CPSC 462 – Lab Manual

Microprocessor System Design Using Coldfire Embedded Processor

Originally developed by Texas A&M students:

Marshall Belew

Delilah Dabbs

Terry Dahlke

Brian Sladecek

© 2000-2002, Texas Engineering Experiment Station

Table of Contents

LAB 1: INTRODUCTION TO THE COLDFIRE DEVELOPMENT ENVIRONMENT 1

1.1 OBJECTIVE 1
1.2 INTRODUCTION 1
1.3 DETAILS OF THE BLOCK DIAGRAM 1
1.4 MONITOR/DEBUG SOFTWARE 6
1.5 SETTING UP THE TERMINAL AND THE BOARD 10
1.5.1 Setup the Coldfire board 10
1.5.2 Setup the Terminal 10
1.5.3 Logging Output to a File 11
1.5.4 Transferring a file from the PC to the MCF5206eLITE 11
1.6 PROCEDURE 13
1.7 ASSIGNMENT 15

LAB 2: ASSEMBLY PROGRAMMING ON THE MCF5206ELITE BOARD 16

2.1 OBJECTIVE 16
2.2 INTRODUCTION 16
2.2.1 Assembly Commands 17
2.2.2 Writing a Program 18
2.2.3 Defining Constants 19
2.2.4 System Calls 19
2.2.5 Assembling the Program 20
2.3 ASSIGNMENT 21

LAB 3: C PROGRAMMING WITH EMBEDDED ASSEMBLY CODE 24

3.1 OBJECTIVE 24
3.2 INTRODUCTION 24
3.3 PROCEDURE 26
3.3.1 – Sample Code 26
3.3.2 – Writing Two Useful Functions 26
3.5 ASSIGNMENT 27
ATTACHMENT A - USING GCC FOR THE LAB 27

LAB 4: LED OUTPUT AND TIMING 28

4.1 INTRODUCTION 28
4.2 PROCEDURE 28
4.3 ASSIGNMENT 29

LAB 5: LCD DEVICE DRIVER 30

5.1 OBJECTIVE 30
5.2 INTRODUCTION 30
5.3 PROCEDURE 30
5.4 ASSIGNMENT 33

LAB 6: SERIAL COMMUNICATION 37

6.1 OBJECTIVE 37
6.2 INTRODUCTION 37
6.3 PROCEDURE 37
6.3.1 – Example Header File for your C code .... **Error! Bookmark not defined.**
6.3.1 Lab Assignment 39
6.3.2 Write up .... **Error! Bookmark not defined.**

REFERENCES 46

Lab 1: Introduction to the ColdFire Development Environment

1.1 Objective

The purpose of this lab is to introduce the M5206eLITE Board and the Monitor/Debug Software. Basic information about the M5206eLITE hardware is covered, and an introduction to the dBUG Software is given.

1.2 Introduction

The M5206eLITE is a versatile single board computer based on the MCF5206e ColdFire Processor. The ColdFire is a descendant of the Motorola 68000 microprocessor family. The board may be used as a powerful microprocessor based controller in a variety of applications. With the addition of a terminal and keyboard, it serves as a complete microcomputer system for development/evaluation, training and educational use.

It is possible to expand the memory and I/O capabilities of the board by connecting additional hardware via the Microprocessor Expansion Bus connectors, although it may be necessary to add bus buffers to accommodate additional bus loading. Since the connectors require use of a printed circuit board (PCB), we do not make use of this expansion capability in class.

Provisions have been made on the printed circuit board to permit configuration of the board in a way which best suits an application. Options available are: up to 32 MB of asynchronous DRAM (ADRAM), 1 MB Fast SRAM (FSRAM), two timers, two general purpose I/O (GPIO) chips, and 1 MB of Flash EEPROM. All of the processor's signals are also available via connectors J1 and J2 for expansion purposes.

The M5206eLITE board provides FSRAM, Flash EEPROM, RS232 and all the built-in I/O functions of the MCF5206e for learning and evaluating the attributes of the MCF5206e. The MCF5206e is a member of the ColdFire processor family. The processor has eight 32-bit data registers, eight 32-bit address registers, a 32-bit program counter and a 16-bit status register, which includes the condition codes. A block diagram of the board is shown in Figure 1 and a diagram of the board level system details is shown in Figure 2. For more detailed information about this board, refer to the M5206eLITE User's Manual. This can be found in Appendix B of this manual.

1.3 Details of the Block Diagram

<u>Flash Rom</u>

The MCF5206eLITE board comes with one Flash EEPROM chip, which is programmed with the debugger/monitor firmware (dBUG). This AM29LV800BB Flash EEPROM is 16-bits wide and gives a total of 1 MB (512Kx16-word) of flash memory. The first 128K and the last 128K are reserved by the Monitor Firmware, however the middle 768K are available to the user. The chip-select signal (-CS0) that is generated by the board enables this chip. In order to avoid accidental

damage to the dBUG monitor, we will not reprogram the flash memory in class.

ADRAM SIMM Socket

The MCF5206eLITE board is equipped with one 72-pin SIMM socket (CN-1) for ADRAM. This socket supports ADRAM SIMM's of 256Kx32, 1Mx32, 4Mx32, and 8Mx32. No special configurations are needed, because the dBUG will detect the total memory installed when the board is powered-up. We do not use this SIMM socket in class.

THE CN1 Connector Pin Assignment

Pin No.	Signal Name	Pin No.	Signal Name
1	VSS	72	VSS
2	DQ0	71	N.C.
3	DQ16	70	PD4
4	DQ1	69	PD3
5	DQ17	68	PD2
6	DQ2	67	PD1
7	DQ18	66	N.C.
8	DQ3	65	DQ15
9	DQ19	64	DQ31
10	VCC	63	DQ14
11	N.C.	62	DQ30
12	A0	61	DQ13
13	A1	60	DQ29
14	A2	59	VCC
15	A3	58	DQ28
16	A4	57	DQ12
17	A5	56	DQ27
18	A6	55	DQ11
19	A10	54	DQ26
20	DQ4	53	DQ10
21	DQ20	52	DQ25
22	DQ5	51	DQ9
23	DQ21	50	DQ24
24	DQ6	49	DQ8
25	DQ22	48	-N.C.
26	DQ7	47	-W
27	DQ23	46	N.C.
28	A7	45	-RAS1
29	N.C.	44	-RAS0
30	VCC	43	-CAS1
31	A8	42	-CAS3
32	A9	41	-CAS2
33	-RAS3	40	-CAS0
34	-RAS2	39	VSS
35	N.C.	38	N.C.

UART's

The board also comes with 2 UARTs for serial communications. The UARTs have independent baud rate generators. The debugger uses UART1 in order to give the user access to the terminal. In other words, the UART1 channel is the "TERMINAL" channel used by the debugger for

communication with the PC. The signals for channel one are passed through external Driver/Receivers to make the channel compatible with RS-232, which is available on connector J9. The "TERMINAL" baud rate is set at 19200. The signals for UART2 are not available off the board.

The J9 (Terminal) Connector Pin Assignment

Pin No.	Direction	Signal Name
1	Output	Data Carrier Detect (shorted to 6)
2	Output	Receive Data
3	Input	Transmit Data
4	Input	Not Connected (shorted to 1 & 6)
5		Signal Grounded
6	Output	Data Set Ready (shorted to 1 & 4)
7	Input	Request to Send
8	Output	Clear to Send
9		Not Used

The J4 Connector Pin Assignment

Pin No.	Direction	Signal Name
1		3.3V
2	Output	Clear to Send
3	Input	Request to Send
4	Output	Receive Data
5	Input	Transmit Data
6		Signal Ground

Parallel I/O Ports

The MCF5206eLITE processor offers one 8-bit general-purpose parallel I/O port. Each pin can be individually programmed as input or output. However this port is not directly available to us on the board, so we do not use it. There are also two memory-mapped bus transceivers that are controlled via chip select 3 (CS-3). The first transceiver (U14) drives the 7-segment LED display and is configured for output only. The A7 line of the U14 also drives the direction control signal of the U15 transceiver. This allows the U15 to be used for both input and output. The signals for U15 are provided by the J10 connector, which is referred to as the GPIO. The value read or written to J10 appears on bits D16 to D23 of the data bus. The direction of the data on J10 is controlled by D31 of the read/write to J10. The GPIO will be used in class. The GPIO value is also available on an open-collector buffer on J11. This is primarily useful for driving high-voltage signals.

The J10 Connector Pin Assignment

Pin No.	Signal Name
1	DATA 0
2	DATA 1
3	DATA 2
4	DATA 3
5	DATA 4
6	DATA 5
7	DATA 6
8	DATA 7

Timer/Counter

The processor has two built in general-purpose 16-bit timer/counters. The J1 and J2 connectors provide the signals for the timer/counters. J1 and J2 actually provide all of the processor signals. The details of these connectors are available on the course Web site.

The J1 Connector Pin Assignment

Pin No.	Signal Name	Pin No.	Signal Name
1	A0	80	BKPT
2	A1	79	DS0
3	A2	78	DSCLK
4	A3	77	DS1
5	GND	76	GND
6	A4	75	-RESET
7	A5	74	TCK
8	A6	73	SDA
9	A7	72	TT1
10	A8	71	TT0
11	A9	70	ATM
12	A10	69	-TS
13	A11	68	-ATA
14	A12	67	-TA
15	A13	66	SIZ0
16	A14	65	SIZ1
17	A15	64	R/-W
18	GND	63	GND
19	A16	62	CLK
20	A17	61	-HIZ
21	A18	60	N.C.
22	A19	59	-CS0_OFF
23	A20	58	-DREQ0
24	A21	57	-ILP2
25	A22	56	-ILP1
26	A23	55	-ILP0
27	D0	54	-BR
28	D1	53	-BR_HW
29	GND	52	GND
30	D2	51	-BG_HW
31	D3	50	-CS3
32	D4	49	-CS2
33	D5	48	-CS1
34	D6	47	-CS0
35	D7	46	PST0
36	D8	45	D15
37	D9	44	D14
38	D10	43	D13
39	D11	42	D12
40	GND	41	GND

The J2 Connector Pin Assignment

Pin No.	Signal Name	Pin No.	Signal Name
1	D16	80	N.C.
2	D17	79	N.C.
3	D18	78	N.C.
4	D19	77	N.C.
5	GND	76	GND
6	D20	75	N.C.
7	D21	74	N.C.
8	D22	73	N.C.
9	D23	72	SCL
10	D24	71	TOUT1
11	D25	70	-JTAG
12	D26	69	N.C.
13	D27	68	VCC
14	D28	67	VCC
15	D29	66	VCC
16	D30	65	VCC
17	D31	64	N.C.
18	GND	63	GND
19	A24	62	N.C.
20	A25	61	N.C.
21	A26	60	N.C.
22	A27	59	N.C.
23	N.C.	58	TIN1
24	N.C.	57	-DREQ1
25	N.C.	56	N.C.
26	N.C.	55	-CTS2
27	PST1	54	-RTS2
28	-TEA	53	TXD2
29	GND	52	GND
30	PST2	51	RXD2
31	PST3	50	-CTS1
32	-BG	49	-RTS1
33	DDATA0	48	TXD1
34	DDATA1	47	RXD1
35	DDATA2	46	-DRAMW
36	DDATA3	45	-CAS3
37	-BD	44	-CAS2
38	-RAS0	43	-CAS1
39	-RAS1	42	-CAS0
40	GND	41	GND

Registers and Memory Map

The MCF5206eLITE has built-in logic and 4 chip-select pins (-CS0 to –CS3) that are used to enable external memory and I/O devices. In addition the ADRAM uses two –RAS lines (-RAS0 and –RAS1) as chip-selects. There are registers to specify the address range, type of access, and the method of –TA generation for each chip-select. –TA is the acknowledgment that is sent to indicate the presence of a new device. The registers are then programmed by dBUG to map the external memory and I/O devices. In order to declare user defined memory spaces you have to

re-compile and up-load the dBUG software. This concept is further explained in Lab 4: Memory Interface. The board uses –CS0 to enable the Flash ROM, -CS2 for FSRAM, and –CS3 for GPIO space. The following table is a memory map of the MCF5206eLITE, which shows the address ranges that correspond to each signal and device.

