SBC5307 Manual & Reference (Spring 2006)

Sanjay Singh, ssingh@ecemail (Document Maintainer)
Irene Huang, yqhuang@ecemail (Automated Testing)

Original Authors:
Mattias Hembruch, mghembru@ece.uwaterloo.ca
Chris McKillop, cdmckill@ece.uwaterloo.ca

Teaching Assistant and Proofreader:
Dave Grant, d2grant@engmail.uwaterloo.ca

This document is the tutorial / reference guide for the ColdFire boards used in E&CE 354. It discusses hardware, GCC, compiler utilities, assemblers, and debugging.

Send changes/corrections and suggestions to either Sanjay or Irene. Document maintainer is currently Sanjay Singh.

Table of Contents

List of Tables

Acknowledgments

This document has borrowed content from several resources, including Motorola's documentation for the ColdFire and lab manuals prepared by lab instructors for this and other courses.

Thanks to Roger Sanderson, Irene Huang, Eric Praetzel and Professors Paul Dasiewicz and Douglas Harder for their help. And a note of acknowledgement to Dave Grant for pointing out some errors and modifications to really improve this manual.

Introduction and Scope of this Document

This document is primarily intended to be a student guide and primer for programming the ColdFire boards used in the ECE 354 course project, as part of the Computer Engineering undergraduate program.

Students should refer to the earlier manual to familiarize themselves with the Arnewsh SBC5206 ColdFire boards if they wish to use them in a fourth year design project. There is a separate manual for this board called 5206_ColdFire_Manual.pdf which can be downloaded from the course website, or requested from a Lab Instructor.

This document assumes the use of the CJ Design MCF 5307 ColdFire board.

UW COLDFIRE WEB PAGE

There is a ColdFire resource page on campus, available at:

`http://sca.uwaterloo.ca/ColdFire/`

Use of the Lab After Hours

Use of the lab after hours is a privilege, not a right. Loss, damage, improper use of equipment or indication of food or beverage consumption in the lab will lead to cancellation of after hour access.

*Please be responsible in the lab.

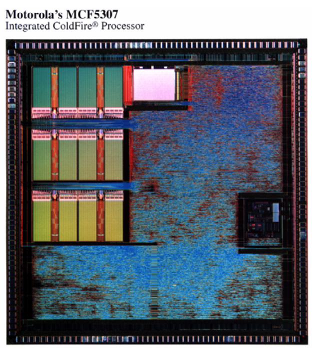
Overview

The ColdFire 5307 board can be used for running your ECE 354 software project. On it is the ColdFire processor designed and built by Freescale Semiconductor, a spinoff from Motorola Corporation. It also provides RAM, serial communication channels (UARTs), parallel I/O ports, timers/counters, and RAM. These boards have at least 64 MB of DRAM installed.

General Information about the ColdFire Processor

ABOUT THE COLDFIRE

The processor, an MCF5307, is a member of the ColdFire family of processors made by Motorola. The MCF5307 is a 32-bit CPU with the ability to handle 32-bit addresses and 32-bit data, and can be thought of as a *RISC*ified Motorola 68000.



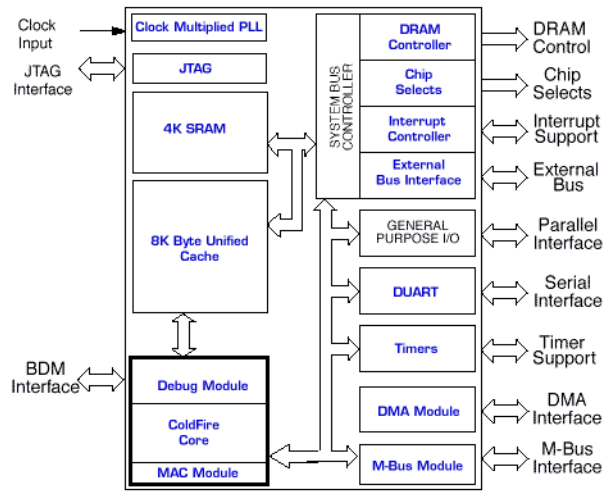


FIGURE 1. **Block diagram of MCF5307.**

It contains eight 32-bit data registers, eight 32-bit address registers, a 32-bit program counter and 16-bit status register. The MCF5307 has a System Integration Module (SIM).

COLDFIRE MANUALS AT UW ON THE WEB

ColdFire Programmers Reference Manual

http://www.ece.uwaterloo.ca/~ece222/ColdFire/CFPRM.pdf

ColdFire User Manual:

http://www.ece.uwaterloo.ca/~ece222/MCF5307BUM.pdf

INTERRUPT INFORMATION

The ColdFire has various types of interrupts that should be understood in order to write robust low-level software. Please read the following sections to have a complete knowledge of these facilities.

TIMER MODULE

*Read chapter 13 of the ColdFire User Manual, just discussed.

The ColdFire processor has two internal timers. You can configure a timer to count until it reaches a reference value, at which time it either starts a new time count immediately or continues to run. The free run/restart (FRR) bit of the Timer Mode Register (TMR) selects the mode. When the timer reaches the reference value, the Timer Event Register (TER) bit is set and issues a maskable interrupt if the output reference interrupt (ORI) enable bit in the TMR is set.

See “The Timer” on page 37. You will find code samples to assist.

SYSTEM INTEGRATION MODULE

The SIM, provides overall control of the bus and serves as the interface between the ColdFire processor core and the internal peripheral devices. The SIM incorporates many of the functions needed for system design. This includes programmable chip-select logic, system protection logic, general-purpose I/O and interrupt controller logic. A hardware watchdog timer (DOS monitor) circuit is included in the SIM which monitors bus activities. If a bus cycle is not terminated within a programmable time, the watchdog timer will assert an internal transfer error signal to terminate the bus cycle.

ENABLING TIMER INTERRUPTS IN THE SIM

To enable the timer interrupt so that can occur, you must set the processors interrupt mask register (IMR) accordingly.

See the Interrupt Pending and Mask Registers (IPR and IMR) section, page 9-6 of the “MCF5307 ColdFire Integrated Microprocessor User’s Manual” for details.

TIMING

The system clock for the MCF5307 is 45 MHz, with the CPU core running at 90 MHz. There are two steps to reducing the clock speed for the timer. First, setting the Input Clock Source for the Timer of the TMR to divide the master clock by 16, and secondly specifying the prescalar value of the TMR to further reduce the number of timer cycles per second.

The actual value the time accounts to a set in the Timer Reference Register (TRR). For more information, please see Table 2, “MCF5307 Timer Addresses and Values,” on page 7.

SUPERVISOR / USER MODE

Unlike the older 68K families, the ColdFire *does not have a separate stack* for User mode and Supervisor mode. Due to this single stack, when an interrupt occurs the stack *must* be aligned on a 4 byte boundary in order to ensure that the handler doesn't attempt to access an unaligned stack. The processor will take care of this alignment for you by padding the stack and encoding the number of padded bytes in the exception frame.

However, to simplify things in both the TRAP handler and the rest of your code, just ensure that all parameters passed on the stack when using a TRAP are 32-bits in length. This assures that the stack will remain aligned on a 4-byte boundary. This is not an issue with interrupts from external sources such as the timer and serial port as the stack is not used to communicate information directly to these routines.

THE VECTOR BASE REGISTER

There is an internal ColdFire register named VBR. This is the *vector base register*. This register points to the place in memory were the read/write vector tables live.

*This must first be moved to point to DRAM before interrupts can be used.

Please see the section “Vector Base Register Initialization Code” on page 37, for code that prepares this register for use.

*Note in particular, the second and third instructions in that section.

MCF5307 Janus Board Hardware Information

MCF5307 MEMORY MAP

The following table describes the arrangement of devices on the processor databus when using the CJ Design board.

TABLE 1. MCF5307 Memory Map

Address Start	Address End	Number of Bits	Description
0x00000000	0x0FFFFFFF	8	Flash
0x10000000	0x1FFFFFFF	32	SDRAM
0x20000000	0x3FFFFFFF	32	Peripheral Space
0x40000000	0xFFFF7FFF	-	Unused
0xFFFF8000	0x0FFFFFFF	32	Internal SRAM

Be aware that the janusROM changes the location of the peripheral space to 0xF0000000.

MCF5307 IMPORTANT ADDRESSES AND VALUES

The following table lists addresses and values that are important when programming the ColdFire’s timers.

TABLE 2. MCF5307 Timer Addresses and Values

Name	Address (Hex)	Notes and Extra Information
MBASE	0xF0000000	MCF5307 module base address
TMR1	MBASE+$100	Timer Mode Register 1 • pg. 13-3, MCF5307 Users Manual • set system clock divisor to 16 • set Free / Run Restart to reset immediately after reaching the reference value • enable the Output Refrence interrupt • set the output mode to toggle output • disable capture events • set the prescalar value
TRR1	MBASE+$104	Timer Reference Register 1 • pg. 13-4, MCF5307 Users Manual • set counter compare value
TER1	MBASE+$111	Timer Event Register 1 • pg. 13-5, MCF5307 Users Manual • write a 1 to the ORI bit to clear the interrupt event

EXCEPTION VECTOR TABLE LOCATIONS

Note that all autovector values are 32-bits.

TABLE 3. MCF5307 Autovector Addresses and Values

Name	Address (Hex)
autovector 5	$74
autovector 6	$78
first user interrupt	$100

KEYBOARD INTERRUPTS

The sample code provided in the **manual_5307.zip** file uses the UART directly. Study this code to see how it works, and then adapt it to your design.

The address of the ISR you write for your interrupt should be stored in the corresponding location in the processor exception table.

Please note: to allow the processor to see interrupts, you must also set the ColdFire's Interrupt Mask Register (IMR) accordingly.

