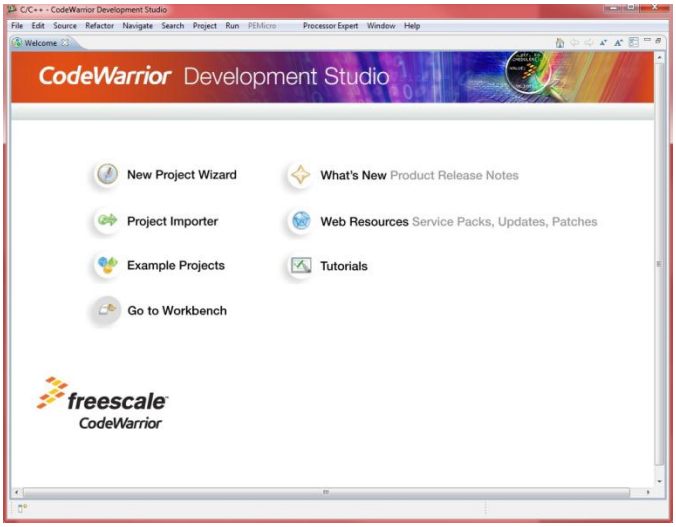
Spring 2013 - CSE 325: Embedded Microprocessor Systems

Project 2: Getting Started with Code Warrior and the MCF52259 Microcontroller Board

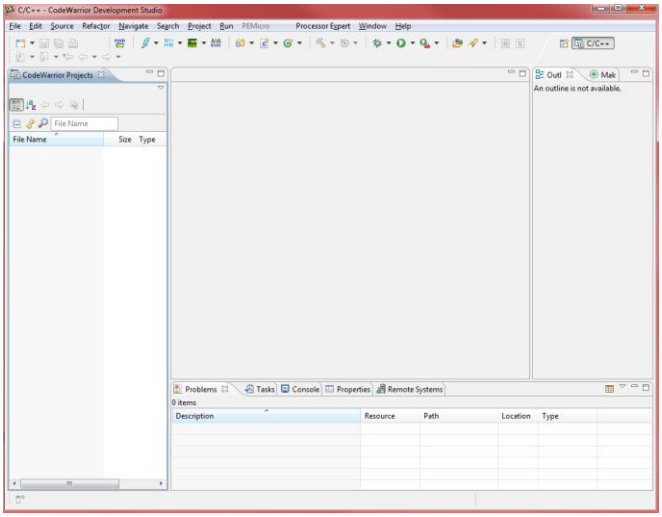
Points: 10, Due Date: Monday, February 04, 2013

1. Launching and Configuring CodeWarrior

Launch CodeWarrior Development Student for MCU Version 10. You might see a window that looks like this,



Click the X button on the Welcome tab to close that pane. You may now see a window that looks like this,



Take some time to look around the menu and configuration options, and set them to your liking. Several things can be configured, including the view, margins, line numbers, tab width, syntax coloring, etc. Some interesting things are:

a. You can configure the comments that will be automatically inserted in the comment header block of newly created files by selecting Files, clicking Edit, and entering something in the Pattern text area. For example:

```
/*****************************************************************************
* FILE: ${file_name}
*
* DECRIPTION
*
*
* AUTHORS
* Kevin R. Burger (burgerk@asu.edu) [KRB]
* Computer Science & Engineering
* Arizona State University
* Tempe, AZ 85287-8809
* http://kevin.floorsoup.com
*
* MODIFICATION HISTORY
* ************************************************************************
DD MMM YYYY Initial revision [KRB].
*****************************************************************************/
```

b. You can configure the comments that will be automatically inserted for new functions by selecting Methods, clicking Edit, and entering something in the Pattern text area. Here is my default template,

```
/*****************************************************************************
* FUNCTION:
*
* DESCRIPTION
*
*****************************************************************************/
```

2. Creating a Project

Let's create a project. On the main menu, click File | New > Bareboard Project.

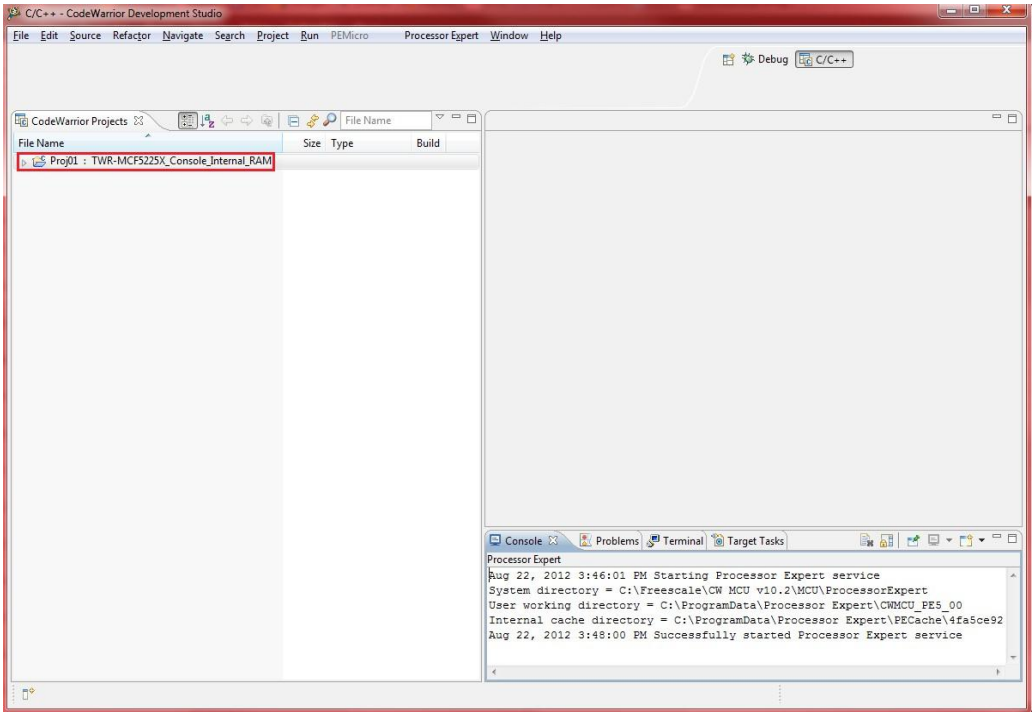
In the dialog, enter a name for the project, e.g., I will use Proj01. By default, the project will be created in the default workspace, but this can be changed if you wish by deselecting Use Default Location and navigating to the desired directory. In any event, click the Next button.

