iPE — latest device settings
  - ProjectName_0.iPE — oldest device settings
  - ProjectName_1.iPE — next device settings
  - ProjectName_2.iPE — next device settings

1. Not all 8-bit microcontrollers have the modules described in the list.

  - ProjectName_nnn.iPE — previous device settings
- Restore Device Settings — This command restores complete design from single configuration file. Directory and file name will be selectable by the user. The user can use also settings from different project — see command Backup Device Settings.
- Update PE from Package — Allows installing a patch or updating from the .PEUpd file.
- Options — Defines the type of code that will be generated and options that will influence the code generation.
- Generate Code — Generates code (Relocatable ASM or C).
- View Report — Submenu:
  - Project Settings — Generates xml file with information about settings of all beans in the design.
  - Register Settings — Generates xml file with information about settings of all control registers modified by the design.
  - Interrupt Usage — Generates xml file with information about settings of all interrupt vectors used in the design.
  - Pin Usage — Generates xml file with information about settings of all pins used in the design.

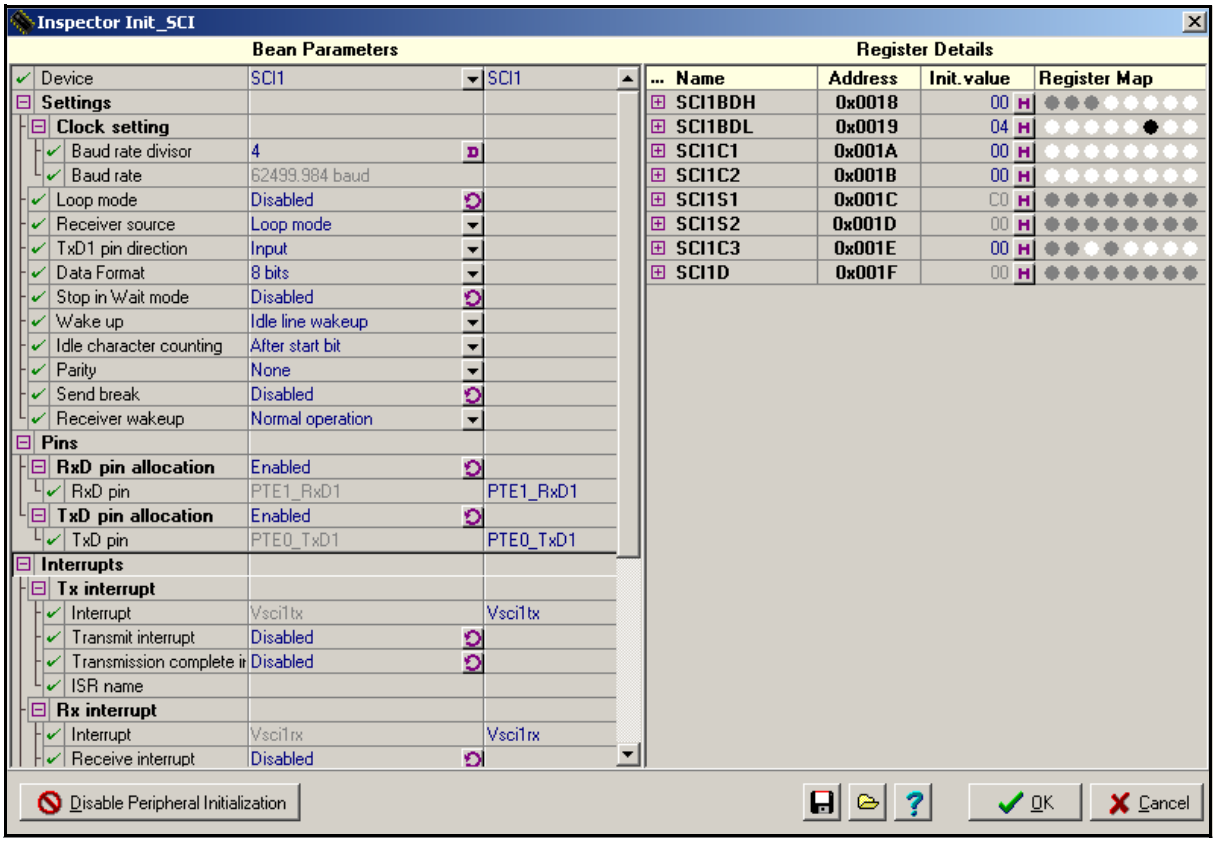
# 3 Target CPU Window




This is the main window where the MCU modules are listed among with their pins. By clicking into the module user can access the configuration menu.

- Unused peripherals are grayed; used ones are highlighted and embossed.
- Single click to init peripheral and open inspector dialog.
- Button for code generation (see top panel of the window).
- CPU peripherals list mode view, which contains all peripherals in the list.
- Closing the window suspends Processor Expert (PE). PE asks the user to save design if it is not saved.
- Closing CW project closes the window.
- Target CPU window will be opened automatically with CW project if there is saved Device Initialization design (and was not suspended).

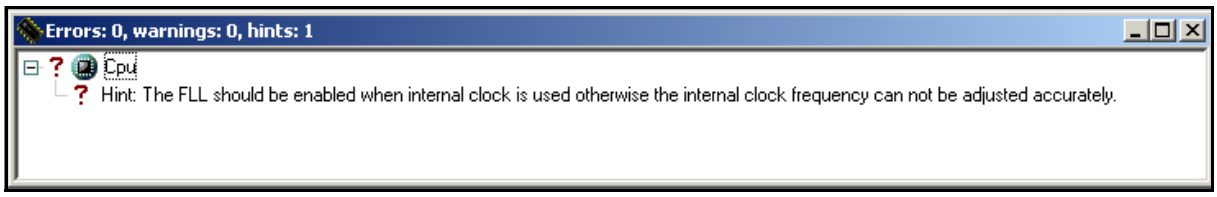
# 4 Inspector Dialog Window




This window shows all the options available for configuration with the selected module in different menus and submenus.

- Cancel restores original design settings (design state before opening the inspector)
- This windows contains corresponding values of control registers (see right side) — based on bean settings. It allows modification of control register values and corresponding bean settings are updated according to the value.

# 5 Error Window




Error window will be displayed only if an error occurs. After resolving errors the window hides automatically. An error is generated when the user misconfigures a module or parameters are missing.

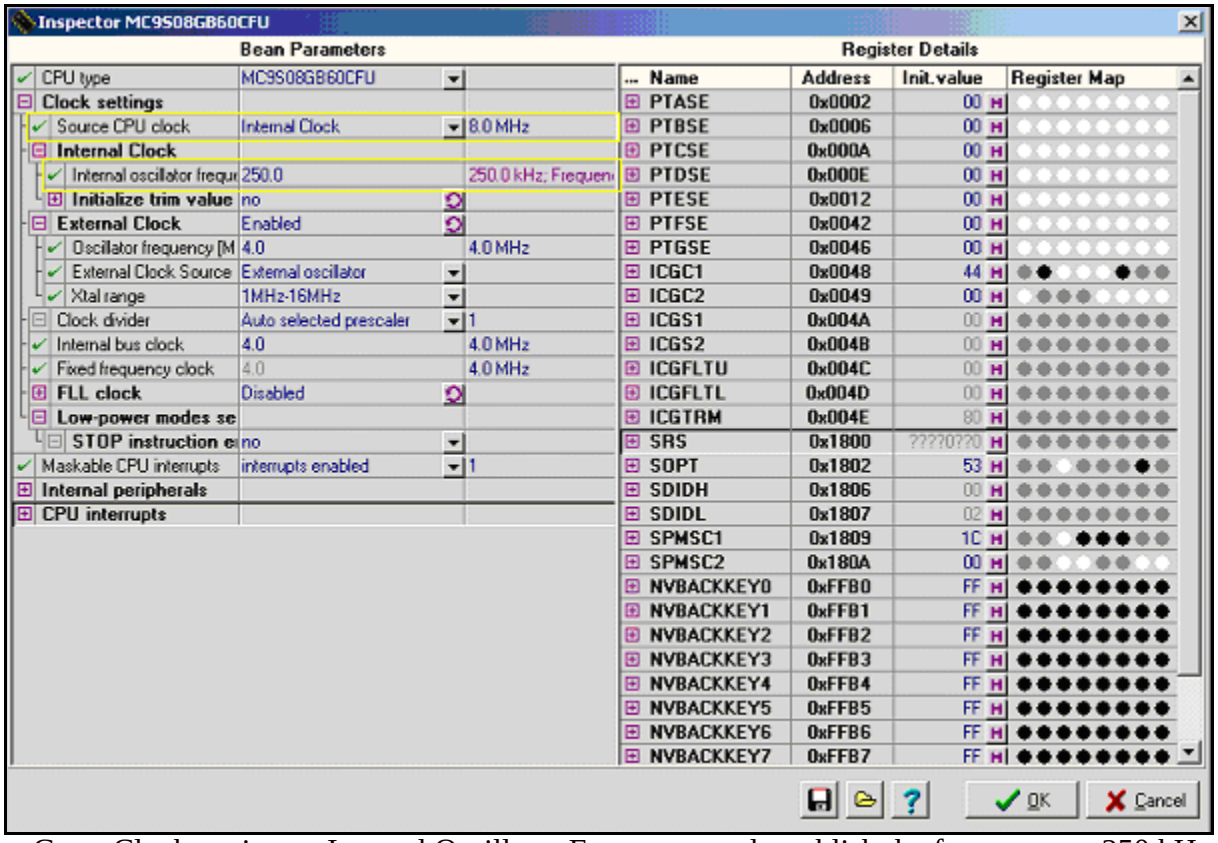
# 6 Description of Generated Files

- Include file (*.inc or *.h) - .*.h for C callable option. Note: Generated file name can be selected using option “Generated file”
- Implementation file (*.asm or *.c) – contains init function MCU_init that initializes selected peripherals, interrupt vector table and selected interrupt service routines.

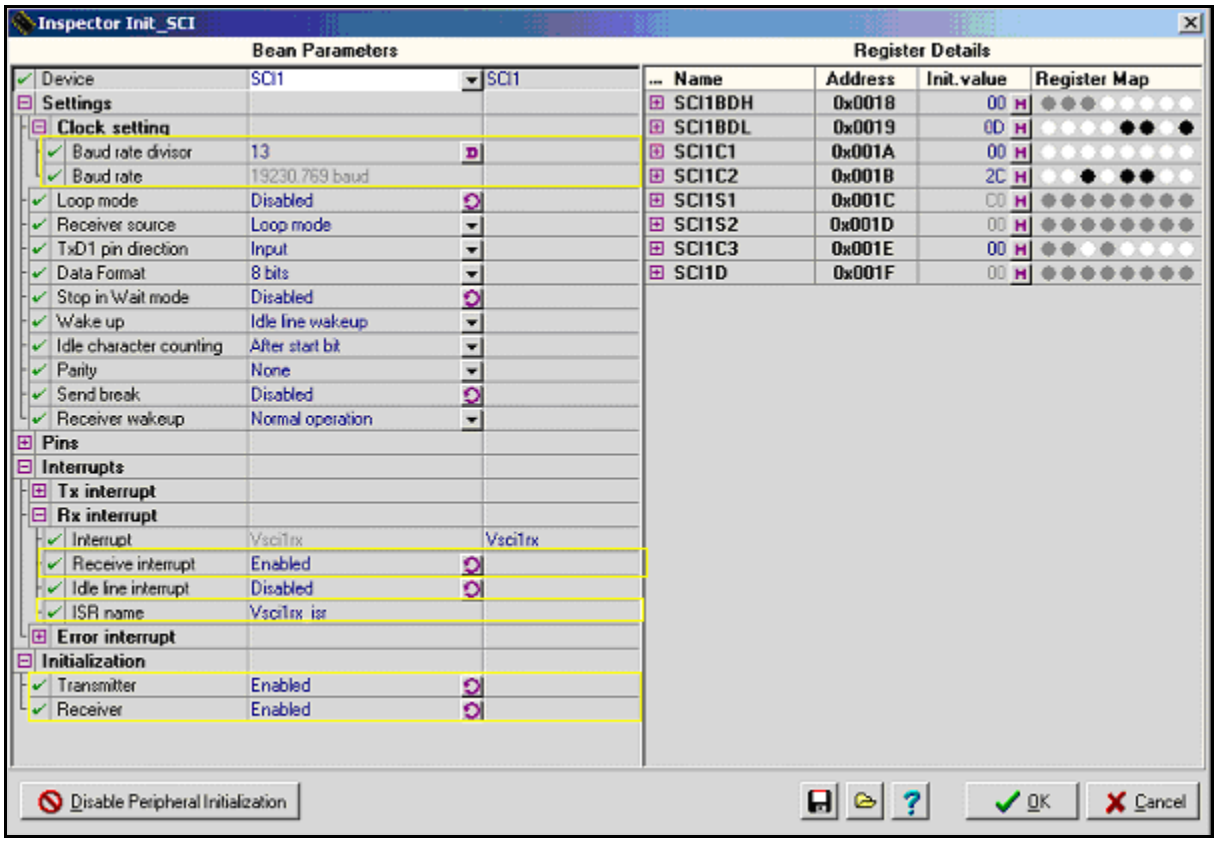
# 7 Example Code and Explanation

This example code shows initializing the SCI module to 19200 baud rate on the MC9S08GB60.

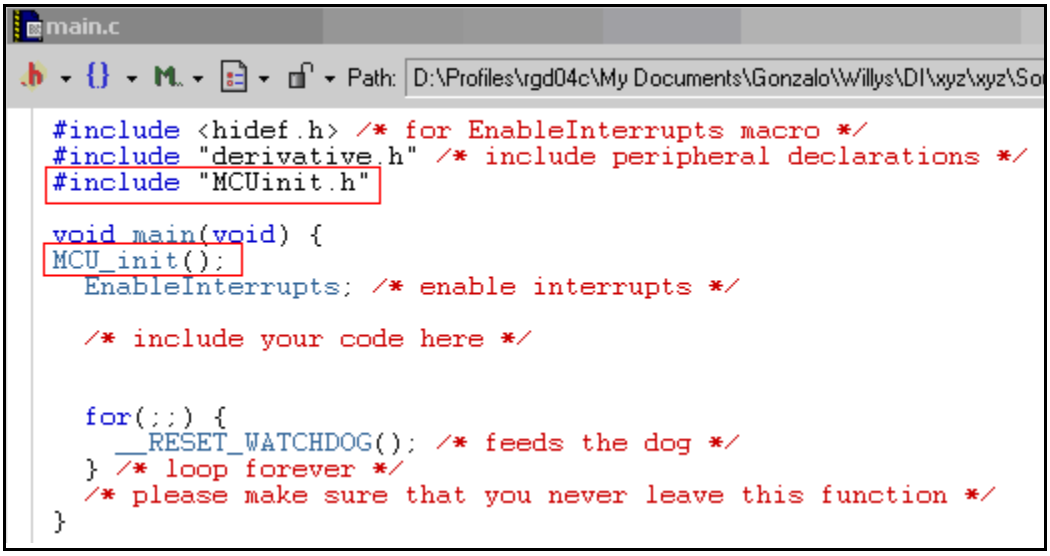
1. Open CodeWarrior version 5.0
2. Create a new project in C
3. Select MC9S08GB60 under the derivative list in the HCS08 derivative.
4. Under Rapid Application Development options, select Device Initialization
5. Select CPU package from the list
6. Click in the CPU module




1. Go to Clock settings…Internal Oscillator Frequency and establish the frequency to 250 kHz
2. Go to Clock settings…Source CPU Clock and select Internal Clock
3. Press OK
4. Click in the SCIx module (x stands for the number of the device)




1. Go to the section Settings…Clock Setting and change the baud rate divisor to 13. (This will lead to a result of a 19230.769 baud rate with a 0.16% of error)
2. Go to Interrupts…Rx Interrupt; enable Receive Interrupt and set a name to ISR for the Receive.
3. Go to Initialization and enable Transmitter and Receiver.
4. Press OK
5. Press Generate Code
6. Select the Generated File type, in this case C callable and Save and add files to project option.
7. Press Generate
8. This will generate two pieces of code, one has the method declaration and the other is the MCU_Init function where all the needed on-chip peripherals are initialized.
9. Include the MCUInit.h in the main file using the command:
   #include "MCUinit.h"
10. Call the MCU_Init (included in the MCUinit.c) function:
    MCU_init();




- Place your code into the main or into the interrupt function located in the MCUInit.c under the Generated Code directory.

**NOTE**

This example was developed using the CodeWarrior Development Studio for Freescale HC(S)08 version 5.0, and was expressly made and tested for the MC9S08GB60. Changes will be required before the code can be used to initialize another MCU. Every microcontroller requires an initialization code that depends on the application and the microcontroller itself.

# Using the Low Voltage Detect System for the HCS08 Family Microcontrollers

by: Andrés Barrilado González
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the low voltage detect (LVD) system on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



### LVD Quick Reference

The LVD function registers are device dependent. Please see the data sheet for your device to check availability / location for these bits. For example, on some devices, the low voltage warning bits are moved to another register (SPMSC3), and there is a PDF (power-down flag) in bit 4 of SPMSC2.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPMSC1 | LVDF | LVDACK | LVDIE | LVDRE | LVDSE | LVDE | | BGBE |

System power management status and control register 1

LVDF — flags low-voltage detections
LVDACK — clears the LVD flag
LVDIE —enables/disables LVD-caused interruptions
LVDRE — enables/disables LVD-caused resets
LVDSE — enables/disables the LVD in stop mode
LVDE — enables/disables the LVD module
BGBE — bandgap buffer enable (not available on all devices — check your data sheet)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPMSC2 | LVWF | LVWACK | LVDV | LVWV | PPDF | PPDACK | PDC | PPDC |

System power management status and control register 2

LVWF — flags low-voltage warnings
LVWACK — clears the low voltage warning flag
LVDV —selects between high or low low-voltage detect trip point voltage
LVWV — selects between high or low low-voltage warning trip point voltage
PPDF — partial power-down flag
PPDACK — partial power-down acknowledge
PDC — power-down control
PPDC — partial power-down control

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The following example configures the LVD using the interrupt-based approach to turn on an LED while voltage levels are low. It also polls the low-voltage warning flag to turn on a second LED in case the low-voltage level is approaching.

The zip contains the following functions:

- main — polls the low-voltage warning flag endlessly and moves the result to a MCU pin where an LED is attached
- MCU_init — Configures hardware and the LVD module to accept interrupts and sets the LVD/LVW trip voltages
- Vlvd_isr — Responds to LVD interruptions.

Using Device Initialization, the LVD configuration applied for this example is:

- LVD interrupt enabled
- High low-voltage detect trip voltage
- High low-voltage warning trip voltage
- No reset in case of low-voltage detection
- No low-voltage detection in STOP mode

Please refer to the code for specifics about the configuration.

After the LVD is configured, and if a low voltage level is detected, a service routine must clear the LVD flag by setting the acknowledge bit. In this example, a bit is also set at a MCU pin in order to turn on a warning LED.

```
__interrupt void Vlvd_isr(void){
PTFD_PTFD2 = 0x00;              /* Turn on PTF2 (and keep it on) */
SPMSC1 |= 0x40;                 /* Acknowledge LVD and clear the flag */
}
```

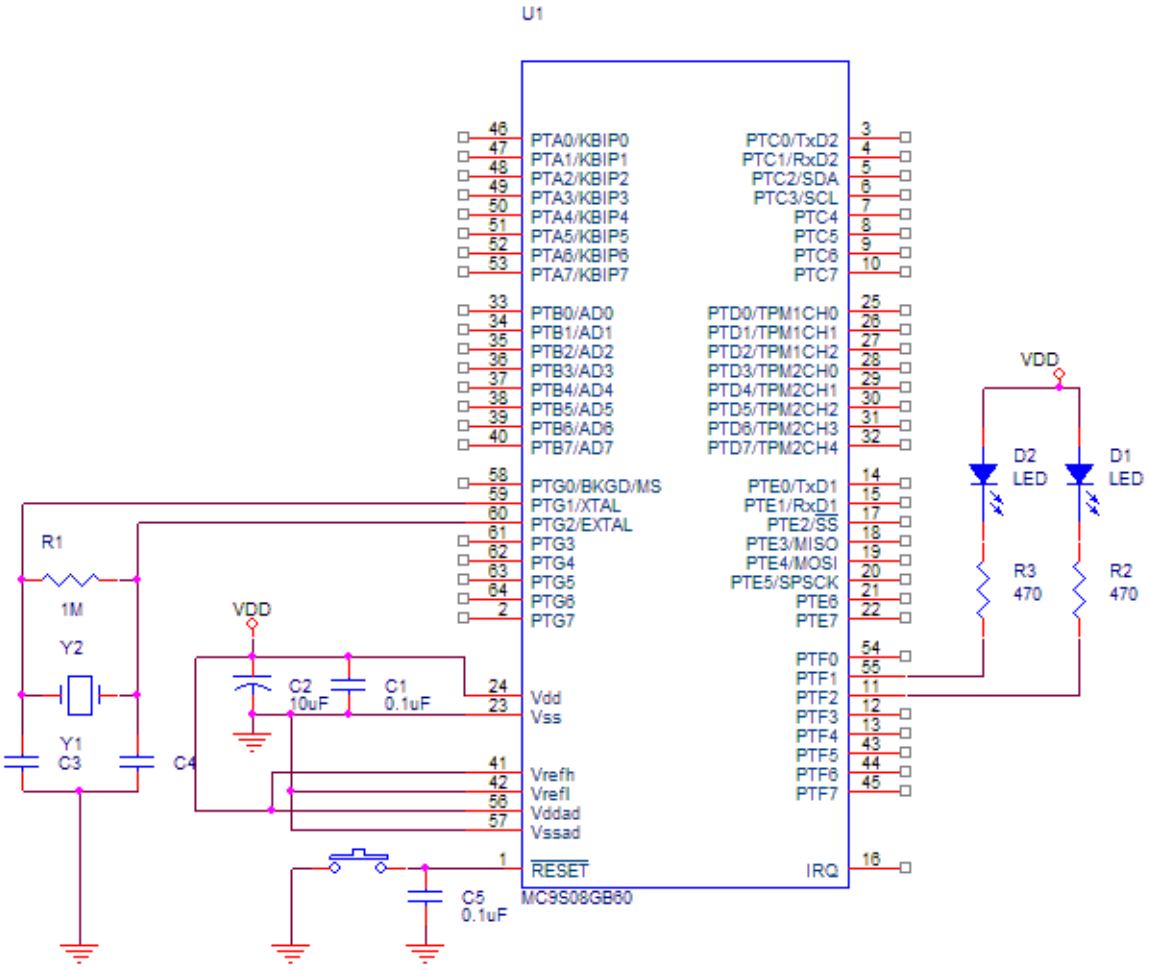
Please refer to the source code for more details.

**NOTE**

This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and tested using a MC9S08GB60 running in self-clocked mode. Coding changes may be needed to initialize another MCU. Every microcontroller needs an initialization code that depends on the application and the microcontroller itself.

# 3 Hardware Implementation

This schematic shows the hardware used to exercise the code provided.



**Freescale Semiconductor**
Users Guide

# Using the Internal Clock Source (ICS) for the HCS08 Microcontrollers

by: Sergio García de Alba Garcin
RTAC Americas
México 2005

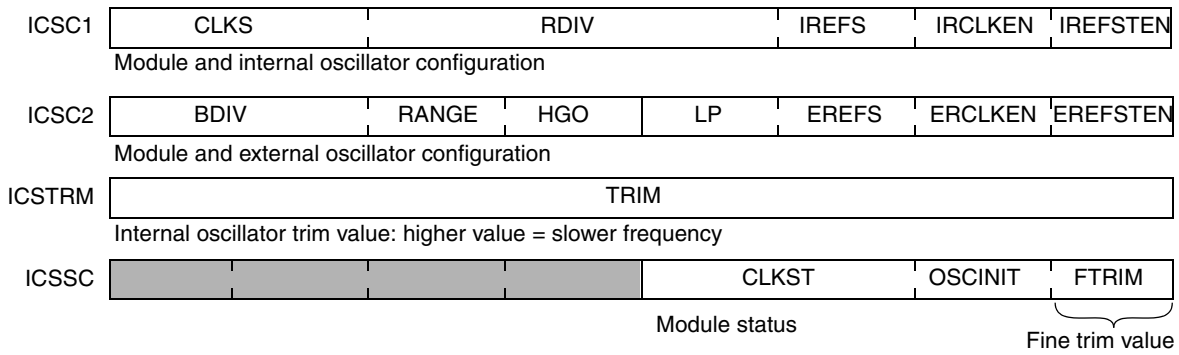
## 1 Overview

This is a quick reference for using the internal clock source (ICS) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**ICS Quick Reference**



# 2 Code Example and Explanation

The ICS provides several options for clock sources. This offers great flexibility when having to choose between precision, cost, current consumption, and performance. The weight of each one of these factors will depend on the requirements and characteristics of the application being developed.

## 2.1 FLL Engaged External Example

Our first example will be configuring the microcontroller for FLL engaged external (FEE) mode using a 4.9152 MHz crystal as an external clock reference. Using this mode we can have a bus frequency in the range of 1 MHz < $f_{bus}$ < 10 MHz, high clock accuracy, and medium/high cost (because a crystal, resonator, or external oscillator is required).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{ext} \div RDIV \times 512 \div BDIV) \div 2$

Where $f_{ext}$ is the frequency of the external reference (in this example we assume a 4.9152 MHz crystal is being used). RDIV bits must be programmed to divide $f_{ext}$ to be within the range of 31.25 kHz to 39.0625 kHz (in this example they divide $f_{ext}$ by 128). Then the FLL multiplies the frequency by 512, and BDIV bits divide it (in this example they are programmed to divide by 2). Finally, the clock signal is divided by 2 to give the bus clock.