M5206eLITE Memory Map	
Address Range	Signal and Device
$00000000 - $003FFFFF	-RAS1, -RAS2, 4M bytes of ADRAM's
$10000000 - $100003FF	Internal Module Registers
$20000000 - $20001FFF	Internal SRAM (8K bytes)
$30000000 - $300FFFFF*	-CS2, External FSRAM (1M byte – 256Kx32)
$40000000 - $4000FFFF	-CS3, 64K bytes of GPIO
$FFE00000 - $FFEFFFFF	-CS0, 1M byte of Flash EEPROM (512Kx16)

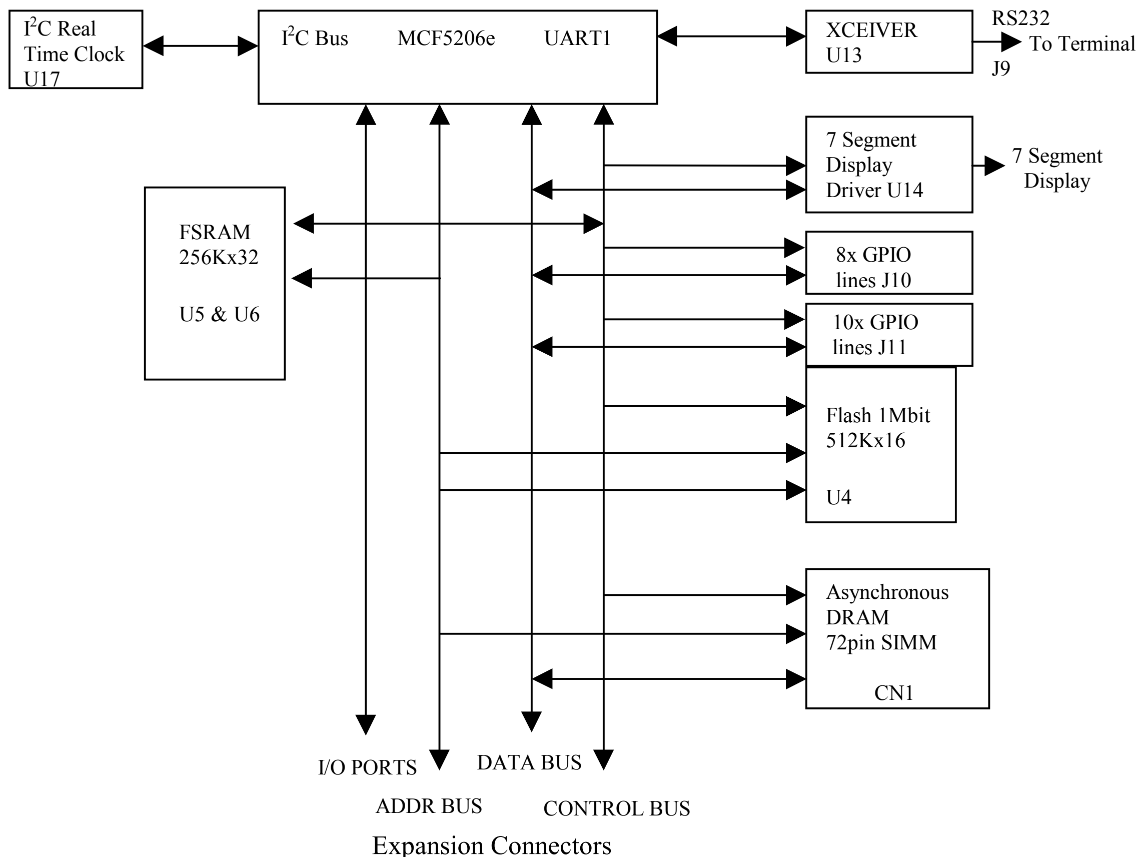


Figure 1. Block Diagram of the Board

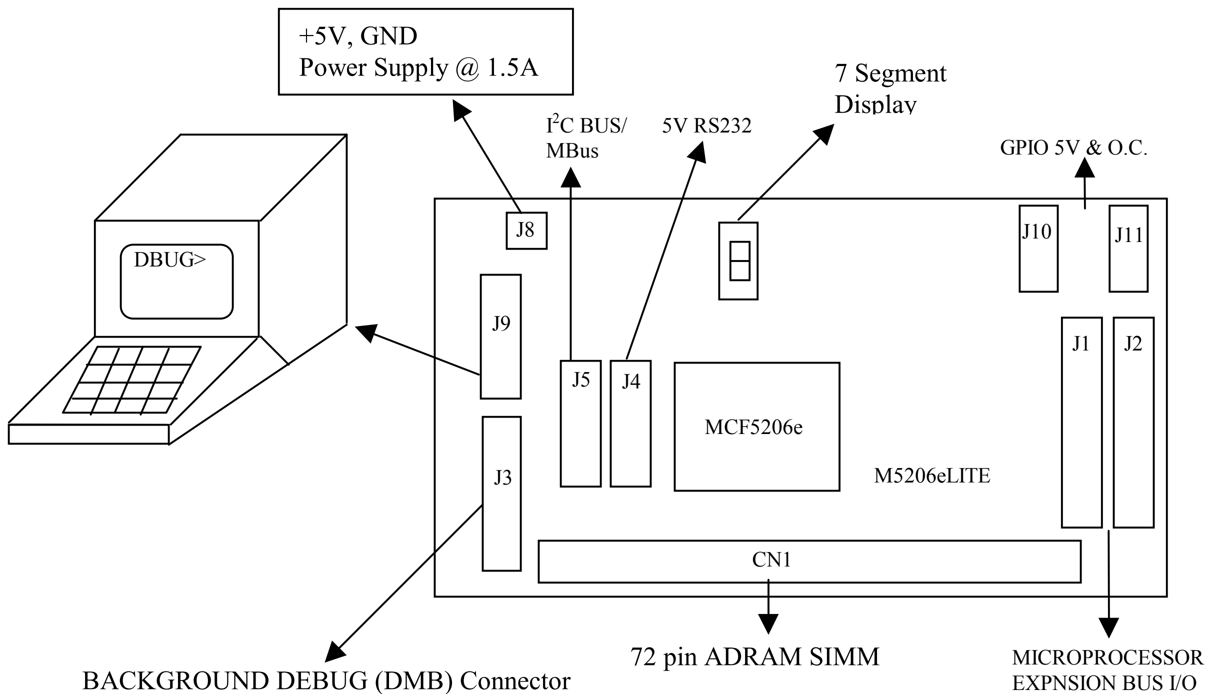


Figure 2. System Configuration

1.4 Monitor/Debug Software

The M5206eLITE Computer Board has a resident firmware package that provides a self-contained programming and operating environment. The firmware, named "dBUG", provides the user with monitor/debug, disassembly, program download, and I/O control functions. The firmware (stored in one 512Kx16 Flash EEPROM device) provides a self-contained programming and operating environment.

The user interacts with dBUG through pre-defined commands that are entered via an RS232 line from the PC. The user interface to dBUG is the command line. A number of features have been implemented to achieve an easy and intuitive command line interface. dBUG assumes that it is communicating with a dumb 80x24 terminal, so since we are using a PC, we use the terminal emulator. For serial communications, dBUG requires the emulator be set to eight data bits, no parity, and one stop bit, 8N1. The baud rate is 19200 baud, which can be changed after power-up.

The command line prompt is "dBUG>". Any dBUG command may be entered from this prompt. Command lines cannot exceed 80 characters. It echoes each character as it is typed, eliminating the need for any "local echo" on the terminal side.

In general, dBUG is not case sensitive. Commands may be entered either in upper or lower case, depending upon the user's equipment and preference. Only symbol names require that the exact case be used.

Most commands can be recognized by using an abbreviated form of the name. For instance, entering "h" is the same as entering "help". Thus, it is not necessary to type the entire command name.

The commands DI, GO, MD, STEP, and TRACE are used repeatedly when debugging. Therefore dBUG allows for repeated execution of these commands with minimal typing. After a command is entered, simply press <RETURN> to invoke the command again. The command is executed as if no command line parameters were provided. ***NOTE: STEP and TRACE may not work correctly with repeated execution.***

An additional function called the "TRAP 15 handler" allows the user program to utilize various routines within dBUG. There are four TRAP #15 functions, which include OUT_CHAR, IN_CHAR, CHAR_PRESENT, and EXIT_TO_dBUG.

OUT_CHAR – sends a character, which is in the lower 8 bits of D1, to the terminal.
IN_CHAR – return an input character from the terminal to D1.
CHAR_PRESENT – checks if an input character is present to receive.
EXIT_TO_dBUG – transfers program control back to dBUG.

Examples of the TRAP #15 functions can be found on pages 34-36 of the M5206eLITE on-line User's Manual.

After the system initialization on power up or reset, the board waits for a command-line input from the user terminal. When a proper command is entered, the operation continues in one of the two basic modes. If the command causes execution of the user program, the dBUG firmware may or may not be re-entered depending on the operation of the user's code. In the alternate case, the command entry mode is entered.

For commands that accept an optional <width> to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no <width> option is provided, the default width is .W.

The core ColdFire register set is maintained by dBUG. These are listed below:
A0 – A7
D0 – D7
PC
SR

All control registers on ColdFire are readable only via the supervisor-programming mode, and thus not accessible via dBUG. User code my change these registers, but caution must be exercised as changes may render dBUG useless.

A reference to “SP” actually refers to “A7”. The user memory is located at addresses $30020000 - $300FFFFF, $300FFFFF is the maximum address of the FSRAM memory installed on the board. The user should limit his/her activities to this area of the memory map. Address range $30000000 - $3001FFFF is used by dBUG.

If a command causes the system to access an unused address, a bus trap error will occur. This results in the terminal printing out a trap error message and the contents of all the MCF5206e core registers. Control is then returned to the dBUG monitor.

Several keys are used as command line edit and control functions. These functions include:

<RETURN> - will enter the command line and cause processing to begin
<Delete> or CTRL-H – will delete the last character entered
CTRL-D – Go down in the command history buffer
CTRL-U – Go up in the command history buffer
CTRL-R – Recall and execute the last command entered

A list of dBUG commands can be found in Table 1. For an individual description of each of these commands, refer to pages 18-33 of the M5206eLITE on-line User’s Manual located in Appendix B of this manual.