In the Device or Board to be Used pane, expand ColdFire Vx Tower Boards. Expand ColdFire V2. Select TWR-MCF5225X and click the Next button.

In the Connections dialog, make sure P&E USB Multilink Universal [FX] / USB multilink is the only item selected and click the Next button.

In the ColdFire Build Options dialog, select Minimal Hardware Support. Select No Optimizations and click the Next button.

Click the Next button. Click the Finish button. You will see your project displayed in the CodeWarrior Projects view,



3. Configuring Project Settings

Before we proceed there are some project settings we need to muck with. In the Project view, right-click on the project name, Proj01: TWR-MCF5225X_Console_Internal_RAM and select Properties from the pop up menu.

In the left-hand pane, expand C/C++ Build. Click on Settings.

In the right-hand pane, click on the Tool Settings tab.

Click on Messages.

Enter 1 in the Maximum Number of Errors and Maximum Number of Warnings text fields.

In the left-hand pane, expand ColdFire Compiler. Click on Warnings.

In the right-hand pane, select every check box except,

1. Unused Result from Non-Void-Returning Functions
2. Pad Bytes Added
3. Undefined Macro in #if
4. Non-inlined Functions
5. Token Not Formed by ## operator

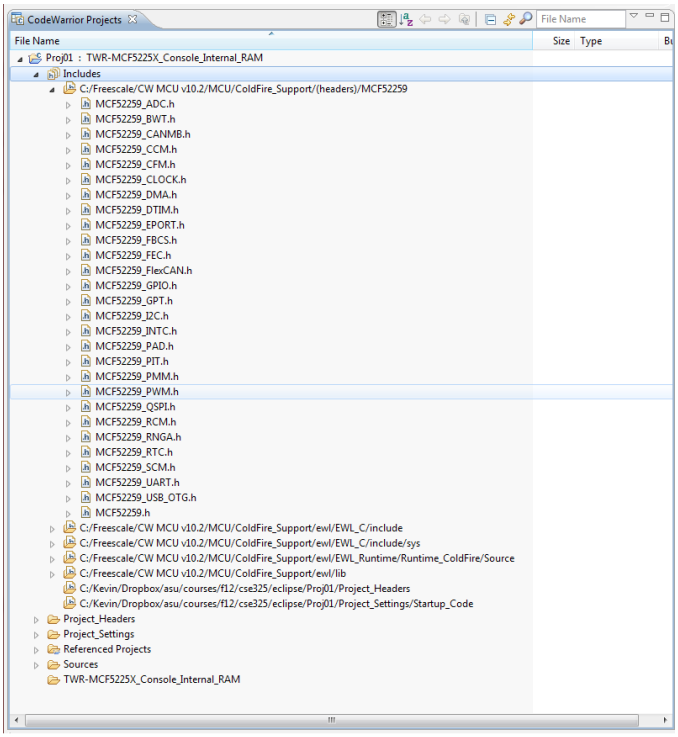
In the left-hand pane, click on Language Settings.

In the right-hand pane, select the following items (and deselect all the rest),

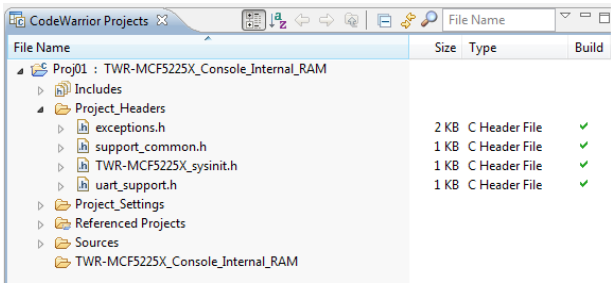
1. Require Prototypes
2. Enable C99 Extensions (this will, among other things, permit you to use // style comments)
3. Reuse Strings.