IMPORTANT OFFSETS FOR INTERRUPT MANAGEMENT

The following table shows important offsets that need to be understood for interrupt management:

TABLE 4. Offsets and Values for Interrupt Management

Name	Address (Hex)	Notes
MBASE	0xF0000000	MCF5307 module base address
MFP	N/A	Multifunction Peripheral no longer exists!
IMR	MBASE+0x044	Interrupt Mask Register • pg. 9-6 to 9-7, MCF5307 User's Manual • enable the Level 4 interrupt
IPR	MBASE+0x040	Interrupt Pending Register • pg. 9-6 to 9-7, MCF5307 User's Manual • makes pending interrupt sources visible
ICR (timer0)	VBR+0x04C	bits 16-23, within 0-31 bits
ICR (timer1)	VBR+0x04C	bits 8-15, not to be used by ECE 354 students
ICR (UART0)	VBR+0x050	bits 24-31, not to be used by ECE 354 students
ICR (UART1)	VBR+0x050	bits 16-23, within 0-31 bits

IMPORTANT: JANUS BOARD ANOMALIES

Please note:

• IRQ1 is in an unknown state and *must* be masked out in the IMASK register.

• *Under no circumstances* should you use the MOVEM instruction in your code. It will not work properly due to the way the board is designed. Write register contents to memory one at a time, rather than with this multiple move instruction.

Janus Board Hardware Expansion Port Specifications

MCF5307 PARALLEL PORT

The parallel port (J1 on the board) provides a flexible 8-bit bi-directional interface to external devices. These signals are also connected to LED's to aid in system debugging, should your project make use of the parallel port. The parallel port signals are contained in the Bus Expansion Control Port Connector J1. The signal description is provided in the following table.

TABLE 5. **MCF5307 Parallel Port Map**

Parallel Port Bit	J1 Pin	LED
7	2	L8
6	4	L7
5	6	L6
4	8	L5
3	10	L4
2	12	L3
1	14	L2
0	16	L1

MCF5307 BUS EXPANSION PORT (4TH YEAR PROJECTS)

For 4th year teams wishing to use it, this port extends the processor bus to external devices. Viewed from the front of the board* the bus expansion port is found on the left side of the board. Use this port to connect external devices to the processor data bus. The specific signals are listed in the following table:

TABLE 6. **MCF5307 Bus Expansion Port Pin Assignments**

Pin Name	J2 Pin Number	Pin Name	J2 Pin Number
GND	1	~AS	2
R/~W	3	~TA	4
A1	5	A0	6
A3	7	A2	8
A5	9	A4	10
A7	11	A6	12
A9	13	A8	14
A11	15	A10	16
A13	17	A12	18
A15	19	A14	20
A17	21	A16	22
A19	23	A18	24

*. For lack of any other reference serial ports are assumed to be at the rear of the board.

TABLE 6. MCF5307 Bus Expansion Port Pin Assignments

Pin Name	J2 Pin Number	Pin Name	J2 Pin Number
A21	25	A20	26
A23	27	A22	28
D30	29	D31	30
D28	31	D29	32
D26	33	D27	34
D24	35	D25	36
D22	37	D23	38
D20	39	D21	40
D18	41	D19	42
D16	43	D17	44
D14	45	D15	46
D12	47	D13	48
D10	49	D11	50
D8	51	D9	52
D6	53	D7	54
D4	55	D5	56
D2	57	D3	58
D0	59	D1	60

MCF5307 BUS EXPANSION CONTROL PORT

As with the previous port, this also extends the processor bus to external devices. It is located beside the bus expansion port, clearly labelled on the left corner of the board. The specific signals are listed in the following table:

TABLE 7. MCF5307 Bus Expansion Control Port Pin Assignments

Pin Name	J2 Pin Number	Pin Name	J2 Pin Number
TIN1	1		2
TOUT1	3		4
TIN0	5		6
TOUT0	7		8
	9		10
	11		12
	13		14
3.3 V	15		16
~CS0_EXT	17	~BG	18
~BWE0	19	~BR	20
~BWE1	21	~BD	22
~BWE2	23		24
~BWE3	25	~TS	26

TABLE 7. MCF5307 Bus Expansion Control Port Pin Assignments

Pin Name	J2 Pin Number	Pin Name	J2 Pin Number
~OE	27	SIZ0	28
~CS6	29	SIZ1	30
~CS7	31	CK-PORT	32
~RESET	33	GND	34

Debugging Commands for CJ Design janusROM

JanusROM is the ROM monitor for the new ColdFire boards that are now in use for ECE 354.

```
*** Janus ->Boot<- ROM 1.0.1 ***

Configuring Chip Selects...
Configuring SDRAM...
Testing SDRAM...
    Data Test:      PASSED
    Address Test:   PASSED
Copying janusrom to DRAM...
Transfering Control To Next Program @ 0x10000000

-------------------------------------------------
  janusROM 1.0.2
-------------------------------------------------
MCF5xxx Development System
(c)2001 CJDesign
Desuckification by David Grant, 2003.

Installed DRAM: 64MB bytes.

Enter help or ? for help system...

janusROM>
```

FIGURE 2. The janusROM initialization sequence.

Once the initialization sequence has completed, you can refer to the table below for working with janusROM

TABLE 8. janusROM Commands and Explanations

Command / Argument	Short Form / Synopsis	Explanation
blockcmp	bc [-h] <first> <second> <lengh> <first> - First address. <second> - Second address. <length> - Number of bytes to compare.	Block compare: Perform a block compare of length bytes between the first and second addresses.
blockfill	bf [-b|w|l] [-h] <begin> <end> <data> <begin> - Starting address. <end> - Ending address. <data> - Data to fill with. -h - Show this message. -b|w|l - Pick fill size (bytes, words, longs)	Block fill: Fill the given range of memory with the given value.
blockmove	bm [-b|w|l] [-h] <start> <end> <dest> <start> - Start address of move. <end> - End address of move. <dest> - Destination address of move. -h - Show this message. -b|w|l - Pick move size (bytes, words, longs).	Block move: Perform a block move of data from start->end to dest.
break	break [-h] [-d] [-l] <address> <address> Address to add/remove breakpoint from. -h - Show this message. -d - Remove breakpoint from the given address instead of adding. -l - List all installed breakpoints.	Manage the breakpoints in the system.
cont	cont [-h] [-t] -h - Show this message. -t - Toggle trace mode.	Continue program execution after a breakpoint is hit or during program tracing.
disasm	di disasm [-h] [-l#] <address> <address> Starting address. -h Show this message. -l # Set the length, in bytes, to dissassemble.	Disassemble machine instructions from memory into human-readable format.

TABLE 8. janusROM Commands and Explanations

Command / Argument	Short Form / Synopsis	Explanation
go	go [-h] [-t] <address> [reg=val,reg=val,...] <address> - Program starting address. reg=val,... - Comma separated pairs of starting register values. -h - Show this message. -t - Start with tracing enabled.	Start program Begin execution of a program in memory at a specific location.
help	?	Show This Information
info		Show Board Information
md	md	Memory Dump Example dumps memory from hex address 4000 to 40FF: janusROM> md 4000
memmodify	mm [-b|w|l] [-h] [-d] <addr> [<data>] <addr> - Address (start) to modify. <data> - Data to put at the given address. -h - Show this message. -d - Disassemble data as entered. -b|w|l - Pick move size (bytes, words, longs).	Memory Modify Modify the values in memory. If no data is given this will be done interactively with the user.
peek	peek [-h] [-s #] [-c #] <address> <address> Address to access. -h - Show this message. -s # - Set the size to access in bits (8,16 or 32). Default: 32 -c # - Number of accesses (address auto-increments). Default: 1	Display memory locations
poke	poke [-h] [-s #] [-c #] <address> <value> <address> - Address to access. <value> - Value to write at address (dec, hex or oct format). -h - Show this message. -s # - Set the size to access in bits (8,16 or 32). Default: 32 -c # - Number of writes to perform (address auto-increments). Default: 1	Write to memory locations

TABLE 8. janusROM Commands and Explanations

Command / Argument	Short Form / Synopsis	Explanation
params	params [-h] [-l] [-i] [-s] [-d] [-w]	Manage parameters
	-h - Show this message. -l - List all parameters. -i name - List the specific parameter. -s name=value - Set the parameter name to the given value. -d name Remove the parameter name from the system. -w - Write paramteres into flash.	
passwd	passwd [-h]	Change the system password
	-h - Show this message.	
ramtest	ramtest [-h] -a addr -s size [-c count]	Test installed SDRAM
	-h - Show this message. -a address - Set address to start tests. -s size - Set size (in bytes) of ram. -c count - Number of test cycles. Default: 1	
regclear	rc	Clear registers by setting PC SR An and Dn to 0
reddisp	rd	Display current values of registers PC SR An and Dn
regset	rs,rm	Set registers
	regset [-h] reg=val[,reg=val,reg=val,...] reg=val[,reg=val] Register name=value pairs. ie: d0=1,d1=2,a0=44 -h - Show this message.	Set registers to given values before and during program execution.
restart		Restart the ROM monitor
tftp	tftp [-h] [-f filename] [-a addr] [-s ip] [-t type]	Download program over network (TFTP)
	-a <addr> - Download address. -f <filename> - File to download. -h - Show this message. -s <x.x.x.x> - Use this IPv4 server. -t binary|srec - File type.	Download files over the ethernet interface using the tftp protocol.
transfer	transfer [-h] [-s baud] [-r] [-p port]	Download program over serial port.
	-b baud - Set transfer baud rate to baud. Default 19200. -h - Show this message. -p port - Select transfer port (terminal|aux). Default "aux"	

Setup Cabling, Serial Ports, Compiler, & Source Files

Before going any further into this document you will need to have the following tools at your disposal.

- ColdFire board (encased in a light blue box)
- Two serial cables
- A PC or a Sun workstation with two serial ports
- The ColdFire gcc compiler
- The source.zip file for this manual: **http://ece/~ece354/source.zip**

All of these are available on campus and any hardware should be located in the ColdFire lab on the second floor of E2, in rooms E2-2363 and E2-2364. All of the machines in the main lab are setup with two serial ports so you will not need to use two machines to continue.