In our example $f_{bus}$ will be: 4.9152 MHz. For this example HGO was programmed to configure the external oscillator for low power operation (reduced amplitude).

The ICS control registers will be programmed in the following way:

| **ICSC1 = 0x38** | | | |
|---|---|---|---|
| Bits 7:6 | CLKS | 00 | Output of FLL is selected |
| Bits 5:3 | RDIV | 111 | Divides reference clock by 128 |
| Bit 2 | IREFS | 0 | External reference clock selected |
| Bit 1 | IRCLKEN | 0 | ICSIRCLK inactive |
| Bit 0 | IREFSTEN | 0 | Internal reference clock disabled in stop |

| **ICSC2 = 0x64** | | | |
|---|---|---|---|
| Bits 7:6 | BDIV | 01 | Set to divide selected clock by 2 |
| Bit 5 | RANGE | 1 | High frequency range selected for the external oscillator |
| Bit 4 | HGO | 0 | Configures external oscillator for low power operation |
| Bit 3 | LP | 0 | FLL is not disabled in bypass mode |
| Bit 2 | EREFS | 1 | Oscillator requested |
| Bit 1 | ERCLKEN | 0 | ICSERCLK inactive |
| Bit 0 | EREFSTEN | 0 | External reference clock disabled in stop |

The following piece of code in C would set this configuration:

```
ICSC2= 0x64;
while(ICSSC_OSCINIT==0);
ICSC1= 0x38;            // Best practice is to enable external clock, then switch to FEE mode
```

**NOTE**

The while loop is used to wait for the initialization cycles of the external crystal to complete.

## 2.2 FLL Bypassed External Example

This time, we will configure the microcontroller to work in FLL bypassed external mode (FBE) using a 4.9152 MHz crystal as a reference. This mode allows for a bus frequency in the range 2 kHz < $f_{bus}$ < 2.5 MHz, very high clock accuracy, low power consumption, and medium/high cost (because a crystal, resonator, or external oscillator is required).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{ext} * 1/BDIV) / 2$

Where $f_{ext}$ is the frequency of the external reference (in this example we assume a 4.9152 MHz crystal is being used). RDIV bits must be programmed to divide $f_{ext}$ to be within the range of 31.25 kHz to 39.0625 kHz (in this example they divide $f_{ext}$ by 128).

In our example, $f_{bus}$ will be: 1.228 MHz. In this example we programmed HGO to configure the external oscillator for high gain to provide higher amplitude for improved noise immunity.

The ICS control registers will be programmed in the following way:

| **ICSC1 = 0xB8** | | | |
|---|---|---|---|
| Bits 7:6 | CLKS | 10 | External reference clock is selected |
| Bits 5:3 | RDIV | 111 | Divides reference clock by 128 |
| Bit 2 | IREFS | 0 | External reference clock selected |
| Bit 1 | IRCLKEN | 0 | ICSIRCLK inactive |
| Bit 0 | IREFSTEN | 0 | Internal reference clock disabled in stop |

| **ICSC2 = 0x74** | | | |
|---|---|---|---|
| Bits 7:6 | BDIV | 01 | Set to divide selected clock by 2 |
| Bit 5 | RANGE | 1 | High frequency range selected for the external oscillator |
| Bit 4 | HGO | 1 | Configures external oscillator for high gain operation |
| Bit 3 | LP | 0 | FLL is not disabled in bypass mode |
| Bit 2 | EREFS | 1 | Oscillator requested |
| Bit 1 | ERCLKEN | 0 | ICSERCLK inactive |
| Bit 0 | EREFSTEN | 0 | External reference clock disabled in stop |

The following piece of code in C would set this configuration:

```
ICSC2= 0x74;
while(ICSSC_OSCINIT==0);
ICSC1= 0xB8;          //Best practice is to enable external clock, then switch to FBE mode
```

**NOTE**

The while loop is used to wait for the initialization cycles of the external crystal to complete.

## 2.3 FLL Bypassed External Low Power Example

This mode is very similar to FLL bypassed external mode (FBE), with the difference that the FLL is turned off to reduce power consumption. For this example, we will also use a 4.9152 MHz crystal as a reference. This mode allows for a bus frequency $f_{bus}$ <= 10 MHz, very high clock accuracy, very low power consumption, and medium/high cost (because a crystal, resonator, or external oscillator is required).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{ext} * 1/BDIV) / 2$

Where $f_{ext}$ is the frequency of the external reference (in this example we assume a 4.9152 MHz crystal is being used). Although this FLL will be disabled in this example, it is best practice to set the RDIV bits to divide $f_{ext}$ to be within the range 31.25 kHz to 39.0625 kHz (in this example, $f_{ext}$ is divided by 128).

In our example, $f_{bus}$ will be: 2.457 MHz. For this example, HGO was programmed to configure the external oscillator for low power operation (reduced amplitude).

The ICS control registers will be programmed in the following way:

| **ICSC1 = 0x80** | | | |
|---|---|---|---|
| Bits 7:6 | CLKS | 10 | External reference clock is selected |
| Bits 5:3 | RDIV | 111 | Divides reference clock by 128 |
| Bit 2 | IREFS | 0 | External reference clock selected |
| Bit 1 | IRCLKEN | 0 | ICSIRCLK inactive |
| Bit 0 | IREFSTEN | 0 | Internal reference clock disabled in stop |

| **ICSC2 = 0x2C** | | | |
|---|---|---|---|
| Bits 7:6 | BDIV | 00 | Set to divide selected clock by 1 |
| Bit 5 | RANGE | 1 | High frequency range selected for the external oscillator |
| Bit 4 | HGO | 0 | Configures external oscillator for low power operation |
| Bit 3 | LP | 1 | FLL is disabled in bypass mode (unless BDM is active) |
| Bit 2 | EREFS | 1 | Oscillator requested |
| Bit 1 | ERCLKEN | 0 | ICSERCLK inactive |
| Bit 0 | EREFSTEN | 0 | External reference clock disabled in stop |

The following piece of code in C would set this configuration:

```
ICSC2= 0x2C;
while(ICSSC_OSCINIT==0);
ICSC1= 0x80;           // Best practice is to enable external clock then switch to FBELP mode
```

**NOTE**

The while loop is used to wait for the initialization cycles of the external crystal to complete.

## 2.4 FLL Bypassed Internal Example

In this example, the microcontroller will be configured to operate in FLL bypassed internal mode (FBI). This mode allows a bus frequency in the range 2 kHz < $f_{bus}$ < 19 kHz, low cost, and good accuracy (if trimmed).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{irc} * 1/BDIV) / 2$

Where $f_{irc}$ is the frequency of the internal reference clock (in this example we assume 32.768 kHz).

In our example $f_{bus}$ will be: 8.19 kHz.

The ICS control registers will be programmed in the following way:

| **ICSC1 = 0x44** | | | |
|---|---|---|---|
| Bits 7:6 | CLKS | 01 | Internal reference clock is selected |
| Bits 5:3 | RDIV | 000 | Divides reference clock by 1 |
| Bit 2 | IREFS | 1 | Internal reference clock selected |
| Bit 1 | IRCLKEN | 0 | ICSIRCLK inactive |
| Bit 0 | IREFSTEN | 0 | Internal reference clock disabled in stop |

| **ICSC2 = 0x40** | | | |
|---|---|---|---|
| Bits 7:6 | BDIV | 01 | Set to divide selected clock by 2 |
| Bit 5 | RANGE | 0 | Low frequency range for the external oscillator |
| Bit 4 | HGO | 0 | Configures external oscillator for low power operation |
| Bit 3 | LP | 0 | FLL is not disabled in bypass mode |
| Bit 2 | EREFS | 0 | External clock source requested |
| Bit 1 | ERCLKEN | 0 | ICSERCLK inactive |
| Bit 0 | EREFSTEN | 0 | External reference clock disabled in stop |

The following piece of code in C would set this configuration:

```
ICSC1= 0x44;| // If switching from FEE, FBE, or FBELP into FBI, delay for a time equal to tIRST
ICSC2= 0x40;
```

## 2.5 FLL Bypassed Internal Low Power Example

This mode is very similar to FLL bypassed internal mode (FBI), with the difference that the FLL is turned off to reduce power consumption. This mode allows for a bus frequency in the range 2 kHz < $f_{bus}$ < 19 kHz, low cost, very low power consumption, and good accuracy (if trimmed).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{irc} * 1/BDIV) / 2$

Where $f_{irc}$ is the frequency of the internal reference clock (in this example we assume 32.768 kHz).

In our example, $f_{bus}$ will be: 16.38 kHz.

The ICS control registers will be programmed in the following way:

| **ICSC1 = 0x44** | | | |
|---|---|---|---|
| Bits 7:6 | CLKS | 01 | Internal reference clock is selected |
| Bits 5:3 | RDIV | 000 | Divides reference clock by 1 |
| Bit 2 | IREFS | 1 | Internal reference clock selected |
| Bit 1 | IRCLKEN | 0 | ICSIRCLK inactive |
| Bit 0 | IREFSTEN | 0 | Internal reference clock disabled in stop |

| **ICSC2 = 0x08** | | | |
|---|---|---|---|
| Bits 7:6 | BDIV | 00 | Set to divide selected clock by 1 |
| Bit 5 | RANGE | 0 | Low frequency range for the external oscillator |
| Bit 4 | HGO | 0 | Configures external oscillator for low power operation |
| Bit 3 | LP | 1 | FLL is disabled in bypass mode (unless BDM is active) |
| Bit 2 | EREFS | 0 | External clock source requested |
| Bit 1 | ERCLKEN | 0 | ICSERCLK inactive |
| Bit 0 | EREFSTEN | 0 | External reference clock disabled in stop |

The following piece of code in C would set this configuration:

```
ICSC1= 0x44; //If switching from FEE, FBE, or FBELP into FBILP, delay for a time equal to tIRST
ICSC2= 0x08;
```

## 2.6 FLL Engaged Internal Example

In this example, we will use the microcontroller in FLL engaged internal mode (FEI), which is the default mode of operation for the ICS module. When this mode is entered out of reset the bus frequency will default to approximately 4.1943 MHz.

This mode allows for a bus frequency in the range 1 MHz < $f_{bus}$ < 10 MHz, low cost, quick and reliable system startup, and good accuracy (if trimmed).

In our example, $f_{bus}$ will be around 4.1943 MHz, which is the default frequency after reset. To operate in FLL engaged internal mode (FEI) no register needs to be written if the default settings are suitable. If required, the default configuration can be changed. For instance, the internal reference clock could be

trimmed writing the ICSTRM register or the bus frequency could be reduced by changing the BDIV bits in the ICSC2 register.

The internal reference must be trimmed to less than 39.0625 kHz before BDIV is set for divide by 1.

# 3 Tips and Recommendations

- When ICS is configured for FEE or FBE mode, input clock source must be divisible using RDIV to within the range of 31.25 kHz to 39.0625 kHz.
- Check the external and internal oscillator characteristics in the data sheet for electrical and timing specifications.
- The external oscillator can be configured to provide a higher amplitude output for noise immunity. This mode of operation is selected by HGO = 1.
- When switching modes of operation, if the newly selected clock is not available, the previous clock will remain selected.
- The TRIM and FTRIM value will not be affected by a reset.
- When using an XTAL (crystal) be sure to use high quality components (XTAL, resistors, and capacitors). Use low inductance resistors such as carbon composition resistors. Capacitors must be high quality ceramic capacitors specifically designed for high frequency applications. If using a resonator, be sure to use a high-quality resonator.
- For the values of the components used with the XTAL or resonator, consult the manufacturer or the device's data sheet (typical values are: C1 and C2 in the range of 5 pF to 25 pF, $R_F$ in the range 1–10 MΩ, $R_S$ in the range of 0–100 kΩ). Take into consideration stray capacitance when sizing C1 and C2.
- Good layout practices are fundamental for correct operation and reliability of the oscillator (crystal or resonator). Have the oscillator's components very close to the XTAL and EXTAL pins to minimize the length of the routing traces. Avoid high frequency/current signals near the oscillator to prevent crosstalk and to minimize noise, etc.
- Freescale recommends an evaluation of the application board and chosen resonator or crystal by the resonator or crystal manufacturer.

**NOTE**

- This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization.
- A small project is included that was tested in the MC9S08QG8 that configures the microcontroller as described in the previous examples depending on which define# is not commented. An LED blinks at a frequency which depends on the ICS mode of operation selected.

# Using the Internal Clock Generator (ICG) for the HCS08 Family Microcontrollers

by: Sergio García de Alba Garcin
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the internal clock generator (ICG) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**ICG Quick Reference**

ICGC1

| | RANGE | REFS | CLKS | OSCSTEN | | |
|---|---|---|---|---|---|---|

RANGE — FLL frequency range
REFS — reference clock select
CLKS — clock mode select
OSCSTEN — oscillator stop enable

ICGC2

| LOLRE | MFD | LOCRE | RFD |
|---|---|---|---|

LOLRE — loss of lock reset
MFD — multiplication factor
LOCRE — loss of clock reset
RFD — reduced frequency divider

ICGS1

| CLKST | REFST | LOLS | LOCK | LOCS | ERCS | ICGIF |
|---|---|---|---|---|---|---|

CLKST — module mode status
REFST — status of reference clock
LOLS — loss of lock status
LOCK — current lock status
LOCS — loss of clock status
ERCS — external reference clock
ICGIF — interrupt flag

ICGS2

| | | | | | | DCOS |
|---|---|---|---|---|---|---|

DCOS — DCO clock stable

ICGFLTU

| | | | | FLT |
|---|---|---|---|---|

ICGFLTL

| FLT |
|---|

FLT[11:0] — DCO frequency control

ICGTRM

| TRIM |
|---|

TRIM[7:0] — internal reference clock trim setting



freescale semiconductor

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The ICG provides several options for clock sources. This offers great flexibility when having to choose between precision, cost, current consumption, and performance. The weight of each one of these factors will depend on the requirements and characteristics of the application being developed.

## 2.1 FLL Engaged External Example

Our first example will be configuring the microcontroller for FLL engaged external (FEE) mode using a 32 kHz crystal as an external clock reference. Using this mode we can have a bus frequency in the range of 0.03 MHz < $f_{bus}$ < 20 MHz, good clock accuracy, and medium/high cost (because a crystal, resonator, or external clock is required).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{ext} \times P \times N \div R) \div 2$

Where $f_{ext}$ is the frequency of the external reference (in this example we assume a 32.768 kHz crystal is being used). P depends on the value of the RANGE bit, because we are using a crystal in the low-frequency range P = 64 (if RANGE = 1 then P = 1). N and R are the multiplication and division factors determined by bits MFD and RFD in ICGC2.

In our example, we will program N/R = 4, therefore $f_{bus}$ will be: 4.19 MHz. The ICG control registers will be programmed in the following way:

**Table 1. ICG Control Register Settings for FEE Mode**

| **ICGC1** | | | |
|---|---|---|---|
| Bit 7 | HGO* | 0 | Configures oscillator for low power operation |
| Bit 6 | RANGE | 0 | Configures oscillator for low frequency range (FLL prescale factor P = 64) |
| Bit 5 | REFS | 1 | Oscillator using crystal or resonator is requested |
| Bits 4:3 | CLKS | 11 | FLL Engaged External mode requested |
| Bit 2 | OSCSTEN | 0 | Oscillator disabled in STOP mode |
| Bit 1 | LOCD* | 0 | Loss of clock detection enabled |
| Bit 0 | | 0 | Unimplemented or reserved |

*Only available in MC9S08AW, for MC9S08GB/GT always write zero

| **ICGC2** | | | |
|---|---|---|---|
| Bit 7 | LOLRE | 0 | Generates an interrupt request on loss of lock |
| Bits 6:4 | MFD | 000 | Sets the MFD multiplication factor to 4 (N) |
| Bit 3 | LOCRE | 0 | Generates an interrupt request on loss of clock |
| Bits 2:0 | RFD | 000 | Sets the RFD division factor to 1 (R) |

The following piece of code in C would set this configuration:

```
ICGC2=0x00;
ICGC1=0x38;              //Best practice is to set MFD/RFD, then enable FEE
while (ICGS1_LOCK==0);
while (ICGS2_DCOS==0);   //Optional
```

**NOTE**

*The while loop is used to pause execution until the FLL has locked. For time critical tasks an additional while loop could be included to wait for DCOS = 1.*

## 2.2 FLL Engaged Internal Example

This time, we will configure the microcontroller to work in FLL engaged internal (FEI) mode. The reference used will be the internal 243 kHz reference clock. This mode allows for a bus frequency in the range 0.03 MHz < $f_{bus}$ < 20 MHz, medium clock accuracy (if IRG has been trimmed), and the lowest cost (because it requires no external components).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = ((f_{IRG} \div 7) \times P \times N/R) \div 2$

Where $f_{IRG}$ is the frequency of the internal reference generator (approximately 243 kHz). In this mode the FLL prescale factor P is always 64. N and R are the multiplication and division factors determined by bits MFD and RFD in register ICGC2.

We will program N/R = 2, therefore $f_{bus}$ will be: 2.221 MHz.

The ICG control registers will be programmed in the following way:

**Table 2. ICG Control Register Settings for FEI Mode**

| **ICGC1** | | | |
|---|---|---|---|
| Bit 7 | HGO* | 0 | Configures oscillator for low power operation; (don't care) |
| Bit 6 | RANGE | 0 | Configures oscillator for low frequency range; (don't care) |
| Bit 5 | REFS | 1 | Oscillator using crystal or resonator is requested; (don't care) |
| Bits 4:3 | CLKS | 01 | FLL engaged internal mode requested |
| Bit 2 | OSCSTEN | 0 | Oscillator disabled in stop mode |
| Bit 1 | LOCD* | 0 | Loss of clock detection enabled |
| Bit 0 | | 0 | Unimplemented or reserved |

*Only available in some MCUs, for other devices, always write zero (see the data sheet for your device)

| **ICGC2** | | | |
|---|---|---|---|
| Bit 7 | LOLRE | 0 | Generates an interrupt request on loss of lock |
| Bits 6:4 | MFD | 000 | Sets the MFD multiplication factor to 4 (N) |
| Bit 3 | LOCRE | 0 | Generates an interrupt request on loss of clock |
| Bits 2:0 | RFD | 001 | Sets the RFD division factor to 2 (R) |

The following piece of code in C would set this configuration:

```
ICGC2=0x01;
ICGC1=0x28;              //Best practice is to set MFD/RFD, then enable FEI
while (ICGS1_LOCK==0);
while (ICGS2_DCOS==0);   //Optional
```

**NOTE**

The while loop is used to pause execution until the FLL has locked. For time critical tasks an additional while loop could be included to wait for DCOS = 1.

## 2.3 FLL Bypassed External Example

Now we will configure the microcontroller to work in FLL bypassed external (FBE) mode using a 32 kHz crystal as a reference. This mode allows for a bus frequency <= 8 MHz (up to 20 MHz if using external oscillator), highest clock accuracy, lowest power consumption, and medium/high cost (because crystal, resonator, or external clock is required).

The bus frequency that will be generated is calculated with the following formula:

$f_{bus} = (f_{ext} \times 1/R) \div 2$

Where $f_{ext}$ is the frequency of the external reference (in this example we assume a 32,768 kHz crystal is being used).

In our example $f_{bus}$ will be: 16.384 kHz.

The ICG control registers will be programmed in the following way:

**Table 3. ICG Control Register Settings for FBE Mode**

| **ICGC1** | | | |
|---|---|---|---|
| Bit 7 | HGO* | 0 | Configures oscillator for low power operation |
| Bit 6 | RANGE | 0 | Configures oscillator for low frequency range (FLL prescale factor P = 64) |
| Bit 5 | REFS | 1 | Oscillator using crystal or resonator is requested |
| Bits 4:3 | CLKS | 10 | FLL Bypass External mode requested |
| Bit 2 | OSCSTEN | 0 | Oscillator disabled in STOP mode |
| Bit 1 | LOCD* | 0 | Loss of clock detection enabled |
| Bit 0 | | 0 | Unimplemented or reserved |

*Only available in some MCUs; for others, always write zero (refer to the data sheet for your device).

| **ICGC2** | | | |
|---|---|---|---|
| Bit 7 | LOLRE | 0 | Generates an interrupt request on loss of lock (don't care) |
| Bits 6:4 | MFD | 000 | Sets the MFD multiplication factor to 4 (N); (don't cares) |
| Bit 3 | LOCRE | 0 | Generates an interrupt request on loss of clock |
| Bits 2:0 | RFD | 000 | Sets the RFD division factor to 1 (R) |

The following piece of code in C would set this configuration:

```
ICGC2=0x00;
ICGC1=0x30;
while (ICGS1_ERCS==0);
```

**NOTE**

The while loop is used to pause execution until the external reference clock is stable and meets the minimum frequency requirement.

## 2.4 Self-Clocked Mode Example

In this example we will use the microcontroller in self-clocked mode (SCM). This is the default mode of operation for the ICG module. When this mode is entered out of reset, the bus frequency will default to approximately 4 MHz.

This is the only mode in which the filter registers (ICGFLT) can be written. The default value of the ICGFLT registers is 0x0C0. Writing a higher value will increase the bus frequency, while a lower value will decrease the bus frequency.

This mode allows for a bus frequency in the range 3 MHz < $f_{bus}$ < 20 MHz (via filter bits), quick and reliable system startup, and poor accuracy.

In our example $f_{bus}$ will be around 20 MHz.

To operate in SCM no register needs to be written, however in this example we will write ICGFLTU and ICGFLTL to increase the bus frequency (by writing ICGFLT we modify ICGFLTL and the four least significant bits of ICGFLTU. The other four bits are unimplemented).

The following piece of code in C would modify the FLT registers:

```
ICGFLT=0x0800;
```

The bus frequency could be reduced by changing the RFD division factor in the ICGC2 register.

# 3 Tips and Recommendations

- Be careful when writing to the ICGC1 register because bits RANGE and REFS are write-once after reset. Also, if the first write after reset sets CLKS = 0x (SCM, FEI) the CLKS bits cannot be written to 1x (FEE, FBE) until after the next reset (because the EXTAL pin was not reserved).
- For minimum power consumption and minimum jitter, choose N and R to be as small as possible when operating in FEE or FEI modes.
- When operating in FEE mode and using a crystal or resonator, make sure its frequency is in the specified range of 32 kHz – 100 kHz for RANGE = 0, or 2 MHz – 10 MHz for RANGE = 1.
- When operating in FBE mode and using a crystal or resonator, make sure its frequency is in the specified range of 32 kHz – 100 kHz for RANGE = 0, or 1 MHz – 16 MHz for RANGE = 1.
- The oscillator can be configured to provide a higher amplitude output for noise immunity. This mode of operation is selected by HGO = 1 (only available on some MCUs — see the data sheet for your device).

- To avoid long oscillator startup times when exiting stop mode, you can program OSCSTEN = 1, this way the oscillator will remain enabled in stop mode (ICG in off mode). The disadvantage is higher current consumption in STOP mode.
- When operating in FEI, trim the internal reference generator. Increasing the value in the ICGTRM register will increase the period and decreasing the value will decrease the period. For a detailed explanation of the trim procedure, please refer to application note AN2496.
- Two very useful bits are LOLRE and LOCRE. They configure whether a reset or an interrupt will be generated in the events of a loss of lock (LOLRE) and of a loss of clock (LOCRE).
- When using an XTAL (crystal) be sure to use high-quality components (XTAL, resistors, and capacitors). Use low inductance resistors such as carbon-composition resistors. Capacitors should be high-quality ceramic capacitors specifically designed for high-frequency applications. If using a resonator, be sure to use a high quality resonator.
- For the values of the components used with the XTAL or resonator, consult the manufacturer or the device's data sheet (typical values are: C1 and C2 in the range of 5 pF to 25 pF, $R_F$ in the range 1 – 10 MΩ, $R_S$ in the range of 0 – 10 kΩ). Take into consideration stray capacitance when sizing C1 and C2.
- Good layout practices are fundamental for correct operation and reliability of the oscillator (crystal or resonator). Try to have the oscillator's components very near to the XTAL and EXTAL pins to minimize the length of the routing traces. Avoid high-frequency/current signals near the oscillator to prevent crosstalk and to minimize noise, etc.
- Freescale recommends an evaluation of the application board and chosen resonator or crystal by the resonator or crystal manufacturer.
- We recommend writing to ICGC2 before ICGC1. This sets the multiplier before enabling the FLL.

**NOTE**

This code was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization.

**Freescale Semiconductor**
Users Guide

# Programming the Low-Power Modes on HCS08 Microcontrollers

by: Gabriel Sanchez Barba
Gonzalo Delgado
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the low-power modes on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

Freescale's HCS08 microcontrollers include several stop modes that permit the user to achieve low power consumption. This provides great flexibility and may be used to provide ideal conditions for many different types of applications. The HCS08 MCUs support three[1] different stop modes that may be entered when a stop instruction is executed if the STOPE bit in the system option register is set. If the STOPE bit is not set, then an illegal opcode reset will be forced.

