



Introduction to
Embedded Systems
CMPE2200

AN INSTITUTE OF TECHNOLOGY COMMITTED TO STUDENT SUCCESS

NAIT CoursePack CP1503
Revision R12

All Rights Reserved

This publication © The Northern Alberta Institute of Technology (2019). All rights are reserved. No part of this publication may be reproduced, or transmitted in any form or by any means, or stored in a database and retrieval system, without the prior written permission of the copyright holder.

Address all inquiries to:
The Northern Alberta Institute of Technology
11762 - 106 Street, Edmonton, Alberta T5G 2R1

(This page intentionally blank)

Table of Contents

Topic 1 – Embedded Systems Theory and the 9S12X Device	1
Required supporting materials.....	1
Rationale.....	1
Expected Outcomes	1
Where are you at?	1
Embedded Controllers	2
The MC9S12XDP512 Microcontroller	2
The CNT MC9S12XDP512 I/O Board.....	5
Types of Interfaces	11
Port Addressing	12
Switches and LEDs.....	14
S12XCPU Assembly Language and the S12XCPU Microprocessor Core	16
Accumulators and Registers	17
Memory	19
Memory Map	21
Topic 2 –Microcontroller Programming.....	22
Required supporting materials.....	22
Rationale.....	22
Expected Outcomes	22
Connection Activity.....	22
Assembly Language Fundamentals.....	23
Assembler Directives.....	23
Instructions	23
Rudimentary Debugging Skills.....	26
Documentation and Comments	28
Using the Skeleton.txt File	29
Flowcharting.....	31
Subroutines	32
Libraries of Subroutines	33
S12XCPU Addressing Modes	35
Inherent - INH	35
Immediate - IMM	35
Extended – EXT	36
Direct – DIR	36
Relative – REL	36
Indexed – IDx, IDx1, IDx2, [IDx2], [D,IDx]	38

Frequently-Used Instructions.....	39
Masks and Bitwise Boolean Logic.....	41
Commands affecting an entire register or memory location.....	41
Commands affecting selected bits	41
Commands responding to selected bits.....	42
Using Variables and Constants	43
Programming in C.....	45
Setting Up an ANSI C Project.....	45
ANSI C Skeleton File	46
Switches and LEDs with ANSI C.....	47
Functions	47
Libraries of Functions.....	48
Summary.....	48
Numeric Manipulation.....	49
Understanding Base 10.....	49
Converting Binary to Decimal.....	49
Converting Hexadecimal to Decimal.....	50
Converting Hexadecimal to Binary	50
Converting Binary to Hexadecimal	51
8 Bit Arithmetic	51
Working with 2's Complement.....	52
Topic 3 –Interfacing With Internal and External Devices.....	53
Required supporting materials.....	53
Rationale.....	53
Expected Outcomes	53
Connection Activity.....	53
Disclaimer.....	53
Interfacing the ICM7218A 8-Digit LED Display Driver.....	54
ICM7218A Programming Tables	56
Sending Data to the ICM7218A	57
Seven Segment Display Library Components	58
Seven-segment Display Control Using ANSI C.....	59
SevSeg_Lib.h.....	59
SevSeg_Lib.c.....	59
Binary-Coded Decimal Representation and Manipulation.....	61
Converting Hexadecimal Values to BCD.....	62
Misc_Lib.h.....	63
HexToBCD.....	64
BCDToHex.....	65

Switch Management.....	67
Detecting Switch Change of State	67
Debouncing.....	69
SwCk() Debounced Switch Routine	69
Parallel Interfaces: Get On the Bus.....	70
Data Bus	70
Address Bus.....	70
Control Lines.....	70
LCD Displays Using the Hitachi HD44780U Controller.....	71
The HD44780-controlled LCD on the 9S12X Development Kit.....	71
Operation	71
HD44780 Instructions.....	73
LCD Controller Initialization	74
LCD_Init.....	75
LCD_Ctrl.....	78
LCD_Busy.....	78
LCD_Char	79
LCD_String	79
LCD_Addr.....	80
LCD_Pos	80
Character Generation.....	81
LCD_CharGen Example.....	83
LCD_CharGen8 Example.....	84
ASCII Code Manipulation.....	85
ASCII Table.....	85
Upper and Lower Case ASCII Codes	87
Hexadecimal to ASCII conversion	87
The Serial Communications Interface.....	88
Initializing the Serial Communications Interface	91
SCI0 Library	95
Communicating through the Serial Communications Interface.....	96
Terminal Emulation.....	97
SCI0_TxString.....	100
The VT100/VT52 Terminal.....	102
Escape Sequences.....	102
Floating-Point Math in ANSI C.....	105
<stdio.h>.....	106
<math.h>.....	107
Interrupts.....	108

Interrupts in S12XCPU Assembly Language	108
Interrupts using ANSI C	112
Input-Driven Interrupt	114
Accurate Timing	115
Periodic Interrupt Timer (PIT).....	115
Enhanced Capture Timer	117
Timer Initialization	119
Setting the Timer Compare Event Duration	121
Delays vs. Intervals.....	122
Delay Function for Misc_Lib	122
Interrupt-Driven Timer	123
Real-Time Loop.....	126
Input Capture and Pulse Accumulation	128
Input Capture	128
Pulse Accumulation	130
A To D Conversion	132
Setting up V_{RH}	133
Configuring ATD0	134
Using ATD0	137
Pulse-Width Modulation	138
Generating Waveforms.....	139
True Pulse-Width Modulation	144
I ² C Bus	147
Basic I ² C Communication Using the 9S12X	149
LTC2633HZ12 I ² C DAC – 16-bit Data Writes	152
MPL3115A2: Standard 8-bit Reads and Writes	157
M41T81 Real-Time Clock – Standard 8-bit Reads and Writes	160
Position Information with the LSM303DLHC – Standard 8-bit Reads and Writes.....	163
3-Axis Accelerometer.....	163
3-Axis Magnetometer and Temperature Sensor	168
Device with 16-bit Internal Addresses (e.g. EEPROM) – Write and Read Functions.....	171
I ² C Reliability Measures.....	172
Parting Words.....	172

Topic 1 - Embedded Systems Theory and the 9S12X Device

Required supporting materials

- This Module and any supplementary material provided by the instructor
- Device documentation provided in the appendix of this CoursePack
- CNT MC9S12XDP512 Development Kit and 12 VDC Power Adapter
- USBDM Pod or BDM Pod and "A to B" USB Cable
- CodeWarrior

Rationale

Embedded microcontrollers are at the heart of much of modern technology, ranging from automobiles to phones to appliances. An understanding of, and ability to manipulate, these devices is of paramount importance to the Computer Engineering Technologist.

Expected Outcomes

The following course outcome will be partially addressed by this module:

Outcome #1: Develop and debug assembly language programs using an Integrated Development Environment (IDE).

Outcome #2: Create assembly language programs that manipulate data using operations and expressions.