Make sure that you have the serial cables plugged into both the ColdFire board and the PC. In Nexus, open HyperTerminal (Programs >Accessories > HyperTerminal) or any other communications program. You will need to setup direct access the two (COM1 and COM2) serial ports on the PC and open two copies of the terminal program.

SERIAL PORT SETTINGS

The proper serial port (or COM port)settings for communicating with the ColdFire board are:

- 19200 baud
- No start bits, 8 data bits, 1 stop bit (N,8,1)
- No Hardware Flow Control (cts/rts)
- No Software Flow Control (xon/off)

On Nexus, TeraTerm observing COM1 is setup as follows:

```
Start => Run... => tterm
```

VERIFYING PORT CONNECTIVITY

Once the communication programs are opened and setup for the two serial ports trying hitting enter in both of them a few times. In one, there should be nothing (this is the one that your RTX will interact on) and in the other there should be a prompt that looks like this:

```
janusROM>
```

This is the internal monitor of the SBC. This is were you will be able to setup through memory, download your compiled code, etc. This should look familiar. It is what you used in ECE 222.

PREPARING SOURCE FILES FOR USE

Now, make a directory on your N: drive, or your Unix home directory and unzip **manual_5307.zip**. This has the various source files used in the rest of this manual so you don't have to type them back in again. Everything is ready to do some work now. If you are already familiar with the ROM monitor commands, proceed to “Software Tools” on page 17.

Using the ColdFire Server

This section describes in detail how to use the ColdFire server from Unix, which is the preferred place for groups to work together.

WORKING ON UNIX WITH THE COLDFIRE SERVER

We strongly suggest that you do your work on Unix. File sharing can be setup by using groups and directory permissions. Also the gcc compiler is known to work. The cfcp command can quickly copy your RTX to the coldfire server. The downloads to the Coldfire server are easily 10 times faster than serial downloads. Everyone has a Unix account and you can easily access your account via telnet or ssh/putty from Windows.

CONNECTING TO THE COLDFIRE COMPUTER

To use the server just use telnet. On Windows or Unix this is just:

telnet cf-server 8000

The second serial port of the Coldfire computer that you're assigned to will be printed on the screen. If no Coldfire computers are available then try again in 10 seconds or longer. Each user has a computer for a certain amount of time (typically 5 or 10 min.) and then they are kicked off and the first person to connect gets that computer.

DOWNLOADING TO A COLDFIRE COMPUTER

The "cfcp" command is with the compiler tools on our Unix machines. The cfcp utility should already be in your path. See "Preparing to use the software tools" on page 17.

It's also on Nexus in Q:\eng\ece\util ... Note that the Nexus tool will download your file preserving it's name - while the unix *cfcp* command lets you rename your file. I'd suggest using your group number or a userID with a version number appended.

Files are left on the Coldfire server and typically deleted when they are unused for a week. Uploading non-course related files is a violation of campus policy and is not welcome, so please don't take up disk space unnecessarily.

Once your compiled OS is on the server you may download it to the particular board you are connected to. Use the *tftp* command from the ROM prompt of the board. Do not use the tftp command on harshrealm. Typically you would use it like this: tftp -f <your_file_name>

COLDFIRE SERVER DETAILS

The Coldfire Server is a Redhat Linux computer which has 64 serial ports (Cyclades) and two network cards. This server is connected to several Motorola 5307 Coldfire computers for use in ECE 222 and ECE 354. ECE 354 is the primary user of the these boards. One network card feeds the private subnet which contains the cluster of Coldfire boards. The other is connected to the UW network. The tftp (trivial file transfer protocol) utility is used to download software to the Coldire computers.

A C program similar to a terminal server, listens, and establishes port bindings and shuttles data between a TCP/IP network socket and the serial port of a given ColdFire board. When a user connects to port 8000 it checks to see if there is a Coldfire computer available. If there is, then that Coldfire computer is reset and the user is given a connection to it's primary serial port. A message is printed given the port at which you can find the second serial port. This number is randomly assigned, so take note of it when its printed.

Software Tools

There are two types of tools available to develop your code with: A C compiler, or assemblers. We expect most people to work with the C compiler, but the assembler is always available.

OBJECT-ORIENTED DESIGN

In the future, we would like to have tools available for object oriented design, so if you plan to do further work with the ColdFire after you are done with this course, check the 354 course page for any new tools we might have added. Also, if you find tools to assist with OO-design, perhaps you could let us know about it. We'd appreciate that.

PREPARING TO USE THE SOFTWARE TOOLS

Put the following text in your .cshrc file to tell your shell (csh or tcsh) where to find the gcc cross compiler, and the ColdFire simulator:

```
##################################
# ECE 354 Stuff
##################################

setenv PATH "${PATH}:/opt/gcc-mcf5307/bin"
setenv PATH "${PATH}:/opt/ColdFire-0.3.2/bin"
```

If you are using *bash*, you're on your own, but you should be able to put in the corresponding changes to your PATH environment variable, if you're astute enough to be using bash in the first place, right?

GCC CROSS COMPILER

The C compiler available on the Solaris server named *harshrealm* is a variant of the popular gcc C/C++ compiler. This compiler is a special version of gcc specifically configured to produce ColdFire opcodes. In particular you will be using a *cross-compiler.* A cross-compiler runs on one platform (such as Unix on UltraSPARC or Nexus on Intel) but produces code for a different platform, in this case the ColdFire processor.

Obviously, you can't run the compiled code on the host platform, as discussed in "Downloading from the Computer to ColdFire Board" on page 29. You must move this compiled binary code to the ColdFire board, or load it into the emulator in order to run it.

INVOKING AND USING GCC

Depending on which platform you want to use, the method of invoking the compiler will vary. Check with the Lab Instructors if you wish to use Nexus. A Windows version of the compiler is available, and can be installed if needed. On Solaris, you have to ensure that the directory:

```
/opt/gcc-mcf5307/bin
```

is in your PATH environment variable, since this tells the Unix shell where the compiler utilities may be found. Type echo $PATH to make sure, if you wish. Look for the above string.

Or type:

```
which m68k-elf-gcc
```

Some things to keep in mind are that C is used as a top-level organizational tool only. You *cannot call* any normal C-library functions (such as printf, fopen, etc). Your project is to *implement* much of the low-level functionality that these familiar C routines would use. The point is to use the control-flow elements of C (if, while, for, functions, assignment, operators) to organize *your* code. With a little work, you can code your own puts() equivalent, which you can then call from within your code.

In order to do this, you will also be using gcc in a way that few people do. You will be combining C and ColdFire assembly to gain a very fine level of control over the ColdFire board. Combining C and assembly is easy, but you do have to pay attention. For more information about this, see "Combining C and Assembly Language" on page 32.

TRAP #15 HANDLER INFORMATION

The janusROM monitor includes a "TRAP #15" handler to simplify input and output. This function can be called by the user program to utilize various routines within janusROM. There are four TRAP #15 functions available: OUT_CHAR, IN_CHAR, CHAR_PRESENT and EXIT_TO_DEBUG. The codes for these functions are as follows:

TABLE 9. TRAP #15 Functions

Name	Value	Parameters
OUT_CHAR	0x0013	Char to output in lower 8 bits of d1
IN_CHAR	0x0010	Char returned in d1
CHAR_PRESENT	0x0014	non-zero if a character is available
EXIT_TO_JANUSROM	0x0000	<none>

For examples of how to use these within your code, please refer to the SBC5307 User's Guide.

ADVANCED UNIX USERS: ROLL YOUR OWN GCC

*Skip to the next section if you plan on using on-campus computers for your project work.

This is a brief and concise set of instructions for those advanced students who want to develop on Linux or Solaris at home, and would also like to gain experience compiling their own software. E-mail the Lab Instructors if you have any questions about this section.

- Get binutils-2.10. Any versions higher than this have a bug in the 5XXX assembler.
- Get gcc-3.3-core from any GNU-friendly website.

For binutils unpack with something similar to the following set of commands:

```
tar xvzf binutils-2.10.tar.gz
cd binutils-2.10
./configure
make
make install
```

*Please note that if you plan to install in any of the default directories like /usr/local you will need to have root privileges on your computer. Or you will have to add your installation directory as an argument to the "configure" command, as shown in the example below. Again, it is assumed you have experience with Unix-based software development before attempting something like this. Either you have the experience, or you should know someone that does.

For gcc, unpack and:

```
cd gcc-3.3
./configure --target=m68k-elf --prefix=/opt/ColdFire
make
make install
```

These are fairly straightforward for those who have used Linux and/or the GNU tools. When you have done this, add:

```
/opt/ColdFire/bin
```

to your PATH, and reinitialize your shell environment variables, and internal hash tables with (assuming you are using csh as your shell):

```
source ~/.cshrc
```

MAKE AND MAKEFILES

The *make* utility is designed to help keep object files up to date from source files while minimizing recompilation. The input to make is a file called "Makefile" or "makefile" (note: Unix files are case sensitive). A Makefile will contain a series of targets, listing which source files a given target depends on, and how to rebuild the target from its source files. The default target is the first one listed in the Makefile, called "all" by convention.

A makefile usually contains two things: Macro definitions and targets. A macro definition would look like this:

```
make CC=cc
```

The above statement will define CC to be cc, overriding any such definitions in the makefile itself, such as:

```
CC=gcc
```

A target has the following form:

```
target: dependency.1 dependency.2 etc
    command 1
    command 2
    etc
```

Note that the first character of each command line *must* be a "tab" character, *not* a series of spaces. This is important, so it bears repeating: the make utility will *only* work with a tab. The way this target line is interpreted is as follows: the file *target* depends on the files dependency.1, dependency.2, etc. If the time/date of last modification/creation of any of these files is LATER than the time/date of last modification/creation of *target*, this means target is out of date. As a result, to bring target up to date, *command 1* is issued, then *command 2*, etc. For example:

```
rtx: rtx.c rtx.h process.o process.h
     $(CC) $(CFLAGS) -c rtx.c
     $(CC) $(LDFLAGS) rtx.o process.o -o rtx $(LIBS)
```

This says that the file rtx depends on rtx.c, rtx.h, process.o and process.h. Another rule would specify how to create target process.o (and it would be checked to ensure process.o is up to date as well), and so on.