1. Not all stop modes are available on all devices. Refer to the data sheet for your device.

**Table of Contents**



**Low Power Modes Quick Reference**

The stop modes function uses device dependent registers. Please see the data sheet for your device to check availability / location for these bits. For example, on some devices, the low voltage warning bits are moved to another register (SPMSC3), and there is a PDF (power-down flag) in bit 4 of SPMSC2.

| SOPT | COPE | COPT | STOPE | | | | | |
|---|---|---|---|---|---|---|---|---|

STOPE — enables the stop modes

| SPMSC2 | LVWF | LVWACK | LVDV | LVWV | PPDF | PPDACK | PDC | PPDC |
|---|---|---|---|---|---|---|---|---|

System power management status and control register 2

LVWF — flags low-voltage warnings
LVWACK — clears the low voltage warning flag
LVDV —selects between high or low low-voltage detect trip point voltage
LVWV — selects between high or low low-voltage

PPDF — partial power-down flag
PPDACK — partial power-down acknowledge
PDC — power-down control
PPDC — partial power-down control

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

It contains the following functions:

- main — Checks for a stop2 recovery, and if so, enters stop1, otherwise, it enters stop3 and waits for an external interrupt. After it receives an external interrupt, it services the interrupt routine, and returns to the main loop where it enters stop2 and waits for another external interrupt. When this interrupt occurs, the MCU will go through reset and see that it came out of stop2 and so configures and enters itself into stop1.
- MCU_init — Configures MCU hardware, as well as the external interrupt.
- Delay — This is a simple loop routine.
- Virq_isr — Services the IRQ pin interrupt.

Following these four simple steps, the user can enter any of the stop modes available on the device:

1. Set the STOPE bit in SOPT to enable stop modes:
   This will enable stop instructions, otherwise, an illegal opcode reset will be forced.
2. Set up the SPMSC1 register:
   This register sets up low-voltage detect. Low-voltage detect must be disabled to be able to enter stop2 and stop1.
3. Set up all interrupts that will exit the MCU from stop mode.
   This is needed so that the MCU may successfully exit stop modes by other means than just reset.
4. Check and set up SPMSC2 register.
   There are two main purposes to this: (a) check partial power down flag, and (b) set up stop mode to be used.

After these steps have been done, you may enter the stop modes in the MCU by executing a stop instruction.

## 2.1 Important I/O Configuration Information

When the HCS08 goes into stop2 or stop3 mode, the content of its registers remain unchanged. In particular, the ports keep their configuration. So, it is important to set the ports in a state that may not lead to a current consumption increase at the application level. Software and hardware engineers should follow these guidelines in order to avoid additional current consumption:

- Do not leave any I/O configured as an unconnected inputs — instead, tie them to $V_{DD}$ or $V_{SS}$. Or, you can set unconnected I/O as output, thus forcing a steady level.
- The same recommendation applies to unbonded I/O on small packages (on the package QFP44 vs. LQFP64 for instance). In this case, set the unbonded I/O as output.
- For inputs whose logic state is uncertain (for a Hall-effect sensor signal, for instance), use external pullup or pulldown resistors instead of the internal ones that are weak (typically between 20 kΩ and 50 kΩ). This way, the power consumption is minimized in case the level of these inputs changes during the low-power mode.

# 3 In-Depth Reference Material

The information in this section is provided as reference material for those who would like to learn more about the stop modes in the HCS08 Family of MCUs.

## 3.1 Stop3 Overview (PDC = 0, PPDC = 1 or 0)

This is the same stop mode the 68HC08 Family of MCUs uses. The states of all of the internal registers and logic are maintained. Because of this, I/O conditions are not affected by stop3 and do not have to be re-initialized after exit. RAM is maintained. All peripherals are disabled with the exception of the RTI (if enabled). Stop3 can be exited using an external interrupt (IRQ), real-time interrupt (RTI), keyboard interrupt (KBI), or a low-voltage warning (LVW) interrupt if enabled. RTI can use either its 1-kHz internal clock or the external oscillator if it is enabled during stop. When waking from stop3 via an asynchronous interrupt or the real-time interrupt, the MCU re-enters the program flow through the interrupt service routine (ISR) and executes the next instruction after the stop. If stop3 is exited by means of the $\overline{\text{RESET}}$ pin, the MCU will be reset and operation will resume after taking the reset vector.

## 3.2 Stop2 Overview (PDC = 1, PPDC = 1)

This is a lower power mode than stop3. Stop2 can be entered only if the low-voltage detection is disabled. I/Os are latched at the pin, but the states of the internal registers are lost during stop2. If the application requires the I/O pin conditions to be maintained, the contents of the appropriate registers should be saved to RAM. RAM is maintained. All peripherals are disabled with the exception of the RTI. If using the RTI, only the internal time base can be used because the internal oscillator circuitry is disabled in stop2. Stop2 can be exited using an external interrupt (IRQ), a real-time interrupt (RTI), or a keyboard interrupt [1](KBI). When waking from stop2, no ISR code is processed because the MCU re-enters the program flow through the reset vector. The user must determine whether this is a stop2 event or a true power-on reset (POR) and take appropriate action. All internal registers are set to their POR states. If the I/O pin conditions are to be maintained, the appropriate registers can be restored from RAM before acknowledging the stop2 condition.

## 3.3 Stop1 Overview (PDC = 1, PPDC = 0)

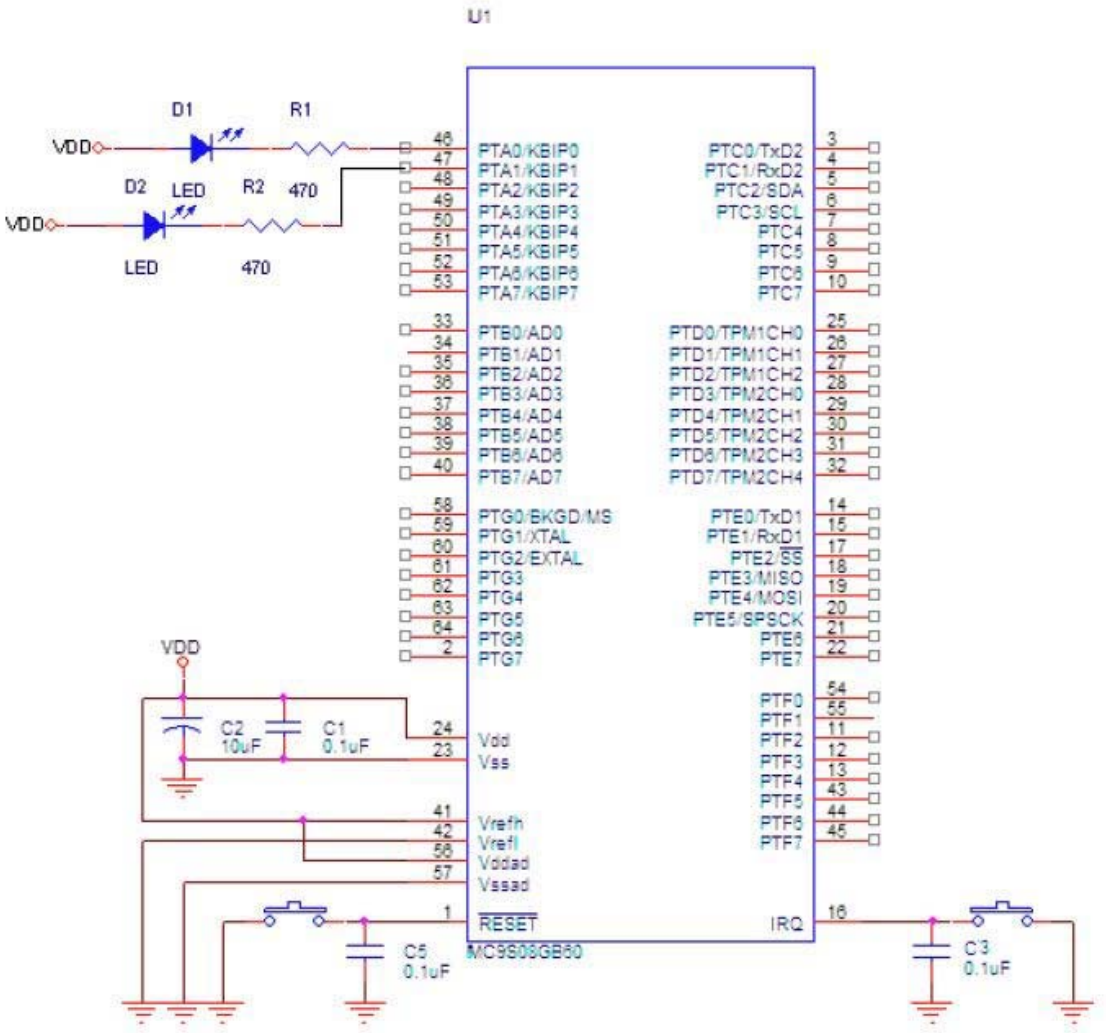
HCS08 devices that are designed for low voltage operation (1.8 V to 3.6 V) also include stop1 mode. The stop mode to be entered is selected by setting the appropriate bits of the SPMSC2 register.

This is the lowest power mode. Basically the device is switched off and can only be exited through $\overline{\text{RESET}}$ or IRQ if enabled. Stop1, just as stop2, can be entered only if the low-voltage detection is disabled. IRQ will be active-low in stop1 regardless of how it was configured before entering stop1. When you wake from stop1, you re-enter the program flow through the reset vector. No ISR code is processed.

1. Not all devices support exiting stop2 through the KBI — refer to the data sheet for your device.

# 4 Hardware Implementation

The schematic below shows the hardware used to exercise the code provided.




**NOTE**

This example was developed using the CodeWarrior IDE version 5.0 for HC(S)08, and was expressly made for the MC9S08GB60. Changes to the code may be required before using it to initialize other MCUs. Every microcontroller needs an initialization code that depends on the application and the microcontroller itself.

# Using the External Interrupt Request Function (IRQ) for the HCS08 Family Microcontrollers

by: Laura Delgado
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the external interrupt request (IRQ) function on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. This example may be modified to suit your application — refer to the data sheet for your device.

Note that on some devices, the IRQ signal is active low ($\overline{IRQ}$), while on others, the polarity is selected with

**Table of Contents**



**IRQ Function Quick Reference**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IRQSC | | IRQPDD | IRQEDG | IRQPE | IRQF | IRQACK | IRQIE | IRQMOD |

Interrupt request status and control

IRQPDD — disables the internal pullup device when the IRQ pin is enabled (IRQPE = 1), allowing an external device to be used (not available on all devices — check your data sheet)

IRQEDG — selects the polarity of the edges or levels that will be monitored in the IRQ pin (i.e., rising edge or falling edge) (not available on all devices — check your data sheet)

IRQPE — enables the IRQ pin to be used as an interrupt request; basically, it enables the whole IRQ function

IRQF — flags an edge- or level-event in the IRQ pin

IRQACK — allows device to acknowledge IRQ interrupt requests

IRQIE — determines whether the IRQ events will trigger hardware interruptions; if not enabled, the IRQ flag (IRQF) can still be used for software polling

IRQMOD — selects the kind of event that will be detected in the IRQ pin (i.e., edge or edge-and-level events)

IRQEDG., so the IRQ pin name does not have an overbar.

When the IRQ function is enabled, the IRQ pin is monitored for an event to trigger an interruption. Some microcontrollers in the HCS08 Family have a pin assigned specifically for this function. The IRQ interrupt can be programmed to detect either edge-only or edge-and-level events, as well as the polarity of such events.

The configuration register for the IRQ function is the interrupt pin request status and control register (IRQSC).

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The project IRQ.mcp implements the IRQ function, selecting a falling-edge and low-level event as desired to trigger a hardware interrupt. The functions for this example code are:

- main — Endless loop waiting for the IRQ interrupt to occur.
- MCU_init — Initializes MCU in the IRQ function
- IRQIsr — Toggles the LED when an external interrupt request is made

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by device initialization

In this application, the IRQ function will be exemplified by turning an LED on and off because of an IRQ hardware interrupt triggered by the IRQ pin.

This is the initialization code for the external interrupt IRQ using the MC9S08GB60 microcontroller. During the initialization phase, the interrupts are masked because it takes time for the internal pullup (typically 26 kΩ) to reach a logic 1. After the false interrupts are cleared, the IRQ interrupt is unmasked.

```
  IRQSC &= (unsigned char)~0x02;  /* Disable IRQ Interrupt to avoid
                                     false interrupt requests */
  IRQSC |= (unsigned char)0x11;   /* Enables the IRQ function */
  IRQSC |= (unsigned char)0x04;   /* clears flag */
  IRQSC |= (unsigned char)0x02;   /* enable IRQ interrupt */
```

After this, the IRQ is initialized and the program is ready for any external interrupt request (from the IRQ pin). Whenever one occurs, the IRQ interrupt is serviced. This interrupt routine acknowledges the interrupt, and then changes the logic state of an LED that will be fed from PTF3 output pin. PTF3 will blink the LED on and off every time an IRQ pin event is detected.

```
interrupt void IRQIsr (void) {

     IRQSC_IRQACK = 1;  /* Acknowledges flags */
     PTFD_PTFD3 = ~PTFD_PTFD3; /* Toggle LED */

}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the device initialization tool if the option is enabled. The user must define its contents.

# 3 Hardware Implementation

For this example, only four pins of the MCU are used, which makes hardware implementation fairly simple. These pins are:

- Supply voltage pin
- Ground reference pin
- IRQ pin as interrupt input
- I/O pin as output; an LED is used as visual display of the IRQ interrupt routine serviced

After the IRQ pin and interruption is enabled, the IRQ pin will be prepared to receive and detect desired events. Some MCUs share the IRQ pin with other functions. As soon as the IRQ pin is enabled in IRQSC, the pin will be used exclusively for the IRQ function. Depending on the sort of event to be detected (falling/rising edge, falling/rising edge-and-level), an optional pulldown/pullup resistor is available (i.e., if the IRQ pin is configured to detect rising edges, the pulldown resistor will be available rather than a pullup resistor – these variables are defined by the configuration bits IRQEDG and IRQMOD in IRQSC).

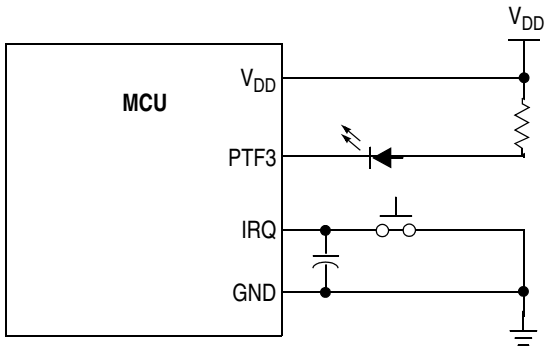



**Figure 1. Four Pins of the MCU Needed for IRQ**

In the code presented before, the IRQ pin is configured to have a pullup resistor because it will be detecting falling-edge and low-level events. The internal pull-up resistor sets a logic 1 as the default state on the port. According to the IRQSC configuration stated in the code before, whenever the button is pressed, the pin on IRQ will read a logic 0 and trigger a hardware interrupt. Pin PTF3 is set as an output and will turn on and off the LED with inverse logic. This means that the LED will turn on with a logic 0 on PTA1 and it will turn off with a logic 1.

**NOTE**

- The software shown here was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and was expressly made for the MC9S08GB60. Changes may be required before the code can be used on another MCU.
- A delay (20 ms typical) within the software is needed to take into consideration the mechanical stabilization time of the push button. An

alternative choice is to use a debounce circuit in the IRQ input as shown in Figure 1.

- The IRQ pin does not have a clamp diode to $V_{DD}$. IRQ should not be driven above $V_{DD}$.

**Freescale Semiconductor**
Users Guide

# Using the Keyboard Interrupt (KBI) for the HCS08 Family Microcontrollers

by: Laura Delgado
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the keyboard interrupt (KBI) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**KBI Quick Reference**

Because there is more than one KBI module on some devices, there may be more than one full set of registers on your device. In the register name below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the register that is on KBI1 from that on KBI2.

KBIxSC

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | KBF | KBACK | KBIE | KBMOD |

Module configuration
KBF — set when event occurs
KBACK — clears KBF
KBIE — interrupt enable
KBMOD — mode select

KBIxPE

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| KBIPE7 | KBIPE6 | KBIPE5 | KBIPE4 | KBIPE3 | KBIPE2 | KBIPE1 | KBIPE0 |

KBI Pin Enable
KBIPE[7:0] — enables and disables each port pin to operate as a keyboard interrupt pin

KBIxES

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| KBEDG7 | KBEDG6 | KBEDG5 | KBEDG4 | KBEDG3 | KBEDG2 | KBEDG1 | KBEDG0 |

KBI Pin Enable
KBEDG[7:0] — determines the polarity of the edge that will be recognized as a trigger event for the corresponding pin
(KBEDG[3:0] not available on all devices — check your data sheet. On some devices, KBEDG[7:4] are located in the KBIxSC register.)

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

In this application, one of the KBI pins will be used to trigger an interruption routine that toggles an LED 10 times, every time a keyboard event is detected. The MCU will be programmed to:

- Have the KBI pin 7 as the interrupt trigger
- Detect falling edges on the selected pin, as well as a following low level
- Generate a hardware interrupt where the LED toggle routine will be serviced

The functions for project “KBI.mcp” are:

- main — Endless loop waiting for a KBI interrupt.
- MCU_init — MCU initialization in KBI module configuration.
- Vkeyboard_isr — Makes an LED toggle 10 times, every time a KBI interrupt is detected.
- Delay — Makes a delay to make the LED toggling visible

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

This is the initialization code for the keyboard interrupt using the MC9S08GB60. For this example, both KBI registers (KBI1SC and KBI1PE) will be used to customize the module as mentioned above. During the initialization phase, the interrupts are masked, because it takes time for the internal pull up (typically about 26 kΩ) to reach a logic 1 (KBI1 in this case). After the false interrupts are cleared, the keyboard interrupt is unmasked.

```
  void MCU_init(void)
{


  /* ### Init_KBI init code */
  /* KBI1SC: KBIE=0  */
  KBI1SC &= (unsigned char)~0x02;  // Enables any keyboard event to cause a
// hardware interruption



  /* KBI1PE:KBIPE7=1,KBIPE6=0,KBIPE5=0,KBIPE4=0,KBIPE3=0,KBIPE2=0,KBIPE1=0,KBIPE0=0 */

KBI1PE = 0x80; ; // Enables KBI PIN 7 to operate as
// a keyboard interrupt pin this pin detects
// falling edges (KBI1SC_KBEDG7 = 0)

 /* KBI1SC: KBIMOD=1 */
  KBI1SC |= (unsigned char)0x01; // Chooses an edge-and-level event as
// valid to cause an interruption

  /* KBI1SC: KBACK=1 */
```

```
  KBI1SC |= (unsigned char)0x04;

  /* KBI1SC: KBIE=1 */
  KBI1SC |= (unsigned char)0x02; // Enables any keyboard event to cause a
// hardware interruption


} /*MCU_init*/
```

After this, the KBI is initialized and the program is ready for any keyboard interrupt. Whenever one occurs, the keyboard interrupt is serviced. In this case, only PTA7 will trigger an interrupt because it's the only one enabled (KBI1PE_KBI1PE7 = 1). This interrupt routine acknowledges the interrupt, and then toggles the LED in PTF0 ten times.

```
__interrupt void Vkeyboard_isr(void)
{
             int i = 0;
             KBI1SC_KBACK = 1; // Clears KBI interrupt flag (KBIF)


    while (i<10) {// Toggles the led 10 times
    PTFD_PTFD0 = ~PTFD_PTFD0;
             i++;
       Delay();
    }
}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the Device Initialization tool if the option is enabled. The user must define its contents.

# 3 Hardware Implementation

For this example the hardware implementation is fairly simple, because we are only using PTA7 as a KBI input. Only four pins of the MCU will be needed:

- Supply voltage pin
- Ground reference pin
- KBI pin as interrupt input
- I/O pin as output. An LED is used as visual display of the KBI module proper function.

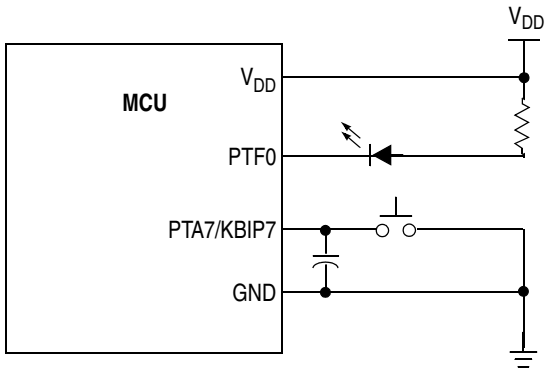



**Figure 1. Four Pins of the MCU Needed for KBI**

Whenever the button in PTA7 is pressed, a hardware interrupt will be triggered. In the code presented before, the KBI interrupt was configured to accept falling edges and low levels events. The internal pull-up resistor makes the default state on PTA7 pin a logic 1. Pin PTF0 is set as an output and it will turn the LED on and off with inverse logic. This means that the LED will turn on with a logic 0. For information on the calculations needed to find the value of R1, refer to application note AN1238: *HC05 MCU LED Drive Techniques Using the MC68HC705J1*.

**NOTE**

- This example was developed using the CodeWarrior IDE version 5.0 for the HCS08 family, and was expressly made for the MC9S08GB60. There may be changes needed in the code to initialize another MCU.
- A delay (20 ms typical) within the software is needed to take into consideration the mechanical stabilization time of the push button. An alternate choice is to use a debounce circuit in the KBI input like shown in the image below.

**Freescale Semiconductor**
Users Guide

# Using the Analog Comparator (ACMP) for the HCS08 Microcontrollers

by: Oscar Luna González
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the analog-to-digital comparator (ACMP) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

The ACMP module provides a circuit for comparing two analog input voltages or for comparing one analog input voltage with an internal reference voltage. Inputs of the ACMP module can operate across the full range of the supply voltage.

**Table of Contents**



**ACMP Quick Reference**

Because there is more than one ACMP module on some devices, there may be more than one ACMP status and control register on your device. In the register name below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the register that is on ACMP1 from that on ACMP2.

| ACMPxSC | ACME | ACBGS | ACF | ACIE | ACO | ACOPE | ACMOD |
|---|---|---|---|---|---|---|---|

Module configuration

ACME — enables module
ACBGS — select bandgap as reference
ACF — set when event occurs
ACIE — interrupt enable

ACO — reads status of output
ACOPE — output pin enable
ACMOD[1:0] — sets mode

The analog comparator (ACMP) module has two analog inputs named ACMP+ and ACMP–, and one digital output named ACMPO. ACMP+ serves as a non-inverting analog input and ACMP– serves as an inverting analog input. ACMPO serves as a digital output and can be enabled to drive an external pin. The ACMP module can be configure to connect the output of the analog comparator (ACMPO) to TPM input capture channel 0 by setting ACIC in SOPT2. With the input capture function, the TPM can capture the time at which an external event occurs. Rising, falling, or any edge may be chosen as the active edge that triggers an input capture.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The project (QG8_ACMP.mpc) implements the ACMP function, selecting a rising- or falling-edge event to trigger hardware interrupt. The main functions are:

- main — Endless loop waiting for the ACMP interrupt to occur.
- MCU_init — Configures MCU to work with the internal oscillator, and enables the ACMP module.
- ACMP_Isr — Toggles an LED after a rising or falling edge event occurs.

This example consists of comparing two different input voltages using the ACMP module. ACMP– will be feed with a static voltage (1.5 V) and will serve as a reference voltage. ACMP+ will be fed with a variable voltage (0 to 3 V). Every time the ACMP+ voltage crosses the ACMP– reference voltage, a hardware interrupt will be triggered turning on and off a pin at port B (PTBD_PTBD0).

In this application, the ACMP module will be demonstrated by turning an LED on and off due to an ACMP hardware interrupt triggered by the comparison voltage between ACMP+ and ACMP–.

Please refer to the source code for more details.

Following these steps, the user will be able to use the ACMP module for this example:

1. Configure the analog comparator register (ACMPSC).

```
                                /* ACMPSC: ACME=1, ACBGS=0, ACF=1, ACIE=1, ACO=0, ACOPE=0,
                                ACMOD1=1, ACMOD0=1 */

ACMPSC = 0xB3;                  /*Analog Comparator Enable, External pin ACMP+
                                selected as Non-inverting input,Compare event
                                has not occurred, Enables ACMP interrupt,
                                Analog Comparator Output not available on ACMP,
                                Sets ACF flag when compare event detects rising
                                Or falling edge */
```

2. Declare ACMP interrupt service routine

```
__interrupt void ACMP_Isr(void))    /* Declare ACMP vector address interrupt*/
                                    /* ACMP Vector Address = 20 */
```

Because an interrupt based algorithm is being implemented, the global interrupt enable mask must be cleared as follows:

```
EnableInterrupts;   /* __asm CLI; */
```

From this point on, the code execution is performed inside the ACMP interrupt service routine. The code inside does the following:

1. Clear ACMP interrupt flag.

   ```
   ACMPSC_ACF = 1;    /* clear ACF flag  */
   ```

2. Next the ISR will contain the code that toggles an LED each time a rising or falling edge event occurs.

# 3 Hardware Implementation

This schematic shows the hardware used to exercise the code provided.


**NOTE**

- This example code was developed using the CodeWarrior Development Studio for HC(S)08 v5.0 using Device Initialization, and was expressly made for the MC9S08QG8 in the 16-pin package. Changes may be needed before the code can be used with other HCS08 MCUs.
- ACMP module can operate comparing one analog input to an internal reference voltage. This example code was expressly made to configure the ACMP module to work without using the internal reference voltage.
- The analog comparator circuit is designed to operate across the full range of the supply voltage. Please see the data sheet for your device.

**Freescale Semiconductor**
Users Guide

# Using the 10-Bit Analog-to-Digital Converter (ADC) for the HCS08 Family Microcontrollers

by: Andrés Barrilado González
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the 10-bit analog-to-digital converter (ADC10) module on an HCS08 microcontroller (MCU). **The ADC module is different from the ATD module — check the data sheet for your device.** A functional description and basic information on the configuration of the module is provided. The following examples may be modified to suit your application — again, refer to the data sheet for your device.