As this course progresses, you will refine the basic skills and understanding of embedded systems and assembly language programming you learn as you complete this topic.

Where are you at?

In most automobiles today, there's at least one "computer module", controlling the door locks, brakes, ignition, fuel injection, lights, and engine monitoring, just to name a few of the diverse applications of the microcontrollers in the system.

When using your computer, if you hit "print", you expect to get ink on a sheet of paper, following a pattern you see on-screen. In order for that to happen, though, at least one microcontroller in your printer kicks into action, activating motors, solenoids, relays, LEDs, and probably an LCD display, all the while monitoring a set of switches on the front panel in case you decide to pause or cancel the print job, along with a bunch of switches and sensors that check for the presence of paper, a paper jam, or an empty ink cartridge. The microcontroller also communicates with your computer, providing status messages or alarms.

These are just a couple of examples of embedded microcontrollers at work, doing the background work we rarely think about, until something goes wrong. Sometimes, even if something does go wrong, the microcontroller might recover before you even notice.

Wouldn't you like to be in control of a device capable of such a diverse array of abilities?

Embedded Controllers

When it comes to working with a microcontroller like the 9S12X, neatly dividing up what you need to know into discrete packages is nearly impossible: In order to interface with peripherals, you need to know how to write programs in S12XCPU Assembly Language and/or a higher level language like ANSI C, how to address registers and ports, how to do bit-wise masking, how to get around the Integrated Development Environment (IDE), how to debug a program, and so on. Consequently, this CoursePack will not be divided into nicely packaged "Objectives" that cover one concept each. Instead, you will be introduced to the main outcomes for a particular module, and will be taught whatever else you need in order to master these outcomes.

What's the difference between a microprocessor and a microcontroller?

A **microprocessor** is a device that can be programmed to perform computational or decision-making tasks following instructions found in program memory, as it manipulates addressed locations in storage memory. Although these storage memory locations may actually be digital logic interfaces (for example, a bank of switches for input or an array of LEDs for output), the microprocessor treats all addressed locations as memory.

A **microcontroller** consists of a microprocessor embedded within a collection of peripheral modules, each designed to carry out specific tasks under the control of the embedded microprocessor. The microprocessor-to-peripheral interface is designed to operate "seamlessly" – all controls and handshaking are managed internally, providing the user with a greatly-simplified task when it comes to programming (although you may not feel that way initially – if you doubt this, try getting a microprocessor like the MC6809 to talk to a Comm port, as compared to asking your 9S12X to use its built-in SCI Port!)

The MC9S12XDP512 Microcontroller

In the "Data Sheet" for the MC9S12XDP512, you will find a block diagram of the microcontroller you will be working with on page 35. This is a huge document that you will occasionally need to access. There's no need to have a paper copy of this (it's over 1300 pages long!), but make sure you can access it. The link below is in Moodle, too.

http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

The block diagram, partially annotated, is shown on the following page. The parts that are labelled are of interest to us in this course. You may, for your own work, find that you can use other modules that aren't covered here, such as the Serial Peripheral Interfaces (SPI) used for talking to a number of commercially-available devices, or the Controller Area Network Buses (CAN Bus) used as the standard communication interface between electronic devices in automobiles and other vehicles.

One thing that should stand out to you when looking at this block diagram is that the microprocessor is deeply embedded in this device, surrounded by a wide range of peripherals, interfaces, and ports that are under its control – hence the term "microcontroller".

