PURDUE UNIVERSITY

Programing the Microprocessor in C

Microprocessor System Design and Interfacing
ECE 362

Course Staff

1/31/2012

1 Introduction

This tutorial is made to help the student use C language for programing the microcontroller. It assumes that the student knows how to program using C language. For more information about programming in C you can refer to online sources like: http://users.ece.utexas.edu/~valvano/embed/toc1.htm

2 Starting a new project in C

The purpose of a using high level language like C is to speedup programming while focusing on functionality. Specific details, such as memory addresses, CPU registers, and assembly language instructions are transparent to the programmer. Such low level details are managed by the language compiler. In CodeWarrior, this is done automatically when a new C project is created.

To create a new C project in CodeWarrior follow the steps below:

- Select File then New Project. The new project wizard will start.
- From Device and Connection step - Select your processor (MC9S12C32) and connection (P&E USB BDM Multilink). You may also select Full Chip Simulation if you wish to simulate the hardware.
- From Project Parameters step – Select the programming language (C), the project name (ex. Test1.mcp), and the location for the files.
- Select Finish

After CodeWarrior creates your project you will see the project window. In the left-hand window pane you will see a folder structure for the files that are part of your project.

- The “Sources” folder contains your source files. You should add you “C” files here.
- The “Includes” folder contains files specific to model of microcontroller you are using. The “mc9s12c32.h” file declares variable names you can use to read and write the internal registers of your microcontroller.
- The “Libs” Folder contains files to support the specific model of microcontroller being used as well as the ANSI-C libraries you may use.
- The “Startup Code” folder contains code used to allow the processor to run C-language code.
- The “Linker Files” folder, contains files used the CodeWarrior program to compile and link your program. The most important file is the “.prm” file that specifies the overall memory map of your program (Like RAM/ROM areas) and also provides the ability to register your interrupt service routines (ISR’s). It defines “SEGMENTS” which identify portions of memory and your code and variables can be placed in those segments. The “PLACEMENT” section of the PRM file defines the names are associated with each segment.

3 First Program

```
#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 1. Skeleton main.c file generated by *CodeWarrior*.

The "*MC9S12C32.h"* header has the definitions for the registers.

Clearly "*EnableInterrupts"* enables interrupts.

The microcontroller does not have an operating system that can take over control once the program is finished. This means the program should never exit (loop forever!).

4 Reading and Writing Internal Registers

All the internal registers in the microcontroller are defined by variables and can be read and written in the same manner as any other C variable. The definitions for the registers and individual register bits are defined in the header file "mc9s12c32.h" which can be found under the "Includes" folder in your project.

For example, ports A, B are defined by variables named PORTA, PORTB. You may perform operations on these variables just as you would any other.

```
unsigned char var1 = PORTA;

PORTA = 0xf4;                /* write value 0xf4 to PORTA  */

PORTB = 3 + PORTA;
```

Individual bits in the registers can be accessed using the names defined for each bit.

```
PORTA_BIT0 = 1;

if ( PORTA_BIT0 = = 0) { ... }          /* tests bit 0 of PORTA */
```

Alternatively, you can use bitwise operations such as AND and OR.

```
bit0 = PORTA & 0 x01;    /* masks all but bit 1 of byte copied to
variable bit0 */
```

When writing individual bits in the registers it is important to be aware that this will cause the microcontroller to first read the full eight-bit byte, modify the byte, and then write the full byte back. Some of the internal functions of the microcontroller are affected by reading a register so it is important to be aware of any potential side effects when accessing the registers.

5 Using Assembly Language Code in C

You can write assembly instructions and call assembly language routines from within your C code by using the "asm{ }" construct. Below are some guidelines for writing assembly code.

- Write each assembly instruction on a separate line.
- Labels must be defined on their own line.
- Global variables can be accessed using their name.
- Local variables in a function are stored as part of the stack frame and must be accessed using the stack pointer.
- The stack must be returned to its original state before starting your assembly code.

6 Standard C Library

CodeWarrior includes a standard C library that contains many of the routines that programmers are used to having available to use in their programs. Routines like "sprintf" and "sscanf" can be used the same way as on larger systems.

The location where printf should emit its output is very application specific. Some devices have a serial port, some have multiple and others do have none at all. Because the ANSI library cannot cover all those cases at once, the provided printf implementation calls a TERMIO_PutChar function which is not implemented in the library itself. A sample implementation of TERMIO_PutChar is provided in termio.c, however it's also perfectly fine to just implement "void TERMIO_PutChar(char ch);" as declared in termio.h on your own. The termio.c file contains a second method TERMIO_Init to be called before printf is used, but the ANSI library on its own is not using TERMIO_Init, so implementing this function is optional, of course the serial port controller has to be configured before it is used. In order to use termio.c, copy this file from the library into the project directory and then add it to the project.

7 Interrupt Service Routines

Writing interrupts can be done in a couple of ways:

1. The keyword "interrupt" can be used in the routine declaration, as is shown below.

```
void interrupt myISR( void)
{
// ISR code
}
```

2. Another way is to use a #pragma keyword to tell the compiler it is an ISR.