Table 1. dBUG Commands

COMMAND	DESCRIPTION	SYNTAX
AS	ASSEMBLE	AS <addr><instruction>
	BLOCK COMPARE	BC<FIRST><SECOND><LENGTH>
	BLOCK FILL	BF<WIDTH>BEGIN END DATA
BM	BLOCK MOVE	BM BEGIN END DEST
BS	BLOCK SEARCH	BS<WIDTH> BEGIN END DATA
BR	BREAKPOINT	BR ADDR <-R><-C COUNT> <-T TRIGGER>
DATA	DATA CONVERT	DATA VALUE
DI	DISASSEMBLE	DI <ADDR>
DL	DOWNLOAD SERIAL	DL <OFFSET>
GO	EXECUTE	GO <ADDR>
GT	Go TILL BREAKPOINT	GT <ADDR>
HELP	HELP	HELP <COMMAND>
IRD	INTERNAL REGISTER DISPLAY	IRD <MODULE.REGISTER>
IRM	INTERNAL REGISTER MODIFY	IRM MODULE.REGISTER><DATA>
MD	MEMORY DISPLAY	MD <WIDTH><BEGIN><END>
MM	MEMORY MODIFY	MM <WIDTH>ADDR<DATA>
RESET	RESET	RESET
RD	REGISTER DISPLAY	RD <REG>
RM	REGISTER MODIFY	RM REG DATA
SET	SET CONFIGURATIONS	SET OPTION<VALUE>
SHOW	SHOW CONFIGURATIONS	SHOW OPTION
STEP	STEP(OVER)	STEP <NUM>
SYMBOL	SYMBOL MANAGEMENT	SYMBOL <SYMB><-a SYMB VALUE> <-R SYMB><-C|L|S>
TRACE	TRACE(INTO)	TRACE <NUM>
UPDBUG	UPDATE DBUG	UPDBUG
UPUSER	UPDATE USER FLASH	UPUSER
VERSION	SHOW VERSION	VERSION

The STEP and TRACE commands may only work correctly with a single iteraction (i.e. do not give them a numerical argument).

1.5 Setting up the Terminal and the Board

Be sure to read the Assignment section below before proceeding so that you will know exactly what is required. You will need to capture the output of the following steps into a file called "debugger.txt"on your PC.

1.5.1 Set up the ColdFire board - already done by instructor

Note: the computer MUST be off when working with the parallel port connections.

- Plug serial connection from COM2 on computer to J9 on Coldfire.
- Connect BDM ribbon cable into J3 on Coldfire.
- Connect parallel extension cable from BDM to computer parallel port.
- Connect J8 (Coldfire power) to a 5 volt 1.5 amp source.
- Power the computer, but leave the Coldfire board powered down for now.

1.5.2 Setup the Terminal

Go to Start->Accessories->HyperTerminal->HyperTerminal

Window: New Connection
For 'Name' use "ColdFire"
Click OK

Window: Connect To
For 'Connect using' select COM2
Click OK

Window: COM2 Properties	
For 'Bits per second'	select 19200
'Data bits'	select 8
'Parity'	select None
'Stop bits'	select 1
'Flow Control'	select Xon / Xoff

Power on or reset the ColdFire board.
If the MCF5206eLITE is powered on and properly connected, hitting the ENTER key should always give you the dBUG> prompt.

```
Hard Reset
FSRAM Size: 1M

Copyright 1997-1999 Motorola, Inc.  All Rights Reserved.
ColdFire MCF5206e EVS Debugger v1.4.7 (Mar  2 1999 13:04:24)
Enter 'help' for help.

dBUG>
```

1.5.3 Logging Output to a File

Use the following steps to transfer the output of a program to a file.

1. Choose the "Capture Text" option from the HyperTerminal "Transfers" menu.

2. Choose the file that you want to save to.

   Warning: If you want to use a file that does not exist you have to explicitly create one. To create a new file, right-click in the file browser area and choose "New->Text Document".

3. In order to stop the data transfer, go to the "Capture Text" option under the "Transfer" menu and select "Stop". This command will also allow you to pause your data transfer.

4. Your new file can be viewed using the Notepad.

The dBUG command set has a help facility. If you want to see a list of all the available commands, type "help" at the dBUG prompt. If you want specific help on a particular command, type "help" followed by the command name. For example, "help rd" will give you the syntax of the register dump command.

1.5.4 Transferring a file from the PC to the MCF5206eLITE

Programs are downloaded to the ColdFire board from the PC. For the following example, the file "TEST.x68" containing MCF5606e assembly code can be found on the C drive of your computer. You need to assemble the program and convert it to a S-Record before transferring it to the MCF5206eLITE board.

Assembling and converting into S-Record

1. Open a command prompt
2. The assembler is located in the directory C:\68k, make sure the path is set to this directory
3. Assemble the TEST.x68 file by typing:

   x68k –L TEST

4. Now the S-Record needs to be created from the.BIN which should be generated from the previous command, you do this by typing the following:

   s68k TEST

5. This will create a new file called TEST.REC, you can view this file in wordpad.

The next step is to transfer the file to the MCF5206eLITE

1. Start a terminal session.

2. At the dBUG> prompt, type "dl" and press Enter. This tells the assembler to store the instructions you input starting at the memory location specified in the TEST.x68 program with the "ORG" (origin) command. This is location 30020000. If a program does not specify the starting memory location, you can do so by including it with the "dl" command, as in "dl 30020000". Note that if the program has an "ORG" statement and you also add a location in the "dl" command, the two values will be added together, probably putting the program out of memory and generating an error message during the transfer below.
3. Now choose the "Send Text File" option from the "Transfers" menu. When the dialog box appears, choose the appropriate drive, directory and file name then press Enter. The program will automatically stop the transfer when it reaches the end of the file. You will not see any indication that they file is being transferred. Also, this may take a few minutes, so be patient.
4. To view the transferred data type "di 30020000" at the dBUG> prompt.
5. Compare the contents of "TEST.x68" with the disassembled code displayed in the terminal window. Verify that the transfer was done correctly.

1.6 Procedure

1. Put the following assembly code into memory, starting at memory location 30020000, using the assembler command (AS).

```
CLR.L    D0
CLR.L    D1
CLR.L    D2
CLR.L    D3
CLR.L    D4
CLR.L    D5
CLR.L    D6
CLR.L    D7
MOVE.L   #$0000,D0
TRAP     #15
.
```

2. Now type `DI 30020000` to invoke the disassembler and verify the code that you just typed.

```
dBUG> di 30020000
30020000: 4280                CLR.L     D0
30020002: 4281                CLR.L     D1
30020004: 4282                CLR.L     D2
30020006: 4283                CLR.L     D3
30020008: 4284                CLR.L     D4
3002000A: 4285                CLR.L     D5
3002000C: 4286                CLR.L     D6
3002000E: 4287                CLR.L     D7
30020010: 203C 0000 0000      MOVE.L    #0x00000000,D0
```

3. Dump the register set.

```
dBUG> rd
PC: 00000000 SR: 2700 [t.Sm.111...xnzvc]
An: 0000FFFF 00000000 00000000 00000000 00000000 00000000 00000000 300FFFF0
Dn: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

4. Using the register modify command (RM), change the value of the program counter (PC) to $30020000 and change the values of registers D0-D7 to $FFFF0000. Set the values of registers A0 through A6 to $0000FFFF.

```
dBUG> rm pc 30020000
dBUG> rm d0 ffff0000
dBUG> rm d1 ffff0000
dBUG> rm d2 ffff0000
dBUG> rm d3 ffff0000
dBUG> rm d4 ffff0000
dBUG> rm d5 ffff0000
dBUG> rm d6 ffff0000
dBUG> rm d7 ffff0000
dBUG> rm a1 0000ffff
dBUG> rm a2 0000ffff
dBUG> rm a3 0000ffff
dBUG> rm a4 0000ffff
dBUG> rm a5 0000ffff
dBUG> rm a6 0000ffff
```

5. Dump the contents of the register set again.

```
dBUG> rd
```

```
PC: 30020000 SR: 2700 [t.Sm.111...xnzvc]
An: 0000FFFF 0000FFFF 0000FFFF 0000FFFF 0000FFFF 0000FFFF 0000FFFF 300FFFF0
Dn: FFFF0000 FFFF0000 FFFF0000 FFFF0000 FFFF0000 FFFF0000 FFFF0000 FFFF0000
```

6. Now, run the program that you input above (in Step 1) by typing the command "GO". This command begins execution at the memory address contained in the program counter.

7. Look at the register again and note the effect that your program has had on the registers.

```
dBUG> rd
PC: 30020018 SR: 2704 [t.Sm.111...xnZvc]
An: 00000000 0000FFFF 0000FFFF 0000FFFF 0000FFFF 0000FFFF 0000FFFF 300FFFE8
Dn: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

8. Using the memory modify command, enter the following words starting at address $30010000:
ADAD ADAD ADAD DADA 0000 FFFF F0F0 CCCC

```
dBUG> mm 30010000
30010000:  [08EA]  ADAD
30010002:  [EF9F]  ADAD
30010004:  [CDC7]  ADAD
30010006:  [10D6]  DADA
30010008:  [90F7]  0000
3001000A:  [B7CF]  ffff
3001000C:  [8BE3]  f0f0
3001000E:  [8CA4]  cccc
30010010:  [0A7F]  .
```

9. Verify the data that you just entered by using the memory display command. Use the parameters of the command to group the data as it is grouped above and display only the values that you just typed in.

```
dBUG> md 30010000 30010010
       30010000:  ADAD ADAD ADAD DADA 0000 FFFF F0F0 CCCC ................
```

10. Using the memory modify command, write the string "CPSC462!" into the block of memory beginning at address $30010000.

```
dBUG> mm.b 30010000
30010000:  [AD]  43
30010001:  [AD]  50
30010002:  [AD]  53
30010003:  [AD]  43
30010004:  [AD]  34
30010005:  [AD]  36
30010006:  [DA]  32
30010007:  [DA]  21
30010008:  [00]  .
```

11. Use the memory display command to show the string you just typed in byte format. Once again, set the arguments of the command to show only the data you have entered.

```
dBUG> md.b 30010000 30010007
30010000:  43 50 53 43 34 36 32 21 00 00 FF FF F0 F0 CC CC CPSC462!........
```

12. Invoke the assembler at memory location 30020000 and type in the following code:

```
MOVE.L #$7,D0
MOVE.L #$9,D1
ADD.L  D0,D1
loop:  BRA loop
```

13. Run the program you entered in the last step. If you entered the program correctly, your board should get stuck in an infinite loop at memory address 30020000E.

Press the ABORT (s1) button on your board to interrupt the program's execution. Dump the register set, and return to the dBUG prompt.

```
dBUG> go 30020000


Hard Reset
FSRAM Size: 1M

Copyright 1997-1999 Motorola, Inc.  All Rights Reserved.
ColdFire MCF5206e EVS Debugger v1.4.7 (Mar  2 1999 13:04:24)
Enter 'help' for help.

dBUG> rd
PC: 00000000 SR: 2700 [t.Sm.111...xnzvc]
An: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 300FFFF0
Dn: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

14. Set the program counter to 30020000 and registers D0 and D1 to 0F00AAAA using any of the commands you have already learned.