4. Becoming Familiar with the Project Structure

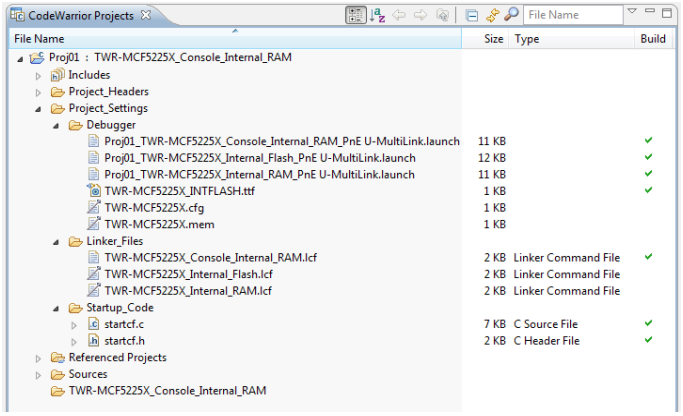
Click the icon to expand the project tree. You will see six folder groups. Expand Includes. Expand C:/Freescale/CW MCUv10.2/MCU/ColdFire_Support/(headers)/MCF52259.
These are system header files containing stuff that we will be using in our projects. Over the course of the semester, you will become familiar with several of these files. In this project we will be using declarations in MCF52259_GPIO.h so you might take a glance at it.



Expand Project Headers. These are MCF52259-specific header files that CodeWarrior provides. In general you will be using these header files, but not modifying them.



Expand Project_Settings. The Debugger folder contains some files that are used by (ta da) the debugger. You will not be messing with these files.

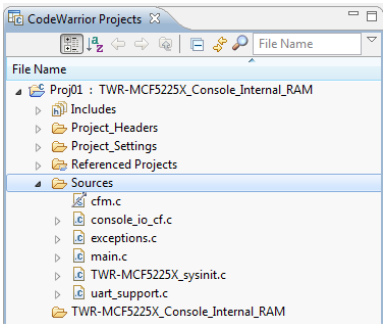


The Linker Files folder contains settings that control the linker. Double-click on TWR-MCF5225X_Console_Internal_RAM.lcf to open it. Read chapters 12 and 13 of the CodeWarrior Build Tools Reference Manual. Read the .lcf file.

Expand Startup_Code.

Double-click on startcf.c to open it. This is the code that is executed to start the system when you launch your project. If you look at line 296 you will see an instruction, jsr main. If you ever wondered when you write a C program how main() gets called and by whom; well, now you know.

Expand Sources,



This is the folder where you will create your source code files. CodeWarrior has already added some source code files to this folder. We will not be modifying any of these except for main.c which CodeWarrior added that will contain your main() function. Edit main.c now.

This program (if CONSOLE_IO_SUPPORT is defined when it is compiled) will display "Hello World..." on the Console window of CodeWarrior and then drop into an infinite loop.

To add a new C source code file to your project, right-click on the Sources folder icon and select New > Source file. To add a new header file, right-click on the Project_Headers folder icon and select New > Header file. To delete a source code file, right-click on the file name and select Delete from the pop up menu. You can rename files similarly.

5. Building the Project

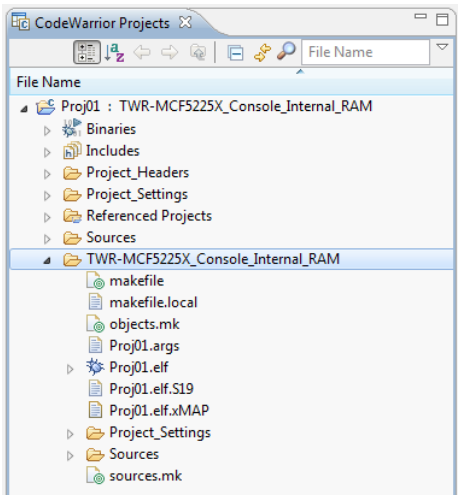
Let's build the project. Click Project | Build All. As the project builds, you should see a lot of text scroll by in the Console window. These strings are outputs from the makefile, the compiler, and the linker. At the very end, you should see a message which states, "Finished building target: Proj01.elf". If you do, congratulations, your project built correctly. If you have syntax errors, you can view them (and the locations of the errors) in the Problems view. If the Problems view is not displayed, click Window | Show View > Problems.

What the heck is an ELF file?

Briefly, an ELF file is a binary file containing the machine language code and data for an executable program. When you create a C or C++ project in Linux, the output from the linker will be an ELF file (we call this a "binary"). ELF files are stored using a particular file format that is well documented 10. It would not hurt you at all to familiarize yourself with, at least, the basics of the ELF file format.

In general, the file format defines various "sections" that constitute the program to be loaded into memory. For example, one of these sections is the .text section and this section contains the binary machine language instructions for your compiled program. More in a bit.

Expand TWR-MCF5225X_Console_Internal_RAM



CodeWarrior automatically generates a makefile for you. You can view it by editing makefile, but I've been told you should not modify this file yourself.

Edit Proj01.elf.xMAP. This file contains information generated by the linker including: (1) the names of various sections that appear in the ELF file, e.g., .userram, .code, .vectorram, .vectors, .text, .data, .bss, .custom, and .romp; (2) the names and memory addresses of some global variables defined in the linker command file, see the .data, .bss, .custom, .romp sections; (3) the names and addresses of functions that appear in the ELF file, see the .text section); and (4) the names, addresses, and sizes of the sections that appear in the ELF file, see the "Memory Map" part.

