# LAB 2: Developing and Debugging C Programs in *CodeWarrior* for the HCS12 Microcontroller

The objective of this laboratory session is to become more familiar with the process for creating, executing and debugging application programs, written in the C language, for the Freescale MC9S12C32 (HCS12) microcontroller, using the *CodeWarrior* Development Studio. You are to design a C program, containing a "main" program and two subroutines, to exercise various I/O ports and elements of the microcontroller. In the lab, you will exercise a number of the debug support elements of CodeWarrior, so that you will be better prepared for debugging larger projects later in the semester.

For reference, Chapter 10 of the ELEC 2220 text (*Software and Hardware Engineering: Assembly and C Programming for the Freescale HCS12 Microcontroller*, *2nd Ed.* by Frederick M. Cady), introduces program development for the HCS12 in C using *CodeWarrior*. Other C program examples are provided in Chapters 11-18.

## Interfacing To and Controlling HCS12 Input/Output (I/O) Pins

The MC9S12C microcontroller family block diagram[1] is given in Figure 1. The microcontroller includes the HCS12 CPU, 32K bytes of flash memory, 2K bytes of RAM, analog to digital converter (ATD), timer module (TIM), PWM module, asynchronous (SCI) and synchronous (SPI) serial communication interfaces, controller area network (CAN) interface, clock generator, system integration module (SIM), and a number of parallel I/O ports. Shown in Figure 1 are 60 programmable signal pins, plus power, ground, clock and test pins. The chip comes in 48-pin LQFP, 52 pin LQFP, and 80-pin QFP packages. In the 80-pin package, all signals shown in Figure 1 are bonded to pins on the package. On smaller packages, only a subset of these signals is bonded to pins. The DragonFly12-Plus application module used in this lab contains the 48-pin chip package, shown in Figure 2. Two 20-pin headers, arranged in DIP package format as shown in Figure 3, are used to insert the module into the project board. Therefore, the 40 pins shown in Figures 3 and 4 carry the signals available to users of the DragonFly-Plus modules.

As discussed in ELEC 2220, most of the signal pins have multiple functions. The default in most cases is simple parallel I/O; special functions are selected by software. Table 1 lists the functions of the pins on the 48-pin package.