**Table of Contents**



**ADC Quick Reference**

Because there is more than one ADC module on some devices, there may be more than one set of registers on your device. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the register that is on ADC1 from that on ADC2. For the specific pin control registers and bits on your device, please refer to your data sheet.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ADCxSC1 | COCO | AIEN | ADCO | ADCH | | | | |
| | Interrupt enable; continuous conversion enable; input channel select | | | | | | | |
| ADCxSC2 | ADACT | ADTRG | ACFE | ADFGT | | | | |
| | Compare function, conversion trigger, and conversion active control | | | | | | | |
| ADCxRH | | | | | | | ADR9 | ADR8 |
| ADCxRL | ADR7 | ADR6 | ADR5 | ADR4 | ADR3 | ADR2 | ADR1 | ADR0 |
| | Result of ADC conversion | | | | | | | |
| ADCxCVH | | | | | | | ADCV9 | ADCV8 |
| ADCxRCVL | ADCV7 | ADCV6 | ADCV5 | ADCV4 | ADCV3 | ADCV2 | ADCV1 | ADCV0 |
| | Compare value | | | | | | | |
| ADCxCFG | ADLPC | ADIV | | ADLSMP | MODE | | ADICLK | |
| | Mode of operation, clock source select, clock divide, sample time, and low power configuration | | | | | | | |
| APCTL1 | ADPC7 | ADPC6 | ADPC5 | ADPC4 | ADPC3 | ADPC2 | ADPC1 | ADPC0 |
| APCTL2 | ADPC15 | ADPC14 | ADPC13 | ADPC12 | ADPC11 | ADPC10 | ADPC9 | ADPC8 |
| APCTL3 | ADPC23 | ADPC22 | ADPC21 | ADPC20 | ADPC19 | ADPC18 | ADPC17 | ADPC16 |
| | Pin control: ADC or I/O controlled | | | | | | | |

The HCS08 10-bit analog-to-digital converter (ADC) is a successive-approximation converter available for 10-bit resolution. Some HCS08 microcontrollers include an ADC module with a wide range of options for the user. See the data sheet for family-specific features.

- Two options for resolution: 8-bit or 10-bit, configurable by software
- Conversion type adaptable to each application: allows single or continuous conversion
- Includes a conversion complete flag and a conversion complete interrupt, allowing the user to choose either polling or an interrupt-based approach
- Selectable ADC clock frequency: includes a bus clock prescaler

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The following example describes the initialization code for the 10-bit ADC module using the interrupt-based approach, 10-bit resolution, and continuous-sample mode.

The zip file contains the following functions:

- main – Loops forever
- MCU_init – Configures hardware and the ADC module to accept ADC interrupts, selects channel 1 as the input channel, and formats the result
- Vadc_isr – Responds to ADC interrupts and turns an LED on or off accordingly

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

Following these five simple steps, you can use the ADC module:

1. Configure the APCTL1, APCTL2 and APCTL3 registers for the ADC module:

   ```
   APCTL1 = 0x80;
   ```

   This will select which pins the MCU will use as input for ADC conversions. For this example, pin 1 has been enabled for ADC use.
2. Configure the AD1CFG register for the ADC module:

   ```
   ADCCFG = 0x78;
   ```

   There are three main purposes for this step: (a) format the result, (b) establish the speed at which the signal will be sampled, and (c) select the power mode.

   For the first part, one must first choose between 8-bit resolution and 10-bit resolution. For this example, 10-bit resolution has been selected.

   The second part of the configuration of this register allows the user to set the sampling speed of the ADC by selecting the ADC clock source, and an ADC-clock prescaler. The speed of the conversion depends on the resolution selected (8- or 10-bit), the frequency of the clock source, and the value of the prescaler. Table 1 presents different scenarios where an estimate of the number of cycles is shown for a complete conversion.

**Table 1. Total Conversion Time vs. Control Conditions**

| Conversion Type | ADICLK | ADLSMP | Max Total Conversion Time |
|---|---|---|---|
| Single or first continuous 8-bit | 0x, 10 | 0 | 20 ADCK cycles + 5 bus clock cycles |
| Single or first continuous 10-bit | 0x, 10 | 0 | 23 ADCK cycles + 5 bus clock cycles |
| Single or first continuous 8-bit | 0x, 10 | 1 | 40 ADCK cycles + 5 bus clock cycles |
| Single or first continuous 10-bit | 0x, 10 | 1 | 43 ADCK cycles + 5 bus clock cycles |
| Single or first continuous 8-bit | 11 | 0 | 5 µs + 20 ADCK + 5 bus clock cycles |
| Single or first continuous 10-bit | 11 | 0 | 5 µs + 23 ADCK + 5 bus clock cycles |
| Single or first continuous 8-bit | 11 | 1 | 5 µs + 40 ADCK + 5 bus clock cycles |
| Single or first continuous 10-bit | 11 | 1 | 5 µs + 43 ADCK + 5 bus clock cycles |
| Subsequent continuous 8-bit; $f_{Bus} \geq f_{ADCK}$ | xx | 0 | 17 ADCK cycles |
| Subsequent continuous 10-bit; $f_{Bus} \geq f_{ADCK}$ | xx | 0 | 20 ADCK cycles |
| Subsequent continuous 8-bit; $f_{Bus} \geq f_{ADCK}/11$ | xx | 1 | 37 ADCK cycles |
| Subsequent continuous 10-bit; $f_{Bus} \geq f_{ADCK}/11$ | xx | 1 | 40 ADCK cycles |

In this example, the bus clock has been selected as the ADC clock, with a divide-by-8 prescaler value.

Finally, it is important to select the power consumption of the module. Low power consumption does not enable the converter to operate at maximum speed. Long sample times also help with low power consumption. In this example, low power configuration with long sample times has been selected.

3. Configure the compare function for the ADC module:

   If enabled, the compare function will raise the conversion complete flag only when the ADC result is greater- or less-than a pre-established value. Setup is a two step process. First, the pre-established 10-bit value must be set.

   ```
   ADCSC2 = 0x30;
   ```

   Next, the AD1SC2 register has to be configured. In doing so, the automatic compare function can be enabled and configured to flag greater- or less-than values. Also, ADC hardware or software triggering can be selected. In this example, the compare function is enabled with greater-than comparison and software triggering is selected.

4. Configure the AD1SC1 register for the ADC module:

   ```
   ADCSC1 = 0x67;
   ```

   The AD1SC1 register allows the user to select either the polling method or the interrupt method to handle conversions. If the interrupt method is selected by setting the interrupt enable bit, when the conversion is complete, the read-only conversion complete flag in this register will be set. The program will then jump to the interrupt service routine. If the polling method is selected by clearing the interrupt enable bit, the software must continuously poll the conversion complete flag to determine when the conversion is done. In this example, ADC interrupts are enabled in continuous-conversion mode, and channel 1 is selected.

5. Read the result after the ADC conversion is done:

   ADCRL;

```
PTBD_PTBD6 = ~ADCRH_ADR8;
PTBD_PTBD7 = ~ADCRH_ADR9;
ADCSC1 &= 0x7F;
```

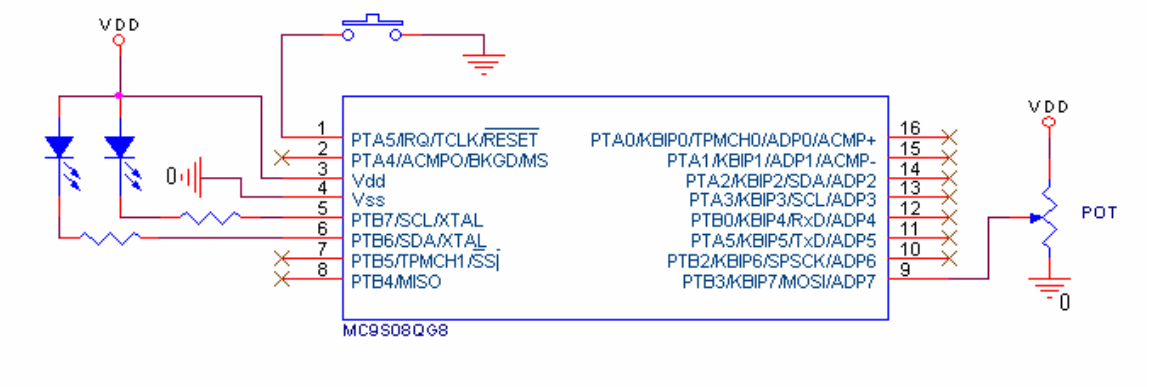
After the ADC conversion is done, the conversion complete flag will be set, and the program will jump to the ADC interrupt service routine. The resulting conversion is placed in the ADC result data registers (ADC1RH/ADC1RL). For 8-bit conversions, the ADC1RL contains the result; for 10-bit conversions, the ADC1RH register contains the most significant bits, and the ADC1RL register contains the least significant ones. After any of the ADC result data registers is read, the conversion complete flag will be cleared. To start a new conversion, the AD1SC1 register must be written again. The same configuration can be re-written to start a new conversion.

# 3 Hardware Implementation

As mentioned before, ADP7 is the selected pin for our analog input, which, for the purposes of this example, is a variable resistor. The variable resistor (potentiometer) allows ADP7 to receive voltages between $V_{DD}$ and $V_{SS}$. The LEDs used in this application are set in inverse logic. This means, the LED will turn on with a logic 0 on ADP7 and will turn off with a logic 1. The ADC also requires four supply/reference/ground connections: Analog power ($V_{DDAD}$), used as the ADC power connection; analog ground ($V_{SSAD}$), used as its ground connection; voltage reference high ($V_{REFH}$), the high reference voltage for the converter; voltage reference low ($V_{REFL}$), the low reference voltage for the converter. Depending on the package, these ports can be externally available. If so, always connect them, also connect the $V_{REFL}$ pin to the same voltage potential as $V_{SSAD}$, the MC9S08QG8 microcontroller have these connections internally. $V_{REFH}$ may be connected to the same potential as $V_{DDAD}$, or may be driven by an external source that is between the minimum $V_{DDAD}$ spec and the $V_{DDAD}$ potential ($V_{REFH}$ must never exceed $V_{DDAD}$, for more information refer to the specific data sheet).

**NOTE**

This software example was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and tested using a MC9S08QG8 running in self-clocked mode. Coding changes may be needed to initialize another MCU. Every microcontroller's initialization code depends on the application and the microcontroller itself.



# Using the Analog-to-Digital Converter (ATD) for the HCS08 Microcontrollers

by: Andrés Barrilado González
RTAC Americas
México 2005

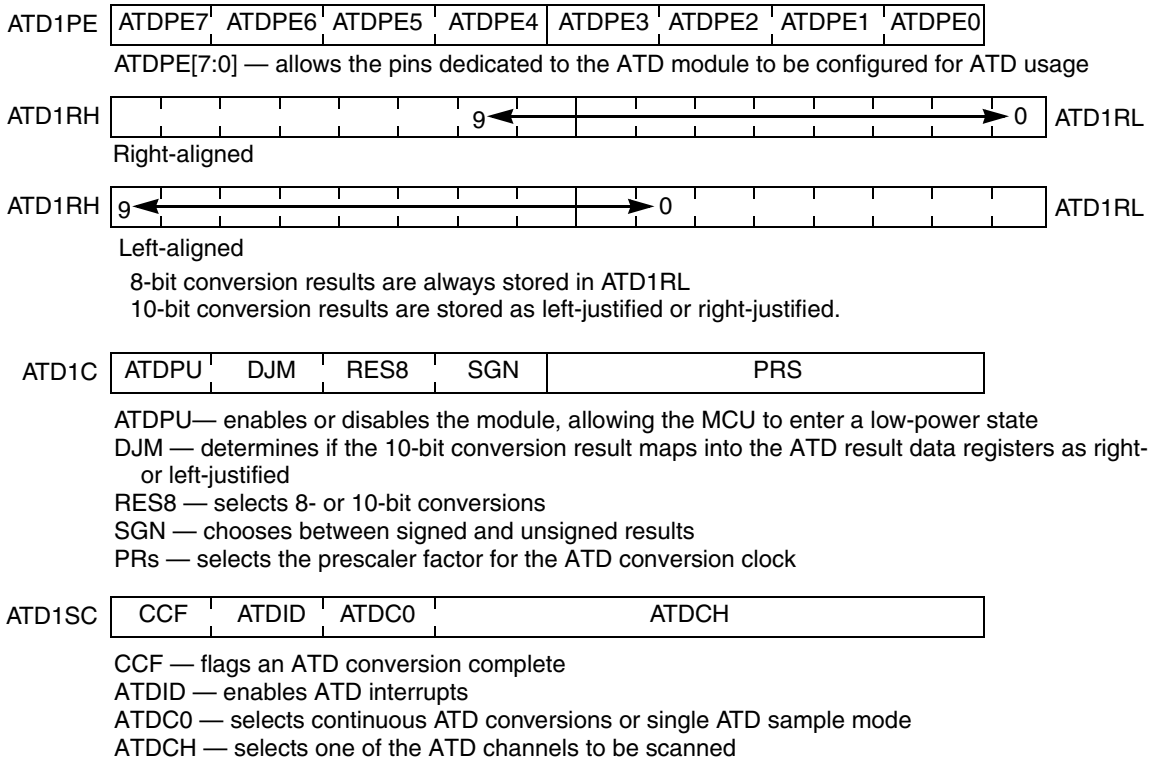
## 1 Overview

This is a quick reference for using the analog-to-digital converter (ATD) module on an HCS08 microcontroller (MCU). **The ATD module is different from the ADC module — check the data sheet for your device.** Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — again, refer to the data sheet for your device.

**Table of Contents**



**ATD Quick Reference**



# 2 Code Example and Explanation

The following example describes the initialization code for the 10-bit ATD module using the interrupt-based approach, 10-bit resolution, and continuous-sample mode. This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

It contains the following functions:

- main — Loops forever
- MCU_init — Configures hardware and the ATD module to accept ATD interrupts, selects channel 1 as the input channel, and formats the result

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

Following these four simple steps, the user can use the ATD module:

1. Configure the ATD1PE register for the ATD module:
   ATD1PE = 0x02;  /* Write breaks the conversion */

This will select which pins of the MCU will be used as input for ATD conversions. For this example, pin 1 has been enabled for use by the ATD.

2. Configure the ATD1C register for the ATD module:
   ATD1C = 0xA0;  /* Write breaks the conversion */

There are two main purposes to this step: (a) format the result and (b) establish the speed at which the signal will be sampled.

For the first part, one must first choose between 8-bit resolution and 10-bit resolution. When selecting the latter, it is also necessary to select between right- or left-justification; for 8-bit resolution, this field is not relevant. It is also possible to select between signed (two's compliment) and unsigned format using this register. For this example, 8-bit, unsigned data format has been selected.

The second part of the configuration for this register enables to decide on the sampling speed of the ATD via a bus-clock prescaler. The ATD conversion clock must operate between a specific range of frequencies for correct operation. If the selected prescaler is not fast enough, the ATD will generate incorrect conversions. According to the bus-clock speed, the prescaler must be set according to the formulas below:

$$Clk_{MaxBus} = (ATDClk_{Max}) * ((\Pr eScaler + 1) * 2)$$
$$Clk_{MinBus} = (500kHz) * ((\Pr eScaler + 1) * 2)$$

Where the *Maximum Bus Clock* is defined by the ICG configuration for the MC9S08GB60 microcontroller. The *Maximum ATD Conversion Clock* is 2 MHz when $V_{DD}$ is greater than 2.08 V and 1 MHz when under this same value, and the *prescaler* is the value to be set. For the ATD to operate correctly, prescaler values must be between the values obtained for this variable in this two equations. For this example, a prescaler value of 0 has been selected, the bus clock is 4 MHz, and $V_{DD}$ is 3 V.

Finally, it is important to power up the module. If this bit is not set (ATDPU in ATD1C register), the ATD is not enabled, and it will not work.

3. Configure the ATD1SC register for the ATD module:
   ATD1SC = 0x41;  /* Write starts a new conversion */

The configuration of the ATD1SC register allows the selection of which of the eight ATD channels will be used. Configuration of the polling method (polling or interruption), and continuous or single conversion modes are also included. If the interrupt method is selected, when the conversion is complete the read-only *Conversion Complete Flag* (also in this register) will be set and the program will jump to the interruption routine. For this example, ATD interrupts are enabled in single-conversion mode, and channel 1 is selected.

4. Read the result after the ATD conversion is done:

After the ATD conversion is done, the *Conversion Complete Flag* will be set and the program will jump to the ATD interrupt routine. The resulting conversion is placed at the *ATD Result Data* registers.

```
__interrupt void Vatd1_isr(void)
{
    result = ATD1RH;        /* Read result and acknowledge interrupt */
    PTFD_PTFD3 = ~ATD1RH_BIT15;
    PTFD_PTFD2 = ~ATD1RH_BIT14;
    PTFD_PTFD1 = ~ATD1RH_BIT13;
    PTFD_PTFD0 = ~ATD1RH_BIT12;
    ATD1SC = ATD1SC;            /* Re-Start Conversion for Ch1 */
}
```

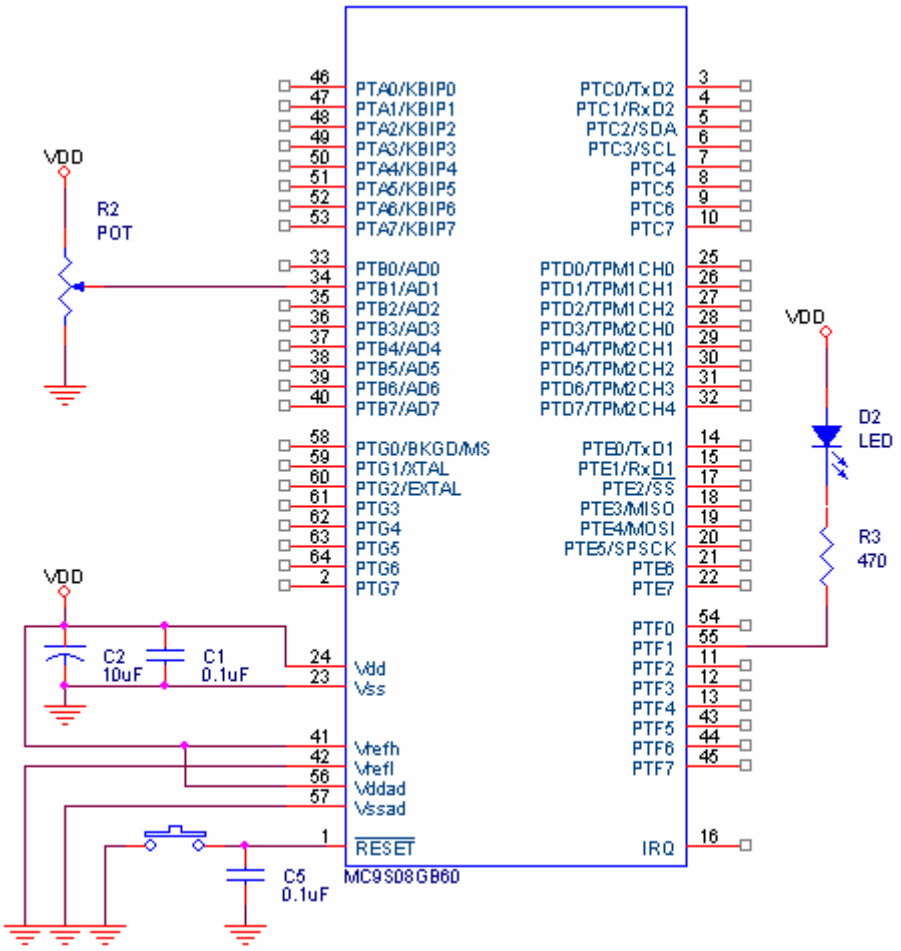
This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the Device Initialization tool if the option is enabled. The user must define its contents.

Because this example is configured for 8-bit resolution conversion, the ATD1RH register contains the result. After any of the *ATD Result Data* registers is read, the *Conversion Complete Flag* will be cleared (acknowledged). To start a new conversion, the ATD1SC register must be written again. The same configuration can be re-written to start a new conversion.

# 3 Hardware Implementation

AD1P1 is the selected pin for our analog input, which, for the purposes of this example, is a variable resistor. The variable resistor (potentiometer) allows AD1P1 to receive voltage values between $V_{DD}$ and $V_{SS}$. Analog power ($V_{DDAD}$) is used as the ADC power connection, analog ground ($V_{SSAD}$) is used as its ground connection. Voltage reference high ($V_{REFH}$) is the high reference voltage for the converter. $V_{REFH}$ may be connected to the same potential as $V_{DDAD}$, or may be driven by an external source that is between the minimum $V_{DDAD}$ spec and the $V_{DDAD}$ potential ($V_{REFH}$ must never exceed $V_{DDAD}$. For more information, refer to the specific data sheet). Voltage reference low ($V_{REFL}$) is the low reference voltage for the converter. If externally available, always connect the $V_{REFL}$ pin to the same voltage potential as $V_{SSAD}$. Finally, the LED used in this application example (turns on every data conversion) is set in inverse logic, which means that the LED will turn on with a logic 0 on AD1P1 and it will turn off with a logic 1.

The schematic below shows the hardware used to exercise the code provided.




**NOTE**

This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and tested using a MC9S08GB60 running in self-clocked mode. Coding changes may be needed to initialize another MCU. It is important to consider that every microcontroller needs an initialization code that depends on the application and the microcontroller itself.

# Using the Inter-Integrated Circuit (IIC) Module on the HCS08 Microcontrollers

by: Miguel Agnesi Meléndez
RTAC Americas
México 2005

## 1 Overview

This document is a quick reference for programming and erasing the Flash memory included in the HCS08 Family microcontrollers (MCUs). Basic information about the functional description and configuration are provided. The example may be modified to suit the specific needs for your application — refer to the data sheet for your device.

**Table of Contents**



**IIC Quick Reference**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IICA | ADDR | | | | | | | 0 |
| Address to which the module will respond when addressed as a slave (in slave mode) | | | | | | | | |
| IICF | MULT | | ICR | | | | | |
| Baud rate = BUSCLK / (2 x MULT x (SCL DIVIDER)) | | | | | | | | |
| IICC | IICEN | IICIE | MST | TX | TXAK | RSTA | 0 | 0 |
| Module configuration | | | | | | | | |
| IICS | TCF | IAAS | BUSY | ARBL | 0 | SRW | IICIF | RXAK |
| Module status flags | | | | | | | | |
| IICD | DATA | | | | | | | |
| Data register; Write to transmit IIC data read to read IIC data | | | | | | | | |

## 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

The example shown here consists of generic code to use the HCS08 MCU IIC module to communicate with another IIC device, using the HCS08 MCU IIC interrupt routine to handle most of the communication. This

example is intended to be used in a basic scenario where the communication just addresses the slave and in the next byte starts sending or receiving data.

The code can be used in a master or slave implementation with out any modification, in order to make the code as generic as possible a 128 bytes array is used as a buffer to store the data received trough the IIC module and also to send data from this buffer when data is sent trough the IIC module.

The code breaks the IIC communication process into several steps, which are tracked by a global variable called I2C_STEP, this makes easier for the main code to know if the IIC module is ready for a new communication, if an error has occurred or the actual status of the communication.

The defined IIC steps are:

```
#define IIC_ERROR_STATUS            0
#define IIC_READY_STATUS            1
#define IIC_HEADER_SENT_STATUS      2
#define IIC_DATA_TRANSMISION_STATUS 3
#define IIC_DATA_SENT_STATUS        4
```

The code needs global variables to control the IIC communication in an easy manner, by modifying these variables the interrupt routine can handle the desired communication steps.

```
unsigned char I2C_STEP
```

Used to store the actual status of the IIC communication.

```
unsigned char I2C_LENGTH
```

When the device is configured as master this variable stores the number of bytes to be read from the slave or sent to the slave.

```
unsigned char I2C_COUNTER
```

Stores the bytes that are sent or received.

```
unsigned char I2C_DATA[128]
```

Array used as transmit or receive buffer for IIC communications.

```
unsigned char I2C_DATA_DIRECTION
```

Used to indicate if data should be sent to the salve or read from the slave.

## 2.1 Configure IIC Function

Module configuration is accomplished by the configureI2C function; this function receives the device self address and sets the IIC bus speed to the desired frequency. This value may change between devices and clock configuration; please refer to the data sheet for detailed information.

```
    /* Function to configure the IIC module. */
void configureI2C(unsigned char selfAddress){
IICC_IICEN = 1; /* Enable IIC */
IICA = selfAddress; /* IIC Address */
IICF = 0x8D; /* Set IIC frequency */
```

## 2.2 Write Bytes and Read Bytes Functions

The code has implemented two functions as an example on how to use the global variables to read data from the slave or to send data to the slave, because we are initializing the communication when using these routines both of these functions set this device as the master device, send the address of the selected slave and after this step the following IIC communication will be handled by the interrupt handler routine according to the IIC global variables settings.