Chapter 1 Device Overview MC9S12XD-Family

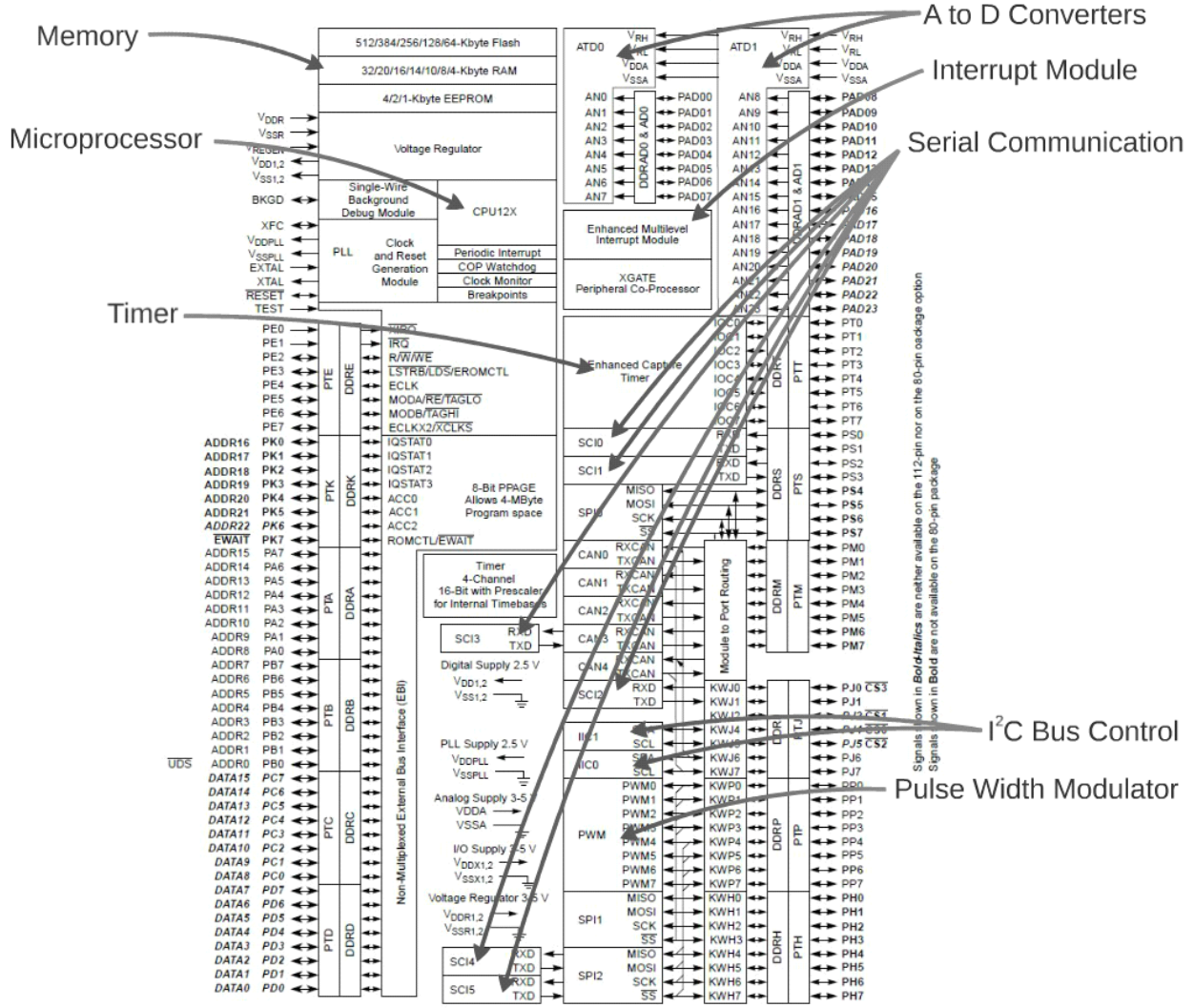
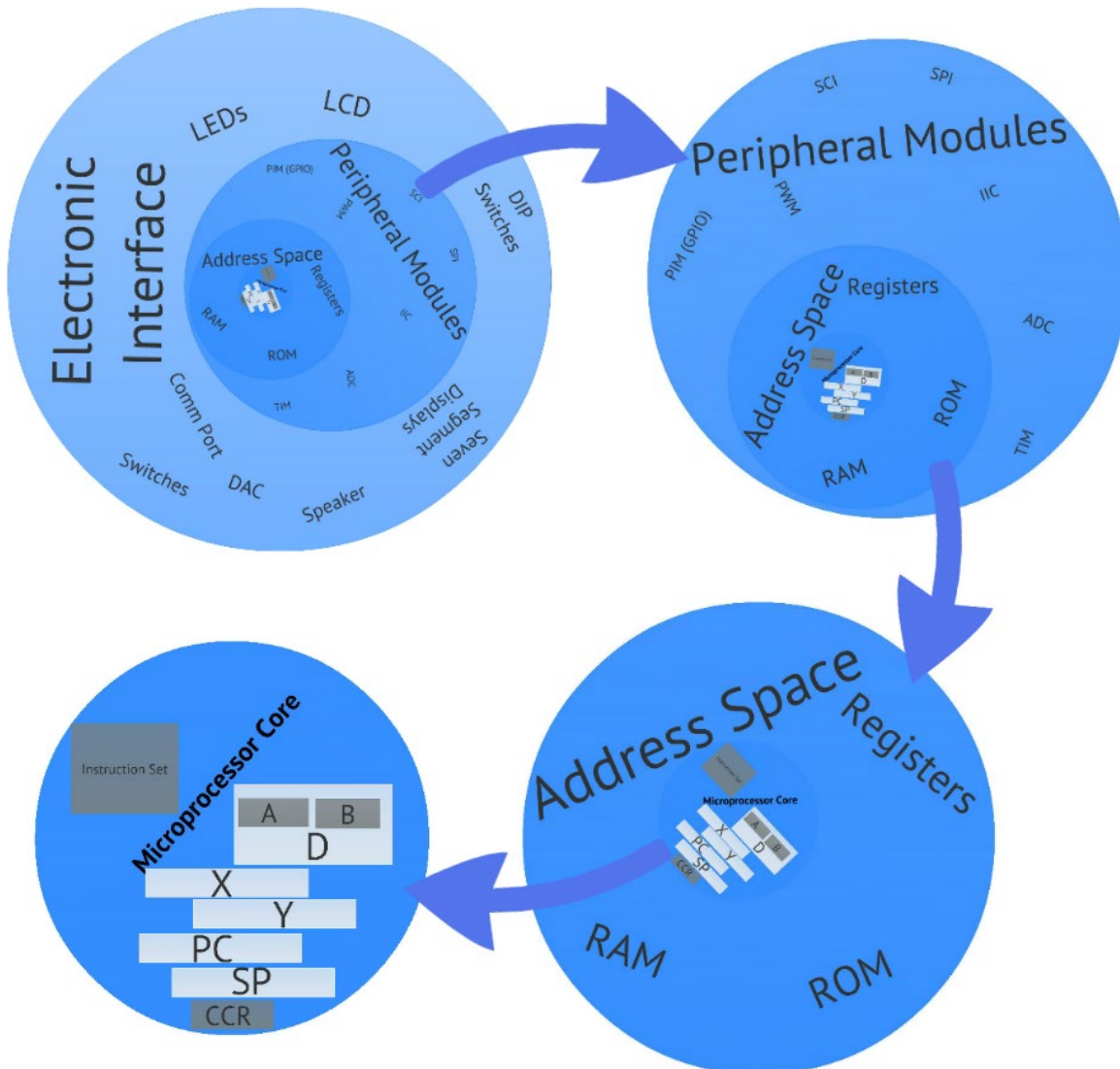


Figure 1-1. MC9S12XD-Family Block Diagram

MC9S12XDP512 Data Sheet, Rev. 2.21

The microcontroller kit used in this course goes one step further: we now embed the 9S12XDP512 microcontroller into an electronic interface with external peripherals also under its control. Again, in the following image, you can see that the microprocessor core ends up being a pretty small, but central, part of the hardware used in this course.



Let's work backwards through the diagram above.

The **microprocessor core** is made up of a group of registers that you should have become acquainted with in an previous course (A, B, D, X, Y, PC, SP, and CCR), along with the actual logic unit and an instruction set. The logic unit acts according to the instructions in the instruction set, as presented to it in a program written by you and stored in ROM, and it does all of its work using the registers.

The core operates within the **address space**. This is where it gets its instructions from (in ROM). While it operates, it may read from RAM or ROM, and it may write to RAM. More importantly, though, it reads from and writes to a set of registers that are directly connected to the microcontroller's peripherals.

The **peripheral modules** are the interface with the “outside world”. We’ve identified a number of these previously. Between the modules and the pins in the following diagram is the “PIM” – Port Integration Module – which allows us either to connect to the peripheral or to use the pins as “GPIO”. GPIO is “General Purpose Input/Output”, and refers to pins available on the IC that can be used individually or in groups as digital inputs and outputs, under the programmer’s control. On this microcontroller, as with most, almost all of the pins associated with other peripherals can, instead, be used as GPIO.

The biggest circle represents the **electronic interface**, which contains all the components on the printed circuit board. Most of the external peripherals on the printed circuit board are accessed using GPIO, although the speaker is intended for operation using the pulse-width modulator module (PWM), the Comm port is controlled by one of the Serial Communication Interface (SCI) modules, and the DAC is accessed using one of the Inter-Integrated Circuit (I²C) busses. Here’s how our electronic interface is wired to the microcontroller in the kit designed for this course.