The .text section is where the machine language executable instructions for your program reside. When the program is loaded into the MCF52259 controller board for execution, the .text section will start at address 0x2000_0900 and it is 0x1218 bytes in size.

Question 1: What would be the address of the last byte of the last instruction in the .text section when it is loaded in memory.

Let's configure the linker to produce a "listing file". In the Project view, right-click on the project icon labeled

Proj01 : TWR-MCF5225X_Console_Internal_RAM and select Properties from the pop up menu. In the left-hand pane, expand C/C++ Build. Click on Settings. In the right-hand pane, expand ColdFire Linker and select Output. Select the check box labeled Generate Listing File. Click the OK button to close the Properties dialog.

Let's rebuild. Click Project | Build Project on the main menu bar. In the Project view, expand the TWR-MCF5225X_Console_ Internal_RAM folder and edit the Proj01.elf.lst file. The listing file gives you detailed information about the contents of the ELF file (Proj01.elf).

Question 2: See the document in footnote 10. ident[EI_DATA] = 2 in the Proj01.elf.lst file tells you something about the way multibyte data is stored in memory in the ColdFire architecture. Explain what it tells you.

Hint: it has something to do with Gulliver's Travels.

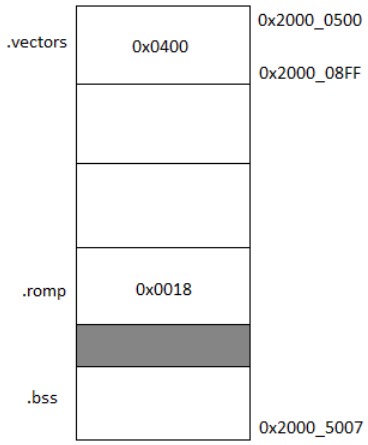
Question 3: What is the address in memory to which the system first transfers control, thus starting the process? In other words, what is the address in memory of the first instruction that is executed when the program is loaded and executed.

Hint: The answer is in the ELF header.

Question 4: There is an assembly language label associated with the address from Question 3. What is the name of this label? In which source code file is it located and on what line number of that source code file?

Question 5: What is the memory address of the main() function? How many bytes of memory (in decimal) do the instructions for main() consume?

Question 6: Edit Proj01.elf.xMAP. Locate the Memory Map near the bottom of the file. The memory map tells you the names of various nonempty sections (.vectors, .text, .data, .bss, and .romp), their physical (in memory) addresses (see column p_addr), and their sizes (in bytes). Finish this memory map diagram by showing each of the sections, the beginning and ending addresses of the section, and the size of the section.


.vectors
0x0400
0x2000_0500
0x2000_08FF
.romp
0x0018
.bss
0x2000_5007

6. Configuring the MCF52259 Microcontroller Board

Before running the project, we need to check the configuration of the MCF52259 controller board. There are some jumpers 11 that need to be placed correctly. The jumper documentation can be found in the TWR-MCF5225X User's Manual (available on the course website; hereafter referred to as the User's Manual). Open this document for reference.

Read Section 3.1. We must configure the clock source to be the 48 MHz crystal (Y1 on the TWR-MCF5225X Board Schematic which is hereafter referred to as the Schematic).

Looking at the Schematic, locate Y1 on sheet 3. Notice that pin 1 of Y1 is connected to pin 105 (XTAL) on the MCF52259 microcontroller. Pins 2 and 4 of Y1 are unconnected and pin 3 is connected to pin 1 of jumper J5 (CLK_SEL). Look at the bottom of page 5 in the User's Manual and note that J5 has two settings. If the jumper connects pins 1 and 2 then pin 1 of Y1 will be connected to pin 1 of J5 which is connected to pin 2 of J5 which is connected to pin 106 (EXTAL) on the MCF52259. Therefore, EXTAL will be connected to the output of Y1. (If pins 2 and 3 of J5 are connected then the microcontroller clock signal will be derived from the CLKIN0 signal on the elevator connector; that's not what we want).

Open the ColdFire MCF52259 Integrated Microcontroller Reference Manual (hereafter referred to as the IMRM) and move to Section 2.4 on page 2-9. Here we see that EXTAL (External Clock In) is the crystal oscillator input. Therefore, this is the clock signal which drives the entire microcontroller. Therefore, ensure that there is a jumper on pins 1 and 2 of J5 on your board so the microcontroller's clock source will be the 48 MHz crystal Y1.

Read Section 3.2 of the User's Manual. Jumper J4 routes 3.3 V to the MCF52259. Look at the bottom of page 5 and note that if J4 is on then 3.3 V is supplied to the MCF52259. Therefore, ensure there is a jumper on J4.

Load the MCF52259 Technical Data Sheet (hereafter referred to as the Datasheet). Scroll to page 15 which shows the pinout configuration of the 144-pin LQFP MCF52259. The power input signal in a digital circuit is frequently labeled V_{DD} and the ground signal is usually labeled V_{SS}.

Question 7: Which pin(s) on the MCF52259 are connected to power and which pin(s) to ground?

Look at page 5 in the User's Guide. Table 1 describes the jumper settings. Let's focus on J3.