A small delay is used after the master bit is set to 1. This delay is used to stabilize the bus signals in noisy environments. This delay can be modified or deleted according to the specific application bus characteristics.

```
unsigned char WriteBytesI2C (unsigned char slaveAddress,unsigned char numberOfBytes){
      unsigned char Temp;

  I2C_LENGTH = numberOfBytes;
  I2C_COUNTER =0;
  I2C_STEP = IIC_HEADER_SENT_STATUS;
  I2C_DATA_DIRECTION = 1;

  /* Format the slave address to place a 0 on the R/W bit (LSB).*/
  slaveAddress &= 0xFE;

  IICC_IICEN = 0;
  IICC_IICEN = 1;
  IICS; /* Clear any pending interrupt */
  IICS_IICF=1;
  IICC_MST = 0;
  IICS_SRW=0;
  IICC_TX = 1; /* Select Transmit Mode */
  IICC_MST = 1; /* Select Master Mode (Send Start Bit) */
  for(Temp=0;Temp<5;Temp++); /* Small delay */
  ICD=slaveAddress; /* Send selected slave address */

  return(1);
}

unsigned char ReadBytesI2C (unsigned char slaveAddress,unsigned char numberOfBytes){
  I2C_LENGTH = numberOfBytes;

  I2C_COUNTER =0;
  I2C_STEP = IIC_HEADER_SENT_STATUS;
  I2C_DATA_DIRECTION = 0;

  /* Format the Address to fit in the IICA register and place a 1 on the R/W bit. */

slaveAddress &= 0xFE;

slaveAddress |= 0x01; /* Set the Read from slave bit. */
IICS; /* Clear any pending interrupt */
IICS_IICIF=1;
IICC_TX = 1; /* Select Transmit Mode */
IICC_MST = 1; /* Select Master Mode (Send Start Bit)*/
IICD=slaveAddress; /* Send selected slave address */
return(1);


}
```

## 2.3 Main Function

The main function in this example configures the internal bus to 20 MHz and initializes the IIC module with a different address depending if the MASTER variable is defined on the code, the address assigned are random selected values, these values can be modified to fit the specific application.

After initialization the global interrupts are enabled, and if the MASTER variable is defined, data is read from the slave and then data is sent to the slave by calling the respective functions.

```
void main(void) {

  /* Configure internal clock reference.
   * Internal clock and 19,995,428 bus frequency. */
  ICGC1 = 0x28;
  ICGC2 = 0x70;

  /* Configure interfaces. Set our IIC address. */
#ifdef MASTER
  configureI2C(0x50);
  I2C_DATA[0]='A';       /* test data */
#else
  configureI2C(0x52);
#endif

  EnableInterrupts; /* enable interrupts */

#ifdef MASTER
  ReadBytesI2C(0x52,6);
  WriteBytesI2C(0x52,6);
  while(I2C_STEP>IIC_READY_STATUS)__RESET_WATCHDOG(); /* wait for memory to be read */
#endif

/* Application is based on interrupts so just stay here forever. */
  for(;;) {
      __RESET_WATCHDOG(); /* feeds the dog */
  }                       /* loop forever */
  /* please make sure that you never leave this function */
}
```

## 2.4 Interrupt Handler Routine

The most important part of the code is the IIC interrupt handler routine, which acknowledges the interrupt and then depending on the IIC communication status, follows the appropriate steps to send or receive the remaining bytes.

This routine handles the master and slave interrupts in both transmit and receive modes. The routine determines whether the device is acting as a master by verifying the MST bit in the control status is set. If so, it follows the master logic to read write the next byte. If the device is configured as slave, it follows the slave logic to read or write the next byte.

```
interrupt 24 void IIC_Control_handler(void)
```

The handler routine clears the interrupt flag.

```
IICS; /* ACK the interrupt */
IICS_IICIF=1;
```

Then verifies if a collision as occurred on the bus to set the IIC_ERROR_STATUS and stop the communication.

```
if(IICS_ARBL==1){ /* Verify the Arbitration lost status */
IICS_ARBL= 1;
IICC_MST = 0;
          I2C_STEP = IIC_ERROR_STATUS;
          return;
   }
```

Verify if our module is the IIC master device by reading the MST bit.

Notice this bit is cleared automatically if an arbitration lost has occurred.

```
if(IICC_MST==1){ /* If we are the IIC Master */
If the last byte was not ACK stop communication and set the error flag.
if(IICS_RXAK==1){ /* Verify if byte sent was ACK */
    IICC_MST = 0;
    I2C_STEP = IIC_ERROR_STATUS;
      return;
    }
```

Verify whether this interrupt was generated due to the first byte transmission complete (byte containing slave address and data direction bit). If so, configure the module direction bit according to the desired read from slave or write to slave configuration.

Set the global variable I2C_STEP to data transmission status.

```
if(I2C_STEP == IIC_HEADER_SENT_STATUS){ /* Header Sent */
     IIC1C_TX = I2C_DATA_DIRECTION;
     I2C_STEP = IIC_DATA_TRANSMISION_STATUS;
```

If we are about to read data from slave read the data register to clock in the first byte sent from the slave and return from the interrupt handler to wait until the requested byte is received.

```
if(IICC_TX==0){

     IICD;
     return;
     }
}
```

If in the data transmission status, verify if we are sending or receiving data from the slave.

```
if(I2C_STEP == IIC_DATA_TRANSMISION_STATUS){
```

If we are sending data to the slave load IIC data register with the next byte, verify whether we have reached the number of bytes to be sent to set the global variable I2C_STEP value to DATA_SENT_STATUS and then wait for this byte to be transmitted to the slave.

```
if(IICC_TX==1){
IICD = I2C_DATA[I2C_COUNTER]; /* Send the next byte */


    I2C_COUNTER++;
if(I2C_LENGTH <= I2C_COUNTER){
                                        I2C_STEP=IIC_DATA_SENT_STATUS;
    }
    return;
    }
```

If the master is reading data from the slave, verify whether we are about to read the last byte to change the TXAK bit to 1 and avoid acknowledging the next byte read to indicate the slave we are done reading data.

```
        else{

            if((I2C_COUNTER+1) == I2C_LENGTH)
            IICC_TXAK = 1; /* to indicate end of transfer */
```

Read the next byte.

```
            I2C_DATA[I2C_COUNTER] = IIC1D; /* Read the next byte */
            I2C_COUNTER++;
```

If we have finished reading data set the global variable I2C_STEP value to DATA_SENT_STATUS.

```
        if(I2C_LENGTH <= I2C_COUNTER){
         I2C_STEP=IIC_DATA_SENT_STATUS;
}
```

Wait until next byte is read.

```
          return;
        }
}
```

After we have finished with the data transmission or reception and the last byte has been sent/received, the device should generate the stop signal on the bus and set the global variable I2C_STEP value to READY_STATUS

```
if(I2C_STEP==IIC_DATA_SENT_STATUS){
I2C_STEP=IIC_READY_STATUS;
IICS;
IICS_IICIF=1;
IICC_TX=0;
IICS_SRW=0;
IICC_MST=0;

return;
}
}
If the device is acting as the slave device on the IIC bus
else{ /*  SLAVE OPERATION  */
```

Verify whether this is the first byte received (address and data direction byte) by looking at the actual global variable I2C_STEP value. If we were in a ready status, this is the first byte received.

```
if(I2C_STEP <= IIC_READY_STATUS){
    I2C_STEP = IIC_DATA_TRANSMISION_STATUS;
```

Configure the module data direction to the desired slave transmit or slave receive according to the less significant bit on the data received.

```
        IICC_TX = IIC1S_SRW;
        I2C_COUNTER = 0;
```

If we are receiving data, we should read the IIC1D (containing the address byte) to free the IIC bus and get the next byte (which will be the first data byte sent to the slave).

```
        if(IICC_TX==0){
           IICD;
```

```
        return;
        }
        }
```

If this is not the first byte received

```
            if(IICS_TCF==1){
```

If we are receiving data store the received byte on the buffer and return.

```
            if(IICC_TX == 0){
            I2C_DATA[I2C_COUNTER]=IIC1D;
            I2C_COUNTER++;
            return;
         }
```

If data is sent from the slave to the master

```
else{                           /* Data sent by the slave */
```

Verify if the last byte sent was acknowledged, it not the transmission has finished so we clear the flags and free the IIC bus.

```
            if(IICS_RXAK==1){
                IICC_TX = 0;
                IICD;
                I2C_STEP = IIC_READY_STATUS;
                return;
           }
```

If the byte was acknowledged place the next byte in the data register so the bus signals allow the master to read the next byte.

```
                    IICD = I2C_DATA[I2C_COUNTER];
                    I2C_COUNTER++;
                    return;
            }
           }
    }
}
```

# 3 In-Depth Reference Material

Physically, the IIC bus is a simple bidirectional bus, based on two wires, serial data (SDA) and serial clock (SCL). Each device on the bus must have open collector lines to interface with the bus. Because the bus is populated with open collector devices, pullup resistors must be used for each line on the bus.

Data and control signals share the same bus. The IIC standard specifies that when data bits are sent, SDA may change only when SCL is low. When SCL is high and SDA changes, it indicates a start or stop signal.

- SCL high and SDA goes from high to low it is a start signal.
- SCL high and SDA goes from low to high it is a stop signal.

The IIC bus specification is oriented for serial 8-bit data transfers with an extra acknowledge bit in the serial communication. This 9$^{th}$ bit is held down by the receiving device to indicate successful data reception (acknowledge the byte transfer).

The logical implementation of the IIC bus is designed to operate in a master slave relationship, allowing direct communication between any two devices located on the IIC bus. Each device on the IIC bus must have defined a unique address which will be used by the master device to establish communication with the desired slave.

The master starts the communication by generating a start signal on the bus and then sending the unique address of the selected slave device. This calling address should be acknowledged only by the slave with the self address equal to the calling address. Communication between these devices continues until a stop condition is detected on the bus.

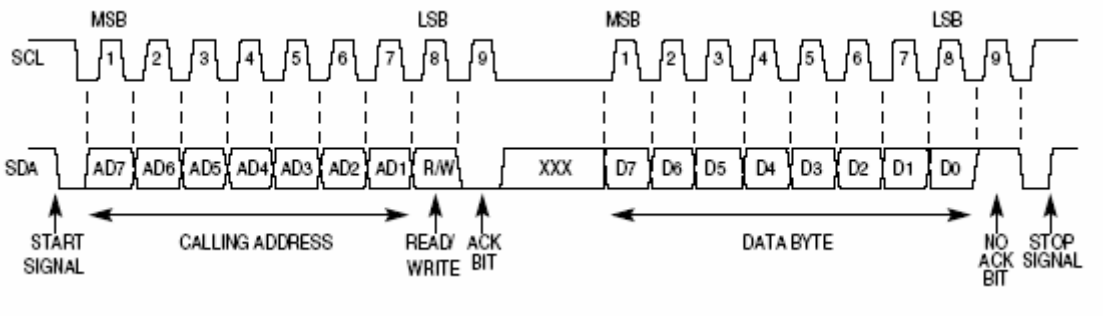



**Figure 1. IIC Bus Transmission Signals**

The first byte after a start condition is used as the calling address. Only the device with a self address equal to the calling address should acknowledge this byte. The device should send or receive the following bytes until a no acknowledge bit is found or a stop signal is generated on the bus.

The calling address uses the less significant bit to indicate if the master is trying to read from the slave (LSB = 1) or sending data to the slave (LSB = 0). If master is reading from the slave, the slave should start transmitting when the master generates the next SCL pulses. If the slave is not fast enough to match the master speed the slave can delay the master until ready to send or receive the next bytes by holding the SCL low.

IIC can handle 100 kbps in standard mode, allowing faster speeds if the bus is properly configured. The HSC08 IIC module can manage data transfers up to clock/20 using reduced bus device loading to meet the electrical characteristics of the bus for a faster transfer.

## 3.1 HCS08 IIC Module Functional Description

IIC module functional description can be divided into two main sub-sections, when the module is acting as the IIC bus master and when the module is acting as a slave on the IIC bus.

This functional description assumes the IIC module and IIC interrupts are already enabled (IIC control register bits IICEN and IICIE set).

When the device is acting as the IIC master this device should start the communication.

- IIC should be configured for data transmission setting the IIC control register TX bit.
- The user sets the IIC control register MST bit and the module generates a start signal on the IIC bus.

- The user writes the desired slave address into the IIC data register with the less significant bit value according to the desired data direction (read from the slave LSB = 1, write to the slave LSB = 0) the module sends the byte trough the IIC bus.
- After the byte transmission is completed, the IIC module sets the IIC interrupt flag.
- Interrupt handler routine should clear the interrupt flag, check if the byte was acknowledged by the slave and if so continue with the communication procedure.

If master is reading data from the slave:

- The IIC control register TX bit should be cleared to enable the reception of data from the slave.
- A dummy read to the IIC data register should be performed to generate the necessary SCL signals on the IIC bus to read the first data byte from the slave into the IIC data register.
- After the byte is received, the acknowledge bit for this byte is automatically generated by the IIC module if the IIC control register TXAK bit is clear; after that, the IIC interrupt flag is set.
- Reading the IIC data register will clock in the next byte.
- The last byte read should not be acknowledged to indicate to the slave the reading is over. To accomplish this, the IIC control register TXAK bit should be set before reading IIC data register.

If master is writing data to the slave:

- The next byte should be written to the IIC data register and the module will send the next byte through the IIC bus, setting the IIC interrupt flag after the byte transmission is complete.
- The IIC status register RXAK bit indicates if the slave acknowledged the byte transfer.

Clearing the IIC control register MST bit will generate the stop condition on the IIC bus.

When the device is acting as the slave device:

- Interrupt flag will be set when the calling address is matched with the device self IIC address. The acknowledge bit is handled by the IIC module.
- Interrupt handler routine should clear the interrupt flag, and continue with the communication procedure.
- If this is the first byte received it is the address byte and the LSB bit indicates if the master wants to read or write to this device, IIC control register TX bit should be set or cleared according to the desired communication direction pointed by the IIC status register SRW bit when the address byte is on the IIC data register.
- If the device is reading data from the master, IIC data register should be read in order to free the SCL line and allow the master to send the next byte.
- If the device is sending data to the master, next byte should be written to the IIC data register in order to free the SCL line and allow the master to read the next byte.
- The IIC interrupt will be generated each time a byte transmission or reception is complete.
- Communication will be finished when the master generates a stop condition on the bus.

**NOTE**

This example code was developed using the CodeWarrior IDE version 5.0 for HC08 using Device Initialization, and was expressly made for the MC9S08GB60.

# Using the Serial Communications Interface (SCI) for the HCS08 Family Microcontrollers

by: Laura Delgado
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the serial communications interface (SCI) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. This example may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**SCI Quick Reference**

Because there are two SCI modules on some devices, there are two full sets of registers. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the registers that are on SCI1 from those on SCI2.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxBDH | | | | SBR12 | SBR11 | SBR10 | SBR9 | SBR8 |
| SCIxBDL | SBR7 | SBR6 | SBR5 | SBR4 | SBR3 | SBR2 | SBR1 | SBR0 |

Baud rate = BUSCLK / (16 x SBR12:SBR0)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxC1 | LOOPS | SCISWAI | RSRC | M | WAKE | ILT | PE | PT |

Module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxC2 | TIE | TCIE | RIE | ILIE | TE | RE | RWU | SBK |

Local interrupt enables; Tx and Rx enable; Rx wakeup and send break

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxS1 | TDRE | TC | RDRF | IDLE | OR | NF | FE | PF |

Interrupt flags; Rx error flags

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxS2 | | | | | | BRK13 | LINR | RAF |

Configure LIN support options and monitor receiver activity
BRK13 and LINR are not available on all devices — see the data sheet for your device.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxS3 | R8 | T8 | TXDIR | | ORIE | NEIE | FEIE | PEIE |

9th data bits; Rx/Tx pin direction in single-wire mode; Local interrupt enables

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SCIxID | R7/T7 | R6/T6 | R5/T5 | R4/T4 | R3/T3 | R2/T2 | R1/T1 | R0/T0 |

Read: Rx data; Write: Tx data

**SCI Module Initialization**

1. Write: SCIxBDH:SCIxBDL
   — to set baud rate

2. Write: SCIxC1
   — to configure 1-wire/2-wire, 9/8-bit data, wakeup, and parity, if used.

3. Write: SCIxC2
   — to configure interrupts
   — to enable Rx and Tx
   — to enable Rx wakeup (RWU), SBK sends break character

   RWU, Rx wakeup, and SBK are used infrequently during initialization.

4. Write: SCIxC3
   — to enable Rx error interrupt sources.
   — Also controls pin direction in 1-wire modes.
   — R8 and T8 only used in 9-bit data modes.

**Module Use**

Wait for TDRE (transmit data register empty flag), then write data to SCIxD
Wait for RDRF (receive data register full flag), then read data from SCIxD
A small number of applications will use RWU to manage automatic receiver wakeup, SBK to send break characters, and R8 and T8 for 9-bit data.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

In this example, the MCU will interact with a serial protocol interface (Hyperterminal) using the SCI module. With the Hyperterminal, the user will send a byte to the MCU, the MCU will add an integer value of 1 to the received data and return the result to the Hyperterminal. The configuration used for the Hyperterminal is described under the Notes section. The application will configure the baud rate registers to have 9600 bps, using the internal bus clock in its default mode (self-clocked mode / 8 MHz).

The functions of the project SCI.mcp are:

- main — Endless loop sending characters to the SPI module
- MCU_init — Initializes MCU and customizes and enables the SCI module
- Vsci1rx_isr — happens every time the SCI receiver full flag (RDRF flag) is detected, it loads the data received and adds a value of 1, to then sends it back.

MCU_init is a function generated by Device Initialization and is located in MCUinit.c also generated by the Device Initialization, which was included in the project.Following these steps, the user will run the SCI1 module in a 9600 bps baud rate:

1. Configure the SCI control registers 1, 2, and 3:

SCI1C1 = 0x00; /* Loop mode disabled, disable SCI, Tx output not inverted,

8-bit characters, idle line wakeup, disable parity bit */

SCI1C2 = 0x2C; /* Enable SCI receive interrupts, Enable transmitter and

receiver */

SCI1C3 = 0x00; /* Disable all error interrupts */

2. Configure the SCI baud rate register

```
*****************************************************************
*                  BUSCLK                 4 MHz                 *
* Baud Rate = -------------------- = ---------- = 9600 bps      *
*                [SBR12:SBR0] x 16     26 x 16                 *
*****************************************************************
/* For this example, the internal bus clock is used,
            ICGOUT
BUSCLOCK = ---------
              2
The default ICGOUT is 8MHz (Self-clocked mode), therefore
BUSCLOCK is 4 MHz. In order to get a 9600 bps baud rate,
following the baud rate formula in MC9S08GB60, the value
for [SBR12:SBR0] is 26 */

SCI1BDH = 0x00; // SCI1BDH has [SBR12:SBR8] bits and SCI1BDL has [SBR7:SBR0],
SCI1BDL = 0x1A; // altogether SCI1BDH and SCI1BHL control the 13 bit
                // prescale divisor for the SCI module baud rate.
```

From this point on, the code execution is performed inside the SCI receive interrupt service routine.

3. Define contents of the interrupt function

```
__interrupt void Vsci1rx_isr(void)
```

4. Clear SCI receiver full interrupt flag.

```
SCI1S1; // Acknowledge SCI Receiver Full Flag
```

5. Read the received data in a global variable called ReceivedByte and increment it.

```
ReceivedByte = SCI1D; // Load received data into a global variable
ReceivedByte += 1; // Increment received data by 1
```

6. Wait for the transmitter to be empty, so that we can queue a new transmission.

```
while (SCI1S1_TDRE == 0); // Wait for the transmitter to be empty
```

7. Store the new computed byte in the SCI data register.

```
SCI1D = ReceivedByte; // Stores new data to be transmitted
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the Device Initialization tool if the option is enabled. The user must define its contents.

# 3 SCI In-Depth Reference Material

The SCI allows full-duplex, asynchronous, NRZ serial communication among the MCU and remote devices, including other microcontrollers. Some features of this module are:

- Full-duplex operation.
- Standard mark/space non-return-to-zero (NRZ) format.
- Programmable baud rate (13-bit modulo driver).
- Programmable 8-bit or 9-bit character length.
- Two receiver wakeup methods: idle line wakeup and address mark wakeup.
- Interrupt driven operations with eight interrupt flags: transmitter empty, transmission complete, receiver full, idle receiver input, receiver overrun, noise error, framing error and parity error.

The data transmission and reception functions are handled by one logical register: the SCI data register (SCIxD). The SCIxD is actually two separate registers: one is written to define the next data to be transmitted, and the other one is read to get the last data received. This allows the SCI transmitter and receiver blocks to operate independently.

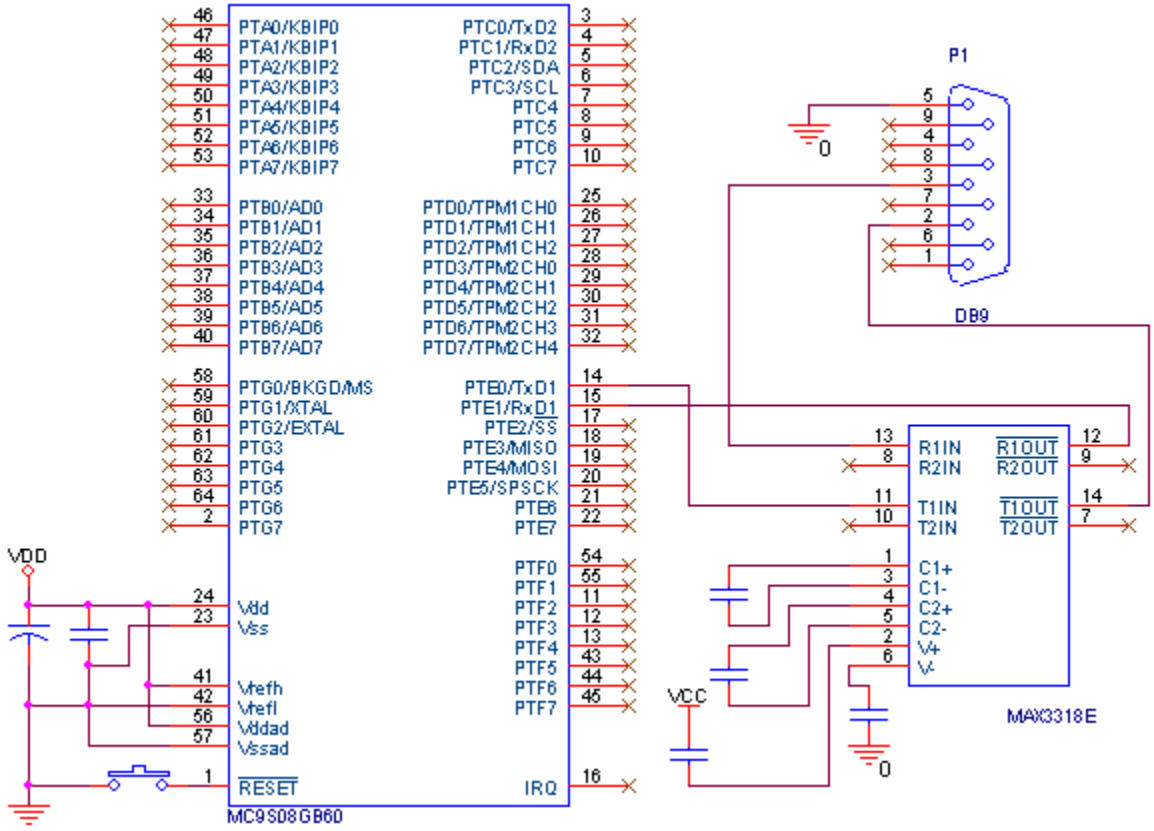
Physically, two MCU pins are used: the transmission pin (TxD) and the reception pin (RxD). Both of these pins transfer data to and from the SCIxD. The SCI module controls transmission and reception using its status interrupt flags. During normal operation, if the transmit buffer is empty, the TDRE flag (transmit data register empty flag) is set and the MCU is permitted to write the next character to be transmitted. In the same way, if the receiver buffer is full, the RDRF flag (receiver data register full flag) is set and the character received can be processed. More conditions are notified with the module's flags, these are discussed and illustrated in the example application.

Both transmission and reception blocks work at the same baud rate. The SCI module has a 13-bit modulo driver that allows a wide range of options for baud rate generation. The clock source for the SCI baud rate generator is the bus-rate clock. Depending on the MCU, the bus-rate clock source can be configured to be internal, external, etc. Refer to your device's data sheet to learn more about the bus-rate clock.

Many microcontrollers in the HCS08 Family have more than one SCI modules. The SCI modules are referred to as SCIx. While programming, register names should include placeholder characters to identify which of the SCI modules is being referenced.

# 4 Hardware Implementation

The schematic below shows the hardware used to exercise the code provided.




**NOTE**

- The software of this note was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and was expressly made for the MC9S08GB60. There may be changes needed in the code to be used in other MCUs.
- The hardware used for the example is shown under the *Schematics* section.
- The Hyperterminal was configured to have:
  - A baud rate of 9600 bps
  - 8-bit mode
  - No parity checked
  - 1 stop bit
  - No flow control
- Not all MCU packages have the RxD and TxD physical pins, even though they have the SCI module.
- It's important for the user to verify the SCI module availability for the microcontroller, because not every part in the HCS08 Family has one. See the data sheet for your device.

# Using the Serial Peripheral Interface (SPI) for the HCS08 Family Microcontrollers

by: Rogelio Gonzalez Coppel
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the serial peripheral interface (SPI) module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**SPI Quick Reference**

Because there are two SPI modules on some devices, there may be two full sets of registers. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the registers that are on SPI1 from those on SPI2.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPIxC1 | SPIE | SPE | SPTIE | MSTR | CPOL | CPHA | SSOE | LSBFE |

SPIE — enables the interrupts generated by the SPRF bit (receiver interrupt) and MODF bit
SPE — enables the SPI module
SPTIE — enables the interrupts generated by the SPTEF bit (transmitter interrupt)
MSTR — selects master mode (1) or slave mode (0) operation
CPOL — configures the SPI clock signal to idle high (1) or low (0)
CPHA — selects clock phase format so first edge occurs at start or middle of data transfer
SSOE — slave select output enable
LSBFE — LSB first (shifter direction)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPIxC2 | | | | MODFEN | BIDIROE | | SPISWAI | SPC0 |

MODFEN — enables master mode-fault function
BIDIROE — enables bidirectional mode output
SPISWAI — SPI stop in wait mode
SPC0 — enables single-wire bidirectional SPI operation

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPIxBR | | SPPR2 | SPPR1 | SPPR0 | | SPR2 | SPR1 | SPR0 |

SPPR[2:0] — selects one of eight divisors for the SPI baud rate prescaler
SPR[2:0] — selects one of eight divisors for the SPI baud rate divider

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SPIxS | SPRF | | SPTEF | MODF | | | | |

SPRF — flags when the receiver's data register becomes full
SPTEF — flags when the transmitter's data register becomes empty
MODF — indicates mode-fault error detected on data input

| | |
|---|---|
| SPIxD | SPID[7:0] |

Read: Rx data; Write: Tx data

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

## 2.1 SPI Master Project

The project SPI_Master implements the SPI in master mode. The main functions are:

- main — Endless loop sending characters to the SPI module
- MCU_init — Configures the hardware and the SPI module as a master
- Vspi1_isr — Responds to the “Receive Full” interrupt
- SPISendChar — Function used to send a byte

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

This example configures the MCU as a single master in the SPI bus:

```
SPI1C1 = 0xD0;
SPI1C2 = 0x00;
```

The SPI clock is configured to run at a bit rate of 64 µs with a 5 MHz bus clock. To obtain a 15.625 kHz (64 µs) SPI bit rate the calculation:

$$\frac{5MHz}{(\text{Prescaler Divisor}) \times (\text{Rate Divisor})} = 15.625\ \text{kHz}$$

$$(\text{Prescaler Divisor}) \times (\text{Rate Divisor}) = \frac{5MHz}{15.625kHz} = 320 = 5 \times 64$$

Given this, SPIBR needs to be configured with a value of 0x45 hex to achieve a 15.625 kHz SPI bit rate with a 5 MHz bus clock.