```
dBUG> rm pc 30020000
dBUG> rm d0 0F00AAAA
dBUG> rm d1 0F00AAAA
dBUG> rd
PC: 3005FFFF SR: 2700 [t.Sm.111...xnzvc]
An: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 300FFFF0
Dn: 0F00AAAA 0F00AAAA 00000000 00000000 00000000 00000000 00000000 00000000
```

15. Trace through the program one step at a time until you have crossed the branch instruction at least three times.

1.7 Assignment

1. Demonstrate that you are able to transfer a file from the PC to MCF3206e memory and that you are able to log the output of the dBUG to a file on the PC.

2. Turn in a printout of the file “debugger.txt”.

3. Give a brief (1-2 paragraph) summary of the each of the components listed in the Block Diagram.

4. Use each of the following debugger commands and capture the output of each to a file. Turn in a printout of this captured output and briefly explain how each of the commands works.
   BM BS

5. How can you specify the number of instructions to be traced by dBUG’s trace command?

Lab 2: Assembly Programming on the MCF5206eLITE Board

2.1 Objective

The following exercises will introduce you to some of the basics of assembly language programming on the M5206eLITE microprocessor. Read through the lab in its entirety before you begin. The 68000 assembly language instructions provided in your textbook are mostly identical the MCF5206eLITE instructions. If you need further explanation about certain assembly language instructions, consult the ColdFire Microprocessor Family Programmer's Reference Manual in Appendix C.

2.2 Introduction

The ColdFire Family programming model consists of two register groups: User and Supervisor. Programs executing in the user mode use only the registers in the user group. System software executing in the supervisor mode can access all registers and use the control registers in the supervisor group to perform supervisor functions.

Figure 2.1 illustrates the user programming model. The model is the same as for the M68000 Family microprocessors described in your textbook. It consist of the following registers:

- 16 general purpose 32-bit registers (D0-D7,A0-A7)
- 32-bit program counter(PC)
- 8-bit condition code register (CCR)

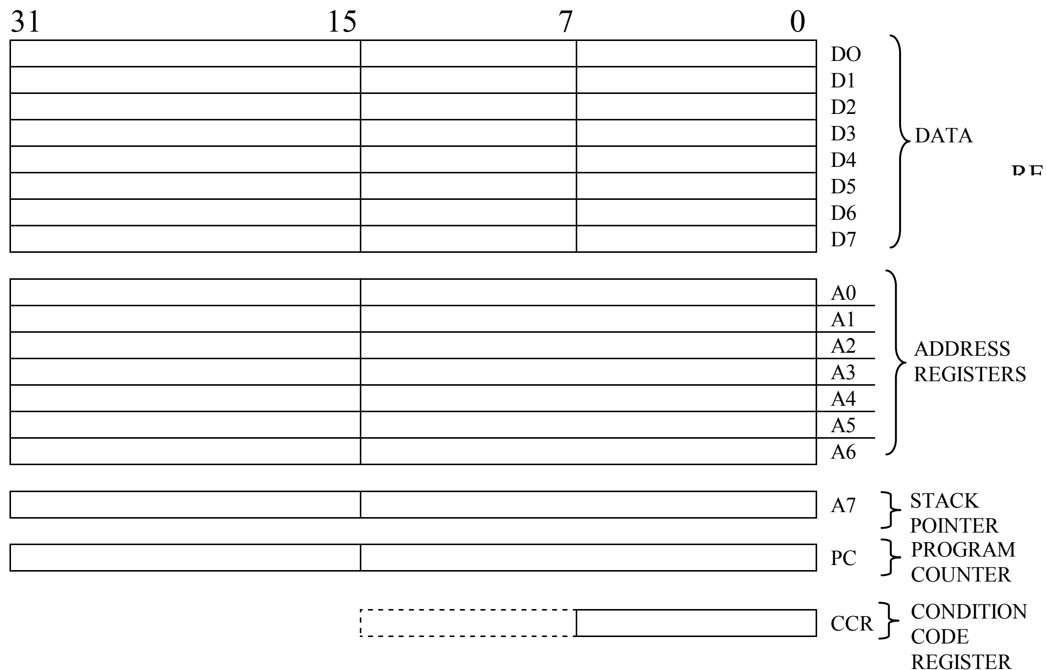


Figure 2.1 – User Programming Model

Each data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Longword operands occupy the entire 32 bits of integer data registers. Because address registers and stack pointers are 32 bits wide, address registers cannot be used for byte-size operands. When an address register is a source operand, either the low-order word or the entire longword operand is used, depending on the operation size.

ColdFire supports a single hardware stack pointer (A7) for explicit references or implicit ones during stacking for subroutine calls and returns and exception handling. The PC contains the address of the currently executing instruction. The CCR is the least significant byte of the processor status register (SR). Bits 4-0 represent indicator flags based on results generated by processor operations.

4	3	2	1	0
X	N	Z	V	C

- X – extend bit
- N – negative bit; set if most significant bit of the result is set
- Z – zero bit; set if the result equals zero
- V – overflow bit; set if an arithmetic overflow occurs
- C – carry bit; set if a carryout of the MSB occurs for an addition

Full access to the SR is only allowed in supervisor mode. The following diagram is a layout of the status register.

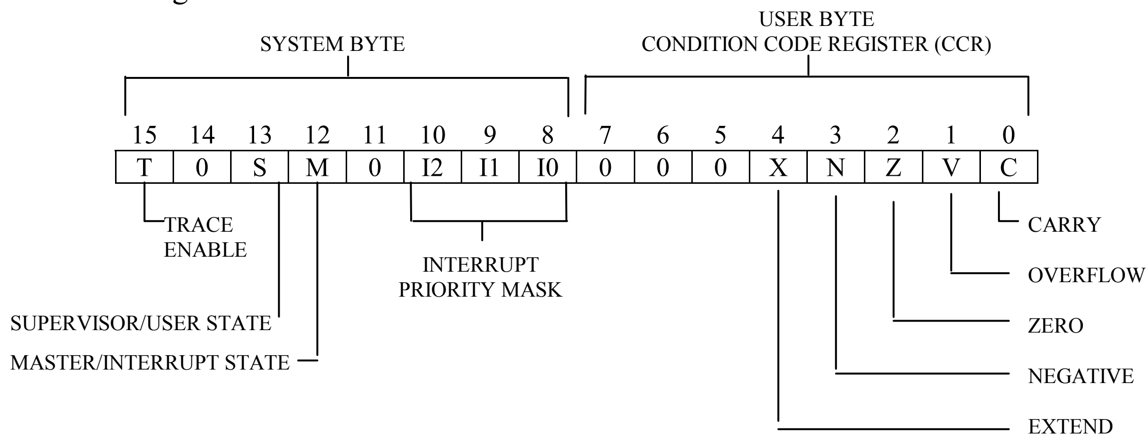


Figure 2.2 – Status Register

2.2.1 Assembly Commands

In order to complete this lab, you will need to familiarize yourself with some of the basic assembly instructions. Four of them have been provided here, but a complete list of all the available instructions can be found in Appendix C.

LEA (Load Effective Address) – Loads the effective address into the specified address register. This instruction affects all 32 bits of the address register. A more detailed description of this instruction can be found on page 4-44 of the Programmer's Reference Manual.

MOVE (Move Data from Source to Destination) – Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or longword. A more detailed description of this instruction can be found on page 4-53 of the Programmer's Reference Manual.

CMPI (Compare Immediate) – Subtracts the immediate data from the destination operand and sets the condition codes according to the result; the destination location is not changed. The size of the operation and the immediate data is specified as a longword. A more detailed description of this instruction can be found on page 4-31 of the Programmer's Reference Manual.

Bcc (Branch Conditionally) – If the specified condition is true, program execution continues at location (PC) + displacement. The program counter contains the address of the instruction word for the Bcc instruction, plus two. The displacement is the two's complement integer that represents the relative distance from the current program counter to the destination program counter. Condition code cc specifies one of the following conditional tests:

E	=
GE	>=
GT	>
LE	<=
LT	<
NE	!=

A more detailed description of this instruction can be found on page 4-15 of the Programmer's Reference Manual in Appendix C.

2.2.2 Writing a Program

First of all, it is important that every line that does not have a label begin with a TAB character. Lines with labels should also have a TAB immediately after the label. Also, comment strings should begin with the "*" character. It can be used on any line in the program and all text after a "*" is ignored by the assembler.

Every assembly program needs to be located somewhere in memory. In the cross assembler this is accomplished by the ORG statement. For example:

```
ORG $30020000
```

will locate a code section at the address $30020000. There can be several ORG instructions in a program, one for each module. The only restriction is that the addresses are in increasing order from the beginning of the program. ORG $30030000 followed by ORG $30020000 will not be accepted.

The END instruction is needed to tell the assembler where to stop assembling. This also requires an address argument. You can use the address of the first ORG as the argument to the END. There should only be one END per program file.

2.2.3 Defining Constants

The assembler provides an EQU instruction for declaring constants much like the #define directive in C. For example:

```
BUFFER EQU $30030000
```

Declares a constant called "BUFFER" which is equal to $30030000. Everywhere in the program where "BUFFER" is referenced, the assembler will replace it with the constant value $30030000.

> Note: The lack of a prefix indicates a decimal number, a percent sign (%) indicates a binary number, and a dollar sign ($) indicates a hexadecimal number.

To create data constants in memory, the assembly instruction DC is used. It can either be byte, word or longword. For example:

```
PROMPT     DC.B $16
```

Creates a data constant named PROMPT that is set to $16. You can also declare space for variables with the DS instruction. For example:

```
INPUT      DS.L   1
STRING     DS.B   30
```

Declares space for 1 longword at the address of label "INPUT" and an array of 30 bytes at the address of "STRING".

2.2.4 System Calls

System calls are used to perform input/output functions that are already available through the MCF5206eLITE hardware. For system calls on this cross assembler, you must use the TRAP #15 instruction.

> **TRAP** (Trap) – Causes a TRAP #<vector> exception. The instruction adds the immediate operand (vector) of the instruction to 32 to obtain the vector number.

The four TRAP functions were introduced in Lab #1. For example:

```
MOVE.L    #$0000,DO
TRAP      #15
```

Is used for a normal exit to the dBUG. The codes associated with OUT_CHAR, IN_CHAR, CHAR_PRESENT, and EXIT_TO_dBUG are #$0013, #$0010, #$0014, and #$0000 respectively. A very thorough explanation of traps can be found on pages 463 – 465 of your textbook.

2.2.5 Assembling the Program

The cross-assembler software is located in the directory on the C:\68k drive of the PC's. After you write your program in a DOS text file, save it with the filename extension ".x68" in this directory. This will enable the cross-assembler to find the program. Make sure that your FILENAME is in caps.

You should save your program in C:\cpsc462. Before assembling , make sure that the path is set correctly. To set the path, goto **Control Panel->System->Environment** and in the box where it says Variable, type PATH and in the box where it says Value, type c:\68k and press Set button.

Open a DOS prompt, change directory to C:\cpsc462.

Then assemble the program by typing:

```
x68k -L FILENAME
```

If this is successful it will create a .BIN and .LIS listing.

Uploading and Running the M5206eLITE

Once you have assembled the program successfully, you will need to upload it to the M5206eLITE board. In order to upload your file to the board you will need to create an s-record. You do this by typing the following:

```
s68k FILENAME
```

This will create a new file called filename.rec, which is created from the .BIN file. Type at the dBUG> dl
then go to the "TRANSFER" menu and select "SEND TEXT FILE" then select the file "FILENAME.rec."

Note: If your program does not have a ORG statement that tells the loader where to put it in memory, then type
dBUG>dl 30020000 (i.e you are explicitly specifying the address where to dump the code, in this case its $30020000)

2.3 Assignment

1. Writing a Subroutine in Assembly

You will program a subroutine that starts at address $30021000 to copy a block of memory (in bytes) from one location to another without using any extra memory for temporary storage. In your program, assume that before entering the subroutine the following addresses are loaded into the registers:

A0 = The starting address of the source block
A1 = The length of the block to be copied
A2 = The starting address of the destination block

For Example:

In order to copy a memory block of 8 bytes from location $30020000 to $30020024, the user should only have to enter the starting address of the source block, the length of that block, the starting address of the destination block, and type 'GO'. Before you run the subroutine, you want to dump the memory at both the source and destination address. Then Lines 2-4 will set the registers A0, A1, and A2 to the starting address, length, and destination address respectively. Line 5 will run your subroutine. After you run the subroutine, you want to dump the memory at both the source and destination address to verify that the memory was moved to the proper location. The following is an example of what your terminal screen should look like when you are running the program.

```
1.  dBUG> MD 30020000
30020000:   0534 00FF 1357 0031 FFFF 3210 0000 1111
30020010:   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
30020020:   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF

2.  dBUG> GO
3.  dBUG>  MD 30020000
30020000:  0534 00FF 1357 0031 FFFF 3210 0000 1111
30020010:  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
30020020:  FFFF FFFF 0534 00FF 1357 0031 FFFF FFFF
```

Also, your subroutine should take care of all possible special cases.
The special cases are:

`Case 1: (A0+A1) < A2`	The subroutine should operate normally. Set the carry bit to zero, and the negative bit to zero.
`Case 2: A0 < A2 <= (A0+A1)`	This will cause part of the data to be overwritten before it is moved unless you find a way to move it in a different order. Set the carry bit to one and the negative bit to zero.

`Case 3: A0 = A2` — In this case, it should move nothing at all. Set both the carry and the negative bit to one.

In order to set the carry and negative bits you have to use the MOVE statement. However, you have to find a way to set just the carry and negative bits without changing the values of the other CCR bits.

2. Integrating A Calling Function
   Write a calling function that will set the values of A0, A1, and A2. Your calling function should store these values onto the program stack (A7). Upon return, the values should be popped off of the stack. The last value on the stack should be the information from the status register.

   You will modify your subroutine to read the address values from the stack. When the subroutine is finished, it should return back to the main program. When branching to and from the subroutine, you will need to use the BSR and RTS instructions. The statement: `BSR $30020500` will branch to the subroutine that is located at $30020500. When the subroutine reaches a `RTS` statement is will return to the calling program.

3. Demonstrate your program to the T.A.

4. Turn in a hard of your subroutine (Question 1) and the calling function (Question 2).

Notes:

- When you are using instructions like CPMA or ADDA , make sure that you also specify the length(.B or .W or .L), like **CMPA.L A0,A1** or **ADDA.L A0,A1** (sometimes after disassembling you might see DC.W instead of the instruction you typed in, usually this happens if you miss out the length)
- Some 68k commands are partially compatible with the Coldfire, so when you disassemble it and it looks no way similar to what you typed, then its probably not compatible. This might mean that the opcode is too large or too small. Look in the Coldfire instruction set manual to compare the instruction you are trying to use. Normally for small programs the x68k compiler should work fine.

- Here is a outline of how your program might look like, this is just an example, you are free to write it the way you want to.

```
MAIN        EQU   $30020000   ADDRESS WHERE THE MAIN PROGRAM WILL RESIDE
MOVEMEM     EQU   $30021000   ADDRESS WHERE THE MOVEMEM SUBROUTINE WILL
RESIDE

MEMFROM     EQU   $30022000   BEGINNING ADDRESS OF MEMORY BLOCK TO BE COPIED
LENGTH      EQU   $00000008   LENTGH OF MEMORY BLOCK TO BE COPIED
MEMTO       EQU   $30022012   BEGINNING ADDRESS OF MEMORY BLOCK COPYING TO
```

```
                ORG     MAIN            THE MAIN PROGRAM..i.e THE CALLING FUNCTION

                BSR     MOVEMEM         BRANCH TO THE SUBROUTINE

EXIT            MOVE.L  #$0000,D0        SELECT EXIT_TO_dBUG
                TRAP #15                MAKE THE TRAP CALL AND EXIT TO dBUG

*--------------------------------------------
*     MOVEMEM SUBROUTINE
*--------------------------------------------

                ORG MOVEMEM

* INITIALIZE THE UPPER AND LOWER BOUNDS
                LEA     LENGTH,A4         Move the length into A4
                LEA     MEMFROM,A0            Load the beginning address into A0
                LEA     MEMTO,A2              Load the beginning copy to address into A2

                CLR.L   D0              CLEAR D0 - USED FOR TEMPORARY MEMORY STORAGE


*               YOUR BLOCK MOVING ALGO SHOULD BE HERE
                .............
                .............
                .............


                RTS


*--------------------------------------------
*     DATA SEGMENT
*--------------------------------------------
                ORG MEMFROM             THIS PIECE OF CODE JUST FILLS UP
*                                       THE MEMORY BLOCK WITH SOME DATA SO THAT
*                                       YOU KNOW WHAT IS BEING MOVED

                MOVE.L  D0,D1             FILL MEMORY WITH A RECOGNIZABLE
                MOVE.L  D1,D2             PATTERN.
                MOVE.L  D2,D3
                MOVE.L  D3,D4
                MOVE.L  D4,D5
                MOVE.L  D5,D6
                MOVE.L  D7,D0

                END MAIN
```

Lab 3: C Programming with Embedded Assembly Code

3.1 Objective

The focus of this lab will be to integrate assembly code into a C program. Microprocessor system designers are starting to use high-level languages instead of assembly language to control hardware interfaces. However the utilization of assembly still allows for much faster execution for high-availability systems. This lab will introduce you to the basic concepts of using the high-level language C for systems programming.

3.2 Introduction

From previous labs, you know that the M5206e family supports three basic data sizes: the byte (8), word (16) and longword (32), which can either by signed or unsigned. However, the C language supports four basic data types: int (integer), char (character), float (real), and double (double-precision floating point). Just like the M5206e data types can be signed or unsigned, the C data types can be signed, unsigned, longword (long) or word (short). Long and short indicate the number of bits that will be assigned, which will be either 32 or 8 respectively. The following table was taken from page 147 of your textbook. It gives a better description of the C data types.

Data Type	C name	Width (bits)	Range
Integer	Int	16	-32,768 to 32,767
Short Integer	Short int	8	-128 to 127
Long integer	Long int	32	-2,147,483 to 2,147,483,647
Unsigned integer	Unsigned int	16	0 to 65,535
Character	Char	8	0 to 255
Single-precision floating point	Float	32	10^{-38} to 10^{+38}
Double-precision floating point	Double	64	10^{-300} to 10^{+300}

Once you have compiled the C code, it is converted into assembly. However, you can directly embed assembly code into your program. The following C instruction will embed the assembly instruction "MOVEA.L   A0,A1" into a C program. The "__asm__" is a C directive that tells the compiler the following text is an assembly instruction.

```
__asm__ (" MOVEA.L %A0,%A1" );
```

Also instead of directly embedding each line of assembly, you can write an assembly file and link it to your C program. Then you can call your assembly subroutine like any other C subroutine. Here is an example of a C program (test.c) that calls an assembly function (_writeme (P)) located

in testasm.s):

```
test.c

int _writeme(int *P);

int main(void) {

        int *P=0x30010000;
        _writeme(P);

        return 0;
}
```

```
testasm.s

_writeme:
                move.l  4(%sp),%A0
                move.l  #0xff013,%D0
                move.l  %D0,(%A0)
                rts

               .globl _writeme
```

The first line of the C program is a prototype for function '_writeme'. The underscore in the name tells GCC that we are using an imported assembly function. The arguments of _writeme is a pointer of type 'int'. The main program declares the pointer, and assigns it to a memory location. Note that in most programming practices, we do not assign pointers to absolute memory locations. In systems programming, we can break this rule. The argument 'P' passed to _writeme is the absolute address that we are going to access from the assembly program.

The first line of the assembly program is a label. The compiler turns this label into an address for use in a jump subroutine. The arguments from the calling C program are pushed onto the program stack (sp). We do NOT pop the value from the stack, this is taken care of once the procedure returns. We merely access it by copying a long word (`0x30010000`) into A0, starting from the 5th byte of the stack pointer. The last line is a compiler directive that exports _writeme to the compiler's symbol list. _writeme from the C program will now be the same _writeme in the assembly program.

Notice also that we have used the '%' symbol in front of the register. This is a syntax used by GCC for accessing all registers. GCC is capable of recognizing two different syntax variations: MIT and Motorola. Motorola is the preferred and most widely used syntax for 68000 chipsets. Refer to **Appendix F** for discrepancies.

The following code is the compiled version of test.c:

test.c compiled version

```
      .file "test.c"
gcc2_compiled.:
__gnu_compiled_c:
.text
      .even
.globl main
main:
      link.w %a6,#-4
      jsr __main
      move.l #805371904,-4(%a6)
      move.l -4(%a6),-(%sp)
      jsr _writeme
      addq.l #4,%sp
      clr.l %d0
      jbra .L1
      .even
.L1:
      unlk %a6
      rts
```

3.3 Procedure

3.3.1 – Sample Code

Copy and save **test.c** and **testasm.s**. Compile and link the files using the 'make' command with the given makefile. See the TA for the makefile source code.

The output file is **test.x**. This is an s-record, much like lab 2's .rec type of file. Upload this to the board and execute it. Use the debugger to find the answer to question 1.

3.3.2 – Writing Two Useful Functions

For this lab you need to write two subroutines in assembly language that will function similarly to *printf* and *scanf* statements. Then write a C program that calls these assembly subroutines. The C program should prompt the user to input a word or a sentence, and then ask them to verify their input by typing it again. Then the program should tell the user if their inputs match. Do not worry about implementing %c, %d, %f type values. Assume all input is of type string. See section 3.4 Question 4 for further instructions.

Example:
Enter a word:
prompt> hello
re-enter your word:
prompt> hello
Correct!
Enter a word:
prompt> hello
re-enter your word:
prompt> good-bye
Incorrect!

3.5 Assignment

1. What address does the first line of testasm.s get mapped to?

2. We want to use the command ‘movea.l (a0)+,d0’ . How does this look in Motorola syntax? MIT syntax?

3. What are some reasons to use absolute addressing?

4. From the assignment given in Section 3.3.4, turn in the following three files: your assembly code , your C code, and the modified makefile.

Attachment A - Using GCC for the lab

GCC stands for GNU C Compiler. GNU is pronounced “ganoo”, and cleverly stands for “GNU’s Not UNIX”. It is a command-line driven compiler that is executed using a makefile. A makefile is nothing more than a script that puts together command-line options. We will use GCC to compile both C and assembly.

From an MS-DOS prompt, type ‘c:\coldfire\setenv.bat’. This will set up your path for using GNU’s make command.

Three files are required to compile test.x: test.c, testasm.s, my5206elite.ld, and makefile. The given makefile can be modified to work with any program to be uploaded to the 5206. The file called my5206elite.ld is a text file that contains the memory map for the board. The board comes with 1 megabyte of physical memory.

In the directory that contains these files, type ‘make test’. Make will first look at target ‘test’ and check to see if the dependent targets are created:

```
test:   test.o testasm.o
     $(CC) $(LDFLAGS) -o test.x test.o testasm.o
```

if not, it will then compile test.c and testasm.s into the two target object files test.o and testasm.o. The output file is s-record test.x.

The memory map ensures that the program will start at user address 0x30020000.

Lab 4: LED Output and Timing

4.1 Introduction

The goal of this lab is to learn how to do basic output with the 7-segment LED display, how to do timing loops, and how to do basic input/output with the functions you wrote in Lab 3.

The MCF5206eLITE board contains an 8-segment LED display that contains the standard seven segments plus a decimal point. The goal of this lab is for you to learn how to do basic output to this display, and how to generate accurate timing using loops. You will also use the string I/O functions you wrote in Lab 3.

The LED appears as a byte at address 0x40000000. The segments are turned on by writing a 1 into the corresponding bit of the byte. You must determine the correspondence by reading the schematics and experimentation. The 8-bit general-purpose I/O (GPIO) interface is located at address 0x40000001. Both the LED and GPIO interfaces are controlled with the same write signal, rather than separate byte write enables. Therefore this memory location actually looks like a word (e.g. unsigned short int) at 0x40000000, with the high byte being the LED and the low byte the GPIO.

4.2 Procedure

The read and write enable for the LED and GPIO are controlled via chip select 3 (CS3), as part of the ColdFire's chip select logic. This logic is explained in detail in the ColdFire Microprocessor User's Manual. Your code must have the following chip select initialization before you can write to the LED:

```
#define MBAR 0x10000000

/* Chip select 3 address, mask, control registers */
#define CSAR3 (*(unsigned short* volatile) (MBAR + 0x00000088))
#define CSMR3 (*(unsigned long *volatile) (MBAR + 0x0000008C))
#define CSCR3 (*(unsigned short* volatile) (MBAR + 0x00000092))
…
main(void)
{
  /* CS3 0x40000000 - 0x4000FFFF */
  CSAR3 = 0x4000;       /* start at 0x40000000 */
  CSMR3 = 0x00000000;   /* 64kb of address space */
  CSCR3 = 0x0183;       /* 2-byte wide with autoacknowledge */

  <insert rest of program >
}
```

You will see in the MCF5206eLITE board manual that by default the GPIO address space starts at 0x40000000. However it is not set up for the fact that the LED and GPIO cannot acknowledge the CPU and it does not have byte write enable. That is why the above code is needed for your program to work correctly.

You are to write a C program that does the following:

1. Prompt the user and then read a hexadecimal number from the terminal. The number may include a decimal point. You must handle both upper and lower case alphabetical characters. The number may optionally be preceded by “0x” or “$” that must be stripped off. You must use the input and output functions written in Lab 3.
2. Write the number on the LED in an unambiguous and readily understandable fashion, displaying each character for one second, starting with the most significant character. Generate your delays using a delay subroutine. This subroutine should take the number of milliseconds as an argument, and then execute the appropriate number of iterations of a loop to generate a delay. Your loop should be parameterized by a compile-time constant giving the ColdFire clock frequency, which is 54 MHz. That way the code can be readily modified for a processor of different speed. The display time should be as close to one second as you can make it. Some instruction timings are listed in the ColdFire Microprocessor User’s Manual.
3. Prompt the user again when the number display is complete. Exit the program if the user types “exit”.

4.3 Assignment

1. Implement your program and demonstrate it to the TA.
2. Document and justify how you display each character. Explain how each is unambiguous and readily understandable.
3. Document how you made your character timing very close to one second.
4. Explain what each statement in the chip select code given above does. The documentation is given in the ColdFire Microprocessor User’s Manual.

Lab 5: LCD Device Driver

5.1 OBJECTIVE

The purpose of this lab is to write a device driver interfacing an LCD character display to the GPIO. The device driver provides a high-level interface to communicate with a hardware device, and hides many of the details of using it. You will write a simple interactive program using the driver.

5.2 INTRODUCTION

In order to implement this lab, you will use the GPIO interface of the MCF5206e. As discussed in Lab 4, the GPIO is located at address 0x40000001, after the proper chip select initialization code has been executed. This forms a word with the LED display. You will use the GPIO to communicate with the Optrex LCD DMC20434 character display.

5.3 PROCEDURE

You will first write a device driver for the LCD. This will consist of a library of subroutines and state variables that hide the details of communicating with the LCD. You will then write a client program using the device driver.

The Device Driver
A device driver is a set of subroutines which simplify the interface with I/O devices so that client programs can communicate with them. For this lab you will write an LCD device driver and a client program using its functions. Your device driver will handle the bit control and read/write operations while your client program will simply call the subroutines and pass data in parameters.

The LCD device driver must implement the subroutines given below. The parameters will be passed on the stack, and any strings should be passed by reference (i.e. address).

- **Reset()** - This function should not only clear the display, but perform all necessary steps to initialize the display after a system reset. The cursor mode after a reset should be blinking.
- **CursorPosition(x)** - Moves the cursor to column **x** on the display. Must not affect any data already displayed.
- **ShiftDisplay(x,y)** - **x** is the character 'r' or 'l' specifying a right or left shift, respectively. **y** is an integer which specifies how many characters to shift the display. This should be able to shift any number from 1 to 80 in decimal.
- **Backspace()** - Should behave just like the "Backspace" key on your keyboard. i.e., it moves the cursor back one space and deletes the character at that position.
- **Write(string)** - Writes a null-terminated character string to the display at the current position. **string** should be the address of a null-terminated array of ASCII characters.
- **CursorControl(x)** - **x** will be the letter 'b' for blinking, 'l' for line or 'x' to turn off the

cursor.

- **DisplayOff()** - Deactivates the display.
- **DisplayOn()** - Reactivates the display with the same settings as when it was turned off with DisplayOff. This means the cursor should be in the same mode and the same text should be visible as before.

Client Program

Now it will be necessary to develop an application which uses the device driver subroutines. The client should be a command line user interface that reads 4-character command strings with parameters and parses those to determine which subroutine to call. The following commands should be recognized: *rset, cpos, shft, bksp, writ, crsr, dpof, dpon*. These correspond to the device driver subroutines. Whenever a command is entered that does not match one of these, the client should respond with ``What?". Here is an example of how these commands should work.

```
dBUG> GO 30020000

LCD_CONTROL> rset         [Initialize the display]
LCD_CONTROL> crse b       [bad instruction]
What?
LCD_CONTROL> crsr b       [Set the cursor to blink]
LCD_CONTROL> shft l 3     [Shift the display left 3 characters]
LCD_CONTROL> cpos 5       [Move the cursor to position 5]
LCD_CONTROL> writ Hello   [Type ``Hello'' on the display]
LCD_CONTROL> DPOF         [Turn off the display]
LCD_CONTROL> DPON         [Turn on the display]
LCD_CONTROL> quit         [Return to dBUG]