[1] *Freescale "MC9S12C Family Reference Manual*

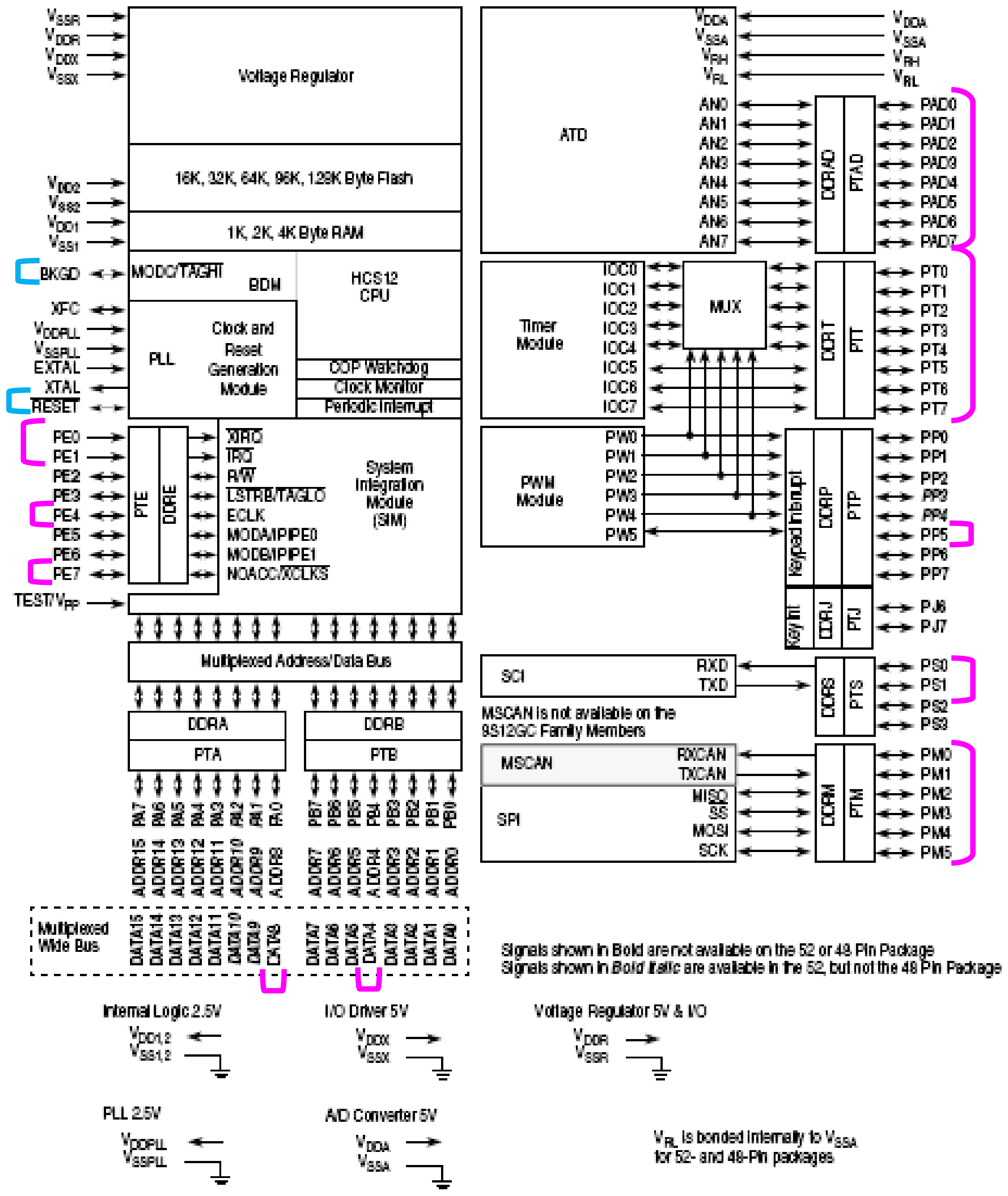


**Figure 1. MC9S12C-Family Block Diagram**[2]

[2] ***Freescale "MC9S12C Family Reference Manual***

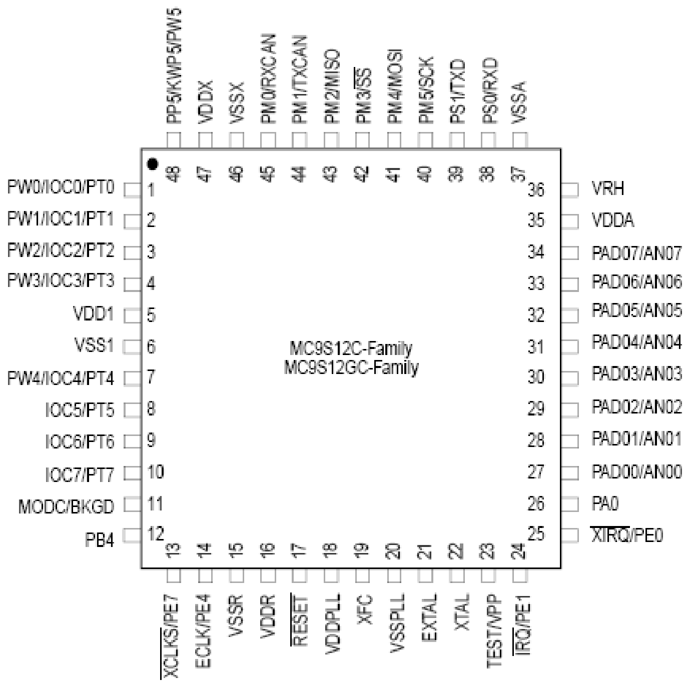


**Figure 2. Pin assignments on the 48-pin LQFP package**[3]

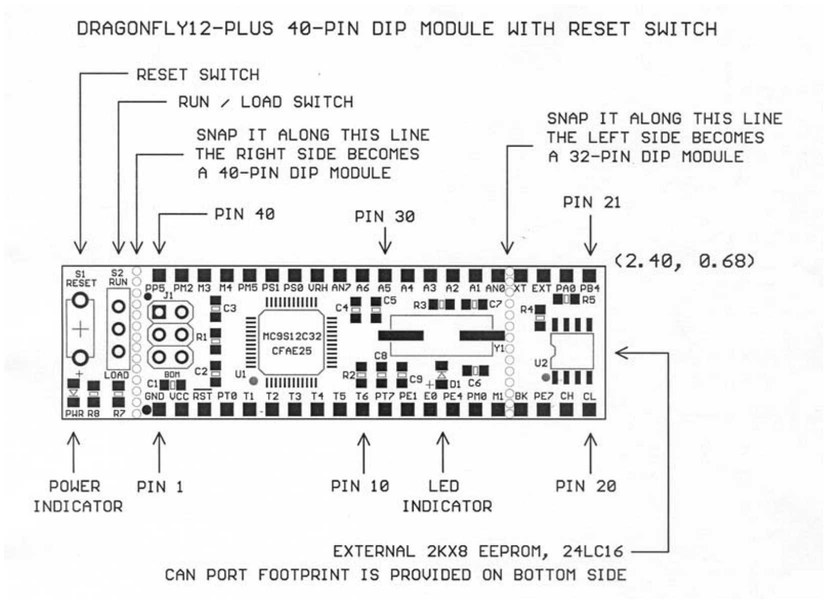

DRAGONFLY12-PLUS 40-PIN DIP MODULE WITH RESET SWITCH
RESET SWITCH
RUN / LOAD SWITCH
SNAP IT ALONG THIS LINE THE RIGHT SIDE BECOMES A 40-PIN DIP MODULE
SNAP IT ALONG THIS LINE THE LEFT SIDE BECOMES A 32-PIN DIP MODULE
PIN 40
PIN 30
PIN 21
(2.40, 0.68)
POWER INDICATOR
PIN 1
PIN 10
LED INDICATOR
PIN 20
EXTERNAL 2KX8 EEPROM, 24LC16
CAN PORT FOOTPRINT IS PROVIDED ON BOTTOM SIDE

**Figure 3. 40-pin connector J1 on the DragonFly12-Plus module**[4]

[3] ***Freescale "MC9S12C Family Reference Manual***

[4] ***Application Module Student Learning Kit Users Guide featuring the Freescale HCS12***