```
SPI1BR = 0x45;    /*  64us SPI Clock @ 5MHz Bus Clock  */
```

The SPI module is normally used with various slaves. To communicate with a specific slave, its $\overline{SS}$ signal must be low and the $\overline{SS}$ signals from its neighboring slaves must be high to avoid collisions. Therefore, the $\overline{SS}$ signal must be generated by software using a GPIO. This approach must also be used for data transfers of more than one byte because a SPI transaction must be framed within a slave select low-level.

In this example, the master will only interface with one slave. The $\overline{SS}$ line is implemented using a GPIO pin to manage it.

```
PTED_PTED2   = 1;    /* SS Initial State will be 1 (no activity on SPI) */
PTEDD_PTEDD2 = 1;    /* Configure SS as output */
```

The SPISendChar function, which is used to send a byte through the SPI module. It waits for the transmit buffer to be empty and then pulls the $\overline{SS}$ line of the device down and then moves the data into the transmit buffer to start transmission.

```
    void SPISendChar (unsigned char data){


      while (!SPI1S_SPTEF);     /* wait until transmit buffer is empty*/
     PTED_PTED2 = 0;           /* Slave Select set in low*/
     SPI1D = data;             /* Transmit counter*/
}
```

Vspi1_isr (receive full interrupt function) waits for the clock to go low, and then puts the $\overline{SS}$ line high. Then, it acknowledges the interrupt by reading SPI1S and SP1D. The SPI module has only one interrupt vector to service all events associated with the SPI system (receive full, transmit buffer empty and mode fault). Because transmit interrupts and mode fault are disabled for this example, only receive interrupts will be generated. If all interrupts are enabled, the SPI interrupt service routine (ISR) must check the flag bits to determine what event caused the interrupt.

```
__interrupt void Vspi1_isr(void)
{
  while (PTED_PTED5);        /*wait for clock to return no default*/
  PTED_PTED2 = 1;            /*Set Slave Select high*/
  SPI1S;                     /*Acknowledge flag*/
  SPI1D;                     /*Acknowledge flag*/
  PTFD_PTFD1 = ~PTFD_PTFD1; /* Toggle LED*/
}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the Device Initialization tool if the option is enabled. The user must define its contents.

Further details of the actual coding are in the project.

## 2.2 SPI Slave Project

The project SPI_Slave implements the SPI in slave mode. The main functions are:

- main — Endless loop waiting to receive characters through the SPI module
- MCU_init — Configures the hardware and the SPI module as a slave
- Vspi1_isr — Function used to receive a byte

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

This project simply configures the SPI as a slave. When the SPI module receives a byte, it interrupts the MCU and executes the Vspi1_isr function, which outputs on GPIO port F the byte received.

Please refer to the source code for more details.

**NOTE**

- This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization.
- Both projects were tested using MC9S08GB60 running with internal oscillator.
- SPI is a protocol designed for in-board communication. However, if a cable is needed, be sure that it is not longer than 20 cm.

# Using the 8-Bit Modulo Timer (MTIM) for the HCS08 Family Microcontrollers

by: Miguel Agnesi Meléndez
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for enabling the 8-bit modulo timer (MTIM) functionality on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. This example may be modified to suit your application — see the data sheet for your device.

The HCS08 8-bit MTIM clock input can be selected from the bus clock, an internal fixed clock, the rising edge of an external reference, or the falling edge of the external reference. To make the counter more flexible, a prescaler can be enabled to generate larger time bases using this 8-bit counter. The prescaler can be configured to divide the selected input clock by 1, 2, 4, 8, 16, 32, 64, and 256.

The counter allows user firmware to reset, stop, and select the counter clock source, as well as the clock source prescaler value, in an easy manner.

**Table of Contents**



**MTIM Quick Reference**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| MTIMSC | TOF | TOIE | TRST | TSTP | | | | |
| Overflow status, interrupt enable, counter reset, and stop | | | | | | | | |
| MTIMCLK | | | CLKS | | PS | | | |
| Counter clock select and prescaler select | | | | | | | | |
| MTIMCNT | COUNT | | | | | | | |
| Current counter value | | | | | | | | |
| MTIMMOD | MOD | | | | | | | |
| Counter modulo value | | | | | | | | |

The counter overflow is generated after the counter reaches the value in the modulo register, or, if the modulo register is 0x00, when the counter reaches the maximum 8-bit value (0xFF). The TOF flag becomes set as the counter changes to 0x00. If the interrupt enable bit is set, an interrupt will be generated.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

This example uses the HCS08 modulo timer (MTIM) to generate a square wave on PTB6 using the modulo timer overflow interrupt.

Depending on the selected input clock and prescaler for the MTIM, the desired overflow time value can be calculated according to the next formula:

$$\frac{Clock\ source\ prescaler}{Input\ clock\ \ frequency} * (MTIMMOD\ value + 1)$$

Remember, MTIMMOD = 0x00 will set the modulo timer in a free running mode. Notice the (MTIMMOD value +1) is due to the behavior of the counter: when the counter equals the value in the MTIMMOD register it waits until the next MTIM clock pulse to set the TOF flag and reset the counter as shown in Figure 1.

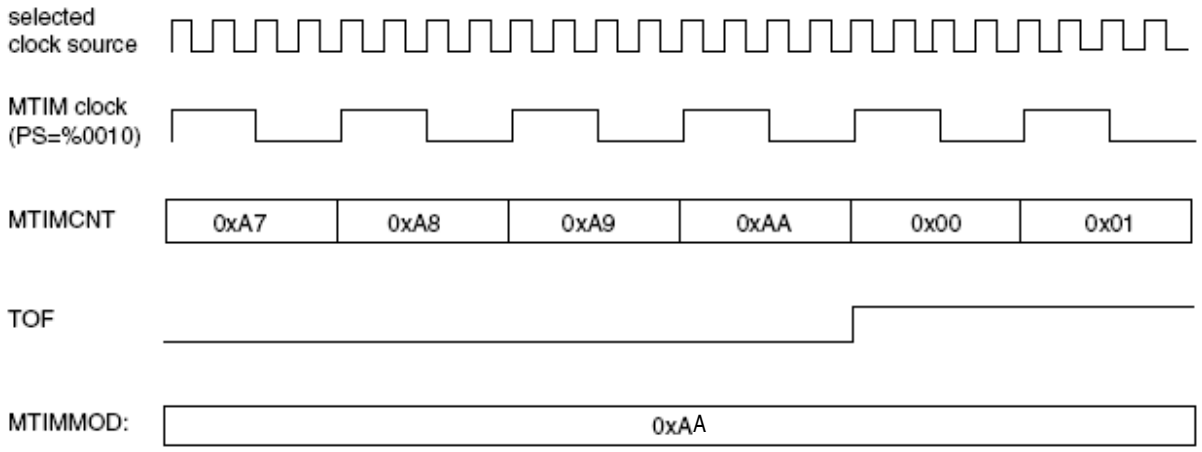



**Figure 1.**

In this example, the device will be configured with a bus clock running at 4 MHz, which is the default bus frequency for the device. The bus will be used as the input clock for the module using a clock divider of 256. With these configuration values, it is possible to obtain an overflow interrupt ranging from 128 µs to 16.32 ms, depending on the MTIMMOD register value.

$$\frac{256}{4000000} * (1+1) = .000128\,s$$

MTIMMOD = 1

$$\frac{256}{4000000} * (255+1) = .0163\,s$$

MTIMMOD = 255

The actual square wave frequency will be affected due to the internal oscillator 2% deviation and the interrupt latency time.

The initialization routine is contained in the MCU_init function. MCU_init is a function generated by device initialization and is located in MCUinit.c, also generated by the device initialization, which is included in the project. It sets the PTB6 pin as output, configures the module timer to use the bus clock divided by 256, setting the modulo timer to 0xFF, enables the modulo timer interrupt and finally starts the timer. This results in an MTIM time out period of 7.68 ms and a PWM period of approximately 15.36 ms.

```
PTBDD |= (unsigned char)0x40;/* Set PTB6 as output   */

MTIMCLK = 0x08;              /* Bus clock 256 divider */
MTIMMOD = 0x77;              /* Count to 0xFF */
MTIMSC  = 0x60;              /* Enable overflow interrupt and start counter */
```

The interrupt handler toggles the PTB6 pin and clears the TOF flag.

```
__interrupt void Vmtim_isr(void)

{

     PTBD_PTBD6 = ~PTBD_PTBD6;  /* Toggle the PTB pin */

     MTIMSC;             /* Clear the TOF flag */

     MTIMSC_TOF = 0;

}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the device initialization tool if the option is enabled. The user must define its contents.

By modifying the MTIMMOD value, the overflow can be generated at any counter value desired.

**NOTE**

This example code was developed using the CodeWarrior IDE version 5.0 for M68HC08, and was expressly made for the MC9S08QG8.

# Using the Real-Time Interrupt (RTI) Function for HCS08 the Microcontrollers

by: Oscar Luna González
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the real-time interrupt (RTI) function on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

The RTI can be used to generate a hardware interrupt at fixed periodic rate. The RTI function in MC9S08QG8 has two source clock choices, the 1-kHz internal clock or an external clock (if available). The RTICLKS bit in SRTISC is used to select the RTI clock source. Both clock sources can be used when the MCU is in run, wait, or any stop mode.

After the RTI module is enabled (by setting RTIE = 1), this interrupt will occur at the rate selected by the SRTISC register. At the end of the RTI time-out period, the RTIF flag is set and a new RTI time-out period starts immediately. Before starting to use the RTI module, the user must select which clock reference will be used to select the RTI clock source.

**Table of Contents**



**RTI Function Quick Reference**

| | | | | | | |
|---|---|---|---|---|---|---|
| SRTISC | RTIF | RTIACK | RTICLKS | RTIE | | RTIS |

RTIF — flags a time-out of the RTI timer; setting this flag will clear the interrupt flag
RTIACK — setting this bit will acknowledge real-time interrupt requests
RTICLKS — selects the clock source to be used by the RTI module (external/internal)
RTIE — enables real-time interrupts
RTIS — sets the period for the RTI based on the internal or external clock source

The data sheet for your device shows the distribution of the different RTI clock sources.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

This example shows how to generate a real-time clock (RTC) through the use of the 1-kHz internal reference and three LEDs. Each LED will indicate the status of its assigned time function (hours, minutes, and seconds, respectively).

Following these steps, the user will be able to use the RTI module for this example:

1. Configure the microcontroller's pins as outputs for initialization purposes

```
PTBD_PTBD1   = 1;       /* Turn Hour LED Off                */
PTBDD_PTBDD1 = 1;       /* Initializes Port B bit 0 as output*/
PTBD_PTBD2   = 1;       /* Turn Minute LED Off              */
PTBDD_PTBDD2 = 1;       /* Initializes Port B bit 1 as output*/
PTBD_PTBD3   = 1;       /* Turn Second LED Off              */
PTBDD_PTBDD3 = 1;       /* Initializes Port B bit 2 as output*/
```

2. Define alias for each pin port for readability purposes

```
/* Defines */
#define LED_Hour        PTBD_PTBD1
#define LED_Minute      PTBD_PTBD2
#define LED_Seconds     PTBD_PTBD3
```

3. Configure the real-time interrupt register (SRTISC). See the data sheet for a detailed description of the different RTI interrupt periods.

```
SRTISC = 0x57;    /* Set delay time to interrupt every 1.024s, Real time
                     INTERRUPT ENABLE, RTI request clock source is internal
                     1-KHz oscillator, ACK = 1 to clear RTIF flag */
```

4. Declare RTI interrupt Service Routine

```
__interrupt void Vrti_isr (void)/* Declare RTI vector address interrupt */
                                /*  RTI Vector Address = 23 */
```

Because an interrupt-based algorithm is being implemented, the global interrupt enable mask has to be cleared as follows:

```
EnableInterrupts;   /* __asm CLI; */
```

From this point on, the code execution is performed inside the RTI interrupt service routine. The code inside does the following:

1. *Clear RTI interrupt flag.*

```
 SRTISC_RTIACK = 1;             /* clear RTIF bit */
```

2. Next the ISR will contain the code that emulates the RTC (Real-Time Clock) functionality.

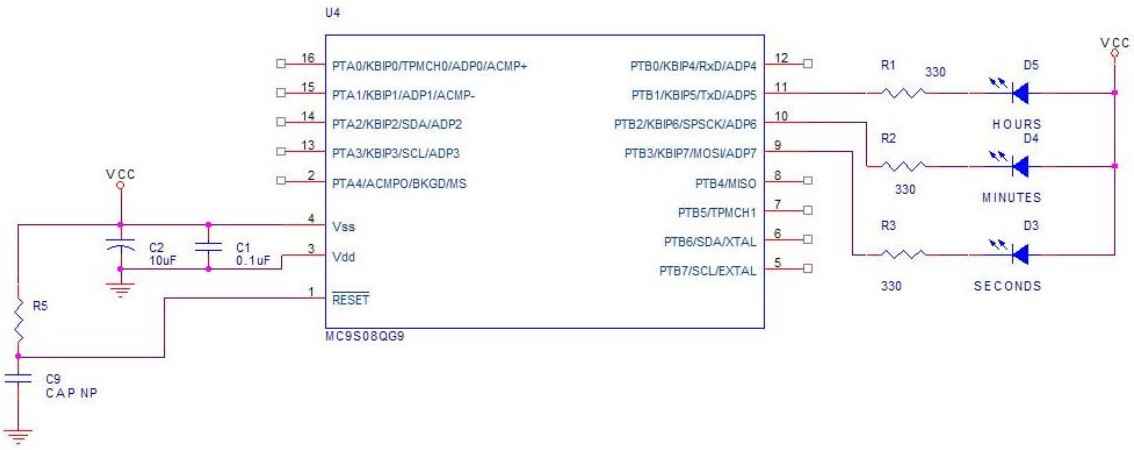
**NOTE**

The following considerations listed must be taken to assure a proper functionality of the RTI module:

- This example code was developed using the CodeWarrior IDE version 5.0 for the HC08 family using Device Initialization, and was expressly made for the MC9S08QG8 using the 16-pin package. There may be changes needed in the code to be used with other HCS08 Families.
- The hardware used for the example is shown under the *Schematics* section.
- The RTI module in this application example takes reference from internal 1-kHz clock source; this 1-kHz internal reference has a 30% margin error. This margin error must be considered by the user because this 30% margin error was characterized at 3.0 V, 25°C and will vary at different voltage and temperatures. Please see the data sheet for your device.
- RTI acknowledge flag must be written with a 1 inside the interrupt service routine to clear real-time interrupt flag (RTIF).
- RTI module has only seven interrupt periods.

# 3 Hardware Implementation

The schematic below shows the hardware used to exercise the code provided.



# Using the Input Capture and Output Compare Functions for the HCS08 Family Microcontrollers

by: Andrés Barrilado González
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for using the timer module on an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**TPM Quick Reference**

Because there is more than one TPM modules on some devices, there are two full sets of registers. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the registers that are on TPM1 from those on TPM2. A small n in the register names below is a place-holder for the channel number.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxSC | TOF | TOIE | CPWMS | CLKSB | CLKSA | PS2 | PS1 | PS0 |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Any write to TPMCNTH or TPMCNTL clears the 16-bit counter

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Modulo value for TPM module; read or write

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnSC | CHnF | CHnIE | MSnB | MSnA | ELSnB | ELSnA | | |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnVH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCnVL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Captured TPM counter for input capture function OR output compare value for output compare of PWM function

The timer/PWM module (TPM) in the HCS08s includes two independent, 16-bit counters, each with several channels that can be configured to work as input capture, output compare, or PWM. Each base counter is considered an independent TPM module. When configured for input capture, an event registered at the channel-related pin will “store” the timer value at the time of the event. When configured for output compare, the channel-related pin can be set, cleared, or toggled at a given value of the timer. When in PWM-mode, a center- or edge-aligned PWM signal can be sent through the channel-related pin with a specific period and duty-cycle. For further information on the TPM differences between each family of microcontrollers, and for information on channel-related pins, refer to the data sheet for your device.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

Two examples have been included: The first one uses the input capture configuration to output the higher value of the timer through a port; The other one is configured to use the output compare configuration to toggle an LED.

All examples contain the same functions:

- main — Cycles endlessly until an interruption occurs.
- MCU_init — Configures hardware and the TPM module to perform as expected in each example. Please refer to each example for specifics of this function.
- Vtpm1ch1_isr — Responds to TPM interruptions according to what is expected in each example. Please refer to each example for specifics of this function.

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which is included in both projects.

## 2.1 Input Capture Code Example

In this example, channel 1 of TPM1 is configured to work in input capture mode. When a rising-edge event is captured through the channel-specific pin, the higher part of the value of the timer at that time is output through port F using an interrupt-based approach. Using a 4 MHz system-bus clock, the TPM is prescaled to overflow approximately every two seconds (Prescaler = 7). To make these settings, the following registers are configured by the device initialization tool.

```
TPM1MOD = 0x00;       /* does not have a modulus value, hence
                         the counter counts up to 0xFFFF */
  /* TPM1C1SC: CH1F=0,CH1IE=1,MS1B=0,MS1A=0,ELS1B=0,ELS1A=1 */
  TPM1C1SC = 0x44;     /* Enable channel interrupt, configures input capture
                  Mode and rising edge event as desired for interrupt*/


  /* TPM1SC: TOF=0,TOIE=0,CPWMS=0,CLKSB=0,CLKSA=1,PS2=1,PS1=1,PS0=1 */
  TPM1SC = 0x0F;       /* Disable overflow interrupt, selects self-clocked
```

```
                    Mode, prescaler of 128*/
```

To estimate the overflow time when setting the timer, the following formula is used:

$$OverflowT = Modulo * \Pr escaler * \frac{1}{TPMclk}$$

When the modulo value is not set, the value *65535* (0xFFFF) should be used instead for this calculation.

For this example, a 4 MHz TPM clock is used (that of the system bus), the modulo value is not set, and a prescaler value of 128 is used:

$$OverflowT = 65535 * 128 * \frac{1}{4000000}$$

$$OverflowT = 2.09712$$

This means the timer will overflow approximately every 2 seconds.

After TPM1 and channel 1 of the TPM1 are configured, and if a rising edge in the channel-specific pin is detected, a service routine must clear the channel overflow flag by reading the flag first and then writing a 0 to it. In this example, the high part of the value stored in TPM1C1V is output through port F.

```
__interrupt void Vtpm1ch1_isr(void)
{

  TPM1C1SC_CH1F = 0;      /*  ACK channel interrupt */
                          /*  Reading flag, then write a 0 to the bit. */
  PTFD = TPM1C1VH;        /* Output high timer result through PTF */
}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the device initialization tool if the option is enabled. The user must define its contents.

Please refer to the source code for more details.

## 2.2 Output Compare Code Example

In this example, channel 1 of TPM 1 is configured for output compare. It is configured to toggle an LED, keeping it roughly half-a-second on and half-a-second off using a 4 MHz system bus clock as clock source. The TPM prescaler value is set to be 5. The timer 1 modulo registers (TPM1MODH:TPM1MODL) and the timer 1 channel 1 value registers (TPM1C1VH:TPM1C1VL) have remained untouched, using the default value of 0. Interruptions are enabled. To make these settings, the following registers are configured by the device initialization tool.

```
TPM1MOD = 0xFFFF;  /* the counter counts up to 0xFFFF */
TPM1C1V = 0x00;  /*Channel interrupt will happen when counter matches
```

```
                    0x00 value*/
TPM1C1SC = 0x54;  /* Enable channel interrupt, configures output compare
                         Mode and toggling of channel pin*/


TPM1SC = 0x0D;  /* Disable overflow interrupt, selects self-clocked
                          Mode, prescaler of 32*/
```

After TPM1 and channel 1 of TPM1 are configured, every time the channel overflows, the interruption service routine will be executed. In it, the channel interrupt flag will by cleared by reading the flag first and then writing a 0 to it.

```
__interrupt void Vtpm1ch1_isr(void)
{


    TPM1C1SC_CH1F = 0;  /*  ACK channel interrupt */
                         /*  Reading flag, then write a 0 to the bit. */
    PTFD_PTFD2 = ~ PTFD_PTFD2;
}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the device initialization tool if the option is enabled. The user must define its contents.

Please refer to the source code for more details.

# 3 Hardware Implementation

The schematic below shows the hardware used to exercise the code provided.

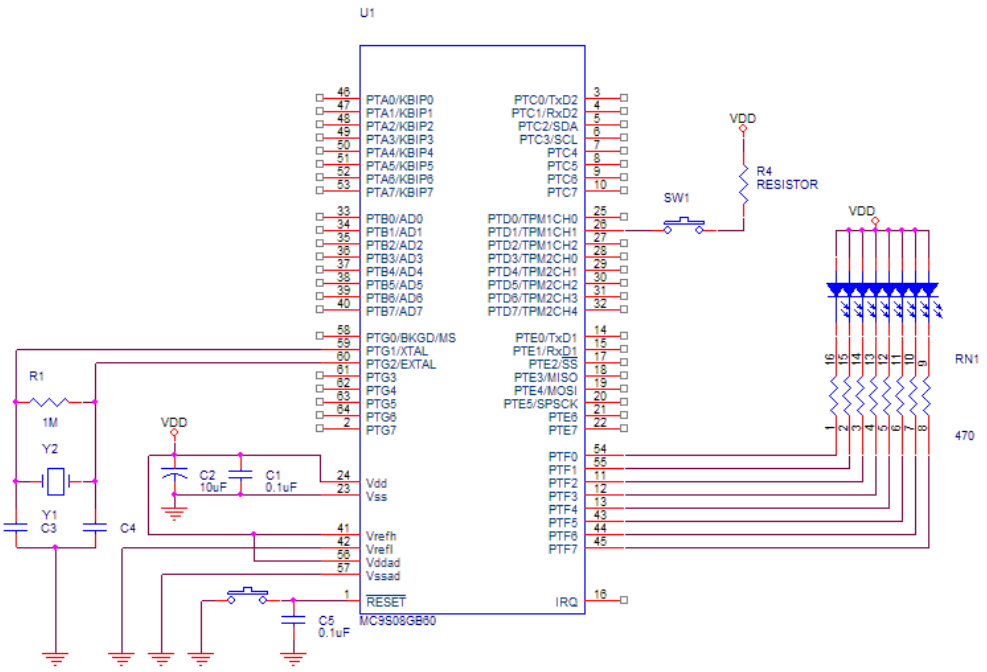



**Figure 1. Schematic of Circuit Used in Example 1**

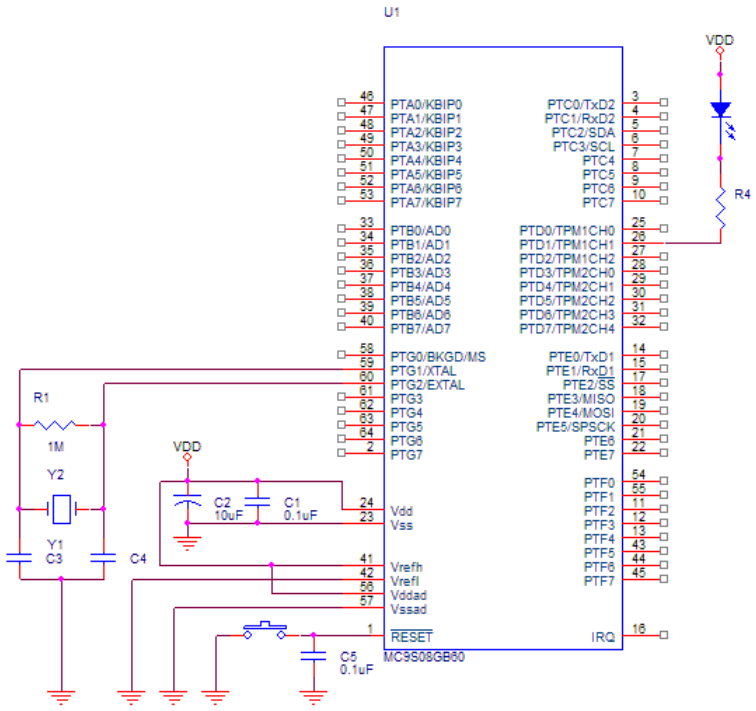



**Figure 2. Schematic of Circuit Used in Example 2**

**NOTE**

It is important to notice that the software presented here was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and tested using a MC9S08GB60 running in self-clocked mode. Coding changes may be needed to initialize another MCU. It is important to consider that every microcontroller needs an initialization code which depends on the application and the microcontroller itself.

# Generating PWM Signals Using the HCS08 Timer (TPM)

by: Miguel Agnesi Meléndez
RTAC Americas
México 2005

## 1 Overview

This is a quick reference for enabling the PWM functionality of the timer module for an HCS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. This example may be modified to suit your application — refer to the data sheet for your device.

**Table of Contents**



**TPM Quick Reference**

Because there is more than one TPM modules on some devices, there are two full sets of registers. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the registers that are on TPM1 from those on TPM2. A small n in a register name below is a place-holder for the channel number.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxSC | TOF | TOIE | CPWMS | CLKSB | CLKSA | PS2 | PS1 | PS0 |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | **BIT9** | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Any write to TPMCNTH or TPMCNTL clears the 16-bit counter

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | **BIT9** | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Modulo value for TPM module; read or write

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnSC | CHnF | CHnIE | MSnB | MSnA | ELSnB | ELSnA | | |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnVH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | **BIT9** | BIT8 |
| TPMxCnVL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Captured TPM counter of input capture function OR output compare value for output compare of PWM function

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

## 2.1 Generating PWM Signals with Common Duty Cycles

Configuring the TPM to generate PWM signals with common duty cycles is straightforward:

1. Load the desired period for all channels on the base timer TPMMOD register.
2. Load the desired duty cycle for each channel on the TPMCnV registers.
3. Select the PWM functionality for each channel that will be used to generate PWM by using the TPMCnSC register of each channel.
4. Select the PWM mode, input clock and prescaler for the main timer in the TPMSC register.

## 2.2 Generating PWM Signals While Changing the Duty Cycle

The project PWM_GB60 implements the TPM module as PWM generator. The main functions are:

- main — Endless loop waiting for timer interrupts
- MCU_init — Configures the hardware and the TPM module as PWM generator
- Vspi1_isr— Responds to the “Receive Full” interrupt
- Vtpm1ch1_isr— Function used to change the PWM’s duty cycle in each interrupt routine service.

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

This is a descriptive example of the TPM module written for the MC98S08GB60 microcontroller. The example will toggle an LED with a varying duty cycle modified every TPM period. The configuration includes the following features:

- PWM period of 524ms (bus clock as source clock, prescaler value of 32, module counter value of 0xFFFF).
- Reset duty cycle value of 0x0F00 (increments a value of 0x1000 on every period)
- PWM is configured to be left-aligned, output pin to be controlled by channel 1 and cleared when channel value is matched.
- Channel interrupt enabled. When serviced, the duty cycle will be incremented a value of 0x1000 on each interrupt request until the max value is reached (0xFFFF), then the initial duty cycle value is restored.

This is accomplished in the device initialization with this initialization code:

```
TPM1MOD = 0xFFFE;       /*Modulo value */
TPM1C1V = 0x0F00;       /*Reset Channel value*/
TPM1C1SC = 0x68;        /*Channel interrupt enabled, PWM mode, clears
                          output on channel value match*/
TPM1SC = 0x0D;          /*Overflow interrupt disabled, edge-aligned
```

```
                        PWM, bus clock selected as source, prescaler
                        value of 32*/
```

This example shows the PWM capabilities by modifying the channel's duty cycle each time the channel interrupt is serviced. The interrupt handler for channel 1 clears the CH1 flag and modifies the duty cycle of channel 1.

```
__interrupt void Vtpm1ch1_isr(void)
{

      TPM1C1SC_CH1F=0;      /* ACK channel interrupt */
                               /* Read flag, then write a 0 to the bit. */

      if (TPM1C1V <= 0xF000) {

          TPM1C1V = TPM1C1V + 0x1000; /* modifies PWM's duty cycle */

      } else {

          TPM1C1V = 0xF00;            /* resets value when max value reached */
      }
}
```

This interrupt function is automatically generated and initialized in a vector array in MCUinit.c by the device initialization tool if the option is enabled. The user must define its contents.

# 3 In-Depth Reference Material

The HCS08 timer module is composed of a 16-bit base counter with one or more channels linked to it. The base counter acts as a reference, shared among all the linked channels. The channels can be independently configured, allowing the user to enable the desired functionality for each channel:

- Capture a time stamp of the base timer to the channel value register when an external event occurs (input capture mode).
- Generate an interrupt or modify an MCU pin value when the base counter reaches a predefined value on the channel value register (output compare mode).
- Pulse-width modulation (PWM) with duty cycle defined for each channel based on a combination of the channel value register and the base timer modulo register.

When using the timer to generate PWM signals, the base timer is used to set the PWM period (which is common to all channels because this counter is the reference), and each channel can be configured to handle a PWM signal with its own duty cycle using the channel value register.

The module can generate an interrupt each time the period is matched in the base timer and each time the duty cycle is reached on any channel. Each channel has its own interrupt vector address so the interrupts can easily be mapped to a specific handler routine.

The timer allows two PWM operation modes:

- Edge-aligned mode

- Center-aligned mode

Edge-aligned PWM operation will count from 0 to the value stored in the base timer modulo register (TPMMOD) resetting the counter, changing the output level and setting the overflow flag when this value is reached. The module will change the output level of each channel again when the base counter equals the value stored in the respective channel value register (TPMCnV), which is an output compare event, as shown in Figure 1.

The level will be changed according to the settings in the channel status and control register:

- If the low-true pulses option is selected, the output level will be set to low when the counter resets and will be set to high when the base counter equals the channel value register.
- If high-true pulses is selected, the output level will be set to high when the counter resets and will be set to low when the base counter equals the channel value register.

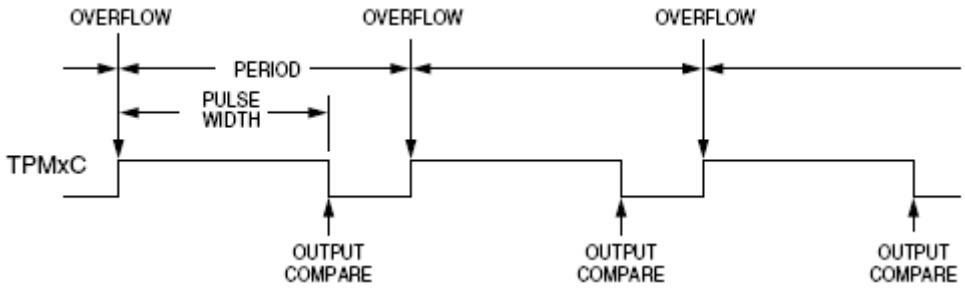



**Figure 1. . Edge-Aligned PWM**

When the timer is configured for center-aligned PWM operation, the timer will count from the value stored in the base timer modulo register (TPMMOD) down to 0 and then up again to the value stored in the base timer modulo register. This changes the output level of each channel when the counter equals the value stored in the respective channel value register (TPMCnV), which is the output compare of the channel, as shown in Figure 2.

When center-aligned mode is selected all the channels linked to this timer and configured as PWM will operate in center-aligned mode.

Notice that when using center-aligned mode, using a 0x0000 value is not allowed. A value higher than 0x7FFF in the modulo register is not recommended because it can generate ambiguous results.

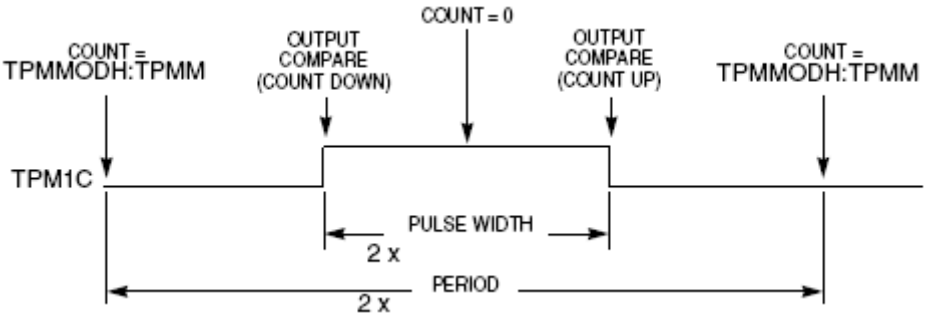



**Figure 2. . Centered PWM Operation with (ELSnA = 0)**

**NOTE**

This example code was developed using the’ CodeWarrior IDE version 5.0 for HC08, and was tested on the MC9S08QG8 device using device initialization. Interrupt vectors must be modified to fit the specific MCU vector table, which is located in the vectors and interrupts section of the data sheet for each MCU.

# Programming and Erasing Flash Memory on HCS08 Family Microcontrollers

by: Gonzalo Delgado
RTAC Americas
México 2005

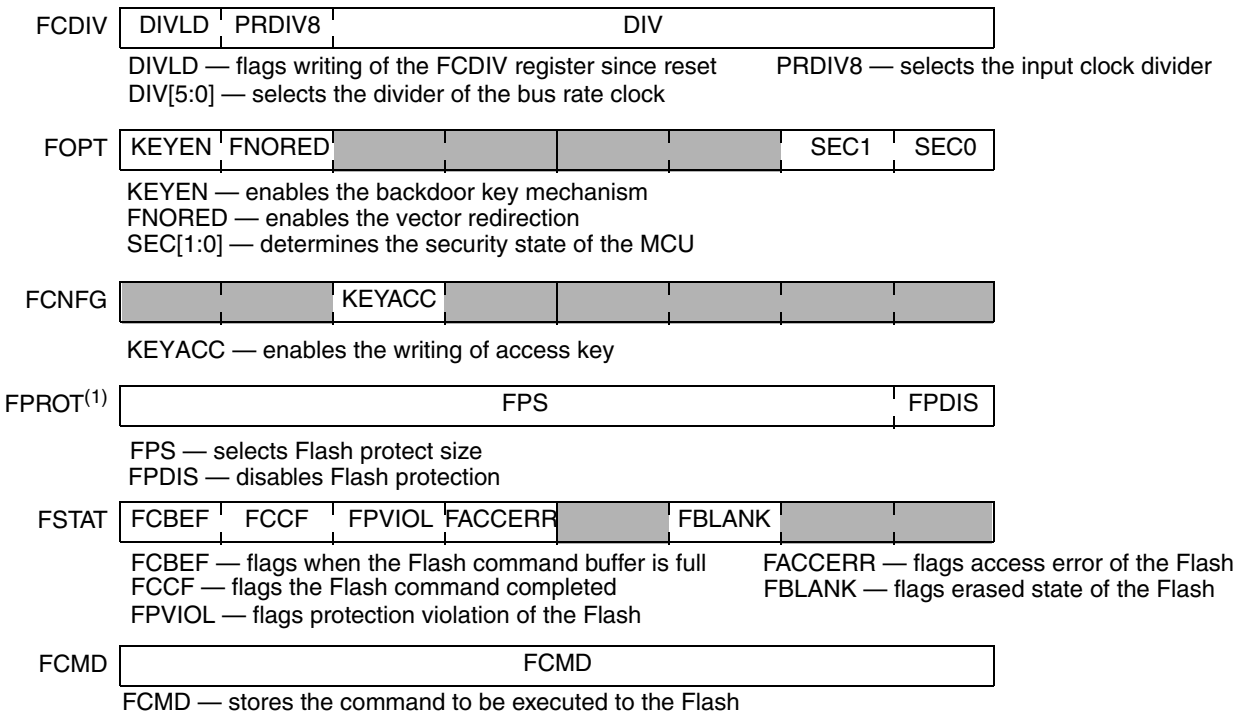
## 1 Overview

This document is a quick reference for programming and erasing the Flash memory included in the HCS08 Family microcontrollers (MCUs). Basic information about the functional description and configuration are provided. The example may be modified to suit the specific needs for your application — refer to the data sheet for your device.

**Table of Contents**



**Flash Quick Reference**




1. FPROT may contain different bits, depending on your device — refer to your data sheet.

Because the application code resides in the Flash memory and is executed from there, it is not possible to program/erase the same block of the Flash that is being read. To program/erase Flash, the code could be placed in RAM and then executed from there. The example code shows how it is done.

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

In this example, the MCU will program byte from 0 to 127 into the Flash memory and erase them after their successful execution. Because the Flash memory can't be programmed while it is being used, an array that contains the opcode[1] of the instructions to erase and program will be defined in a specific address in RAM memory for their execution. This opcode array can be used to program/erase the Flash memory in the members of the HCS08 family. Programming/erasing times have to be very precise in order to extend the life of the Flash and that is why specific and precise instructions are needed. The array of opcode instructions in this particular example will only need 59 bytes of the RAM and 4 bytes of the stack.

- Main — Endless loop for erasing and programming Flash memory address 0x1200.
- MCU_Init — This routine initializes the clock and flash registers.
- Page_Erase — This routine erases the page where the address given is located.
- Program_Byte — This routine programs a byte of data to a given address.

Following the next steps, the user will program and erase the Flash memory:

1. Configure the Flash clock divider and clock registers (using an internal bus clock of 4 MHz, and setting the Flash clock to be 200 kHz):
   (This section was done with the device initialization tool)
2. Declare the array with the opcode of the Erase and Program instructions in RAM:

   It is very important to know that before the programming/erasing subroutine is called the stack is being used to store the byte to be programmed in the Accumulator and the address to program in the HX register. After the subroutine is finished, an error variable is stored in the accumulator (if it is 0xFF, an error occurred)

```
//Array of opcode instructions of the Erase/Program function

void MCU_init(void)
{
  /* ### MC9S08QG8_16 "Cpu" init code ... */
  /*  PE initialization code after reset */
  /*  System clock initialization */
  /* SOPT1: COPE=0,COPT=1,STOPE=0,BKGDPE=0,RSTPE=0 */
  SOPT1 = 0x50;
  /* SPMSC1: LVDF=0,LVDACK=0,LVDIE=0,LVDRE=1,LVDSE=1,LVDE=1,BGBE=0 */
  ICSC1 = 0x04;                        /* Initialization of the ICS control register 1 */
  /* ICSC2: BDIV=1,RANGE=0,HGO=0,LP=0,EREFS=0,ERCLKEN=0,EREFSTEN=0 */
  ICSC2 = 0x40;                        /* Initialization of the ICS control register 2 */
  /* Common initialization of the write once registers */
```

1.Opcode is the numeric value of the assembly instructions, for more information refer to the HCS08 Family Reference Manual.

```
  /* SOPT2: COPCLKS=0,IICPS=0,ACIC=0 */
  SOPT2 = 0x00;
  /* FCDIV: DIVLD=0,PRDIV8=0,DIV5=0,DIV4=1,DIV3=0,DIV2=0,DIV1=1,DIV0=1 */
  FCDIV = 0x13;
```

It is very important to know that before the programming/erasing subroutine is called the stack is being used to store the byte to be programmed in the Accumulator and the address to program in the HX register. After the subroutine is finished, an error variable is stored in the accumulator (if it is 0xFF, an error occurred)

```
//Array of opcode instructions of the Erase/Program function

//Element 0x14 of the array is: (command 0x20 to program a byte, 0x40 to erase a page)
unsigned char FLASH_CMD[] {
0x87,0xC6,0x18,0x25,0xA5,0x10,0x27,0x08,0xC6,0x18,0x25,0xAA,0x10,0xC7,0x18,0x25,
0x9E,0xE6,0x01,0xF7,0xA6,0x20,0xC7,0x18,0x26,0x45,0x18,0x25,0xF6,0xAA,0x80,0xF7,
0x9D,0x9D,0x9D,0x9D,0x45,0x18,0x25,0xF6,0xF7,0xF6,0xA5,0x30,0x27,0x04,0xA6,0xFF,
0x20,0x07,0xC6,0x18,0x25,0xA5,0x40,0x27,0xF9,0x8A,0x81};

/* The opcode above represents this set of instructions
  if (FSTAT&0x10){            //Check to see if FACCERR is set
     FSTAT = FSTAT | 0x10;        //write a 1 to FACCERR to clear
  }
  (*((volatile unsigned char *)(Address))) = data; //write to somewhere in flash

  FCMD = 0x20;                           //set command type.
  FSTAT = FSTAT | 0x80;           //Put FCBEF at 1.
  _asm NOP;                  //Wait 4 cycles
  _asm NOP;
  _asm NOP;
  _asm NOP;
  if (FSTAT&0x30){            //check to see if FACCERR or FVIOL are set
  return 0xFF;            //if so, error.
}
  while ((FSTAT&0x40)==0){        //else wait for command to complete
     ;
}*/
```

3. Disable interrupts to permit execution of the code

```
DisableInterrupts;
```

4. Cycle that writes a byte form value 0 to 127

```
for(counter=0;counter<=127;counter++)
  {
     Program(0x1200 + counter, counter);
  }
```

5. Program one byte in the Flash memory

```
  void Program(int Address, unsigned char data){
      unsigned char dummy;

       asm jsr FLASH_CMD; //jumps to where the Program Routine is located at
       asm sta dummy;
       if (dummy == 0xFF){
          asm NOP;      } //An error occurred during the Programming of the FLASH
              }
```

6. Erase a 512 byte page starting where the first value was written

```
void Erase(int Address){
      unsigned char dummy;
      FLASH_CMD[21] = 0x40;     //Erase command is written into the array

      asm jsr FLASH_CMD;      //jumps to where the Erase Routine is located at
      asm sta dummy;
if ( dummy == 0xFF){ asm NOP;}  //An error occurred during the Erasing of the FLASH
}
```

**NOTE**

- This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 using Device Initialization and was expressly made for the MC9S08QG8. Changes may be required before using the code with other MCUs.
- Verify the memory configuration of the RAM and the Flash for the microcontroller you're using. The size of the memory will depend on the specific configuration of the MCU. Refer to the data sheet for your device.
- The opcode array may be used in members of the HCS08 Family with no modification at all.

# Implementing Interrupt Service Routines (ISR) in C Using CodeWarrior for the HCS08 Family Microcontrollers

by: Laura Delgado
RTAC Americas
México 2005

## 1 Overview

This document is a quick reference to interrupts in CodeWarrior CW08. It provides examples that describe how to initialize interrupts and define their service routines. The example may be modified to suit the specific needs for your application — refer to the data sheet for your device.

**Table of Contents**



**TPM Register Model**

Because there is more than one TPM modules on some devices, there are two full sets of registers. In the register names below, where there's a small x, there would be a 1 or a 2 in your software to distinguish the registers that are on TPM1 from those on TPM2. A small n in a register name below is a place-holder for the channel number.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxSC | TOF | TOIE | CPWMS | CLKSB | CLKSA | PS2 | PS1 | PS0 |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Any write to TPMCNTH or TPMCNTL clears the 16-bit counter

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCNTH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCNTL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Modulo value for TPM module; read or write

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnSC | CHnF | CHnIE | MSnB | MSnA | ELSnB | ELSnA | | |

Interrupt enable and module configuration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TPMxCnVH | BIT15 | BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 |
| TPMxCnVL | BIT7 | BIT6 | BIT5 | BIT4 | BIT3 | BIT2 | BIT1 | BIT0 |

Captured TPM counter of input capture function OR output compare value for output compare of PWM function

# 2 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

In this application, TPM1 of the MC9S08GB60 microcontroller will be used to generate two kinds of interrupt requests to be serviced by two ISRs. Both ISRs will increment a value of 1 to a variable each time they are serviced.

The functions for project Interrupt.mcp are:

- main — Endless loop waiting for a Timer interrupt events.
- MCU_init — MCU initialization and timer module initialization (configures and enables the TPM module and two interrupt sources: timer overflow flag and channel interrupt flags are enabled).
- Vtpm1ch0_isr — Interrupt function where the channel flag is cleared and a value of 1 is added to variable VarA and an LED is toggled for visual display.
- Vtpm1ovf_isr — Interrupt function where the overflow flag is cleared and a value of 1 is added to variable VarB and an LED is toggled for visual display.

MCU_init is a function generated by device initialization and is located in MCUinit.c also generated by the device initialization, which was included in the project.

This is the initialization code for the timer using the MC9S08GB60.

```
TPM1MOD = 0x7FFF; /* sets the number in which the counter will be reset.*/
TPM1C0V = 0x0FFF;  /* sets the number that, if matched by the  counter, will
                        set the channel flag*/

TPM1C0SC = 0x54;  /* sets the mode for the channel and enables channel flag */

TPM1SC = 0x4D;     /*sets timer frequency and enables overflow flag */
```

After the module is initialized, its interrupt sources are enabled and the global interrupt mask is disabled, whenever a timer interrupt request occurs, the ISR is executed. In this case, we handle two interrupts: channel and overflow interrupts. Every interrupt is assigned to one interrupt vector each. For example, for the MC9S08GB60 microcontroller, the vectors that handle these events are vector 8 for the timer 1 overflow flag and vector 6 for the timer 1 channel flag, as shown in Table 1.

**Table 1. Timer 1 Channel and Overflow Interrupts Vectors for the MC9S08GB60 Microcontroller**

| Vector Number | Address (High/Low) | Vector Name | Module | Source | Enable | Description |
|---|---|---|---|---|---|---|
| 8 | $FFEE/FFEF | Vtpm1ovf | TPM1 | TOF | TOIE | TPM1 overflow |
| 6 | $FFF2/FFF3 | Vtpm1ch1 | TPM1 | CH1F | CH1IE | TPM1 channel 1 |

After the vector numbers are identified, the interrupt functions can be defined. The interrupt routines acknowledge the interrupt and add a value of 1 to a variable

```
void interrupt 5 Vtpm1ch0_isr (void){
     TPM1C0SC_CH0F = 0; /* ACK channel interrupt */
                        /* Reading flag, then write a zero to the bit. */

     VarA++;

     PTFD_PTFD0 = ~PTFD_PTFD0;
}
void interrupt 8 Vtpm1ovf_isr (void){
     TPM1SC_TOF = 0; /* ACK timer overflow interrupt */
                     /* Reading flag, then write a zero to the bit. */

     VarB++;