Note the use macros (CC, CFLAGS, LDFLAGS and LIBS are the names that make uses to specify the C compiler to use, the flags to pass to the compiler, the linker flags to use and libraries to link in). This makes it easier to change one and have it apply globally. For example, if all files are to be compiled with -g (debugging), it is easier to change from CFLAGS= to CFLAGS=-g ... This is certainly preferable to finding and changing every compile command, wouldn't you agree?

Certain rules are built into make, such as how to make a .o file from a .c file. Thus, the default rule can be used to create process.o in the above table, assuming it could be compiled with a simple:

```
$(CC) $(CFLAGS) -c process.c
```

make has more built-in macros and options to specify additional generic rules. If you wish to learn more, consult the man page on Unix, or a Unix reference such as the O'Reilly book "Unix in a Nutshell." Two common options:

```
make -f <filename>
```

and

```
make -n
```

shows the commands that would be executed but does not execute them. This is useful for debugging and/or learning to use make.

ASSEMBLER ON NEXUS

The Austex Assembler is the preferred assembler available on Nexus. See http://www.ece.uwaterloo.ca/~ece222/software.html#assemblers for more information.

The Austex Assembler (cfasm), Linker (cflink) and Make (cfmake) utilities have been installed on Nexus in: Q:\Eng\ECE\ColdFire.

Higher Level Software Design and Debugging

There are various software development, design and debugging strategies, and it is impossible to define one that suits every person or team. But there are some guidelines that can help you as you develop your code. The development of reliable, readable, testable and maintainable code requires several steps, detailed in the following sections.

PROBLEM DEFINITION

You must present a clear, rigorous and concise definition of the program requirements. This includes any assumptions or limits (constraints) that may be part of the program. As you are developing your software, you should be able to go back to this definition and ensure that your program meets these program requirements.

PROGRAM DESIGN

The specification (overview) of how the program will be implemented to meet the requirements, from the previous step. This is generally done with a combination of written descriptions, pseudocode, or flowcharts. Its not a bad idea to consider UML as well.

CODING AND REFACTORING

For most engineering students, this is where the action begins. You convert the design into program code. In general, coding should be done in a modular form.That is, the overall program is broken down into a number of subprograms (or procedures) that can be individually tested and debugged, then linked together.

The implementation, though usually larger than straightline code, is easier to read and maintain, because the effect of separating the algorithm into smaller blocks increases the intelligibility of each individual section. Also, the benefit of limited scoping within functions prevents unexpected interactions between variables.

Refactoring is the process of rearranging long complex blocks of code into simpler, more intelligible sections while maintaining program functionality. Good software design helps minimize the need for refactoring, which is usually done with legacy software systems.

DEBUGGING

Alas for the necessity of this step! Rare indeed is the program that *seems to* works right the first time. And even if it does, you still have to subject it to stress testing to ensure it works robustly. In general, the objective of debugging is to create code that will execute on the target machine without errors. There are again, several stages to this process.

The first stage is to obtain code that will compile without syntactical or assembly errors (i.e., non existent opcodes, address faults, etc.).

The second step is to get the code to execute without error on the target machine. This is usually done by setting breakpoints and testing the code in segments. See the handout on debugging techniques and strategies, and “Debugging the OS” on page 45.

TESTING

Just because the code will run without errors does not mean it is correct. A strategy has to be developed to test the code to see if it meets the requirements developed in steps 1 and 2. This usually involves establishing a set of test vectors or sets of known inputs for which the results are known and checking all aspects and functions of the program (or as many as can be practically tested). This includes testing the limits of the program.

Identify *corner cases* and provide atypical or unusual inputs or argumentation to see if you can get the software to fail. You must set your pride aside and force the issue. If you don't others (ie. the TA's and/or the professors, and in the future your customers) will, and you will have to take responsibility for why you didn't listen to your friends in ECE 354 who tried to warn you about this when you were in school.

You will either be embarassed and/or get an average mark, in the case of a course project, or it could cost your company a great deal of money to issue patches to unhappy customers. Software development is, for better or worse, one of the most complex areas of engineering, primarily because of software's extreme sensitivity to change, or an unanticipated input. One small oversight can turn a perfectly running program into something worthless to a customer. So while you are developing your code, try to identify those areas where strange things might happen, and ensure your code can handle it.

DOCUMENTATION

This is one of the *most important* aspects of program design and sadly, one of the most neglected. Uncommented (undocumented) code or poorly documented code is practically useless as anyone who has tried to decipher someone else's or their own undocumented code can attest. To prevent this, the beginning of each program or procedure block must have a clear statement of what the section of code does, what input variables it expects (and how they are passed), what output the code performs (and how) and any side effects of its execution (such as changed registers, etc.).

After you have prepared the *systems documentation* for those who will be maintaining or extending your program, you should also consider the preparation of *user documentation*, which tells people who will use your program how to set it up and get it running, etc.

MAINTENANCE

Software has a *life cycle*. After the software has been distributed to customers, you are now in *maintenance mode*. This is where problems found during use of the program or upgrades to the code are performed. While this is not normally a part of the course, it is a vital step in real life programming. A record of any changes and their motivation must be kept as well as any side effects these changes might have. Changes to the code are generally indicated by version number. Also bear in mind that the source code base of the current product is often the base from which the next version of the product will be created, so fixing bugs now, means preventing future bugs from infesting the next version of your program.

Debugging Your Code: Tips and Strategies

In general, the goal of debugging is to verify the determinism in your individual blocks of program code. Put in simple terms, you must verify that your software does not do anything unpredictable that may affect how it interacts with other blocks of code, or the quality of its computations.

The most powerful tools available for debugging are BReakpoints, TRACE, STEP and the ability to examine registers. If you have used a modular approach to writing the program, then it becomes a relatively simple matter to check the operation of individual modules. In modular programming, you sometimes end up with modules that use other submodules. In this case, if we verify the correct operation of the submodules (and have confidence in their operation) we can then test the modules that use them. Thus, we usually start with the innermost modules and work our way outward until we have tested the complete system.

This can be done by setting up the system (i.e., loading memory locations or registers) with values that produce *known results* at the entry point to the module, setting a breakpoint at the end of the code segment, then checking the results. If the results are as expected (this may require several tests to verify), then we can eliminate the breakpoint and test other modules.

If the results are other than expected, we have several alternatives: we could *single step* through the code segment if it is small enough or we could set breakpoints at intermediate points within the code segment. Many errors occur at conditional branches, usually because the wrong condition is tested or the wrong variable is tested. Thus, if we set breakpoints just before branch operations, then we can check the Status Register to see if the conditions are correct.

Common Processor Errors

The following are some of the more common errors and possible meanings.

IMPORTANT NOTE REGARDING PROGRAM COUNTER

Remember the *program counter* (PC) always points to the next instruction, or the argument (ie an address in memory) of the instruction which generated the exception.

The reason for this is that the ColdFire is a *pipelined processor*, meaning that there could be several instructions in the pipeline at the time an error is detected. The processor stops on the next instruction before it enters the pipeline.

ADDRESS ERROR

The CPU tried to fetch an instruction while the PC pointed to an odd-numbered address. An example would be:

```
JSR $13005
```

The error returned is:

```
Address Error: FS=4 Physical bus error on instruction fetch
```

ACCESS ERROR

This kind of error:

```
Access Error: FS=8, Physical bus error on operand write
```

can be caused by any of the following situations:

- The last instruction tried to access restricted memory locations. Program constructs which lead to this are infinite loops that read or write to memory, or a branch or jump to these restricted memory addresses.
- During a download: The reason is because you tried to download to restricted memory locations.
- The last instruction tried to do something illegal. Examine the last instruction executed and the data it was operating in both memory and registers. The error should become evident, as well as its resolution. Note that the PC will point to the next instruction!

AUTOVECTOR INTERRUPT LEVEL {1-7} ERROR

This may be encountered when using interrupts. The autovector interrupt address has not been set correctly.

YOUR PROGRAM HAS SUDDENLY CHANGED FOR NO APPARENT REASON

If you have found yourself saying "Ack! My program is no longer what it used to be!" please keep reading. Your stack probably overwrote your program. Either you are pushing values onto the stack in an infinite loop, you are jumping to a subroutine in an infinite loop including infinite interrupts, or you have not relocated the stack to a very safe location.

Example Software Program: Hello World!

This section will guide you through getting some basic code onto the ColdFire SBC. First thing will be to generate a basic "Hello World" style application. First though, come to terms with the fact that you don't have a C library anymore. That's right, no printf, malloc, strcat, strdup, fopen, etc, etc, etc. You are writing the bottom of the entire system, the OS. The OS cannot make assumptions like applications can, and in fact, must be the provider of the services that make printf, malloc, and FILE* possible. Sit back and think about what this means for your design.

Okay, enough thinking. All is not lost. It's not "Goodbye, cruel world." You will see why in a moment.

THE ENTRY POINT

Unlike your previous experiences with C programming, the function main is not where it is all going to start. When coding in C under an existing OS, the entry code for the process is provided for you by the compiler. However, since you are going to be writing the OS, it is up to you to provide this entry code. Luckily, this code does not need to perform very much. In order, it must do the following:

- Allocate space for a new stack and a pointer to the monitor's stack
- Jump into the function main
- When main is finished, restore the old stack
- Execute TRAP to get back to the monitor

Sample entry/exit code is provided in the file start.s It performs these exact actions.

```
/* Allocate space for new stack and old stack pointer */
.comm old_stack,4
.comm main_stack,4096

.even

/* Install a new stack */
move.l %a7,old_stack
move.l #main_stack+4096, %a7

/* Jump into main */
jsr main

/* Restore old stack */
move.l old_stack,%a7

/* Store return value from main */
move.l %d0, %d7

/* Get back to the monitor */
move.l #0,%d0
trap #15
```

Onto the C Programming

Now that the function main is getting called properly we can write some actual C code. Look at the file *hello.c* from the **manual_5307.zip** package. You should find that it contains a total of four functions.