J3 selects the Default Clock Mode. Look at the schematic and locate J3 on page 3. Suppose pins 2 and 3 of J3 are connected with a jumper. Pin 3 is connected to MCU_3V3 which is 3.3 V (a digital 1). Pin 3 is connected to pin 2 of J3 which connects to pin 144 (CLKMOD1) on the MCF52259. Pin 143 on the MCF52259 (CLKMOD0) is connected to pin 2 of J6. Suppose pins 2 and 3 of J6 are connected. These two jumper configurations would set both CLKMOD0 and CLKMOD1 to a digital 1.

Go to Section 7.6.4 on page 7-5 of the IMRM. Note that CLKMOD1:CLKMOD0 and XTAL together form a 3-bit binary value. If CLKMOD1:CLKMOD0 is 11 then it does not matter what XTAL is set to, as the MCF52259 clocking mode will be set to "PLL in normal mode, clock driven by external crystal."

PLL is an acronym for Phase Locked Loop12. Without getting into the gory details (which I don't understand anyway), one use of a PLL circuit is to generate a stable clock signal from a crystal oscillator. A PLL can also be used to derive a new clock frequency from the input signal. For example, if the input signal is 48 MHz, this could be divided by 2 to form a 24 MHz clock signal or it could be multiplied by 3 to form a 144 MHz signal which could then by divided by 2 to form a 77 MHz signal.

If you look at Chapter 7 of the IMRM you will note that it discusses the ColdFire Clock Module. The clock module can be programmed to select the clocking method for the MCF52259.

In CodeWarrior, go to the CodeWarrior Projects view and expand the Sources folder. You should see a source code file named TWR-MCF5225X_sysinit.c in that folder. Edit that file and go to line 12 which is the beginning of a function named pll_init(). This function programs the PLL in the clock module to select the desired clock frequency for the ColdFire core.

Go to section 7.7.1.5 on page 7-12 of the IMRM and you will see a discussion of a ColdFire register named the Clock Control High Register (CCHR). Bits 2:0 of this register form a 3-bit field named CCHR and notice at reset, this field is initialized to 101 (5 decimal). Look at the clock module block diagram on page 7-4 and locate the box labeled PLL. Notice the input to PLL is a signal labeled Ref Clock which is the output of a component labeled Pre-Divider. Below Pre-Divider you see a box labeled CCHR. This means that CCHR controls the pre-divider which is applied to the 48 MHz signal coming from the crystal oscillator Y1. Read the description of CCHR on page 7-12. Now, since CCHR is 5 this means the 48 MHz crystal oscillator signal is divided by 6 (if CCHR is 000 the pre-divider value is 1, if CCHR is 001 the pre-divider value is 2, and so on). Therefore, Ref Clock (the input to PLL) is 48 / 6 = 8 MHz.

Now go to section 7.7.1.1 on page 7-7 of the IMRM and you will see a discussion of a ColdFire register named the Synthesizer Control Register (SYNCR). Within this register, bits 14:12 form a 3-bit field named MFD (Multiplication Factor Divisor). The code in pll_init() writes 3 (binary 011) into this 3-bit field. If you look at Table 7-5 on page 7-8 of the IMRM you will see a discussion of what MFD does. In particular, in the column labeled 011 you will see (10x). This means that Ref Clock (coming from the pre-divider) will be multiplied by 10 in the PLL. Therefore, the output of PLL will be an 80 MHz clock signal. And what is the maximum clock frequency of the MCF52259? 80 MHz (see page 1 of the Datasheet).

That 80 MHz clock signal leaves the PLL and feeds the ColdFire core (look at the clock module block diagram on page 7-4). It also feeds various microcontroller peripherals, such as the Interrupt Controllers (INTC), the DMA timers (DTIM), the Queued Serial Peripheral (QSPI) module, the I2C module, and the UART's. The signal is also routed through a divide-by-2 divider (turning it into a 40 MHz signal) which feeds other peripherals such as the Pulse Width Modulation (PWM) module, the General-Purpose Timer (GPT) module, the A/D Converter (ADC) module, and the Programmable Interrupt Timer (PIT) module. You should also note that the 80 MHz signal coming out of the PLL is labeled f_{sys}, i.e., frequency of the system clock. We will refer to the 40 MHz signal going to the peripherals on the right-hand side of the block diagram as $f_{sys}/2$.

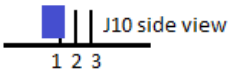
Now you know a bit about how microprocessors are clocked. Oh, by the way, we want CLKMOD1:CLKMOD0 to be 11 so pins 2 and 3 of both J3 and J6 need to be jumpered.

Question 8: Assume CCHR is programmed to 011 (3 decimal). What would be the frequency of Ref Clock assuming 48 MHz crystal oscillator Y1 is the microcontroller's clock input?

Question 9: Continuing with the previous question, suppose we wish to configure f_{sys} to be 72 MHz. What value would we have to write into MFD of SYNCR to obtain that clock frequency?

Question 10: Is it possible to obtain a system frequency f_{sys} = 50 MHz? If yes, what values of CCHR, MFD, and RFD (see the description of field MFD on page 7-8 of the IMRM) would suffice? Hint: look at the formula for f_{sys} below the table in the description of MFD. In this formula f_{ref} is what we referred to earlier as Ref Clock. If no, what would be the closest we could get to 50 MHz without exceeding that value?