dBUG>
```

The interface should also be *case-insensitive*.

You may use the I/O subroutines that you implemented in Lab 3, or you may use I/O subroutines from the C library. The TA can give you advice on I/O subroutines available in dBUG.

5.3.1 Interface Design

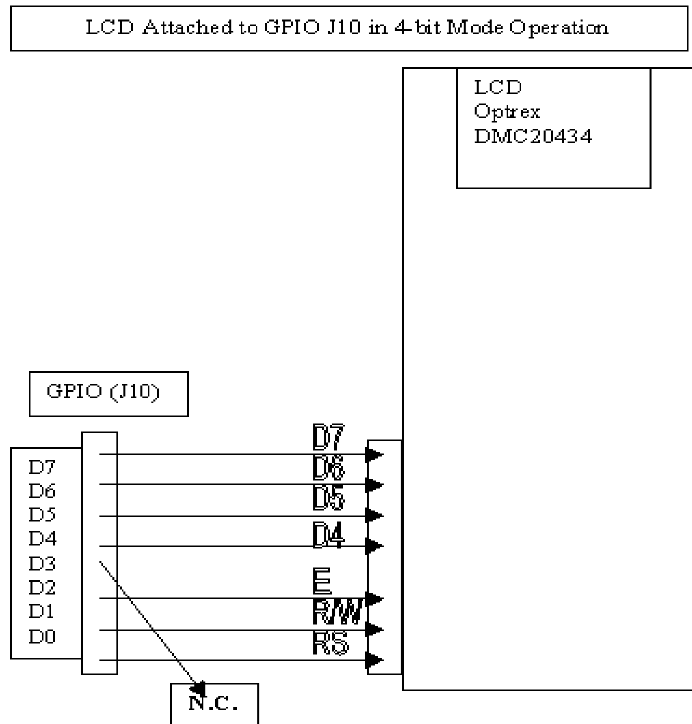
The first step in implementing the LCD device is to figure out how it will be interfaced with the MCF5206eLITE board. Since your parallel port is only 8-bits wide, the LCD must run in 4-bit mode. In this mode you use 4 data I/O lines, 1 Enable Clock, 1 Read/Write control, and 1 Register Select. One pin is not used. The TA will already connect the LCD to the GPIO for you.

You should be very careful with the LCD and GPIO interfaces. All the old LCD modules were burned out by students who hooked up the power supply backwards, etc. If you are not certain, ask the TA.

The direction of the bidirectional GPIO is controlled by bit 7 of the LED byte. When this bit is 0, the GPIO is writing to the LCD. When the bit is 1, it reads from the LCD.

When you write to the GPIO and LED bytes, the data is latched in the MC74LCX646DT bidirectional latch/drivers. (You can find out more about these chips by looking for the 646 data

sheet at http://www.ti.com). It then remains stable on the GPIO outputs as long as the direction control (bit 7 of the LED) is 0. When the GPIO is read, the value returned is the value that is on the GPIO inputs at that instant. The data is also latched, but each read gets the new instantaneous value. The LCD driver can be implemented by doing only output to the LCD, so you can just write to the LED and GPIO bytes without worrying about the details of latching.



Your first interface challenge is to implement the basic write cycle timing. This requires obeying the timing requirements of the interface, using timing loops similar to what you wrote in Lab 4. For long delays you can use your existing routine. For shorter delays you may choose to implement a separate timing routine. The LCD will always work using delays longer than the minimum, but you should not use delays so large that the LCD appears slow to a human. The details of the LCD initialization and command sequences, and the timing of read and write cycles are all given in the Optrex module manual. The manual of the Hitachi controller chip is also available, but is only supplemental.

One simple way to debug your code is to write it to the LED instead of the GPIO using long time delays, so that you can watch the sequence of signals. You can then shift it to the GPIO and speed it up.

5.4 Assignment

1. How much larger are the time delays that you are using compared to the minimum required by the Optrex module?
2. The LCD controller has a Busy bit that you can read to determine if the LCD is ready for the next command. How would you change your program to use this, rather than using time delays to wait? If you use the Busy bit, do you still need timing loops in your program? Why? Would this reduce the amount of CPU time used by the device driver?
3. Demonstrate your device driver and client program to the TA and turn in your code.

Lab 6: Keypad Entry

6.1 OBJECTIVE

The goal of this lab is to learn how to interface an input device to the microcomputer, in this case a keypad. The processor will poll it to determine whether a key press is available. The keypad data will be used to generate a tone using the timer module.

6.2 INTRODUCTION

In order to implement this lab, several modules on the MCF5206e must be used. This board offers one 8-bit general-purpose I/O port, two built in general-purpose 16-bit timer/counters, and two 80-pin connectors for accessing the bus.

6.2.1 INTERFACING HARDWARE

You will get the chance to build some external hardware to interface the keypad to the GPIO. The primary problem is that keys bounce when pressed and released, and your hardware must filter out the bouncing to indicate when a key value is stable and ready for the processor. One of the timer modules of the MCF5206e will be used to drive a speaker interfaced with the board. The input from the keypad will be used to set the frequency of the timer and cause different tones at different clock periods.

6.2.2 OVERVIEW

This lab will give you the opportunity to build the necessary hardware to interface with the board and generate the appropriate tones to the speaker.

6.3 PROCEDURE

This lab consists of two parts, the first part of this lab involves building the keypad interface, and the second part involves generating the speaker tone.

6.3.1 Keypad Interface Design

The keypad is a 4x4 array of keys that implements a switchpoint matrix. When a key is pressed, the corresponding row and column wires are connected. The key bounces for a number of milliseconds when it is pressed or released, and this bouncing must be filtered out, so that only one key press or release is seen by the processor.

The keypad interface hardware must perform two basic functions. The first is to scan the keypad to detect when a key is pressed. The second is to debounce the key press, and present the key information to the GPIO along with a valid bit specifying that the key value is valid.

The simplest way to scan the keypad is to hook the column pins to pullup resistors, so that they float high when no keys are pressed. Then pull the rows low one at a time. If a key is pressed on the row pulled low, then the corresponding column will be pulled low, with bouncing. The

reason for using pullup resistors and active-low rows is that the TTL logic you will use can sink much more current when pulling a wire low than it can source current when pulling a wire high. It is possible for more than one key on a row to be pressed simultaneously, but you can assume that only one key is pressed at a time. You can then take the four-bit column value and encode it as a two-bit value. You then present the key data (4-bit row number, 2-bit column value) and valid bit to the GPIO. This leaves you one spare GPIO pin.

Remember that bit 7 of the LED must be a 1 in order for the GPIO to be an input device. When you read from the GPIO address, the processor gets the value that was on the GPIO input pins at that moment.

Your logic must sequence through the keypad rows fast enough that you don't miss keys being pressed, but sequencing too fast can cause you design problems. You can assume that users do not press keys faster than 3-10 per second. You can generate any necessary clocks from a crystal oscillator, dividing it down to the necessary frequencies.

When you detect that a key has been pressed, you must wait until it stops bouncing, and then put it on the GPIO input and then assert the valid bit. You must have a valid key before asserting the valid bit so that the CPU does not accidentally read an invalid key. When pressed, the key contact will rapidly open and close for a number of milliseconds before staying closed. Similarly when the key is released. As soon as you detect a key being released, you must negate the valid bit before the valid key value is removed from the GPIO. This prevents the CPU from accidentally reading an invalid key as it is released. The valid key value and valid bit should remain stable as long as the key is pressed.

The #1 challenge in this design is the asynchronous nature of the key press and bouncing. They could happen at any time relative to your scanning the keypad rows. You must also decide whether you want to try and stop scanning the rows when a key is pressed, or else scan nonstop.

The TTL data sheets are available at http://www.ti.com under the Digital Logic links. You should look at the LS logic family (e.g. 74LS04 is a hex inverter chip), as this is what we have.

We also have Motorola (now ON Semiconductor) MC14490 debouncer chips. Their documentation can be found at http://www.onsemi.com.

Solutions typically take 8-12 chips, depending on what chips are available.

6.3.2 Software Implementation

System Initialization

You need to set up the chip select for the GPIO as in prior assignments.

Keypad Processing

You need to write a program that periodically polls the GPIO to determine if there is a valid key value present, and if so, record it. A new key value should only be recorded if there was a period of time when there was no key present. This allows the detection of multiple presses of the same key. A key sequence is terminated with the '#' key. You must pull fast enough that you do not miss key presses.

Timer Programming
The number entered on the keypad is the frequency in Hertz that should be put out on the speaker. The speaker is hooked to the Tout pin of Timer 2 (labeled TOUT1). The timer must be programmed so that it puts out a square wave of the appropriate frequency. The speaker will approximate this as a sine wave. Until the first time a value is entered, the speaker should not put out any tone. The timer module description can be found in the Coldfire User's Manual. Your strategy should be to set up the timer module so that each new tone only involves modification of the timer reference register. This process will also require conversion of the number you typed into the suitable reference value.

6.4 Assignment

1. Implement and demonstrate the lab to the TA. Show how you can type a sequence of numbers on the keypad, terminate it with '#', and the corresponding tone in Hertz is produced on the speaker.
2. Turn in schematics and software.
3. What frequencies are audible to the human ear from your speaker?
4. Draw a design to debounce the keypad without using the debouncer chip. Explain how it works.

Lab 7: Software-Based Keypad Entry

7.1 Objective

The goal of this lab is to reimplement the Lab 6 keypad, using the least amount of hardware and using software interrupts.

7.2 Introduction

In Lab 6 you were required to implement keypad scanning hardware so that it would signal a valid bit on the GPIO when a 4 or 6-bit key value was present on the GPIO. You polled the GPIO in software to determine if a key was ready, and then processed it. In this lab you will implement as much of the keypad scanning in software as you can. There were two functions of the hardware in Lab 6. The first was to scan the keypad, using signals generated from a clock. The second function was to debounce a key press or release. In this lab, you will perform as much of those functions in software as possible.

Rather than using timing loops or busy waiting to check for the valid bit, you will use software interrupts to trigger the periodic scanning and polling of the keypad.

7.3 Procedure

Debouncing a key in software is relatively straightforward. When you read the GPIO, if you see a key pressed (e.g. at least one column line pulled low), then you remember what that key was, and then check some suitable time later to see if it is still pressed. If so, it is a valid key press. Similarly, when you see that key released, you record that the key is released and wait a suitable period of time before checking that key again, so that it stops bouncing. Software debouncing eliminates the need for debouncing hardware or valid bit.

One can still do row scanning in hardware, but simplify it by scanning at a faster rate. In the hardware solution the goal was to have the row times be long enough that the key had a chance to stop bouncing during a row time. That eliminated the need for the hardware to remember which key was pressed. However in software it is easy to remember which key was pressed, so row scanning can be much faster, and a key can be checked on a later scan to see if it is still pressed and stopped bouncing.

Rather than busy waiting or using timing loops to periodically access the GPIO, you should use Timer 1 (Timer 2 is used for your speaker output) to generate an interrupt after some time delay. The interrupt routine should then poll the GPIO. This is performed by programming the timer to generate an interrupt when it reaches the reference value, rather than just toggling the TOUT line.

To write an interrupt routine you must do two things. The first is to write an assembly-code interrupt handler wrapper routine that calls the actual C interrupt-handling code, and the second is to specify the interrupt vector. The timers are autovectored devices. You must program the

appropriate interrupt control register (ICR) to specify an appropriate interrupt priority level, which in turn specifies the interrupt vector. Your code must initialize that vector location to point to your interrupt handler wrapper.

Because interrupt handlers are not called by a procedure, they do not use RTS to return. Instead they use RTE. However the C compiler cannot generate an RTE. The solution is to write your interrupt handler in assembly code, with the code being nothing more than a JSR to your C function that actually implements the interrupt handling, followed by an RTE.

You have two basic options for keypad scanning. The first is the have the timer generate regular interrupts, at which time you do keypad processing. This is similar to a typical polling procedure. The second option is to schedule future interrupts depending on what is observed. So in the first approach, you might increment to the next row on one interrupt, then observe that row several times on following interrupts, then increment to the next row on the next interrupt, and so on. In the latter approach, you might increment to the next room and check for a key pressed. If a key is pressed, you could schedule an interrupt, and then check the key again when the interrupt occurs.

The hardware changes for converting to software-based key debouncing are straightforward: you simply feed the keypad columns directly into the GPIO, eliminating the debouncing hardware. To do row scanning in software is more challenging. You need to write out the two-bit row value to the 74LS139 decoder, and have it hold that value until the next time it is changed. The latter can be done by using a 74LS74 dual flip-flop or similar circuit, with 2 GPIO pins providing the data, and 1 pin being used to clock the flip-flops.

The challenge is that when writing to the flip-flops, the other GPIO pins are writing to the keypad columns. If one column is pulling low while a GPIO output is pulling high, a short-circuit occurs. One can avoid this by making sure that the GPIO pins are pulling low, but a software error can result in hardware damage. The solution is to put a series resistor between the GPIO pin and the keypad column, so that even if the GPIO is high and the keypad is low, excessive current will not flow. The problem is that if the resistor value is too high, then the keypad column may not be able to sink enough current for the GPIO to see that value as a zero. The GPIO is implemented by a 74LS646. It will source as much as 0.4 mA with a 0V input, and requires the input voltage to be below 0.5V. This implies that the resistor must be less than 1250 ohms. But the 74LS139 that is sinking the current via the column has a non-zero resistance, so the resistor must be even smaller. A value of 470 ohms is probably safe. The resistance of the 74LS646 output driver is at least 50 ohms, so this would limit the short-circuit current to less than 10 mA. The output values would not be correct, but that is not important.

In the case of the flip-flops, when the GPIO is reading, then the flip-flop data and clock inputs would be floating. The problem with this is that TTL interprets a floating input as high. Since the flip-flops are clocked on the rising edge of the clock input, if the input had been low and then floats high, it would be clocked accidentally. However one cannot use a pulldown resistor to keep it low. The reason is that the clock input can source as much as 3.2 mA when low, which would require a small resistor to keep it low. But the 74LS646 cannot source that much current when driving a high value, so the pulldown resistor would overwhelm it. The solution is to keep

the clock value normally high, and then to pulse it low and then high to clock values into the flip-flops. Then a pullup resistor of about 4.7 kohms can keep the clock high when it is floating. Similarly for the data inputs. Unlike the keypad columns, incorrect software programming will not cause any hardware damage, just incorrect operation.

7.3.1 Lab Assignment

1. Implement and demonstrate the lab to the TA. Show how you can type a sequence of numbers on the keypad, terminate it with '#', and the corresponding tone in Hertz is produced on the speaker.
2. Turn in schematics and software.
3. Explain why you chose the particular interrupt scheme and timing to scan and debounce your keypad.

Lab 8: Serial Communication

8.1 Objective

In this lab you will be introduced to the basics of communications through the serial interface provided by the MCF5206eLite evaluation board. This lab will utilize the serial communication ability of the MCF5206eLite to transmit data from MCF5206eLite board to the PC's HyperTerminal. Before beginning the lab, be sure to read chapter 9 (The Serial Input/Output Interface) in your text book dealing with the operations of serial communications with DUARTs, as well as the synchronous serial data transmissions and serial interface standards sections. It may also be beneficial at this to time read the preparatory material located in Appendix A of this lab in order to fully understand what the provided code is doing. Additionally there is sample code dealing with the UARTs and serial communications provided by Motorola in the appendix of this lab.

8.2 Introduction

The MCF5206e has two independent built in UARTs (Universal Asynchronous/Synchronous Receiver/Transmitters) available to the user that act independently. Each of these UARTs utilizes the system clock, which eliminates the need for an independent crystal. Some of the features of these UARTs are:

- The UARTs can be clocked by the system clock or an external clock
- Full duplex asynchronous/synchronous receiver/transmitter channel
- A Quadruple buffered receiver (bytes)
- A Double buffered transmitter (bytes)
- An independently programmable baud rate for both transmitter and receiver, dependant upon the internal or external clock
- Programmable data format
- Programmable channel modes
- Modem Control Signals (CTS & RTS)

The above features help to reduce the cost of MCF5206eLite board as well as making it more reliable and usable. By having the UARTs clocked by the system clock we no longer need to have a separate crystal on board for the UARTs. Ensuring that the system works in full duplex mode creates the fastest transfers possible using the serial connections. The buffered input/output limits the number of calls that must be made to the processor for menial repeats of data. The programmable baud rate generator allows the MCF5206eLite to be able to communicate with a number of other hardware at a variable speed.

The signals from UART1 are available through the DB9 connector on the MCF5206eLite board. The connections are as follows:

UART1 DB9 Pin Connections

Pin number	Direction (I/O)	Signal Name
1	Output	Data Carrier Detect (shorted to pin 1 & 6)
2	Output	Receive Data RxD
3	Input	Transmit Data TxD
4	Output	Not Connected (shorted to pins1&6)
5	NA	Signal Ground
6	Output	Data Set Ready (shorted to pins1& 4)
7	Input	Request to Send
8	Output	Clear to Send
9	NA	Not Used

Connectors J2 and J4 on the MCF5206eLite are also available to the user for serial communications. Connector J2 provides connection to the signals from both UARTs. Here are the necessary pin connections on J2 for serial communications:

UART 1 & 2 J2 Pin Connections

Signal Name	Pin Number
47	RxD1
48	TxD1
49	RTS1
50	CTS1
51	RxD2
52	GND
53	TxD2
54	RTS2
55	CTS2
57	DREQ1
58	TIN1

For this lab however, it will probably be easier and quicker to use the J4 or DB9 connectors. The pin connections for J4 are as listed below:

UART1 J4 Pin Connections

Pin Number	Direction	Signal Name
1	NA	3.3V
2	Output	Clear to Send (CTS)
3	Input	Request to Send (RTS)
4	Output	Receive Data (RxD)
5	Input	Transmit Data (TxD)
6	NA	Signal Ground

8.3 Procedure

8.3.1 – Example Header File for your C code.

Here is an example of the header file that you will need to use in your serial communications C code. This is only intended to be a starting point so there may be additions that need to be made.