COMPILER IDIOSYNCRACIES: __MAIN

Ignore the first two for now and look at main and __main. What the !@#$ is __main? Well, this is the infamous __main that gcc outputs as the first part of main so that C++-"isms" can be correctly handled. So, that means we get to ignore it but just making a function that does nothing but return to keep the linker happy. That leaves main making a call to rtx_dbug_outs which will print a string to the dBug terminal window. For now, lets ignore the two functions above __main that do all the work and see if we can get this source compiled and linked and onto the SBC.

DATA SIZES FOR C PRIMITIVES

In the C language the variables int, short, long are not guaranteed to provide any given storage size. The file rtx_int.h included at the end of this document provides the correct #defines to generate variable of various bit sizes and signs.

If you want to use your own primitives, always make sure you generate some assembly code from the compiler to ensure your size choices are correct. Use the -S compiler switch to generate assembly code. Below is a segment from the gcc manual page to help clarify this option:

```
-S        Stop after the stage of compilation proper; do not  as-
          semble.   The output is an assembler code file for each
          non-assembler input file specified.
```

EXAMPLE: UINT32

Consider the following #define statement:

```
#define UINT32  unsigned long int
```

The first letter, be it U or S indicates whether it is *unsigned* or *signed*, respectively. The 32 implies the number of bits of this primitive. In the case of the ColdFire, being a 32-bit processor, integer primitives are 32-bits in length.

HELLO.C

```
#include "../rtx_inc.h"

/*
 * Prototypes
 */
VOID rtx_dbug_out_char( CHAR c );
SINT32 rtx_dbug_outs( CHAR* s );



/*
 * C Function wrapper for TRAP #15 function to output a character
 */
VOID rtx_dbug_out_char( CHAR c )
{

    /* Store registers */
    asm( "move.l %d0, -(%a7)" );
    asm( "move.l %d1, -(%a7)" );

    /* Load CHAR c into d1 */
    asm( "move.l 8(%a6), %d1" );

    /* Setup trap function */
    asm( "move.l #0x13, %d0" );
    asm( "trap #15" );

    /* Restore registers  */
    asm(" move.l %d1, (%a7)+" );
    asm(" move.l %d0, (%a7)+" );

}


/*
 * Print a C-style null terminated string
 */
SINT32 rtx_dbug_outs( CHAR* s )
{
    if ( s == NULL )
    {
        return RTX_ERROR;
    }
    while ( *s != '\0' )
    {
        rtx_dbug_out_char( *s++ );
    }
    return RTX_SUCCESS;
}
```

```
/*
 * gcc expects this function to exist
 */
int __main( void )
{
    return 0;
}


/*
 * Entry point, check with m68k-coff-nm
 */
int main( void )
{
    unsigned int i;
    unsigned long j;


    j = 3;
    j = j * 24567;

    i = 10;
    i = i * 35682824;

    rtx_dbug_outs( "Hello World!\n\r" );
}
```

Compiling Code for the ColdFire

This is really pretty easy in this case. Simply type in the following command from a command prompt window under Nexus or ECE Unix:

MCF5307 COMMAND

```
m68k-elf-gcc -Wall -nostdlib -m5200 -Tmcf5307.ld -o hello.bin ../start.s
hello.c
```

This will build a file named hello.bin. To get a better understanding of what that command did, let us pull it apart.

TABLE 10. gcc Command Arguments and Details

Command / Argument	Explanation
m68k-elf-gcc	This is the actual compiler, gcc
-Wall	Enable all warnings and errors.
-nostdlib	This tells gcc that we don't have a standard C library.
-m5200	This tells gcc that we want it to generate code for the 52XX ColdFire family of processors.
-Tmcf5307.ld	This tells gcc's linker how the memory on the SBC is laid out. See Table 1 on page 7

TABLE 10. gcc Command Arguments and Details

Command / Argument	Explanation
-o hello.bin	Tells gcc to put the output binary in this file.
start.s hello.c	Compile and link these files to make hello.bin

Important Compiler Information!

USE A MAKEFILE!

In your project, you should *always* call m68k-elf-gcc with the *-nostdlib* and *-m5200* options. If you don't there will be problems at link time if you forget *-nostdlib,* because the compiler will assume you want to link against the standard C libary. Also, you will have problems at runtime if you forget *-m5200*. Never forget these options. See the Unix man page if one is available.

The best way of doing this is to use a makefile, like that shown in the hello directory, as a starting point. It is usually called either "Makefile" or "makefile" in a directory. Either way, the "make" utility will find it and process it.

A Makefile will allow you to specify compiler options and and linker flags in variables, as well as allow you to define high level commands and actions that will will be carried out. For example, take a look at the CFLAGS and LDFLAGS variables, and look at the "clean:" section of the Makefile and the commands below it. You will see that typing "make clean" will run the "rm -f *.s19 *.o *.map" command.

```
# Makefile
# David Grant, 2004

CC=m68k-elf-gcc
CXX=m68k-elf-g++
CFLAGS=-O2 -Wall -m5200 -pipe -nostdlib
LD=m68k-elf-gcc
AS=m68k-elf-as
AR=m68k-elf-ar
ARFLAGS=
OBJCPY=m68k-elf-objcopy
ASM=../start.s
LDFLAGS = -T../mcf5307.ld -Wl,-Map=hello.map

# Note, gcc builds things in order, it's important to put yhe
#  ASM first, so that it is located at the beginning of our program.
hello.s19: hello.c
$(CC) $(CFLAGS) $(LDFLAGS) -o hello.o $(ASM) hello.c
$(OBJCPY) --output-format=srec hello.o hello.s19

clean:
rm -f *.s19 *.o *.map
```

THE GCC __END VARIABLE

Since the RTX needs to be able to dynamically allocate memory, it needs to know were the free store of memory begins. The gcc compiler provides a nice method for determining this. The compiler will export a symbol that resides at the address where all static code and data end. This variable is named "__end", thats "underscore underscore e n d" OK? By taking the address of this symbol (as if it was a global variable) you can determine the address at which free memory begins. So take note of this. Its going to prove useful.

Converting to S-Records

There is one more step before the file can actually be sent to the ColdFire server, to an individual ColdFire board, or to the software emulator thats available on harshrealm.

The file hello.bin is a binary file and we need a text based S-record file.

The Motorola S-record format is produced by many assemblers and compilers. S-record format encapsulates the opcodes with the addresses and checksums. This S-19, or object file, can be download once you have made one from a binary file.

Thankfully, the gcc toolset comes with a utility to perform this conversion for us. Simply run the following:

```
m68k-elf-objcopy -v --target=srec hello.bin hello.txt
```

This results in a text file, hello.txt, being generated. This is the file that can be sent to the ColdFire. Note that the "-v" switch is optional. Its there so that you can see a message similar to the following example, telling you exactly what was copied where:

```
{harshrealm:36} m68k-elf-objcopy -v --target=srec hello.bin hello.txt
copy from hello.bin(elf32-m68k) to hello.txt(srec)
```

You might want to slightly modify your command to make S-19 files that have the suffix s19, rather than txt, so that you can more easily discern what this file is intended to be used for, such as hello.s19 instead of hello.txt. Also, you can put this command in a Makefile to help automate the creation of S-19 files easily.

Downloading from the Computer to ColdFire Board

Once you have successfully generated an "S" record file, you can transfer it to the ColdFire board. There is an RS-232 cable connected to the serial port of the Nexus PC and to the ColdFire board, or you can also use Unix.

USING HYPERTERM WITHIN NEXUS

HyperTerminal is a Windows-based communications program. To start it, follow this sequence:

Start > Programs > Accessories > Communications > HyperTerminal

To download, make sure your system is sitting at the janusROM prompt. Then, in HyperTerm's menu bar, go to Transfer or Send Text File . Select to view all file types and then choose the file that you previously saved on your N: drive.

At the prompt, type in:

```
janusROM> dl
```

This command, download serial, will cause the monitor to go into a mode where it expects an S-record file.

Under HyperTerminal simply go to the Transfer menu and select “Send Text File”, choose the hello.txt file, and send it to the board. You should see your file displayed on the terminal as it is received by the ColdFire monitor. When it is finished the ColdFire board should report that the download was successful.

You should be able to type in “go 10100000” at the firmware prompt and it should print out “Hello World!”

If it does, congratulations!! Your first C program is downloaded and running on the ColdFire. You now have the ability to compile and run software on the ColdFire.

USING UNIX/LINUX TO DOWNLOAD FILES

If you prefer to use Unix or Linux for software development work* then read this section. Please note that this section applies only to Solaris, but Linux users should be able to adapt this information easily to their platforms.

Firstly connect serial cables from the ports of the Sun workstation to the ports of the ColdFire board. You will likely need hardware adapters, since the Sun machine has both 9 and 25 pin† connectors for its serial ports, the assumption being that at least one will work for most situations.

Open two terminal windows, such as dtterm or xterm, and start “tip” connections to /dev/ttya and /dev/ttyb as shown:

```
tip -19200 /dev/ttya
```

This command will open a 19200 baud tip connection to the first serial port device on the Sun workstation. Do the same thing for ttyb. Depending which serial port on the ColdFire board you have connected the serial cable to, one terminal window or the other will give you a ROM monitor prompt when you turn on or reset the ColdFire board.

*. I (S. Singh) happen to be one of those people that believes Unix is the all around best operating system for software development work of any kind. While you can get gcc and related utilities for Windows, all of these powerful and free utilities came from the Unix world first. In fact, all of the most respected and powerful software comes from the Unix world to Windows. The only exception to this is office productivity software suites such as Office 97, which is the defacto standard in most business environments. Unix platforms are finally have something comparable with OpenOffice, and its free too! So if you are serious about using the latest and greatest technology, you really want to learn more about Solaris and Linux.

†. If you are using a Linux PC, you should be able to use identical cables for each port. I believe that the terminal software to use is called minicom, but I have not verified this.