1. Look at page 6 in the User's Guide. J10 controls the Default Clock Mode Selection. Since we are using the Y1 crystal oscillator, jumper J10 need to be off. If there is a jumper on J10, make sure it is configured this way,



2. J14 configures the BDM/JTAG debugger connection. We are using BDM so connect pins 1 and 2 of J14.

3. J15 needs to have pins 1 and 2 connected.

4. J20 needs to be off.

5. J21 needs to be off.

To summarize, these are the connections that need to be made,

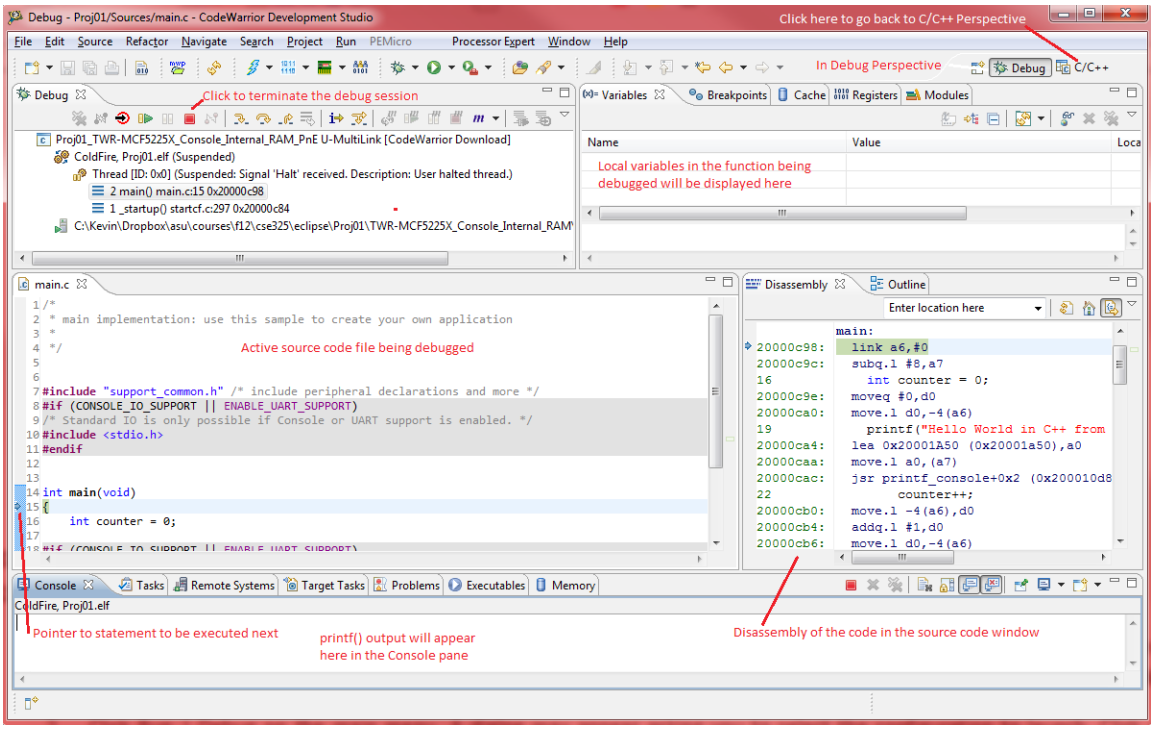
J3	Pins 2 and 3
J4	On
J5	Pins 1 and 2
J6	Pins 2 and 3
J7	All on
J10	Off
J11	Both on
J12	Pins 1 and 2
J13	Pins 1 and 2
J14	Pins 1 and 2
J15	Pins 1 and 2
J16	Pins 1 and 2
J20	Off
J21	Off

7. Debugging the Project

We're now ready to run the project, which we will do by launching the debugger.

Click Run | Debug on the main menu. A dialog may appear asking you to select the configuration to launch. Click on Proj01_ TWR-MCF5225X_Console_Internal_RAM_PnE U-Multilink and click OK.

I got a Windows Security Alert from the Windows Firewall at this point asking me if I wanted to allow CodeWarrior to communicate on this network. I selected Domain Networks and Private Networks and clicked Allow Access. If you don't see this message, well, simply ignore what I just said. The Debugger Perspective opens,



The line to be executed next is highlighted in the editor view. At this point, we can click Run | Step Over (or hit F6) to execute this statement. Do that now.

Notice that the highlighted line is now line 16. Also observe that variable counter is displayed in the Variables view with— probably—some weird value. That is because we have not executed the statement which initializes it to 0 yet. Hit F6 to execute the statement.

Observe that the value of counter is now 0 in the Variables view. Look in the Disassembly view. Notice that the line at memory address 0x2000_0CA4 is highlighted. Notice that the printf() statement actually consists of three assembly language statements,