[4] ***Application Module Student Learning Kit Users Guide featuring the Freescale HCS12***

| | | | |
|---|---|---|---|
| 1 | GND | 40 | PP5/KWP5/PW5 |
| 2 | VCC | 39 | PM2/RXCAN1/RXCAN0/MISO0 |
| 3 | /RESET | 38 | PM3/TXCAN1/TXCAN0/SS0 |
| 4 | PT0/IOC0 | 37 | PM4/RXCAN2/RXCAN0/RXCAN4/MOSI |
| 5 | PT1/IOC1 | 36 | PM5/TXCAN2/TXCAN0/TXCAN4/SCK0 |
| 6 | PT2/IOC2 | 35 | PS1/TXD |
| 7 | PT3/IOC3 | 34 | PS0/RXD |
| 8 | PT4/IOC4 | 33 | VRH |
| 9 | PT5/IOC5 | 32 | AN7/PAD07 |
| 10 | PT6/IOC6 | 31 | AN6/PAD06 |
| 11 | PT7/IOC7 | 30 | AN5/PAD05 |
| 12 | PE1/IRQ | 29 | AN4/PAD04 |
| 13 | PE0/XIRQ | 28 | AN3/PAD03 |
| 14 | PE4/ECLK | 27 | AN2/PAD02 |
| 15 | PM0/TXCAN0/RXB | 26 | AN1/PAD01 |
| 16 | PM1/TXCAN0/TXB | 25 | AN0/PAD00 |
| 17 | BKGD | 24 | XT/ OSC. output |
| 18 | PE7 | 23 | EXT/ OSC, input |
| 19 | CAN - | 22 | PA0 |
| 20 | CAN + | 21 | PB4 |