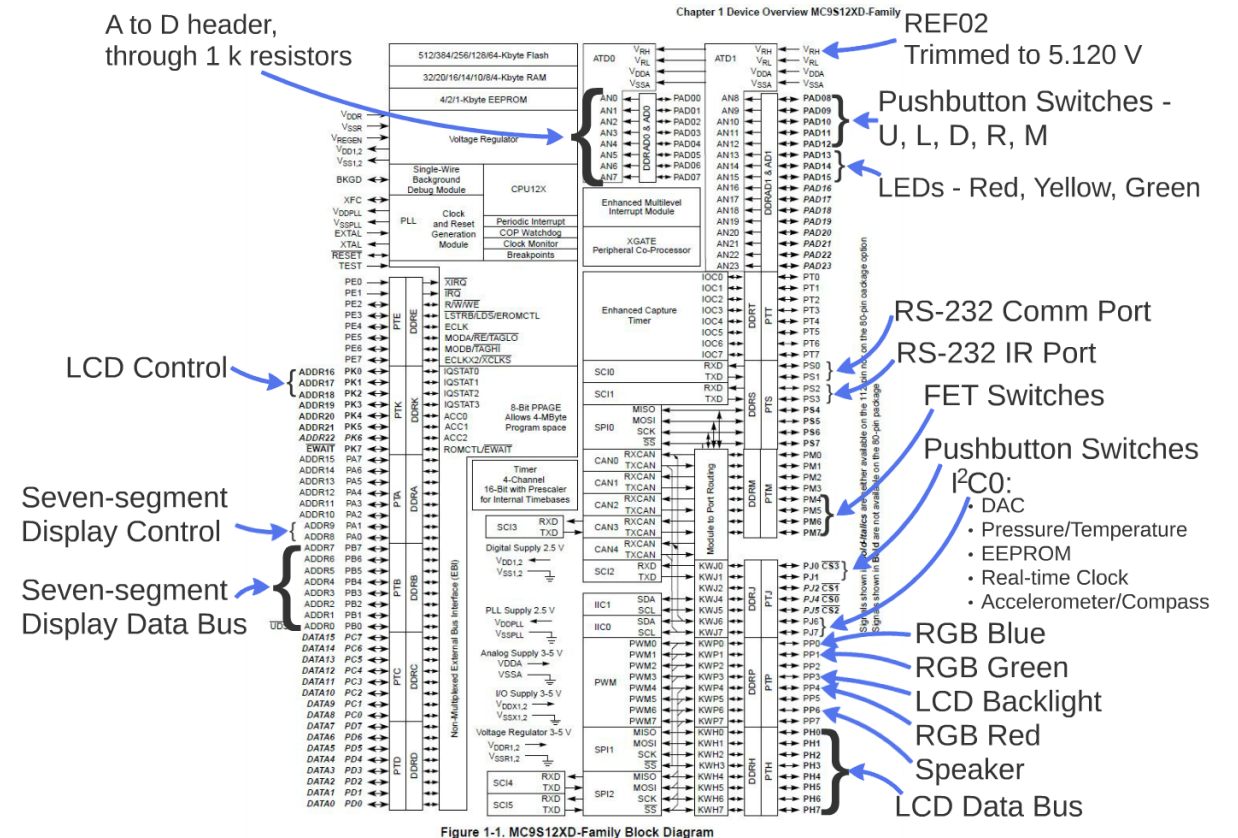
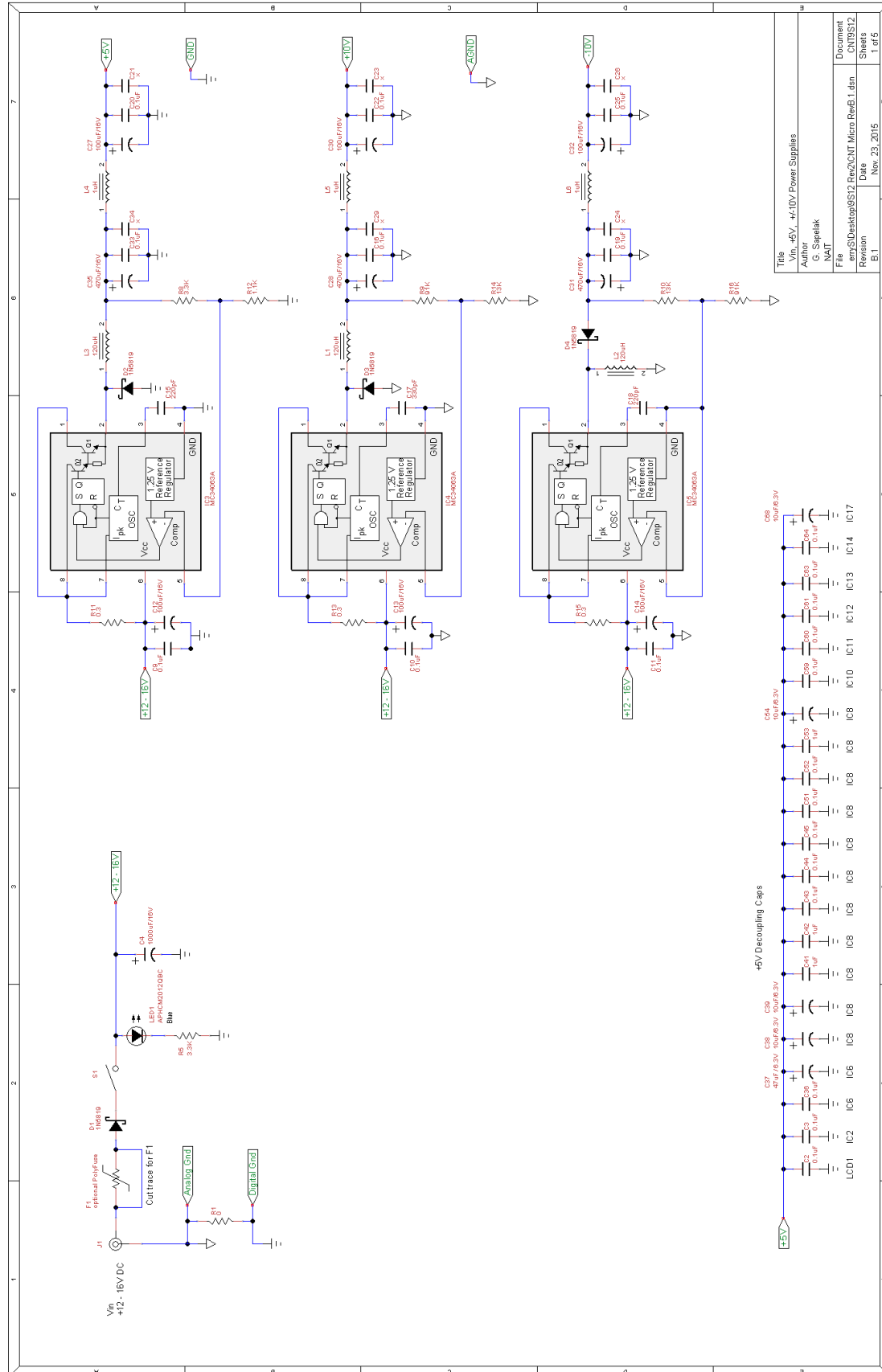