See the man page for the control sequences to copy files to the ColdFire board, in order to download S-Record files to it, as discussed in “Converting to S-Records” on page 29.

Now, continue to “Hello, World!” on page 32 and read that section.

Using the ColdFire Emulator Software

In a terminal window, start the ColdFire emulator on either harshrealm or a Sun workstation with:

```
/opt/ColdFire-0.3.2/bin/ColdFire
```

You should see startup messages similar to the following:

```
Use CTRL-C (SIGINT) to cause autovector interrupt 7 (return to monitor)
Loading memory modules...
Loading board configuration...
        Opened [/opt/ColdFire-0.3.2/share/cjdesign-5307.board]
Board ID: CJDesign
CPU: 5307 (Motorola ColdFire 5307)
        unimplemented instructions: CPUSHL PULSE WDDATA WDEBUG
        69 instructions registered
        building instruction cache... done.
Memory segments: dram  timer0  timer1  uart0(on port 5206)
                 uart1(on port 5207)  sim  flash  sram


*Remember to telnet to the ports specified below if you want to see any
output!

Hard Reset...
Initializing monitor...
Enter 'help' for help.
NOTE: (uart0 on port 5206)
NOTE: (uart1 on port 5207)
dBug>
```

DOWNLOADING S-RECORD FILE TO EMULATOR

You should then have a dBug prompt, similar to the ROM program in the real boards. Type “help” to see the commands available to you, if you like. Then when you are ready, type “dl” and the file name of your S-record file to download it into the emulator:

```
dBUG> dl hello.txt
Downloading S-Record...
Done downloading S-Record.
dBug>
```

OPEN TERMINAL WINDOWS FOR COLDFIRE SERIAL PORTS

In order to see output from the serial ports on the software emulator, you need to open telnet connections to the IP ports mentioned in the “NOTE:” lines printed when you started the ColdFire emulator.

*Important: Keep in mind that these ports may change, and are *not always guaranteed* to be at ports 5206 and 5207. They may be at different port numbers depending on whether other people on the system are running their own copies of the ColdFire emulator. When that happens, they get 2 ports, and whoever comes after gets the next IP ports that are available. This allows multiple copies of the emulator to be running on a computer, maximizing availability while allowing people to still communicate with the emulator remotely.

So go ahead and open two more DOS Windows (if accessing from Nexus) or X-Terminals (if accessing from Sun stations or X-session) and telnet to the computer that is running your copy of the emulator, and be sure to specify the ports you were given at the startup screen. Consider the example below. Its quite simple and straightforward:

```
ssingh@area51[1] telnet harshrealm 5206
Trying 129.97.56.25...
Connected to harshrealm.uwaterloo.ca.
Escape character is '^]'.
uart0
```

See how you are told what serial port you are now connected to: uart0. At this point whatever output is produced by the ColdFire emulator and sent to this virtual serial port, will now be seen by you, since you are connected to it via telnet. Do the same thing for uart1; ie. telnet to the host and port for uart1 using another window, and you're good to go.

HELLO, WORLD!

Switch to your window with the dBug prompt and run your program, which has been compiled and linked to start at 10100000. Thats "101" followed by five zeros. If you really want to see the details of the memory map, check the file mcf5307.ld for the line: "ram : ORIGIN = 0x10100000, LENGTH = 31M" That line in the linker script defines where the beginning of RAM is for the ColdFire board, and this is where your "hello.c" file has been compiled to run from:

```
dBug> go 10100000
dBug>
```

In the uart0/serial port window, you should see the result:

```
uart0
Hello World!
```

Congratulations! You have the ability to compile C and run software on the ColdFire.

Combining C and Assembly Language

There are two ways of mixing C with assembly; putting assembly commands inline directly with your C code or by making a separate assembly source file.

INLINE ASSEMBLY

Inline assembly is the simplest method for combining C and assembly. The assembly code is simply placed one line at a time into the function asm(). You call it like this:

```
asm( "sub.l  #60, %a7" );
```

This simply causes the compiler to emit the given assembly code right along with the assembly code it generates while parsing your C sources. Therefore, *you* are in charge of saving or restoring any registers that were used or modified during the inline code.

The compiler doesn't do anything but blindly compile your code. It does not try to second-guess you, or even save registers for you. It just puts your code right in there with whatever assembly code it generated while parsing the C source before and after your section of assembly.

Inline assembly is most often used in cases when C code is perfectly fine up to the point of actually setting up a system specific register or executing a system specific operation that C does not provide a definition for.

For example, fixing up the stack in your dispatcher and calling the *rte* instruction would be done using this method.

One caveat, however, is that you *cannot* access local variables by name, only global symbols will be recognized. This is because local variables are really just offsets into the current call stack, and the stack space they currently use will be returned to the system when the function you are in is completed. But global variables actually have a more permanent presence with a fixed address and an exported symbol for the linker.

CALLING BETWEEN C AND ASSEMBLY

This form of mixing is a little *purer* and requires a little more work. However, there are often times when the "extra" code generated by the C compiler is harmful to the proper operation of the system and exact instructions must be executed. For instance, the entry point for interrupts must be implemented in this way.

By convention, the extension: .s is used for assembly files. A simple example can be found in the asm directory from the example source. The file main.s can be compiled and linked with startup.s to make a program that does nothing but return to the monitor. Notice the use of the .globl keyword to export the entry point to the function main, which is in fact just a label (main:) which marks were the function begins. Very simple.

MAIN.S

```
     .even
     .globl man
main:
     rts
```

It is a little more complicated when passing arguments to and from C code and assembly code. gcc uses the stack to pass all parameters and uses the register D0 to pass back return values. It is the responsibility of the caller of the function to remove the values from the call stack once a function is completed. In assembly, parameters are simply accessed by performing byte offsets using the register A7. An example of how to make a C callable assembly routine follows. The following example gives a working memset-style function that can be called directly in any C code linked with it.

MEMSET.S

```
#############################################################
#   SINT32 rtx_memset( VOID* ptr, UINT32 value, UINT32 size )
#
#   a0: start address
#   a1: end address
#   d1: value to set (lsb)
#   d0: return code
#############################################################

        .even
        .globl rtx_memset
rtx_memset:
        # save registers
        link    %a6, #0
        sub.l   #12, %a7
        movem.l %a0/%a1/%d1, (%a7)

        # initialize working vars
        clr.l%d0
        move.l  8(%a6), %a0
        cmp.l   #0, %a0
        beq.rtx_memset_2
        move.l  %a0, %a1
        add.l   16(%a6), %a1
        move.l  12(%a6), %d1

.rtx_memset_1:
        cmp.l   %a0,%a1
        beq     .rtx_memset_3
        move.b  %d1, (%a0)+
        bra     .rtx_memset_1

.rtx_memset_2:
# Return an error
        move.l  #-1, %d0

.rtx_memset_3:
        movem.l (%a7), %a0/%a1/%d1
        add.l   #12, %a7
        unlk    %a6
        rts
```

Interrupts in C

Since m68k-elf-gcc is a cross-platform compiler, it does not support any processor specific hardware keywords for generating functions that are safe to call directly as an interrupt, such as in ECE 324.

This requires the use of a small assembly *stub function* that in turn calls the given C function. This allows for maximum flexibilty in terms of *what* gets done *where* (assembly or C), and what registers/states are saved on interrupt. The main sources of interrupts in your RTX design will be TRAPs from processes, the timer, and the serial port. Basic source and implementation details will be given for each of these in the next two subsections.

TRAP.C

```
#include "../rtx_inc.h"

/*
 * Prototypes
 */
VOID rtx_dbug_out_char( CHAR c );
SINT32 rtx_dbug_outs( CHAR* s );


/*
 * C Function wrapper for TRAP #15 function to output a character
 */
VOID rtx_dbug_out_char( CHAR c )
{
    /* Store registers */
    asm( "move.l %d0, -(%a7)" );
    asm( "move.l %d1, -(%a7)" );

    /* Load CHAR c into d1 */
    asm( "move.l 8(%a6), %d1" );

    /* Setup trap function */
    asm( "move.l #0x13, %d0" );
    asm( "trap #15" );

    /* Restore registers  */
    asm(" move.l %d1, (%a7)+" );
    asm(" move.l %d0, (%a7)+" );
}


/*
 * Print a C-style null terminated string
 */
SINT32 rtx_dbug_outs( CHAR* s )
{
    if ( s == NULL )
    {
        return RTX_ERROR;
    }
    while ( *s != '\0' )
    {
        rtx_dbug_out_char( *s++ );
    }
    return RTX_SUCCESS;
}

/*
 * gcc expects this function to exist
 */
int __main( void )
{
    return 0;
}

/*
 * This function is called by the assembly STUB function
 */
VOID c_trap_handler( VOID )
{
```

```
    rtx_dbug_outs( “Trap Handler!!\n\r” );
}

/*
 * Entry point, check with m68k-coff-nm
 */
int main( void )
{
    /* Load the vector table for TRAP #0 with our assembly stub
       address */
    asm( “move.l #asm_trap_entry,%d0” );
    asm( “move.l %d0,0x10000080” );

    /* Trap out three times */
    asm( “TRAP #0” );
    asm( “TRAP #0” );
    asm( “TRAP #0” );
}
```

TRAP_ENTRY.S

```
.globl asm_trap_entry
.even
asm_trap_entry:
move.l %d0, -(%a7)
move.l %d1, -(%a7)
move.l %d2, -(%a7)
move.l %d3, -(%a7)
move.l %d4, -(%a7)
move.l %d5, -(%a7)
move.l %d6, -(%a7)
move.l %d7, -(%a7)
move.l %a0, -(%a7)
move.l %a1, -(%a7)
move.l %a2, -(%a7)
move.l %a3, -(%a7)
move.l %a4, -(%a7)
move.l %a5, -(%a7)
move.l %a6, -(%a7)
jsr     c_trap_handler
move.l (%a7)+, %a6
move.l (%a7)+, %a5
move.l (%a7)+, %a4
move.l (%a7)+, %a3
move.l (%a7)+, %a2
move.l (%a7)+, %a1
move.l (%a7)+, %a0
move.l (%a7)+, %d7
move.l (%a7)+, %d6
move.l (%a7)+, %d5
move.l (%a7)+, %d4
move.l (%a7)+, %d3
move.l (%a7)+, %d2
move.l (%a7)+, %d1
move.l (%a7)+, %d0
rte
```

VECTOR BASE REGISTER INITIALIZATION CODE

The following initialization code is used to prepare for the use of non-autovectored interrupts. This is very simple to change and you will notice this code in the timer and serial port sections to follow. Don't forget to include this code somewhere during the initialization of your RTX. For more information, see “The Vector Base Register” on page 6.

```
/*
 * Move the VBR into real memory
 */
asm( "move.l %a0, -(%a7)" );
asm( "move.l #0x10000000, %a0 "
);
```

The Timer

The ColdFire boards used in the labs are equipped with an external timer. However, the ColdFire itself also has two internal timers that are more flexible than the external chip and are also “cleaner” to deal with. The external chip was used in ECE 222 and will not be discussed here further. However, sample code for setting up the ColdFire's internal timer will follow. Please consult the MCF5307 Users Manual for more detail on the meaning of all the UART's registers.