**Figure 4. DragonFly12-Plus module pin assignments**[4]

| Port | Parallel I/O Pins | Alternate Function 1 | Alternate Function 2 |
|---|---|---|---|
| A | PA0 | | |
| B | PB4 | | |
| E | PE0<br>PE1<br>PE4<br>PE7 | XIRQ* (non-maskable interrupt)<br>IRQ* (maskable interrupt)<br>ECLK (E-clock output)<br>XCLKS* (clock input) | |
| M | PM0,PM1<br>PM2<br>PM3<br>PM4<br>PM5 | RXCAN,TXCAN (CAN control)<br>MISO (SPI controller)<br>SS*<br>MOSI<br>SCK | |
| P | PP5 | KWP5 (Key wakeup) | PW5 (PWM generator) |
| S | PS0,PS1 | RXD, TXD (SCI controller) | |
| T | PT0-PT4<br>PT5-PT7 | IOC0-IOC4 (Timer I/O)<br>IOC5-IOC7 | PW0-PW4 (PWM gen.) |
| AD | PAD00-PAD07 | AN00-AN07 (A/D Converter) | |
| --- | MODC | BKGC (BDM input) | |
| | RESET* | | |

**Table 1. Pin functions on the 48-pin HCS12 microcontroller and test module**

## Microcontroller Software Setup in Assembly Language

Unlike the general-purpose microprocessors in most desktop and portable computers, every microcontroller has its own unique set of fixed memory and I/O resource addresses. Therefore, when creating a new project in CodeWarrior, a specific microcontroller derivative must be identified so that the compiler/assembler/linker can be told where it can place program memory, data, stack, etc. In the HCS12, the following usable addresses are available.

$0000-$03FF – CPU and I/O module registers
$0800-$0FFF – 4K bytes RAM (data storage)
$4000-$7FFF - 16K bytes flash "ROM_4000" fixed memory [C32 and C128]
$8000-$BFFF – 8*16K bytes flash memory (in16K page window) [C128 only]
$C000-$FFFF – 16K bytes flash "ROM_C000" fixed memory [C32 and C128]
(some of this flash used by BDM)

In an assembly language project, data and the stack must be explicitly forced into RAM and program code into flash memory via "origin" (ORG) assembler directives, as shown in Example 1. Note that the four labels *RAMStart*, *RAMend*, *ROM_4000Start*, and *ROM_4000End* are defined in the INCLUDE file "*MC9S12C32.inc*", which also provides symbolic labels for all CPU and module register addresses.

```
;**********************************************************
; include derivative specific macros
INCLUDE 'MC9S12C32.inc'
;RAMStart:          equ   $00000800    ;defined in MC9S12C32.inc
;RAMEnd:            equ   $00000FFF    ;defined in MC9S12C32.inc
;ROM_4000Start:     equ   $00004000    ;defined in MC9S12C32.inc
;ROM_4000End:       equ   $00007FFF    ;defined in MC9S12C32.inc
;ROM_C000Start:     equ   $0000C000    ;defined in MC9S12C32.inc
;ROM_C000End:       equ   $0000FEFF    ;defined in MC9S12C32.inc

            ORG RAMStart
      ; data definitions go here

      ; code section begins here
            ORG ROM_4000Start
Entry:      LDS   #RAMEnd+1    ; initialize the stack pointer
     ; program instructions go here

;**********************************************************
;*                 Interrupt Vectors                       *
;**********************************************************
            ORG   $FFFE
            DC.W  Entry           ; Reset Vector
```