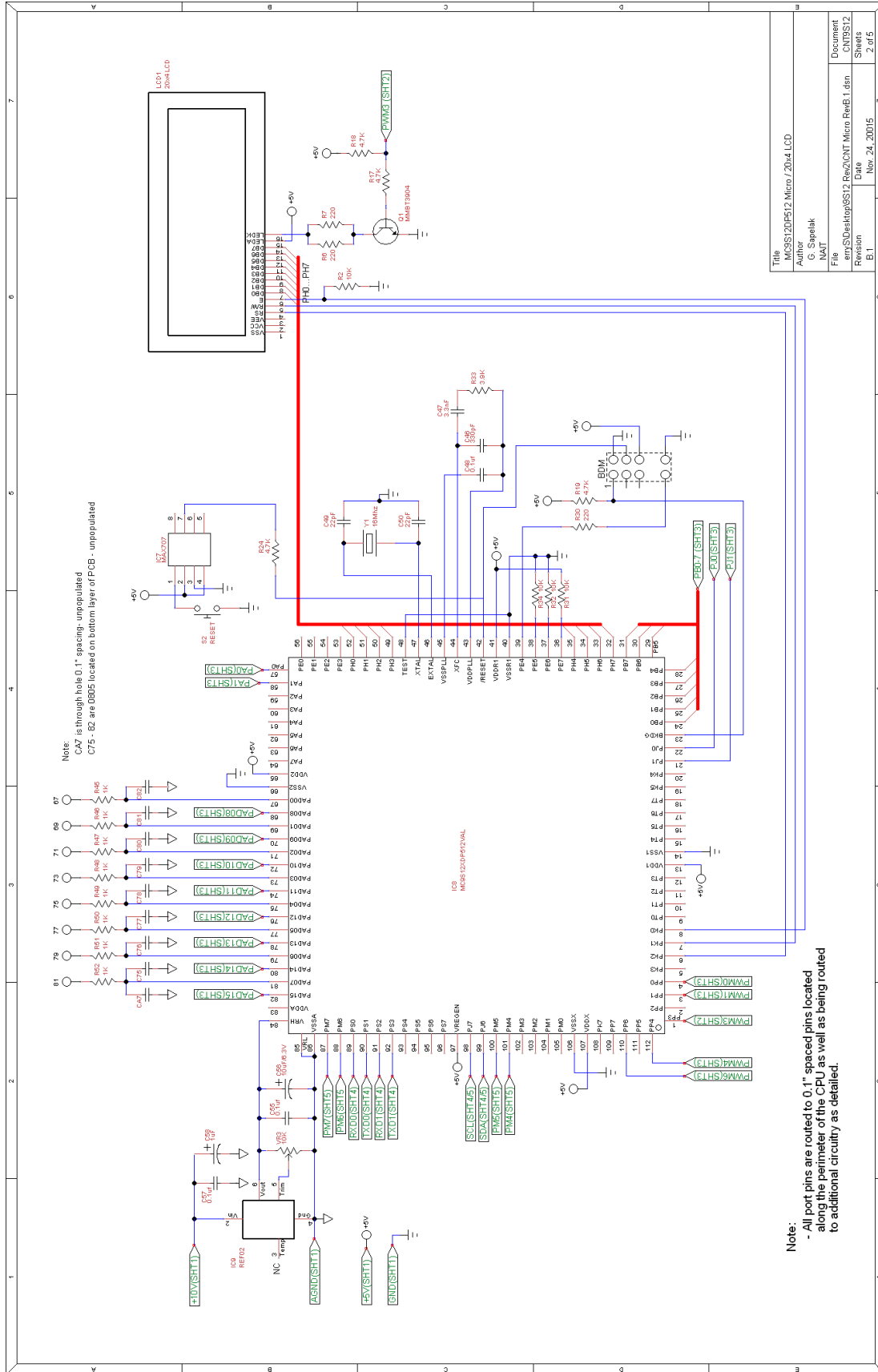
Figure 1-1. MC9S12XD-Family Block Diagram