TIMER.C

```
#include “../rtx_inc.h”

/*
 * Prototypes
 */
VOID rtx_dbug_out_char( CHAR c );
SINT32 rtx_dbug_outs( CHAR* s );

/*
 * Global Variables
 */
SINT32 Counter = 0;


/*
 * C Function wrapper for TRAP #15 function to output a character
 */
VOID rtx_dbug_out_char( CHAR c )
{
     /* Store registers */
    asm( “move.l %d0, -(%a7)” );
    asm( “move.l %d1, -(%a7)” );

    /* Load CHAR c into d1 */
    asm( “move.l 8(%a6), %d1” );

    /* Setup trap function */
    asm( “move.l #0x13, %d0” );
    asm( “trap #15” );

     /* Restore registers  */
```

```
    asm(" move.l %d1, (%a7)+" );
    asm(" move.l %d0, (%a7)+" );
}


/*
 * Print a C-style null terminated string
 */
SINT32 rtx_dbug_outs( CHAR* s )
{
    if ( s == NULL )
    {
        return RTX_ERROR;
    }
    while ( *s != '\0' )
    {
        rtx_dbug_out_char( *s++ );
    }
    return RTX_SUCCESS;
}


/*
 * gcc expects this function to exist
 */
int __main( void )
{
    return 0;
}


/*
 * This function is called by the assembly STUB function
 */
VOID c_timer_handler( VOID )
{
    Counter++;
//    rtx_dbug_out_char('.');
    /*
     * Ack the interupt
     */
    TIMER0_TER = 2;
}


SINT32 ColdFire_vbr_init( VOID )
{
    /*
     * Move the VBR into real memory
     */
    asm( "move.l %a0, -(%a7)" );
    asm( "move.l #0x10000000, %a0 " );
    asm( "movec.l %a0, %vbr" );
    asm( "move.l (%a7)+, %a0" );

    return RTX_SUCCESS;
}


/*
```

```
 * Entry point, check with m68k-coff-nm
 */
int main( void )
{
    UINT32 mask;

    /* Disable all interupts */
    asm( “move.w #0x2700,%sr” );

    ColdFire_vbr_init();

    /*
     * Store the timer ISR at auto-vector #6
     */
    asm( “move.l #asm_timer_entry,%d0” );
    asm( “move.l %d0,0x10000078” );

    /*
     * Setup to use auto-vectored interupt level 6, priority 3
     */
    TIMER0_ICR = 0x9B;

    /*
     * Set the reference counts, ~10ms
     */
    TIMER0_TRR = 1758;

    /*
     * Setup the timer prescaler and stuff
     */
    TIMER0_TMR = 0xFF1B;

    /*
     * Set the interupt mask
     */
    mask = SIM_IMR;
    mask &= 0x0003fdff;
    SIM_IMR = mask;

    /* Let the timer interrupt fire, lower running priority */
    asm( “move.w #0x2000,%sr” );

    /* Wait for 8 seconds to pass */
    rtx_dbug_outs( “Waiting approx. 5 seconds for Counter > 500\n\r” );
    Counter=0;
    while( Counter < 500 );

    rtx_dbug_outs( “Counter >= 500\n\r” );
}
```

TIMER_ENTRY.S

```
.globl asm_timer_entry
.even
asm_timer_entry:
move.l %d0, -(%a7)
move.l %d1, -(%a7)
move.l %d2, -(%a7)
move.l %d3, -(%a7)
move.l %d4, -(%a7)
move.l %d5, -(%a7)
move.l %d6, -(%a7)
move.l %d7, -(%a7)
move.l %a0, -(%a7)
move.l %a1, -(%a7)
move.l %a2, -(%a7)
move.l %a3, -(%a7)
move.l %a4, -(%a7)
move.l %a5, -(%a7)
move.l %a6, -(%a7)

jsr c_timer_handler

move.l (%a7)+, %a6
move.l (%a7)+, %a5
move.l (%a7)+, %a4
move.l (%a7)+, %a3
move.l (%a7)+, %a2
move.l (%a7)+, %a1
move.l (%a7)+, %a0
move.l (%a7)+, %d7
move.l (%a7)+, %d6
move.l (%a7)+, %d5
move.l (%a7)+, %d4
move.l (%a7)+, %d3
move.l (%a7)+, %d2
move.l (%a7)+, %d7
move.l (%a7)+, %d0

rte
```

The Serial Port

As with the timer, the ColdFire board has several different ways of performing serial I/O. There is a UART which the dBUG monitor uses to talk to the PC and you can take over that serial port. However, if you leave it alone a very powerful method of debugging will be made available. To this end, this section will provide the example code needed to get the J6 serial port on the ColdFire boart running with interrupts. This is the port that you should use to provide a simple (or complex) user interface for the RTX. Please consult the MCF5307 Users Manual for more detail on the meaning of all the UART's registers.

This sample code will allow you to finally use both terminal programs at the same time. Once run, every character typed on the RTX terminal will be echoed on that terminal as well as reported back on the dBUG terminal.

SERIAL.C

```
#include “../rtx_inc.h”

/*
 * Prototypes
 */
VOID rtx_dbug_out_char( CHAR c );
SINT32 rtx_dbug_outs( CHAR* s );

/*
 * Global Variables
 */
volatile BYTE CharIn = ‘!’;
volatile BOOLEAN Caught = TRUE;
volatile BYTE CharOut = ‘\0’;
CHAR StringHack[] = “You Typed a Q\n\r”;


/*
 * C Function wrapper for TRAP #15 function to output a character
 */
VOID rtx_dbug_out_char( CHAR c )
{
   /* Store registers */
    asm( “move.l %d0, -(%a7)” );
    asm( “move.l %d1, -(%a7)” );

    /* Load CHAR c into d1 */
    asm( “move.l 8(%a6), %d1” );

    /* Setup trap function */
    asm( “move.l #0x13, %d0” );
    asm( “trap #15” );

   /* Restore registers  */
    asm(“ move.l %d1, (%a7)+” );
    asm(“ move.l %d0, (%a7)+” );

}


/*
 * Print a C-style null terminated string
 */
SINT32 rtx_dbug_outs( CHAR* s )
{
    if ( s == NULL )
    {
        return RTX_ERROR;
    }
    while ( *s != ‘\0’ )
    {
        rtx_dbug_out_char( *s++ );
    }
    return RTX_SUCCESS;
}


/*
 * gcc expects this function to exist
 */
```

```
int __main( void )
{
    return 0;
}


/*
 * This function is called by the assembly STUB function
 */
VOID c_serial_handler( VOID )
{
    BYTE temp;

    /*
     * Ack the interupt
     */
    temp = SERIAL1_UCSR;

    /* See if data is waiting.... */
    if( temp & 1 )
    {
        CharIn = SERIAL1_RD;
        Caught = FALSE;
    }
    /* See if port is ready to accept data */
    else if ( temp & 4 )
    {
        /* Write data to port */
        SERIAL1_WD = CharOut;
        /* Disable tx Interupt */
        SERIAL1_IMR = 2;
    }

    return;
}


SINT32 ColdFire_vbr_init( VOID )
{
    /*
     * Move the VBR into real memory
     *
     * DG: actually, it’ll already be here.
     */
    asm( “move.l %a0, -(%a7)” );
    asm( “move.l #0x10000000, %a0 “ );
    asm( “movec.l %a0, %vbr” );
    asm( “move.l (%a7)+, %a0” );

    return RTX_SUCCESS;
}



/*
 * Entry point, check with m68k-elf-nm
 */
int main( void )
{
    UINT32 mask;
```

```
    /* Disable all interupts */
    asm( “move.w #0x2700,%sr” );

    ColdFire_vbr_init();

    /*
     * Store the serial ISR at user vector #64
     */
    asm( “move.l #asm_serial_entry,%d0” );
    asm( “move.l %d0,0x10000100” );

    /* Reset the entire UART */
    SERIAL1_UCR = 0x10;

    /* Reset the receiver */
    SERIAL1_UCR = 0x20;

    /* Reset the transmitter */
    SERIAL1_UCR = 0x30;

    /* Reset the error condition */
    SERIAL1_UCR = 0x40;

    /* Install the interupt */
    SERIAL1_ICR = 0x17;
    SERIAL1_IVR = 64;

    /* enable interrupts on rx only */
    SERIAL1_IMR = 0x02;

#if 0
    /* The JanusROM takes care of these settings now! */
    /* Set the baud rate (19200) */
    SERIAL1_UBG1 = 0x00;
    SERIAL1_UBG2 = 0x28;

    /* Set clock mode */
    SERIAL1_UCSR = 0xDD;

    /* Setup the UART (no parity, 8 bits ) */
    SERIAL1_UMR = 0x13;

    /* Setup the rest of the UART (noecho, 1 stop bit ) */
    SERIAL1_UMR = 0x07;
#endif

    /* Setup for transmit and receive */
    SERIAL1_UCR = 0x05;

    /* Enable interupts */
    mask = SIM_IMR;
    mask &= 0x0003dfff;
    SIM_IMR = mask;


    /* Enable all interupts */
    asm( “move.w #0x2000,%sr” );

    rtx_dbug_outs( “Type Q or q on RTX terminal to quit.\n\r” );
```

```
    /* Busy Loop */
    while( CharIn != 'q' && CharIn != 'Q' )
    {
        if( !Caught )
        {
            Caught = TRUE;
            CharOut = CharIn;

            /* Nasty hack to get a dynamic string format, grab the charac-
ter
     *  before turning the interrupts back on. */
            StringHack[12] = CharIn;

            /*  enable tx interrupts  */
            SERIAL1_IMR = 3;

    /* Now print the string to debug, note that interrupts
     *  are now back on. */
            rtx_dbug_outs( StringHack );
        }
    }

    /* Disable all interupts */
    asm( "move.w #0x2700,%sr" );

    /* Reset globals so we can run again */
    CharIn = '\0';
    Caught = TRUE;
}
```