**Example 1. Sample assembly language program setup for HCS12.**

CPU and module registers are accessed in assembly language by treating them as memory addresses. Example 2 illustrates initialization of the data direction registers (DDRA, DDRB) of ports A and B as input and output, respectively, reading a byte from port A, and writing it to port B. As stated above, INCLUDE file `' MC9S12C32.inc'` contains symbolic label definitions for all registers. Therefore, you may either include that file or else define them in your program. As shown in the last instruction of Example 2, symbolic labels are not absolutely necessary, but they make the program much more readable, and therefore easier to write, understand, and debug, and less error prone. Therefore, the use of symbolic names for all registers is good programming practice, and highly recommended.

```
INCLUDE ' MC9S12C32.inc'
;ROM_4000Start EQU      $4000    ;defined in MC9S12C32.inc
;PORTA         EQU      $0000    ;defined in MC9S12C32.inc
;PORTB         EQU      $0001    ;defined in MC9S12C32.inc
;DDRA          EQU      $0002    ;defined in MC9S12C32.inc
;DDRB          EQU      $0003    ;defined in MC9S12C32.inc

ORG   ROM_4000Start
movb  #$00,DDRA   ;configure PORTA as input
movb  #$ff,DDRB   ;configure PORTB as output
ldaa  PORTA       ;input byte from PORTA
staa  PORTB       ;output byte to PORTB
ldaa  $0000       ;also inputs byte from PORTA
```

**Example 2. Sample assembly language program to access I/O ports**

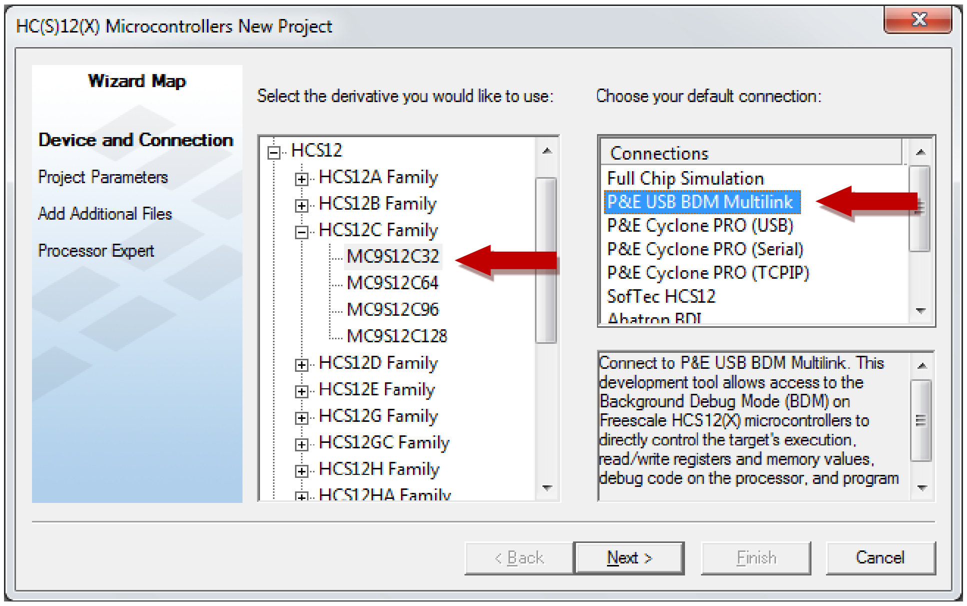
## Microcontroller Software Setup in C

The purpose of a high level language, such as C, is to allow a programmer to concentrate on the algorithm to be performed, independent of the particular processor being used. This is done by making processor-specific details, such as memory addresses, CPU registers, and assembly language instructions, transparent to the programmer. Such low level details are managed by the language compiler. Therefore, the compiler must be provided with ROM and RAM address ranges so that it can position code, data, and stack accordingly. In *CodeWarrior*, this is done automatically when a processor derivative (ex. HCS12) is selected at project creating time.

In assembly language, the programmer must explicitly initialize the stack pointer, background timer, global variables, interrupt vectors, etc. To take care of these things, *CodeWarrior* allows the user to include Freescale-supplied “startup code” in a C project. For this lab, we will use the default “ANSI startup code” (selected on the C/C++ Options Page of the New Project Wizard) to include file “startup.c” in your project. On reset, the instructions in this file are executed to set up the microcontroller, prior to entering the “main” program defined by the programmer. Use the following New Project Wizard selections for C projects in this lab.