```
/***************************************************************
                                UART1
***************************************************************/

#define UMR11  (*(unsigned char* volatile) (MBAR + 0x00000140))
#define UMR21  (*(unsigned char* volatile) (MBAR + 0x00000140))
#define USR1   (*(unsigned char* volatile) (MBAR + 0x00000144))
#define UCSR1  (*(unsigned char* volatile) (MBAR + 0x00000144))
#define UCR1   (*(unsigned char* volatile) (MBAR + 0x00000148))
#define URB1   (*(unsigned char* volatile) (MBAR + 0x0000014C))
#define UTB1   (*(unsigned char* volatile) (MBAR + 0x0000014C))
#define UIPCR1 (*(unsigned char* volatile) (MBAR + 0x00000150))
#define UACR1  (*(unsigned char* volatile) (MBAR + 0x00000150))
#define UISR1  (*(unsigned char* volatile) (MBAR + 0x00000154))
#define UIMR1  (*(unsigned char* volatile) (MBAR + 0x00000154))
#define UBG11  (*(unsigned char* volatile) (MBAR + 0x00000158))
#define UBG21  (*(unsigned char* volatile) (MBAR + 0x0000015C))
#define UIVR1  (*(unsigned char* volatile) (MBAR + 0x00000170))
#define UIP1   (*(unsigned char* volatile) (MBAR + 0x00000174))
#define UOP11  (*(unsigned char* volatile) (MBAR + 0x00000178))
#define UOP01  (*(unsigned char* volatile) (MBAR + 0x0000017C))

/* #define UMR11 (*(unsigned char* volatile) (MBAR + 0x00000140))

    is similar to an Assembly routine

    UMR11   EQU   $00000140

    Both definitions " equate"  or " define"  a register called UMR11
with a memory
    location at Hex address {MBAR + 00000140}.
*/

/**************************************************************
                                UART2
**************************************************************/

#define UMR12  (*(unsigned char* volatile) (MBAR + 0x00000180))
#define UMR22  (*(unsigned char* volatile) (MBAR + 0x00000180))
#define USR2   (*(unsigned char* volatile) (MBAR + 0x00000184))
#define UCSR2  (*(unsigned char* volatile) (MBAR + 0x00000184))
#define UCR2   (*(unsigned char* volatile) (MBAR + 0x00000188))
#define URB2   (*(unsigned char* volatile) (MBAR + 0x0000018C))
#define UTB2   (*(unsigned char* volatile) (MBAR + 0x0000018C))
#define UIPCR2 (*(unsigned char* volatile) (MBAR + 0x00000190))
#define UACR2  (*(unsigned char* volatile) (MBAR + 0x00000190))
#define UISR2  (*(unsigned char* volatile) (MBAR + 0x00000194))
#define UIMR2  (*(unsigned char* volatile) (MBAR + 0x00000194))
#define UBG12  (*(unsigned char* volatile) (MBAR + 0x00000198))
#define UBG22  (*(unsigned char* volatile) (MBAR + 0x0000019C))
#define UIVR2  (*(unsigned char* volatile) (MBAR + 0x000001B0))
#define UIP2   (*(unsigned char* volatile) (MBAR + 0x000001B4))
#define UOP12  (*(unsigned char* volatile) (MBAR + 0x000001B8))
#define UOP02  (*(unsigned char* volatile) (MBAR + 0x000001BC))
```

```
/**************************************************************
                        UART Register Settings
**************************************************************/

#define RESET_TX         (0x30)       /* Reset Transmitter*/
#define RESET_RX         (0x20)       /* Reset Receiver*/
#define RESET_MR         (0x10)       /* Reset Mode Register Pointer*/
#define PARITY_NONE      (0x10)       /* Parity: None*/
#define BPC_8            (0x03)       /* 8 Bits Per Character*/
#define NORMAL_CM        (0x00)       /* Normal Channel Mode*/
#define STOP_BITS_2      (0x0F)       /* 2 Stop Bits*/
#define TIMER_MODE       (0xDD)       /* Timer mode */
#define TX_ENABLED       (0x04)       /* Transmitter Enabled*/
#define RX_ENABLED       (0x01)       /* Receiver Enabled*/
#define TX_READY         (0x04)       /* Transmitter Ready*/
#define RX_READY         (0x01)       /* Receiver Ready*/
```

Step 1:
Using this header, we begin our C code for the serial communications transfer by initializing the transmitter, receiver and mode register pointers in the UCR. This initialization can be accomplished by:

```
UCR1        = RESET_TX;
UCR1        = RESET_RX;
UCR1        = RESET_MR;
```

Step 2:
The next step in the serial communications program would be to initialize the rest of the registers needed. Here is an example of how and what to do.

```
UISR1    = 0;    /*This disables all interrupts */
UMR11 = PARITY_NONE | BPC_8;  /* NO Parity, 8 Bits per Character*/
UMR21 = NORMAL_CM | STOP_BITS_2;  /* Normal Channel Mode, 2 Stop its
UCSR1 = TIMER_MODE;
UBG11 = 0x00;
UBG21 = (SYSCLK/(32))/BaudRate;
```

**** Note:** UMR11 and UMR21 both point to the same address. Resetting the mode register pointer (UCR1 register) sets the pointer to UMR1. After writing to UMR1 the pointer points to UMR2. You should initialize UMR2 immediately after initializing UMR1. *See Appendix A for algorithms used to initialize the UARTs*.

Step 3:
Finally it is time to go ahead and enable the transmitter and the receiver which is done as follows.

```
UCR1    = TX_ENABLED;          /* Enable Transmitter */
UCR1    = RX_ENABLED;          /* Enable Receiver */
```

Step 4:
Now that you have all of the registers initialized, you can actually transmit information from the MCF5206eLite board to the PC and back. You, of course, you have to write these routines first. Here is the code to transmit a character from the MCF5206eLite to the PC.

First the character must be put into the transmit buffer.

```
void UartPutc(int c)
{

        while(!(USR1 & TX_READY));
        URB1 = (byte)c;
}
```

Where ‘c’ is the character that is to be sent.

8.3.1 Lab Assignment

Write a C program that will transmit a string of characters from the MCF5206eLite board to the PC using UART1. Print this message on the screen through the HyperTerminal.

8.3.2 Write up

1. Turn in a printout of the code used to transmit the string from the MCF5206eLite to the PC's HyperTerminal.

2. Make a list of the registers used in the *transfer* of the string and what values that they need to be set to. (i.e. what registers need to be initialized and what initialization values are used. This list of registers should include what registers are used to initialize the UARTs, what registers are used to transfer the string, and what registers are used in the control of the data flow.)

3. What additional registers are needed to *receive* information into the UARTs and what would they be initialized to?

4. What size buffers are included in the UARTs. What are these buffers used for.

BONUS

For the bonus points in this lab, connect two MFC5206eLite boards together to function as a dumb terminal. Whatever is typed into the first PC and MFC5206eLite should appear on the screen of the second PC and vice versa.

References

1. Clements, Alan. “Microprocessor Systems Design,” Third Edition, PWS Publishing Company, 1997.
2. Coldfire Microprocessor Family Programmer’s Reference Manual. Motorola Revision 1.
3. MCF5206 Coldfire Integrated Microprocessor User’s Manual. Motorola 1997.
4. MCF5206eLITE Evaluation Board User’s Manual. Motorola Revision 2.
5. Motorola’s Coldfire HomePage **-** http://www.mot.com/coldfire
6. PDACS II Homepage –
7. http://www.cs.tamu.edu/course-info/cpsc483/common/99c/g4/g4.html