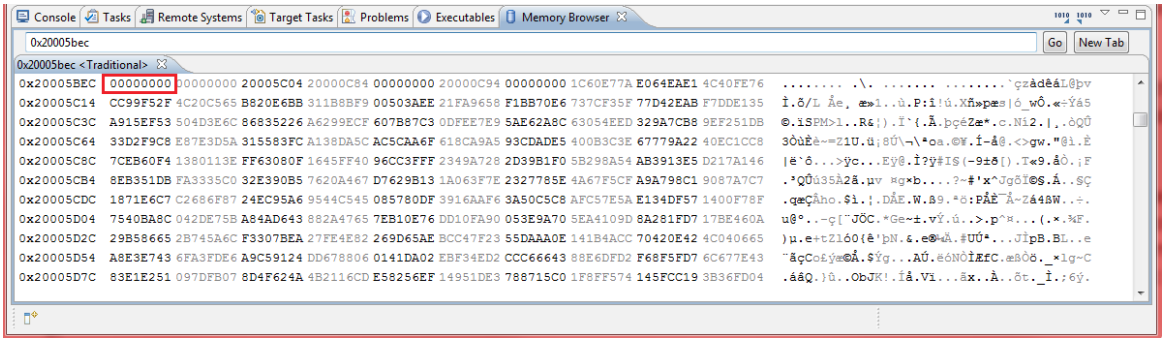
```
printf("Hello world in C++...");
0x2000_0CA4  lea    0x2000_1A50, a0     Loads the address of the first
character of the string literal into register a0
0x2000_0CAA  move.l   a0, (a7)   Pushes a0 onto the runtime stack
0x2000_0CAC  jsr    printf_console+0x2  Calls the printf() code from the
runtime library
```

We can execute these assembly language statement one at a time. Click in the Disassembly view to make it active and click on the address 0x2000_0CAA to highlight that line. Click Run | Run to Line (or hit Ctrl+R). The lea instruction is executed, and the debugger pauses indicating the the move.l instruction is the next instruction to be executed.

The lea instruction should have loaded 0x2000_1A50 into register a0. Let's check. In the upper right-hand view click on the Registers tab. The first entry is General Purpose Registers. Expand that entry. Scroll down and verify that register a0 now contains 0x2000_1A50.

Look at the value of register a7 (I see 0x2000_5BEC). Register a7 acts as the stack pointer register in ColdFire, i.e., it always points to the top item on the runtime stack. We can view memory and see what is at that address. Click Window | Show View > Memory Browser. The Memory Browser view opens at the bottom of the CodeWarrior window (in the same general area where the Console view is). At the top of the Memory Browser view, there is a text field. Enter 0x20005BEC in that field and press the enter key.

This is what I see (where the value that a7 is pointing to is highlighted).



Now go back to the Disassembly view. Highlight the jsr line and hit Ctrl+R to execute the move.l instruction. Go back to the Memory view and verify that the value at address 0x2000_5BEC is now 0x2000_1A50. Cool huh?

Now, we have two choices. If we select Run | Step Into (F5) execution will go into the printf() code. If you want to do that, that's fine, but we don't want to. If we select Run | Step Over (F6) then the printf() code will be executed and we will stop at the line following printf(). Hit F6 now. In general: F5 steps into functions and F6 steps over functions. Remember that.

The output from printf() should have been displayed in the Console view. Verify that.

The counter++ statement on line 22 of main.c is now highlighted. At this point the program drops into an infinite loop, simply incrementing counter over and over. Keep hitting F6 over and over to see what happens. Exciting, huh?

At some point you may get bored with that. Let's stop the debug session. Click the red box on the toolbar of the Debug view in the upper-left corner of the window. That terminates the debug session. You can now click on the C/C++ Perspective button to switch back to, guess..., the C/C++ perspective.

Well, that's your brief introduction to CodeWarrior. It is a very complex program, and we will only be using the most basic features. There are Freescale manuals on the course website that will give you more information about CodeWarrior. In particular, look for these links:

CodeWarrior Getting Started Guide

CodeWarrior IDE Common Features

Guide CodeWarrior Targeting Manual

Freescale Eclipse Extensions Guide

A Small Programming Project

1. Delete the contents of main.c and enter this code,

```
/*****************************************************************************
* FILE: main.c
*
* DECRIPTION
* A first TWR-MCF5225X program. Uses the GPIO module and Port TC to flash the LED's
on the TWR-MCF5225x micro-controller board.
*
* AUTHORS
* Kevin R. Burger (burgerk@asu.edu) [KRB]
* Computer Science & Engineering
* Arizona State University
* Tempe, AZ 85287-8809
* http://kevin.floorsoup.com
*
* MODIFICATION HISTORY
*****************************************************************************
* 27-AUG-2012 Initial revision [KRB].
*****************************************************************************/



#include "support_common.h"


/*****************************************************************************
* FUNCTION: busy_delay()
*
* DESCRIPTION
* Implements a very sleazy busy delay which just kills time. The main problem with
this technique is that it is
* not entirely clear what the value of p_delay should be to delay for n microseconds
or m milliseconds. We will
* see a better way to do this later on using the DMA timers (DTIM module).
*************************/
static void busy_delay(int p_delay)
{
	int i, x = 1;
	for (i = 0; i < p_delay; ++i)
	{
		x = x * 2 + 1;
	}
}
```

```
/*********************************************************************
* FUNCTION: led_off()
*
* DESCRIPTION
* Turns LED 1, 2, 3, or 4 off.
*
* INPUTS
* p_led should be 1, 2, 3, or 4.
*********************************************************************/
static void led_off(int p_led)
{
	MCF_GPIO_CLRTC &= ~(1 << (p_led - 1));
}



/*********************************************************************
FUNCTION: led_on()
*
* DESCRIPTION
* Turns LED 1, 2, 3, or 4 on.
*
* INPUTS
* p_led should be 1, 2, 3, or 4.
*********************************************************************/
static void led_on(int p_led)
{
	MCF_GPIO_SETTC |= 1 << (p_led - 1);
}


/*********************************************************************
FUNCTION: main()
*
* DESCRIPTION
* Flashes LED's 1 and 3 on and off at about a 2 Hz rate.
*
*    declspace(noreturn) tells the compiler that even though main() returns an int,
don't complain about the
* fact that we do not have a return statement at the end of the function. The C
language specification states
* very clearly that main() must return an int. But, this program will never end, so
we forgo the return state-
* ment.
*********************************************************************/
  declspec(noreturn) int main()
{
```

```
        /*
        * Program Port TC Pin Assignment Register (PTCPAR) so pins 0, 1, 2, and 3 are
        configured for the general-purpose I/O (GPIO) function.
        */
        MCF_GPIO_PTCPAR    =    MCF_GPIO_PTCPAR_PTCPAR0(MCF_GPIO_PTCPAR_DTIN0_GPIO)    |
        MCF_GPIO_PTCPAR_PTCPAR1(MCF_GPIO_PTCPAR_DTIN1_GPIO)                            |
        MCF_GPIO_PTCPAR_PTCPAR2(MCF_GPIO_PTCPAR_DTIN2_GPIO)                            |
        MCF_GPIO_PTCPAR_PTCPAR3(MCF_GPIO_PTCPAR_DTIN3_GPIO);

        /* Program Port TC Data Direction Register (DDRTC) so pins 0, 1, 2, and 3 are
        configured as output pins. */
        MCF_GPIO_DDRTC    =    MCF_GPIO_DDRTC_DDRTC0    |    MCF_GPIO_DDRTC_DDRTC1    |
        MCF_GPIO_DDRTC_DDRTC2 | MCF_GPIO_DDRTC_DDRTC3;

        /* Turn off all LED's. */
        led_off(1); led_off(2); led_off(3); led_off(4);

        /* Forever... */
        while (1) {
        /* Turn on LED's 1 and 3. */
        led_on(1);
        led_on(3);


        /* Kill time for around a quarter of a second. */
        busy_delay(500000);

        /* Turn off LED's 1 and 3. */
        led_off(1);
        led_off(3);

        /* Kill time for around a quarter of a second. */
        busy_delay(500000);
        }
}
```

Build the project and run it in the debugger or run it out of the debugger by hitting Ctrl+F11.

Question 11: Using the debugger and the Memory Map view, give the binary contents of the PTCPAR and DDRTC registers after the code enters the endless while loop?

Verify that LED's 1 and 3 on the MCF52259 microcontroller board flash on and off at around a 2 Hz rate 14, i.e., both on for around a quarter of a second, both off for around a quarter of a second, both on for around a quarter of a second, both off for around a quarter of a second, etc.

Once that works, modify the program so the LED's flash the binary numbers from 0000 to 1111 so a binary number is displayed for approximately one quarter of a second, then the LED's are off for approximately one quarter of a second, then the next number is displayed, then off, and so on. When 1111 is displayed, repeat by starting at 0000 again. For example,

LED4	**LED3**	**LED2**	**LED1**	**Comment**
Off	Off	Off	Off	Displaying 0000
Off	Off	Off	Off	*quarter-second delay* All LED's off
Off	Off	Off	On	*quarter-second delay* Displaying 0001
Off	Off	Off	Off	*quarter-second delay* All LED's off
Off	Off	On	Off	*quarter-second delay* Displaying 0010
Off	Off	Off	Off	*quarter-second delay* All LED's off
Off	Off	On	On	*quarter-second delay* Displaying 0010
Off	Off	Off	Off	*quarter-second delay* All LED's off
Off	On	Off	Off	*quarter-second delay* Displaying 0100
...				*quarter-second delay*
On	On	On	Off	Displaying 1110 *quarter-second delay*
Off	Off	Off	Off	All LED's off *quarter-second delay*
On	On	On	On	Displaying 1111 *quarter-second delay*
Off	Off	Off	Off	Displaying 0000 *quarter-second delay*
Off	Off	Off	Off	All LED's off *quarter-second delay*
Off	Off	Off	On	Displaying 0001 *quarter-second delay*

8. Submission Instructions

Type your solution to the question(s) using a word processor or text editor. At the top of the document, type the following information:

(1) Your name and your partner's name if you worked with someone else;

(2) Course number and name, i.e., CSE325 Embedded Microprocessor Systems;

(3) Semester, i.e., Spring 2013;

(4) Lab project number, e.g., Lab Project 2;

(5) Effort division between the team members, e.g., team member 1: 40%, team member 2: 60%.

Type your programming solution in the CodeWarrior IDE and export your main.c file from the project.

Convert answer document into Adobe PDF format and name the PDF file *cse325-s13-p01-lastname.pdf*, where *lastname* is your last name (if you do not have a last name, use your first name). If you worked with a partner then name your PDF file *cse325-s13-p01-lastname1-lastname2.pdf* where *lastname1* and *lastname2* are the surnames of both partners. Zip your source code and answer pdf into one zip file named *cse325-s13-p01-lastname.zip*, where *lastname* is your last name (if you do not have a last name, use your first name). If you worked with a partner then name your PDF file *cse325-s13-p01-lastname1-lastname2.zip* where *lastname1* and *lastname2* are the surnames of both partners.

Submit the zip file on the blackboard before the end of the due time. Late submissions WILL NOT be accepted.