- *Device and Connection Form(Figure 5)* - Select your processor derivative (MC9S12C32) and connection (**P&E USB BDM Multilink)**. You may also select Full Chip Simulation if you wish to simulate the hardware.
- *Project Parameters Form (Figure 6)* – Select the programming language (C), the project name (ex. Lab2.mcp), and the location for the files (your network drive name).

Click "*Finish*" on the Project Parameters Form to accept defaults for the remaining options (i.e. add no existing files to project, disable the "Processor Expert", use ANSI startup code, use the minimal memory model, disable use of floating-point arithmetic, and do not use PC –lint.)

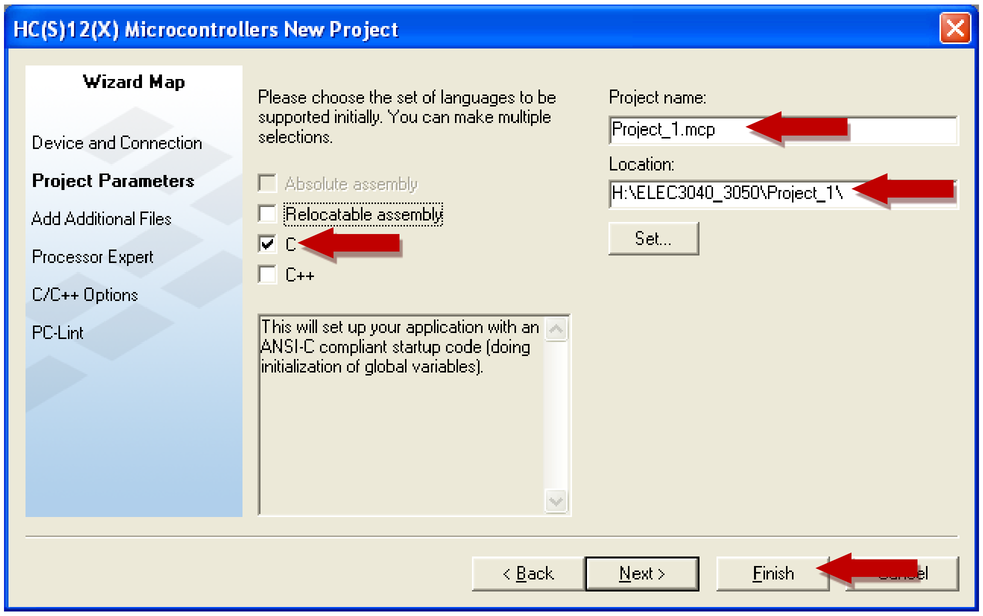


Select the derivative you would like to use:

Choose your default connection:

Connect to P&E USB BDM Multilink. This development tool allows access to the Background Debug Mode (BDM) on Freescale HCS12(X) microcontrollers to directly control the target's execution, read/write registers and memory values, debug code on the processor, and program

**Figure 5. Device and Connection Form**



Please choose the set of languages to be supported initially. You can make multiple selections.

[✓ C]

This will set up your application with an ANSI-C compliant startup code (doing initialization of global variables).

Project name: Project_1.mcp

Location: H:\ELEC3040_3050\Project_1\

**Figure 6. Project Parameters Form**

All I/O ports and functions in the HCS12 microcontroller are accessed and controlled via "registers", each of which is assigned a memory address in the range $0000-$03FF, as defined in the HCS12 Data Sheet, available on the class web page, and in the ELEC 2220 text, *Software and Hardware Engineering: Assembly and C Programming for the Freescale HCS12 Microcontroller, 2nd Ed.* by Frederick M. Cady. Chapter 10 of the text discusses program development using C and *CodeWarrior* and Chapter 11 covers I/O ports.