     PTFD_PTFD1 = ~PTFD_PTFD1;
}
```

Figure 1 shows the code in project interrupts.mcp simulated by the debugger. Five memory displays have been opened to show the values for VarA, VarB, vector 6, vector 8, and timer registers. As well, two breakpoints where set to locate the beginning of the ISRs in the assembly window. The memory locations for the vectors and timer registers are shown in the data sheet for each device.

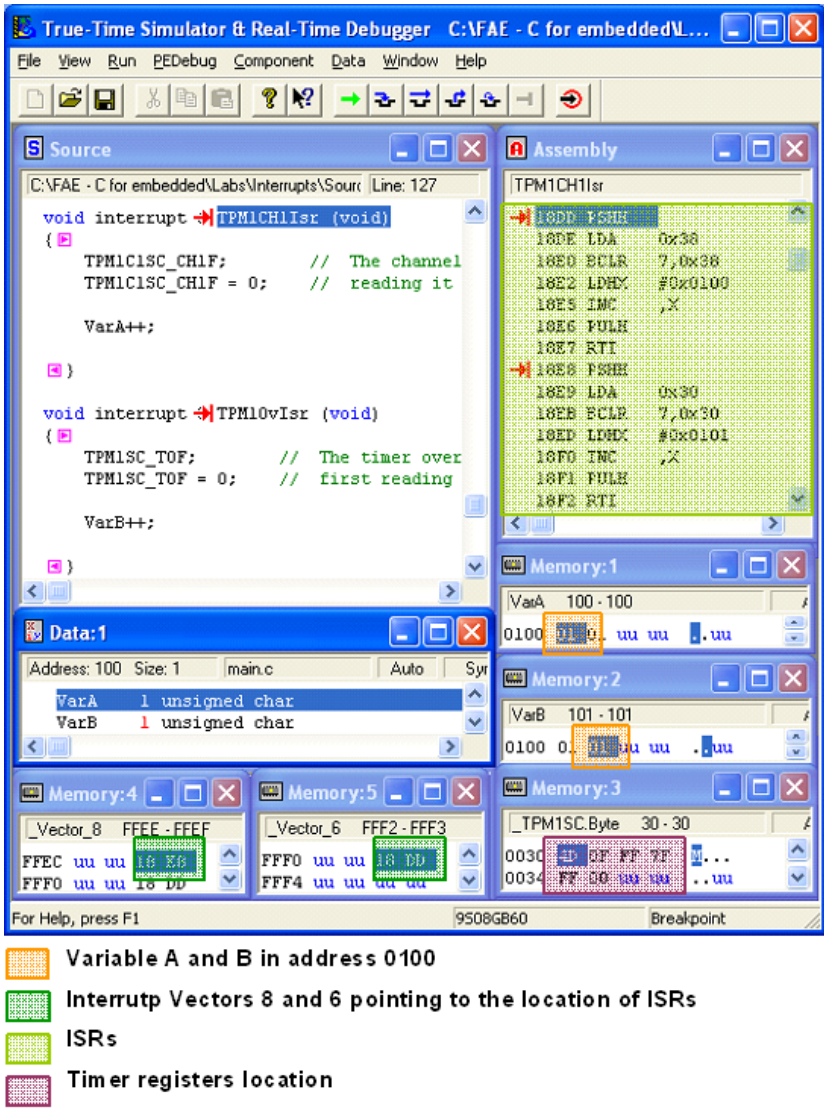



**Figure 1. CodeWarrior's True Time Simulator and Real Time Debugger**

# 3 In-Depth Reference Material

The information in this section is provided as reference material for those who would like to learn more about interrupt functionality in the HCS08 Family of MCUs.

## 3.1 Interrupts

Exceptions are events that change normal flow of a software program. In the case of Freescale's microcontrollers, these events could be a reset instruction or a timeout for the COP watchdog. Interrupts are one type of exception, in which an exceptional event is responded to with an interrupt service routine (ISR). Most of Freescale's 8-bit microcontrollers have several sources of interrupts.

## 3.2 Interrupt Vectors

The set of interrupt sources differs in each microcontroller: timers, peripherals, and input pins are the most common interrupt sources. Vectors are assigned to classify these interrupt sources. Each vector contains the address where its respective ISR is located in memory. Refer to the Reset and Interrupt Vectors table in Section 4, “Memory,” in the data sheet for your device.

Vector numbers are given according to priority. As priority decreases, the vector number increases. The reset instruction is always the highest priority interrupt for all MCUs: it always has the vector number 0 assigned. Not all vector summaries contain the vector numbers, but it can be deduced with the priority order. If the vector you want to use is the third highest priority, its vector name will be 2, and so on. When programming ISRs, having the vector number is essential because it is used to identify the interrupt source referenced.

## 3.3 Implementation

There are three ways to ways to handle interrupt functions:

- Definition of an interrupt function
- Initialization of a vector table
- Placing interrupt function in special memory sections

This document will elaborate only on the definition of an interrupt function. For more information on alternate procedures to achieve interrupt response, refer to CodeWarrior’s HC08 Compiler Manual in the section, “Defining Interrupt Functions.”

There are two main steps to defining an interrupt function:

- Initialization of interrupt source
- Definition of interrupt service routine

During normal operation and if the interrupt mask is disabled, the CPU checks all pending interrupts after every instruction. If more than one interrupt is pending, the highest priority one is serviced first. Every time an interrupt request is made, the interrupt mask is set. After the ISR is serviced, the global interrupt mask is cleared. If the user wants a higher priority event to interrupt the ISR from a lower priority event, the ISR will have to clear the global interrupt mask.

When a qualified interrupt request is made, the CPU completes the current instruction and performs the following steps:

1. Saves the CPU registers (program counter (PC), index register (H:X), accumulator (A), and condition code register (CCR)) on stack.
2. Sets interrupt mask to prevent further interrupts to occur during the ISR.
3. Fetches the interrupt vector for the highest priority.
4. Loads the program counter with the interrupt vector address.
5. Processing continues in the ISR.

### 3.3.1 Initialization of Interrupt Source

Most interrupt sources are part of a module (for example, the timer module, SCI module, ADC module, etc.). Each module has configuration and status registers that select the interrupt-triggering event, and alert when it occurs. Interrupt sources are generally enabled in these registers. The specific configuration will depend on the module and microcontroller (refer to the specific data sheet for more information).

Although each module manages the configuration of its own interrupt sources, there is a control bit in the CPU condition code register (bit I in CCR) that disables all interrupts when set. It is called the global interrupt mask. C doesn't provide a direct tool to accessing the CPU registers. The CodeWarrior CW08 includes the hidef.h library which contains the instructions that manipulate the global interrupt mask:

- EnableInterrupts; — clears the global interrupt mask
- DisableInterrupts; — sets the global interrupt mask

These two instructions, as their names state, enable/disable interrupts. The C compiler also allows the use of assembly instructions within the C code: CLI (enable interrupts), SEI (disable interrupts). In most 8-bit microcontrollers, after any reset, the global interrupt mask is set by default. To clear it, the EnableInterrupts; instruction must be used. It is a good practice to keep the global interrupt mask set during all modules initialization because doing so avoids unwanted interrupt requests while general initialization is in process. This practice will be applied further in this example.

### 3.3.2 Definition of ISR (Interrupt Service Routine)

An interrupt function is defined the following way:

```
void interrupt vector_number function_name ( void ) {
      Flag acknowledgement and
      Interrupt Service Routine are included inside this function
}
```

The interrupt function is where the ISR is executed. The interrupt function name can be chosen by the user. The vector number defines what interrupt source will call that particular interrupt function. It is very important for the user to make sure that the specified vector number matches the wanted interrupt source. For example, if you are monitoring timer module interrupts, for instance, the overflow event, the vector number for the vector that handles the overflow event is needed.

Vector number designation can sometimes be tricky, because different microcontrollers often have different interrupt characteristics: sometimes they handle different vector numbers for similar events, this will have to be taken in consideration when migrating from one device to another. After the vector number is correctly set and the interrupt function is defined, the program is ready to service the interrupt routine every time the wanted event is detected.

Generally if an event occurs in an enabled interrupt source, an associated flag will become set and the interrupt function will be called. The ISR should always include the interrupt flag clearing or acknowledging, otherwise, the corresponding flag will stay set and recalling the ISR becomes impossible. Depending on the microcontroller and module, flag acknowledgment is made in different ways (e.g. reading a register, writing a acknowledgment bit, writing and reading registers), for more information refer to the respective data sheet. Some interrupt vectors handle several interrupt sources, therefore several

flags. For this case, the ISR will have to check the flag bits to determine which of the sources caused the interruption.

**NOTE**

- This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 and was expressly made for the MC9S08GB60. Changes may be needed in the code before it can be used with other MCUs.
- Critical section codes can be protected from unwanted interrupt requests with a NOP instruction after the interrupt masking instruction.

**Freescale Semiconductor**
Users Guide

# Memory Mapping for HCS08 Family MCUs Using CodeWarrior Software

by: Laura Delgado
RTAC Americas
México 2005

## 1 Overview

This document is a quick reference for customizing memory map settings using CodeWarrior in the HCS08 Family microcontrollers (MCUs). Basic information about the functional description and configuration are provided. The example may be modified to suit the specific needs for your application — refer to the data sheet for your device.

Figure 1 shows a general memory map. Most 8-bit microcontrollers contain these memory sections.

**Table of Contents**



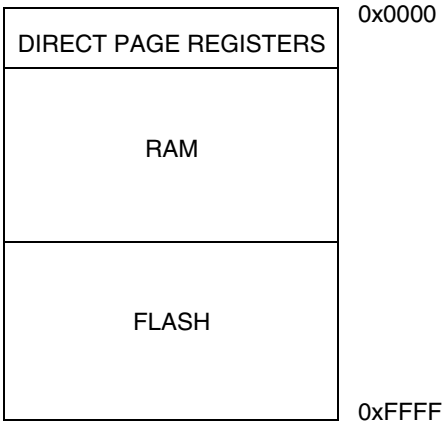



**Figure 1. General Memory Map**

## 1.1 Direct-Page Registers

This section uses direct addressing mode. Direct instructions are used to access operands in the direct page, for example, in the address range 0x0000 to 0x00FF. The high-order byte of the address is not included in the instruction, thus saving one byte and one execution cycle compared to extended addressing. Also, because the bit manipulation instructions support only direct addressing mode, this simplifies management of control and status bits within the system's input/output and configuration registers for the MCU, most of which are in the direct page.

## 1.2 RAM

RAM is where read/write objects[1] are stored (the stack is placed in this memory area). Depending on the microcontroller, RAM could have some space in the direct-page area. This allows the user to have bit addressable variables as well as the most used program objects to be handled more efficiently. When using RAM located in the direct page, the compiler will optimize the code using direct addressing mode (8-bit address) instead of extended mode (16-bit address). Use this area for most used variables in a program.

## 1.3 Flash

Flash memory is intended primarily to store program code. Additionally, CodeWarrior saves read-only objects (e.g., constant variable) in Flash, because they are not meant to be changed. Located in the last memory locations, interrupt vectors are also contained in the Flash area. Every interrupt vector has a 2-byte register that contains ISR address information.

# 2 Linker and Parameter Files

CodeWarrior's software architecture includes several foundation files. These files help characterize the context in which the MCU is working: peripheral definitions (*.h), linker parameter files (*.prm), ANSI C libraries (*.lib), initialization route files. For the purpose of this document, we will elaborate on linker files (PRM files).

Linking is the process of assigning memory to all global objects needed for a given application and combining these objects into a format suitable for downloading into a target system or an emulator. PRM files translate the microcontroller memory map to a linker-readable format. Although the linker takes almost complete control over the placement of objects in memory, it is possible to allocate different groups of functions, variables or constants to different memory areas, this is called segmentation. In the PRM file, segments are set down to establish where you want to allocate certain objects you have defined in your source code. We will further illustrate this with an example.

PRM file contents may vary according to the MCU specific memory map. Figure 2 shows the memory map for the MC9S08GB60 microcontroller and the respective linker file. The SEGMENTS section in the PRM file complies with the memory map.

1. For this document the word *object* is understood as functions, global data, strings, constants or initialization data.

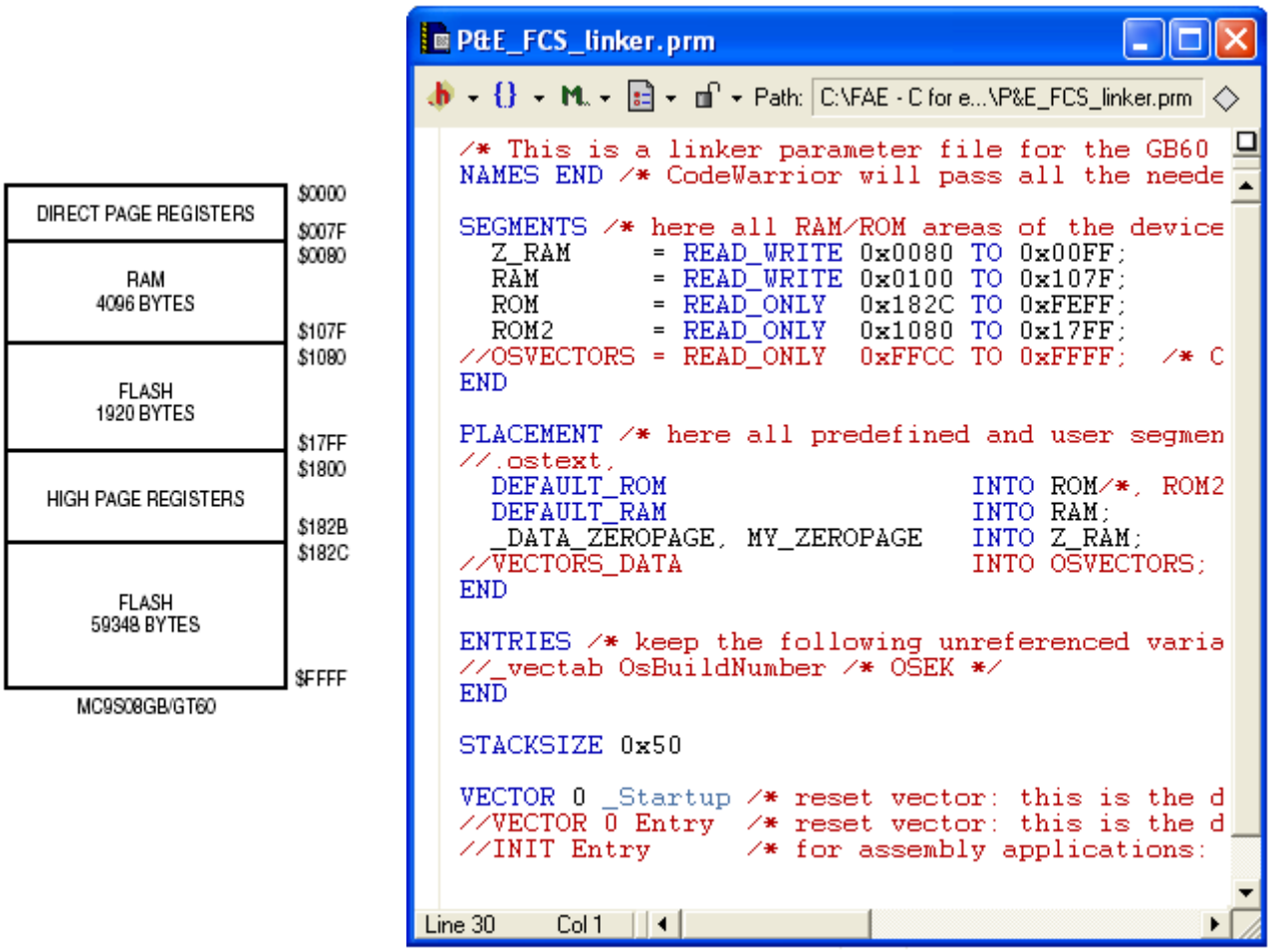



**Figure 2. Memory Map and Parameter File for MC9S08GB60**

The linker file for every MCU has its default settings for memory assignment. In the case of the MC9S08GB60 microcontroller, default ROM space is between 0x182C and 0xFEFF, default RAM space is between 0x0100 and 0x107F, as Figure 2 shows. The rest of the segments won't be used unless referenced, as _DATA_ZEROPAGE, which is the RAM located in the direct address area. When new segments are planned to be addressed, they will have to be made in the PRM file and then referenced in the source code.

# 3 Implementation

Planned object allocation has two main prerequisites:

- To have memory areas defined in the PRM file
- To state which objects are to be allocated in which memory area in the source code

## 3.1 Defining Memory Areas

The whole object allocation is performed through the SEGMENTS and PLACEMENT blocks in the PRM file. The SEGMENTS block is written in the following way:

```
SEGMENTS

        Segment definitions;

END
```

The SEGMENTS block describes the memory map for a certain MCU with a list of all Segment definitions. Segments are defined in the following way:

```
Segment_Name = Segment_Qualifier address TO address ;
```

Because code and data segments are the most used segment types, we are only covering the next segment qualifiers:

- READ_WRITE – for read/write memory segments (i.e., RAM)
- READ_ONLY – for read-only memory segments (i.e., ROM)
- NO_INIT – for read/write memory that is to remain unchanged at startup.

For more information on segment qualifiers, refer to the smart linker manual. The PLACEMENT block allows users to physically place each section from the application in a specific segment. Actually, you can have many sections allocated in one memory segment. The PLACEMENT block is written in the following way:

```
PLACEMENT

        Section placement;

END
```

Section placement is made in the following way:

```
Section_Name1 INTO Segment_Name ;
```

In the case of many sections being placed in the same segment:

```
Section_Name1, Section_Name2, Section_Name3 INTO Segment_Name ;
```

For this instruction, in *Segment_Name*, the objects defined in the section *Section_Name1* are first allocated, then the objects defined in *Section_Name2*, finally the objects defined in *Section_Name3*.

In a similar way, you can place one section in many segments:

```
Section_Name INTO Segment_Name1, Segment_Name2, Segment_Name3 ;
```

For this instruction, *Section_Name* will be allocated first in *Segment_Name1*. When *Segment_Name1* is full, the allocation will continue in *Segment_Name2*. Finally when that segment is full, allocation will continue in *Segment_Name3*.

**NOTE**

It is very important for the user to have the MCU memory map at hand to be sure not to invade configuration and I/O registers, as well as other reserved memory locations. If new segments are to be created, it's a good practice to constrain them within the RAM/ROM memory segments boundary.

## 3.2 Referencing Sections in the Source Code

The CodeWarrior compiler allows attributing a certain segment name to certain global variables or functions, which then will be allocated into that segment by the linker. As mentioned before, where that segment actually lies is determined by an entry in the linker parameter file.

In CodeWarrior software, objects are allocated into the default placements unless otherwise stated with a *#pragma* directive. Because there are two basic types of segments, code and data segments, there are also two basic pragmas to specify segments:

```
#pragma CODE_SEG section_name
#pragma DATA_SEG section_name
```

In addition, there are pragmas for constant data and for strings:

```
#pragma CONST_SEG section_name
#pragma STRING_SEG section_name
```

If no segment is specified, the compiler assumes two default sections named DEFAULT_ROM (the default code segment) and DEFAULT_RAM (the default data segment). If a segment (other than default) has been already specified and you want to return the segment to the default memory allocation, use the segment name DEFAULT to explicitly make these default segments the current segments. This will be better illustrated in the example.

## 3.3 Alternate Option

There is another way to assign global variables to specific addresses. With the next instruction, variables can directly be addressed into a specific address number:

```
#define Var_Name (*(Type *) Address);
```

CodeWarrior has the *global variable address modifier* @ for the same purpose[1]. Direct variable allocation is completed using the following, more simple syntax:

```
Type Var_Name @ Address;
```

Where:

- *Type* is the type specifier; for example, int, char.
- *Var_Name* is the identifier of your global varible.
- *Address* is the address where the global variable is to be allocated.

1. The @ modifier is a non-ANSI operator. Take this into consideration when migrating to other compilers.

Sometimes it is useful to have the variable directly allocated in a named segment. To do this, pragma directives are first stated to make reference of all the sections/segments that are to be used, then any direct allocation(s) using the “@” modifier can be made, without any particular order:

```
#pragma DATA_SEG section1_name
#pragma DATA_SEG section2_name

Type Var1_Name @ "section1_name" = Initializer;
Type Var2_Name @ "section2_name" = Initializer;
Type Var3_Name @ "section1_name" = Initializer;
```

This will be illustrated in the example.

# 4 Code Example and Explanation

This example code is available inside the CodeWarrior project or from the Freescale Web site in HCS08QRUGSW.zip.

In this application, different objects are going to be allocated in memory with different procedures. This example has been created based on the memory map for the MC9S08GB60 microcontroller.

For this example we will reference two of the files included in the project MemAlloc.mcp: the linker parameter file (P&E_FCS_linker.prm) and then the source code (main.c) where data, constants, and code allocation is made.

## 4.1 PRM File

For this example, new segments (MY_RAM and MY_ROM) were created within both, RAM and ROM. In order to avoid the new segments unwanted overwriting, default RAM and ROM have been split into two segments with the help of two other new segments (RAM2 and ROM3). New Default RAM is RAM and RAM2, in a similar way, new ROM is the addition of ROM and ROM3. As well, already existent segment for the MC9S08GB60, _DATA_ZEROPAGE section, was used although it is not the default data allocation section. A new section (MY_CODE) was placed in an already existing segment (ROM2, this is not the default ROM). The section placement is here presented, to see the rest of the PRM file, refer to the CodeWarrior project.

```
SEGMENTS
      Z_RAM      = READ_WRITE 0x0080 TO 0x00FF;
      // Default RAM split into RAM and RAM2
      RAM        = READ_WRITE 0x0100 TO 0x01FF;
      MY_RAM     = READ_WRITE 0x0200 TO 0x0202;
      RAM2       = READ_WRITE 0x0203 TO 0x107F;

      // Default ROM split into ROM AND ROM3
      ROM2       = READ_ONLY  0x1080 TO 0x17FF;
      ROM        = READ_ONLY  0x182C TO 0xEFFF;
      MY_ROM     = READ_ONLY  0xF000 TO 0xF0FF;
      ROM3       = READ_ONLY  0xF100 TO 0xFEFF;
END

PLACEMENT

      DEFAULT_ROM                 INTO ROM,ROM3;
```

```
        DEFAULT_RAM                  INTO RAM, RAM2;
        _DATA_ZEROPAGE, MY_ZEROPAGE  INTO Z_RAM;
        MY_DATA                      INTO MY_RAM;
        MY_CONSTS                    INTO MY_ROM;
        MY_CODE                      INTO ROM2;

END
```

## 4.2 Data Allocation in Source Code

In the source code, MY_DATA section is first referenced because it will be used later for direct variable allocation with the global variable address modifier @. Then, *_DATA_ZEROPAGE* is referenced to state that data will afterwards be located in that section because *VarZeroSeg* is located in the zeropage area[1].

```
#pragma DATA_SEG MY_DATA
#pragma DATA_SEG _DATA_ZEROPAGE

unsigned char VarZeroSeg;

int VarNewSeg@"MY_DATA";
```

Data allocation is then restored to the default settings, where *VarDefSeg* is later allocated.

```
#pragma DATA_SEG DEFAULT
unsigned char VarDefSeg;
```

## 4.3 Constant Allocation in Source Code

For this example, two constant variables are initialized, one allocated in a new section created for this example, called MY_CONSTS (placed in the new segment MY_ROM), and the other allocated in the default ROM.

```
#pragma CONST_SEG MY_CONSTS
const unsigned char ArrayNewSeg[]={0xAA,0xAA,0xAA, 0xAA, 0xAA};
#pragma CONST_SEG DEFAULT
const int ArrayDefSeg[] = {0xBBBB, 0xBBBB};
```

CodeWarrior software has many optimization tools. One of them, *the constants replacement optimization*, must be disabled. Otherwise, the constant value won't be stored in memory and a direct replacement of its content will be used. This option can be found in *Optimizations*, in the compiler option settings. The compiler options are found in the target settings, in the Edit menu in CodeWarrior.

## 4.4 Code Allocation in Source Code

For this kind of allocation, the function prototype and definition will both have to be comprised by the #pragma directives CODE_SEG *segment_name*. In the example, the prototype is written in the following way

```
#pragma CODE_SEG MY_CODE
void FunctionNewSec(void);
#pragma CODE_SEG DEFAULT
```

1.For the MC9S08GB60, Z_RAM is a direct addressing mode memory segment.

and the function definition is written like this:

```
#pragma CODE_SEG MY_CODE

void FunctionNewSec(void){

    VarZeroSeg++;
}

#pragma CODE_SEG DEFAULT
```

In both, settings for code allocation are set back to default, which is considered to be a very good practice. *FunctionNewSec* is placed in ROM2, an already existent segment of Flash memory that is not set as default.

Figure 3 is an image of the CodeWarrior debugger. Five memory windows have been opened to show the location of the variables used in this application and a breakpoint was set to point to the address where the code for *FunctionNewSeg* was allocated.

Table 1 summarizes the type and location for every object that was allocated in this example. The windows, in which each variable's location is shown in Figure 3, are also listed.

**Table 1.**

| Var | Type | Window | Section | Segment |
|---|---|---|---|---|
| VarZeroSeg | char | Memory:1 | _DATA_ZEROPAGE | Z_RAM (0x0080 to 0x00FF) |
| VarNewSeg | int | Memory:2 | My_DATA | MY_RAM (0x0200 – 0x0202) |
| VarDelSeg | char | Memory:3 | DEFAULT_RAM | RAM (0x0100 – 0x01FF),<br>RAM2 (0x0203 – 0x107F) |
| ArrayNewSeg[] | const char | Memory:4 | MY_CONSTS | MY_ROM (0xF000 – 0xF0FF) |
| ArrayDefSeg[] | const int | Memory:5 | DEFAULT_ROM | ROM (0x182C – 0xEFFF)<br>ROM3 (0xF100 – 0xFEFF) |
| FunctionNewSec | void | Assembly | MY_CODE | ROM2 (0x1080 – 0x17FF) |

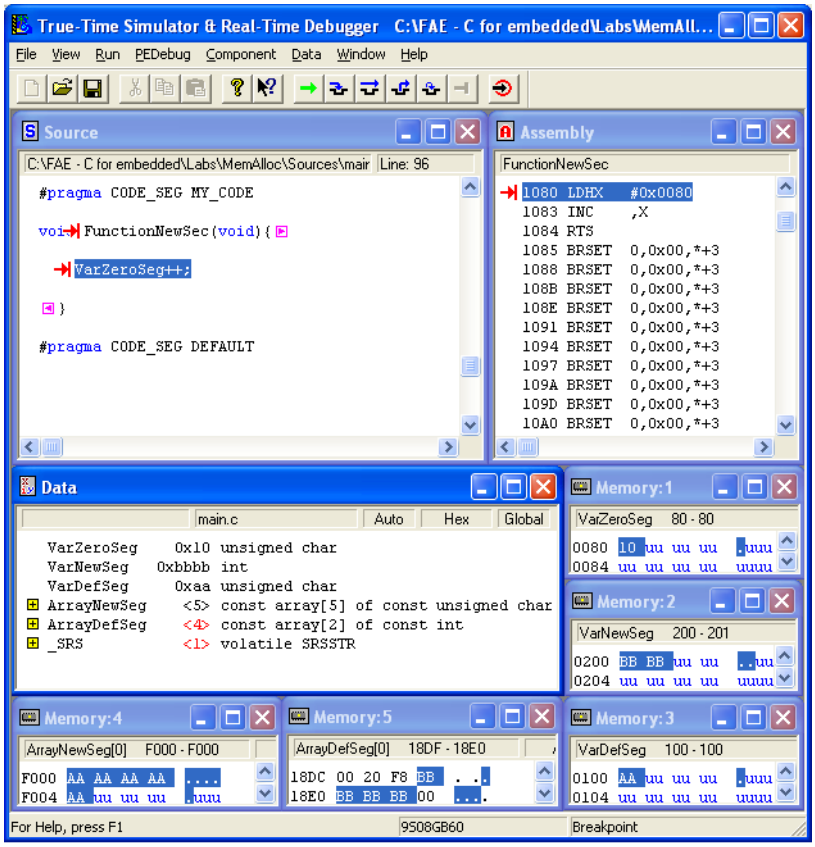



**Figure 3. Debugger Window**

The file P&E_FCS.map located in the project shows a more detailed description of where sections, vectors, and objects have been allocated in memory.

**NOTE**

This software was developed using the CodeWarrior Development Studio for HC(S)08 version 5.0 and was expressly made for the MC9S08GB60. Changes to the code may be required before it can be used with other MCUs.

## *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



HCS08QRUG
Rev. 1, 2/2006

freescale™
semiconductor