```
# pragma TRAP_PROC
void myISR( void)
{
// ISR code
}
```

After you have written your ISR, you must enter its address into microcontroller's interrupt vector table. You can register your ISR in the vector table in the PRM file (under Project Settings/Linker Files) of your project. Use the instruction:

```
VECTOR 7 timerISR
```

This will register the routine named timerISR to the vector table entry 7. Note: choose vector number 7 or greater.

The other way of initializing it is by address:

```
VECTOR ADDRESS 0xFFF6 myISR
```

where "0xFFF6" is the address of the interrupt vector you are using. You may also use the vector number rather than specifying the vector address.

8 Serial Communications Interface (SCI)

The example below shows how to initialize the SCI and how to send and receive characters.

```
void TERMIO_Init(void) {  /* initializes the communication channel */
    SCI0BDL     =   156;   /* baud rate 9600 at 8 MHz */
    SCI0CR2  = 0x0C;   /* 8 bit, TE and RE set */
}

char TERMIO_GetChar(void) {  /* receives character from the terminal
channel */
        while (!(SCI0SR1 & 0x20));       /* wait for input */
    return SCI0DRL;
}

void TERMIO_PutChar(char ch) {  /* sends a character to the terminal
channel */
    while (!(SCI0SR1 & 0x80));  /* wait for output buffer empty */
    SCI0DRL = ch;
}
```

9 Analog-to-digital (ATD)

The example below performs an ATD conversion.

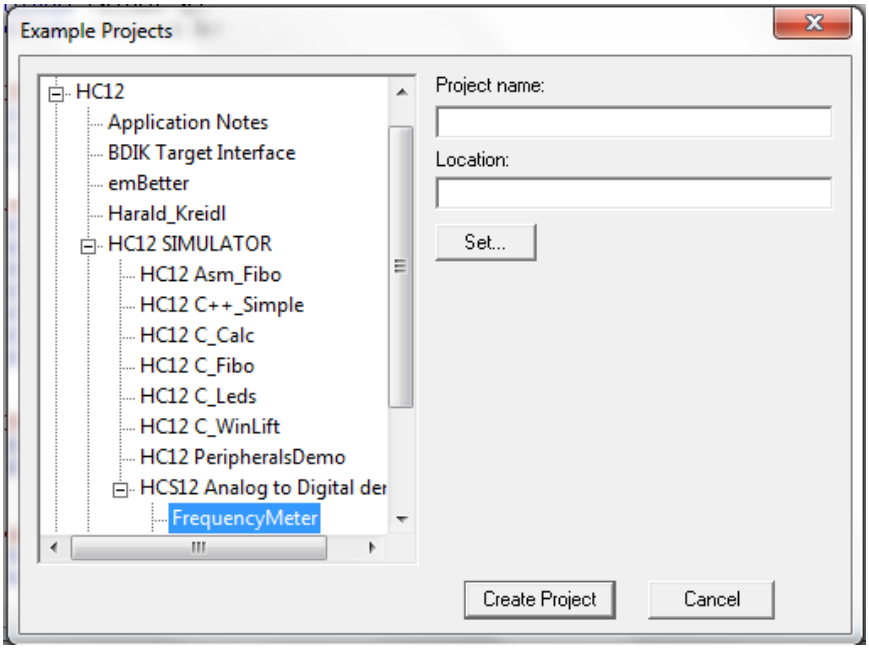
```
void main(void)
{
ATDCTL2 = 0x80; // power up the ADC and disable interrupts
delay_5ms(); // wait for ADC to warm up
ATDCTL3 = 0x00; // select active background mode
ATDCTL4 = 0x01; // select sample time = 2 ADC clks and set
                // prescaler to 4 (2 MHz)
ADC_convert(); // perform conversion and change to usable value
}

void ADC_convert()
{
ATDCTL5 = 0x03; // sets up ADC to perform a single conversion,
                // 4 conversions on a single channel,
                // and store the results ADR0H - ADR3H.
while((ATDSTAT & 0x8000) != 0x8000){} // Wait for conversion to finish
}
```

10 Sample Demo Project

To create a sample demo project which uses RTI and ATD follow the steps below:

1. File -> Startup Dialog -> Load Example Project

2. Choose HC12 -> HC12 SIMULATOR -> HCS12 Analog to Digital demos -> Frequency Meter
3. Fill the project name and location on the right side of the window and create the project.



Appendix

A.1 Preprocessor Directives

Preprocessor directives begin with # in the first column. Preprocessor commands are processed first by the compiler. The two most important are the macro definition (**#define**) used to define I/O ports and bit fields. A second important directive is the #include, which allows you to include another.

A.2 Data types and variables

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

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

A *volatile variable* might represent a location that can be changed by other elements of the computer system at any time, such as a hardware register of a device connected to the CPU. The program would never detect such a change; without the volatile keyword, the compiler assumes that the current program

is the only part of the system that could change the value. 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)))
```

A.3 Operators

operation	Meaning
=	assignment statement
@	address of
?	Selection
<	less than
>	greater than
!	logical not (true to false, false to true)
~	1's complement
+	Addition
-	Subtraction
*	multiply or pointer reference
/	Divide
%	modulo, division remainder
|	logical or
&	logical and, or address of
^	logical exclusive or
.	used to access parts of a structure

operation	Meaning
==	equal to comparison
<=	less than or equal to
>=	greater than or equal to
!=	not equal to
<<	shift left
>>	shift right
++	increment
--	decrement
&&	Boolean and
||	Boolean or
+=	add value to
-=	subtract value to
*=	multiply value to
/=	divide value to
|=	or value to
&=	and value to
^=	exclusive or value to
<<=	shift value left
>>=	shift value right
%=	modulo divide value to
->	pointer to a structure

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 */*

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.