When programming in C for a microcontroller, one must be aware of how data types are defined by the compiler used for each project. For the data types needed in this lab, the *CodeWarrior* HCS12 C compiler uses the definitions in Table 2. You should always select a data type appropriate for each variable.

| Data type declaration | Number of bits | Range of values |
|---|---|---|
| char k;<br>unsigned char k; | 8 | 0..255 |
| signed char k; | 8 | -128..+127 |
| int k;<br>signed int k;<br>short k;<br>signed short k; | 16 | -32768..+32767 |
| unsigned int k;<br>unsigned short k; | 16 | 0..65535 |

**Table 2. Data type definitions for a variable k in the *CodeWarrior* C compiler.**

Variables in C can be automatic, static, or volatile. An *automatic variable* is declared within a procedure and is visible (has "scope") only within that procedure. Space for the variable is allocated on the system stack when the procedure is entered, and deallocated when the procedure is exited. Therefore, values are not retained from one procedure call to the next. If there are only one or two variables, the compiler may choose to allocate them to registers in that procedure, instead of allocating memory.

A *static variable* may be declared either inside or outside a function. These are retained for use throughout the program by using designated RAM locations that are not reallocated during the program. A static variable is defined by inserting the word "static" in front of the variable definition, as in the following character variable declaration.

```
        static char bob;
```

A *volatile variable* is one whose value can be changed by outside influences, i.e. by factors other than program instructions, such as the value of a timer register or the output of an analog to digital converter. The most common use of volatile variable definitions in embedded systems will be for the various I/O ports and module registers of the microcontroller. For example, the following defines the addresses of I/O ports A and B.

```
        #define PORTA    (*((volatile unsigned char*)(0x0000)))
        #define PORTB    (*((volatile unsigned char*)(0x0001)))
```

As an 8-bit input port, values read via port A or B are supplied by sources external to the microcontroller, and therefore such ports are declared as data type "volatile unsigned char" at addresses $0000 and $0001, with identifiers PORTA and PORTB, respectively, defined as pointers to these addresses. This allows these identifiers to be used as any other variable. The following C statements read an 8-bit value from PORTB, assigns it to character variable c, and then writes the value to PORTA.

```
c = PORTB;          /* read value from PORTB into variable c */
PORTA = c;          /* write value to PORTA from variable c */
```

These instructions both transfer one byte of data. To test or change individual bits of a byte, logical operators (AND, OR, XOR) must be used with a "mask", as in assembly language.

```
c = PORTB & 0x01;           /* masks all but bit 0 of byte copied to variable c */
if ((PORTA & 0x01) == 0)    /* tests bit 0 of PORTA */
PORTA = c | 0x01;           /* writes variable c to PORTA with bit 0 set to 1 */
```

Note the difference between bit-parallel operators & (AND) and | (OR), which produce 8-bit results of the corresponding logical operation, and relational operators && (AND) and || (OR), which produce TRUE/FALSE results (any non-zero value is considered "TRUE" and a zero value is considered "FALSE"). Consider the difference between the following two statements, where a and b are 8-bit variables:

```
if (a & b)      /* bitwise AND of variables a and b */
if (a && b)     /* requires a to be TRUE and b to be TRUE */
if (1)          /* non-zero is always TRUE */
if (0)          /* zero is always FALSE */
```

In the first case, the bitwise logical AND of the 8-bit values of a and b is computed, and if all bits = 0, the result would be FALSE, otherwise the result would be TRUE. In the second case, the result is TRUE if variable a is TRUE and if variable b is TRUE. Variables a and b are not combined in the second case. The third case is often used for endless loops – since any non-zero value is always considered TRUE. The fourth case would never be TRUE.