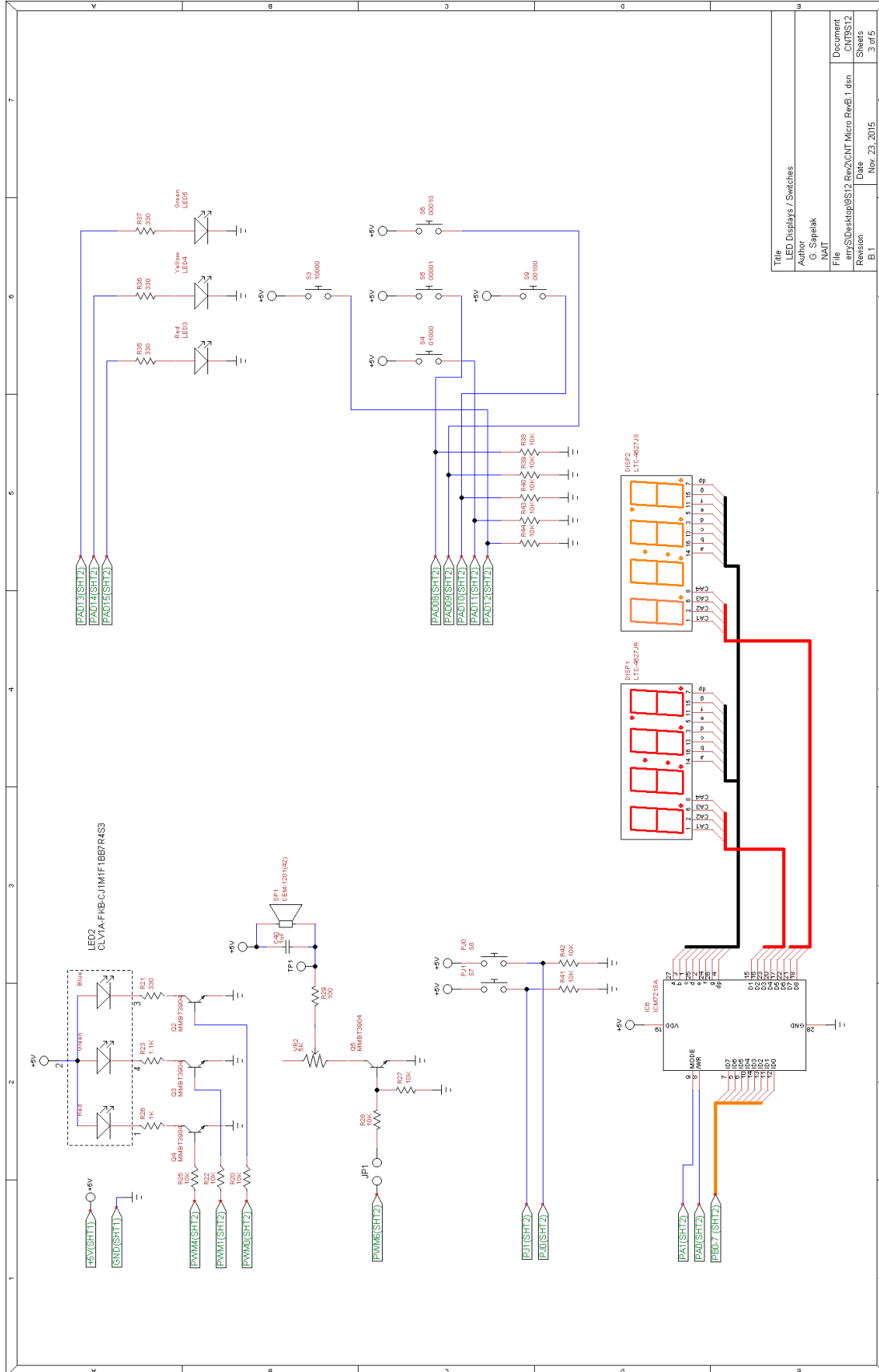
MC9S12XDP512 Data Sheet, Rev. 2.21

The CNT MC9S12XDP512 I/O Board

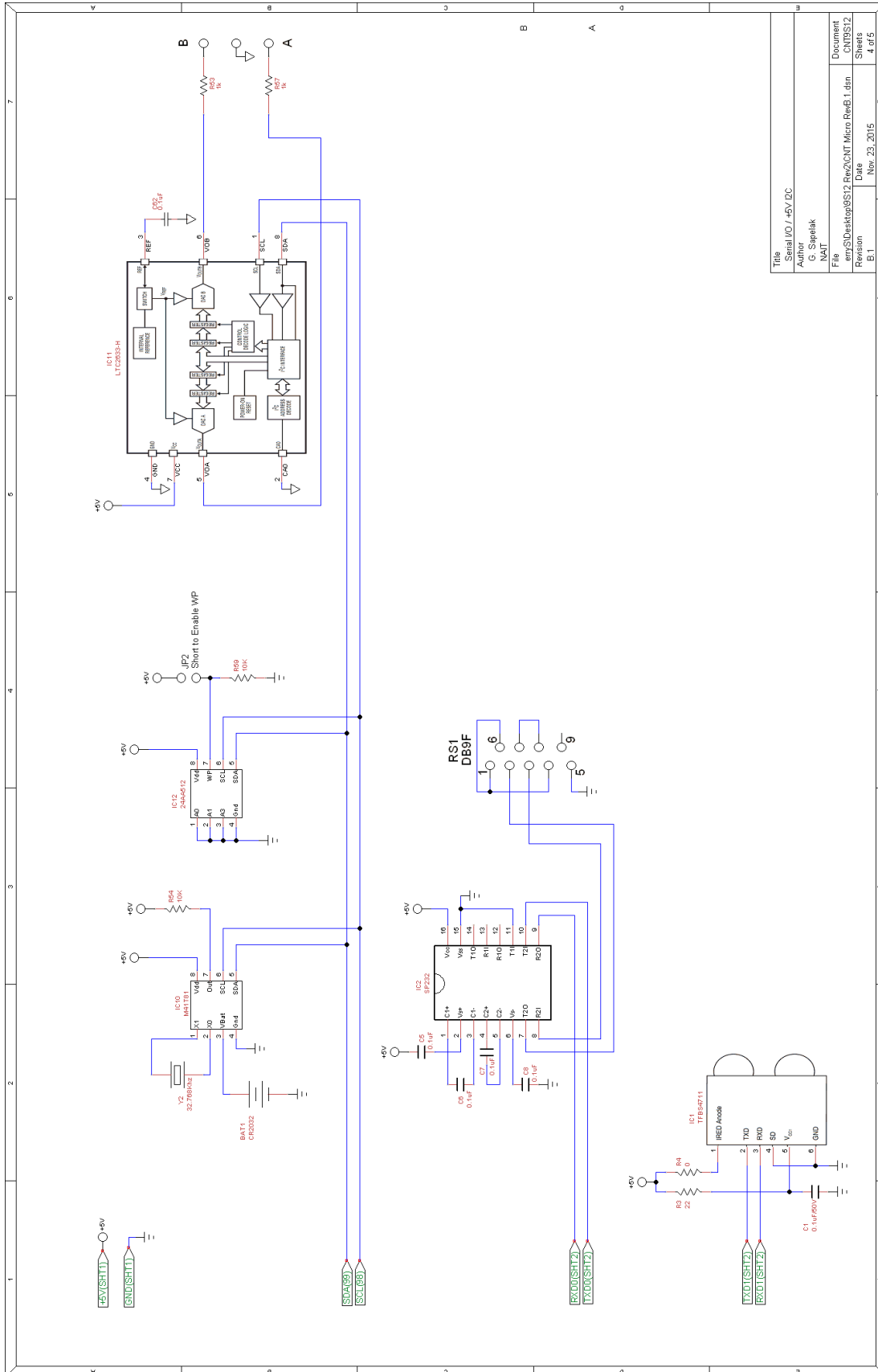
On the pages following, you’ll find the schematics for the microcontroller kit, showing how these connections are actually wired up to provide us with the electronic interface shown in the previous bubble diagrams and block diagrams. Higher-resolution versions should be available in Moodle or from your instructor.