SERIAL_ENTRY.S

```
.globl asm_serial_entry
.even
asm_serial_entry:
move.l %d0, -(%a7)
move.l %d1, -(%a7)
move.l %d2, -(%a7)
move.l %d3, -(%a7)
move.l %d4, -(%a7)
move.l %d5, -(%a7)
move.l %d6, -(%a7)
move.l %d7, -(%a7)
move.l %a0, -(%a7)
move.l %a1, -(%a7)
move.l %a2, -(%a7)
move.l %a3, -(%a7)
move.l %a4, -(%a7)
move.l %a5, -(%a7)
move.l %a6, -(%a7)

jsr c_serial_handler

move.l (%a7)+, %a6
move.l (%a7)+, %a5
move.l (%a7)+, %a4
move.l (%a7)+, %a3
move.l (%a7)+, %a2
move.l (%a7)+, %a1
move.l (%a7)+, %a0
```

```
move.l (%a7)+, %d7
move.l (%a7)+, %d6
move.l (%a7)+, %d5
move.l (%a7)+, %d4
move.l (%a7)+, %d3
move.l (%a7)+, %d2
move.l (%a7)+, %d7
move.l (%a7)+, %d0

rte
```

Debugging the OS

The most difficult part of this project is trying to figure out why things are not working as they should. Since there is currently no source level debugger to run between the board and the PC, debugging will have to be done using the monitor and *printf*-style debugging.

USING TRAP 15

By now it should be apparent that by using the second serial port for your user I/O for the RTX you now have a very powerful debugging tool. The code given for rtx_dbug_outs and rtx_dbug_out_char can be called from within anything such as:

• TRAP handlers
• hardware interrupts
• user processes

For instance, try moving the rtx_dbug_outs(StringHack) into the c_handler_function; no difference. These calls are slow however and should be coded in such a way that they can be stripped out at compile time. A good method to do this is using #define preprocessor directives for conditional compiling. That way a simple switch at compile time can be used to mask out this trap code.

Although printing static strings isn't very flexible, using rtx_dbug_out_char a simple printf-like function could be written to help format numbers and addresses. Having this ability is very useful for trapping "hot-keys" inside the serial ISR and then dumping out kernel information to the dBug/JanusROM terminal including:

• process ID's
• process states
• trace buffers
• whatever else you want to track.

These calls are also very useful because they are serialized through the TRAP function so that the actual order of events within the RTX can be logged.

GENERATING A MAP FILE

A map file can be created by adding a command line parameter:

-Wl,-Map=map

to m68k-elf-gcc as shown in this example:

```
m68k-elf-gcc -I./include -m5200 -nostdlib -Tmcf5307.ld -Wl,-Map=chaos.map
-Wall -o chaos -lgcc chaos.c start.s
```

In either case, a file named *chaos.map* will be generated. This file will have a great deal of low-level information about what object files are included and what symbols and addresses each object file is providing. *You can put this kind of command in your makefile, to make things easier and more automatic.*

USING M68K-ELF-GCC VS. COFF

Note that if you are using elf as your object file format. COFF is an older object file format that is not in widespread use anymore. For ECE 354, you should use the m68k-elf-gcc command. Older versions of the cross compiler used m68k-coff-gcc.

M68K-ELF-NM

Also, there is a utility named *m68k-elf-nm* that will dump out a listing of symbols from the elf binary file. This will allow you to find the location of global variables and functions debugging using the monitor just like in ECE 222. That is why the compile is in two steps so that not only is an S-record file generated, but also a elf binary that can be queried in this manner.

Reference: rtx_inc.h

All of the C source examples include a header file named rtx_inc.h. In it you will find definitions for various constants that will make your own coding easier. Take a look at its contents given here.

RTX_INC.H

```
/*******************************************************************
 *
 *                            ECE 354 RTX
 *                    (c)1998, All Rights Reserved
 *
 *                   Chris McKillop and  Craig Stout
 *                   [cdmckill,castout]@uwaterloo.ca
 *
 ******************************************************************/

/*  $Id: rtx_base.h,v 1.15 1998/04/02 05:49:39 castout Exp $ */

/********************************************************************
 *
 *  $Source: /home/cdmckill/cvsroot/rtx/include/rtx_base.h,v $
 *  $Author: castout $
 *  $Date: 1998/04/02 05:49:39 $
 *
 *  Purpose:   Base include file for the RTX
 *
 *  $Log: rtx_base.h,v $
 *
 *  [LOG REMOVED]
 *
 ******************************************************************/
```

```
#if !defined( RTX_BASE_H__ )
#define RTX_BASE_H__

/*******************************************************************
 *                       CONSTANTS
 *******************************************************************/

/*
 * Data Types by Size
 */
#define SINT32  signed long int
#define UINT32  unsigned long int
#define SINT16  signed short int
#define UINT16  unsigned short int
#define SINT8   signed char
#define UINT8   unsigned char
#define CHAR    signed char
#define BYTE    unsigned char
#define VOID    void
#define BOOLEAN signed long int


#define ESC                 0x1B
#define BKSP                '\b'
#define CR                  '\r'
#define LF                  '\n'


#if !defined( TRUE )
#define TRUE 1
#endif

#if !defined( FALSE )
#define FALSE 0
#endif

#if !defined( NULL )
#define NULL 0
#endif


/*
 * ColdFire system defines
 */
#define RTX_ColdFire_MBAR    (BYTE *)(0xF0000000)
#define SIM_IMR              *( (UINT32 *)( RTX_ColdFire_MBAR + 0x44 ) )


/*
 * ColdFire Timer Defines
 */
#define TIMER0_TMR  *( (UINT16 *)( RTX_ColdFire_MBAR + 0x140 ) )
#define TIMER0_TRR  *( (UINT16 *)( RTX_ColdFire_MBAR + 0x144 ) )
#define TIMER0_TCN  *( (UINT16 *)( RTX_ColdFire_MBAR + 0x14C ) )
#define TIMER0_ICR  *( RTX_ColdFire_MBAR + 0x04d )
#define TIMER0_TER  *( RTX_ColdFire_MBAR + 0x151 )

#define TIMER1_TMR  *( (UINT16 *)(RTX_ColdFire_MBAR + 0x180 ) )
#define TIMER1_TRR  *( (UINT16 *)(RTX_ColdFire_MBAR + 0x184 ) )
#define TIMER1_TCN  *( (UINT16 *)(RTX_ColdFire_MBAR + 0x18C ) )
```

```
#define TIMER1_ICR  *( RTX_ColdFire_MBAR + 0x04e )
#define TIMER1_TER  *( RTX_ColdFire_MBAR + 0x191 )


/*
 * ColdFire Serial Defines
 */
#define SERIAL1_UCR     *( RTX_ColdFire_MBAR + 0x208 )
#define SERIAL1_UBG1    *( RTX_ColdFire_MBAR + 0x218 )
#define SERIAL1_UBG2    *( RTX_ColdFire_MBAR + 0x21C )
#define SERIAL1_UCSR    *( RTX_ColdFire_MBAR + 0x204 )
#define SERIAL1_UMR     *( RTX_ColdFire_MBAR + 0x200 )
#define SERIAL1_ICR     *( RTX_ColdFire_MBAR + 0x51  )
#define SERIAL1_IVR     *( RTX_ColdFire_MBAR + 0x230 )
#define SERIAL1_ISR     *( RTX_ColdFire_MBAR + 0x214 )
#define SERIAL1_IMR     *( RTX_ColdFire_MBAR + 0x214 )
#define SERIAL1_RD      *( RTX_ColdFire_MBAR + 0x20C )
#define SERIAL1_WD      *( RTX_ColdFire_MBAR + 0x20C )

/*
 * RTX Error Codes
 */
#define RTX_SUCCESS 0
#define RTX_ERROR   -1

#endif
```

Suggestions? Comments? Feedback on this Document

Your comments are important. Just as we are talking about fixing bugs in your software, help us fix bugs in this document. Help us make it better, easier to read, all around more useful. Please send suggestions for improvement, comments, corrections, or compliments to Sanjay Singh. If you prefer anonymity, send your comments to Irene Huang, and she will forward them to me, but I cannot reply to comments sent anonymously. I will take them into consideration, and under advisement, as best I can. If you want to discuss aspects of this manual, you will have to contact Sanjay directly.:

TABLE 11. **Contacts for ECE 354**

Contact	E-mail / Phone	Course Role
Sanjay Singh	ssingh@swen.uwaterloo.ca X6165	Unix environment, ColdFire Manual, Compiler, Software Emulator Maintainer
Irene Huang	yqhuang@ece.uwaterloo.ca X3226	Automated Project Evaluation codebase maintainer and CourseBook administrator
Eric Praetzel	praetzel@engmail.uwaterloo.ca X5249	ColdFire board and server, Nexus development environment

Index