*CodeWarrior* provides a derivative-specific include file for each microcontroller which provides symbolic label definitions for all CPU and module registers. Example 3 shows the default main.c file created by *CodeWarrior* for a new project. Note the inclusion of the derivative information file "*MC9S12C32*.h". You should open and review the contents of this file in *CodeWarrior* to see the symbols assigned to various registers.

```
#include <hidef.h>      /* common defines and macros */
#include <MC9S12C32.h>  /* derivative information */
#pragma LINK_INFO DERIVATIVE "MC9S12C32"

void main(void) {
  /* put your own code here */
  EnableInterrupts;     /* keep if interrupts used */

  for(;;) {} /* wait forever */
  /* please make sure that you never leave this function */
}
```

**Example 3. Skeleton main.c file generated by *CodeWarrior*.**

## Program Debugging Support in *CodeWarrior*

- Source window (C statements)
- Assembly window
- Data window
- Register window
- Memory window
- Single step
- Breakpoints (pause program execution at selected points)
- Markpoints (capture variable values at selected points)

**Laboratory Exercise**

To experiment with C program creation and debugging, you are to set up the HCS12 to create a decade up/down counter, i.e. count up from 0 to 9 and repeat, or count down from 9 to 0 and repeat. The program is to test two switches: start/stop switch S1 and direction switch S2, connected to I/O port pins PA0 and PB4, respectively. The "switches" are to be push/pull switches, set up in the Static I/O instrument of *WaveForms* as in Lab 1, connected to EEBOARD digital I/O lines DIO0 and DIO1. Counting is to occur when S1 = 1, otherwise the count should "freeze". While counting is enabled, the count should be up if S2 = 0, and down if S2 = 1. Changing S2 while counting is enabled should change the count direction on the next change. The count is to be displayed on four LEDs in *WaveForms* by writing to Port T bits 3-0. Connect EEBOARD digital I/O lines DI07-DIO4 to the Port T pins on the module with wires, and configure DIO7-DIO4 as LEDs in *WaveForms*. The count is to change approximately once every half second.

This program is to be designed as a main program and two separate "functions" as follows.

> A ***delay*** function is to implement a half-second time delay ("do nothing" for about half a second). If time permits, try to get the delay function to be accurate within 10% of the nominal value (0.5 seconds +/- 10%), or better.
>
> A ***counting*** function is to increment or decrement the count, according to the setting of a direction variable passed as a parameter from ***main***, and display the new value on the LEDs. The count value is to be a static global variable.
>
> The ***main*** program is to initialize port directions and variables, and then execute in an endless loop, calling the ***delay*** function, setting the direction variable based on position of switch S2, and calling the ***counting*** function if counting is enabled by switch S1.

In debugging the program, you will be expected to demonstrate the use of breakpoints, markpoints, single-step, and other debug features in *CodeWarrior*. These are described in the *CodeWarrior* help messages.

Specifically, do the following in the debugger.

1. Set breakpoints in the main program and in each of the two functions, and verify that the program reaches these breakpoints as expected.

2. While stopped at a breakpoint, record the values of your different program variables (from the Data window at the left.) Note that the only data values displayed are global variables, and those in the current "scope", i.e. in the currently-executing function/procedure. So, if you wish to examine a variable in a function F, then you'll need a breakpoint in F.

3. After stopping at a breakpoint, single-step through a few instructions, and record any changes to the variables. Then you may click "run" to continue the program.

4. Using the disassembled program window (upper right), find and record the memory addresses assigned to two of your program variables. Then, in the memory window (bottom right), display the memory area containing these two addresses and verify that the values match those in the variables window.

5. Remove the breakpoints and run the program to verify correct operation.

Prior to lab you should write a first draft of the program and design a "test procedure" (record these in your notebook) so that you will be ready to compile the program and begin testing when the lab period begins.

**Lab 2 Deliverables:**

Lab notebooks are to be submitted to the GTA at the conclusion of the lab period, and they will be returned at the following Monday's lecture. As mentioned in the initial lab lecture, these books are expected to contain your lecture notes, pre-lab work (program design and test procedure), and notes made during the lab session of tests performed, observations, design changes, etc. This will be checked by the instructors when the lab books are collected every second week. Note that it is considered poor engineering practice to make lab book entries at a later time, after the lab session has concluded. Delaying the recording of notes leaves open the possibility of entering inaccurate or incomplete information.