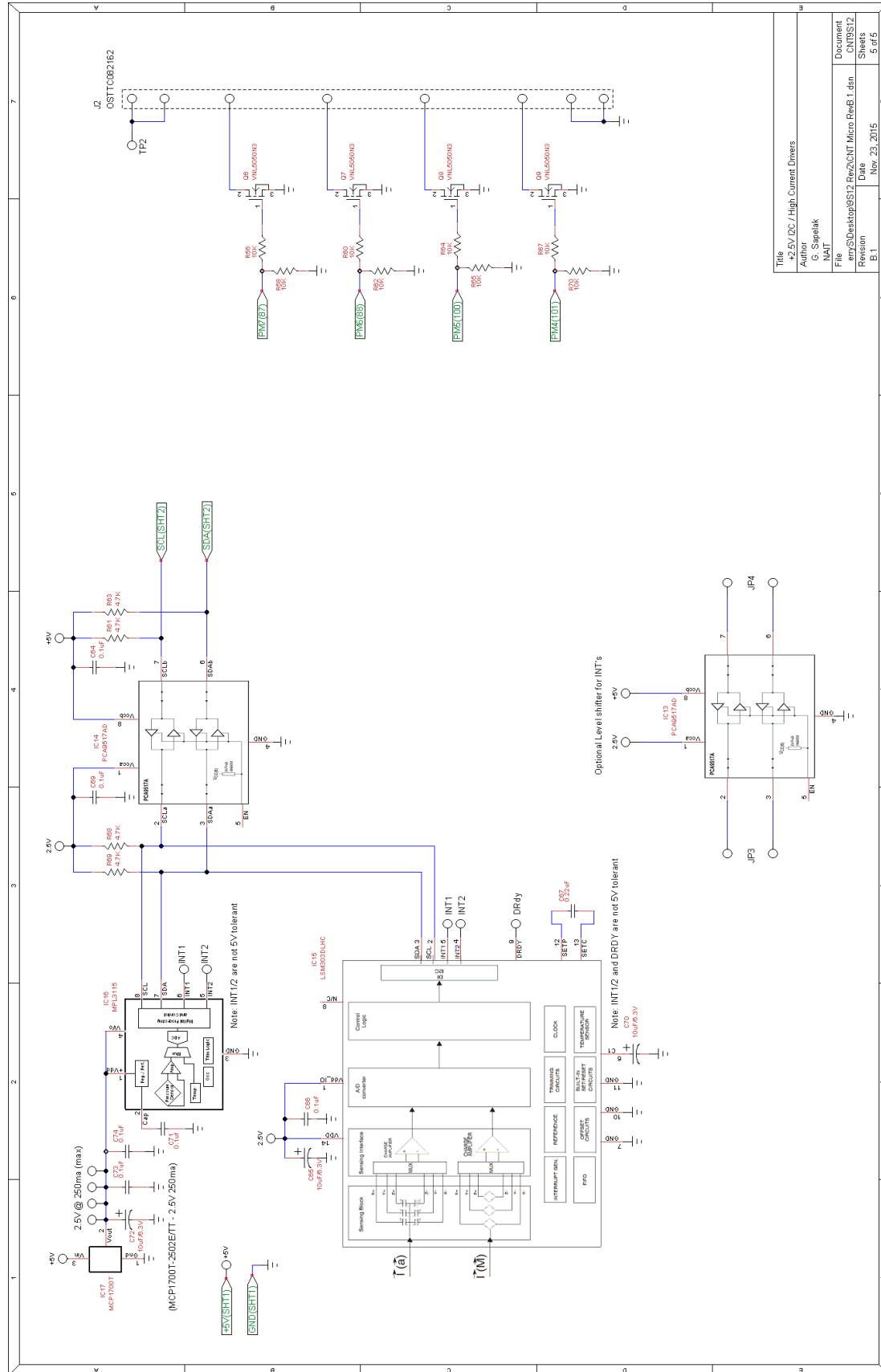




Title	LED Displays / Switches
Author	Chapalak
File	MAE1
Document	CNTSS12
Revision	Rev2/CNT Micro RevB.1.dsn
Date	Nov. 23, 2015
Revision	B.1
Sheets	3 of 5



Title	Serial I/O / +5V I/O
Author	G. Sapek
File	emv/ST/Desktop95512_Res2/CNT_Micro_Resp1.dsn
Revision	B.1
Date	Nov_23_2015
Document	CNT9512
Sheets	4 of 5



Title	U-2.5V/CC / High Current Drivers
Author	G. Sapek
File	emv5\DeskTop9512_Res2\CNT_Micro_Resp1.dsn
Revision	B.1
Date	Nov_23_2015
Document	CNT9512
Sheets	5 of 5

Types of Interfaces

Since a microcontroller can be embedded in a wide variety of systems, there will, of necessity, be different types of interfaces required. The following are the main types of interfaces.

General Purpose Input/Output (GPIO) – GPIO interfacing simply provides or expects logic levels at pins connected to the microcontroller. Conditions in the connected device are read into one or more GPIO pins configured as inputs, and control signals are driven out of one or more GPIO pins, configured as outputs. On the 9S12X, as you have seen, most of the interface pins can be programmed independently to act as GPIO. A logic HIGH or “1” on our device is +5 V, a logic LOW or “0” is 0 V.

We will use GPIO to interface to things like the switches and LEDs on our board.

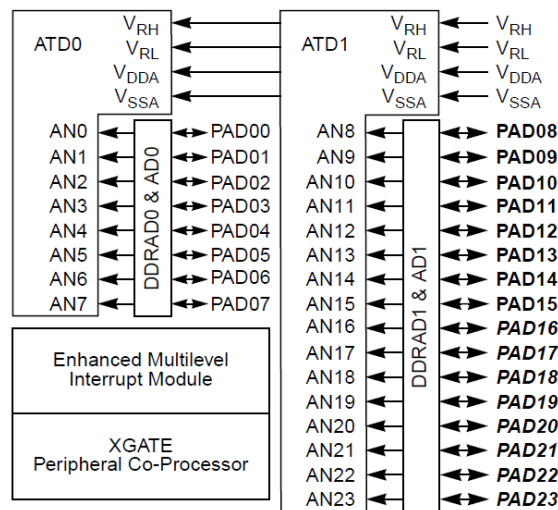
Bussed (Parallel) Interface – A parallel interface involves the simultaneous transfer of multiple bits of information on separate (parallel) copper traces. The microprocessor in a personal computer operates in bussed mode. This requires an address bus capable of locating each unique address in the address space (for a 32-bit address bus, this would be approximately 4.3 billion possible locations). It also requires a data bus capable of delivering all the bits required by that location in a single operation, each on a separate data line (sixteen for a 16-bit data bus). In addition, there will be control lines such as Read/Write, Enable, and Strobe that establish correct communication between the microprocessor and the peripheral. On the block diagram, you can see that PTA and PTB can be used to establish a bussed interface. This would be useful in an application involving a parallel device or where more memory is required than what is available inside the microcontroller (which won't be a problem for us in this course). Most microcontrollers do most of their bus-work internally, taking away the complexity of design and programming. The 9S12X has an internal bus to interface with its memory modules and all of the devices within which it is embedded. All we need to know is the addresses associated with the device we want to talk to, what needs to be communicated, and the speed at which communication takes place, which is based on the internal bus clock or system clock. (For our board, this is half of the 16.000 MHz crystal speed, or 8.000 MHz.)

In this course, we use GPIO to create simpler parallel interfaces to two of the devices on board: we use a simple one-way 8-bit data bus with control lines to control a 7-segment display controller; and we use a two-way 8-bit data bus with control lines to communicate with a second microcontroller embedded in our LCD display.

Serial Communication – Rather than sending all bits simultaneously on separate parallel lines, it is possible to send bits one after the other (sequentially) on a single transmission line (with a current return to complete the circuit). In a system like RS-232 (used by us to communicate with the Comm Port of a computer using an SCI module of the micro, or for communicating using a Bluetooth adapter), separate transmission lines are used for transmitting and receiving. In a system like USB 2.0 or USB 3.0 (used by us to establish a programming link between the computer and our board through the BDM Pod), a single pair of conductors is used for communication in both directions. Serial communication requires protocols establishing voltage levels, timing parameters, and “handshaking” to ensure that data is actually delivered and received.

Port Addressing

Let's focus on one set of port pins shown in the top right corner of the block diagram: PortAD.



The I/O connections shown, PAD00 through PAD23 can either be connected to the A to D converters (ATD0 and/or ATD1) or they can be redirected using the PIM blocks shown as "DDRAD0&AD0" and "DDRAD1&AD1". To begin with, we will be using a subset of these pins as GPIO, because they are connected to three LEDs and five switches on the board (more on that later).

Notice the different typefaces, fitting into the sidebar for the block diagram. PAD00 to PAD07 are in regular type, which means they are available for all flavours of the 9S12XDP512. **PAD08 to PAD15** are in boldface, meaning they are not available on the smaller 80-pin version of the IC. ***PAD16 to PAD23*** are bold italics, meaning they are not available on either the 80-pin version or the 112-pin version, which is installed on our board. Thus, we have access to PAD00 to PAD07, which are connected to ATD0, and PAD08 to PAD15, which are connected to ATD1.

In the Data Sheet, "Chapter 4: Analog-to-Digital Converter (ATD10B16CV4) Block Description" starting on page 125 describes the full functionality of ATD1, and "Chapter 5: Analog-to-Digital Converter (S12ATD108CV2)" starting on page 159 describes ATD0. Clearly, there's a lot of information required to fully implement this corner of the diagram!

The figure below shows ATD1, and there's a similar picture later on that shows ATD0. I've included this figure simply to show one part of the PIM for this set of pins that doesn't appear on the main block diagram: ATDDIEN.

ATDDIEN is used to enable the connection between the associated pins and the digital module. "DIEN" stands for **digital input enable**. This needs to be turned off for A to D functionality, but turned on for GPIO activity for any pin that's used for input.

Chapter 4 Analog-to-Digital Converter (ATD10B16CV4) Block Description

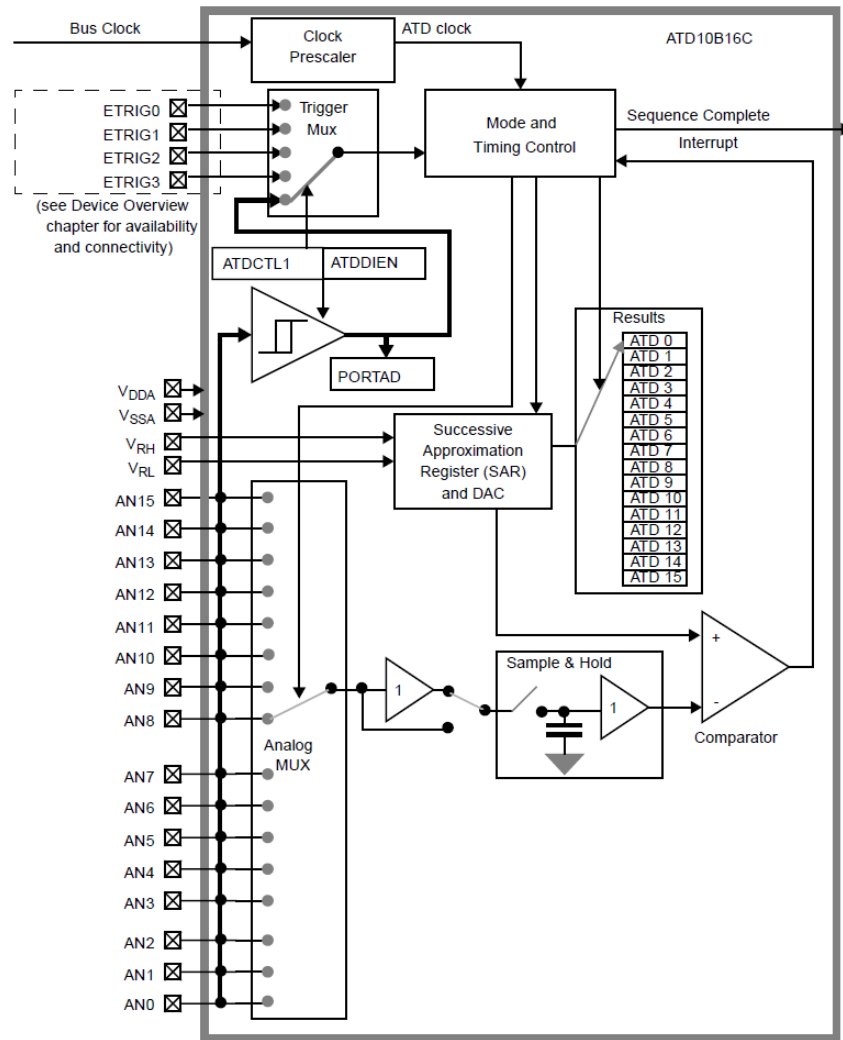


Figure 4-1. ATD10B16C Block Diagram

MC9S12XDP512 Data Sheet, Rev. 2.21

126

Freescale Semiconductor

One other interesting thing to learn from this diagram is that the labels used in the modules don't necessarily correlate to the ones for the whole IC: AN0 to AN15 for the module is actually AN8 to AN23 for the microcontroller!

The ports and control registers are all accessed by the microprocessor by means of unique addresses. For example, the 16 bits we can access of the 24-bit port labelled PAD occupy the addresses 0271_{16} and 0279_{16} ; the two 8-bit Data Direction registers for the accessible parts of this port are at addresses 0273_{16} and $027B_{16}$. The ATDDIEN (digital input enable) registers for the accessible parts are found at addresses $02CD_{16}$ and $008D_{16}$.

Trying to remember all these addresses, and all the rest of the register addresses we'll be using, would be a daunting task. To help with this, the developers at Freescale have

created an "include" file called *mc9s12xdp512.inc* that we'll use when we program in S12XCPU Assembly Language, and a corresponding *mc9s12xdp512.h* file we'll use when we program in ANSI C. This file assigns labels to all the ports (and even to masks for each pin!). These labels are easier to remember than the hex addresses; just remember that the labels represent the actual addresses, but only if the "include" file is actually included.

Here's a little table that summarizes what we've been saying about Port PAD. Notice that the addresses start with "\$", which, in S12XCPU Assembly Language, means hexadecimal. We'll use this notation to indicate numbers in hexadecimal instead of using $nnnn_{16}$, until we start programming in C – at which point we'll revert to the *0xnnnn* format you're used to.

Port (Block Diagram)	Port Name (mc9s12xdp512.inc)	Addresses	Function
AD0	PT1AD0	\$0271	Lowest 8 bits
AD1	PT01AD1	\$0278	Upper 16 bits
	<i>PT0AD1</i>	<i>\$0278</i>	<i>Highest 8 bits (n/a)</i>
	<i>PT1AD1</i>	<i>\$0279</i>	<i>Middle 8 bits</i>
DDRAD0	DDR1AD0	\$0273	Lowest 8 Data Direction
DDRAD1	DDR01AD1	\$027A	Upper 16 Data Direction
	<i>DDR0AD1</i>	<i>\$027A</i>	<i>Highest 8 DDR (n/a)</i>
	<i>DDR1AD1</i>	<i>\$027B</i>	<i>Middle 8 DDR</i>
ATDDIEN	ATD0DIEN	\$02CD	Lowest 8 Input Enable
ATDDIEN	ATD1DIEN	\$008C	Upper 16 Input Enable
	<i>ATD1DIEN0</i>	<i>\$008C</i>	<i>Highest 8 IE (n/a)</i>
	<i>ATD1DIEN1</i>	<i>\$008D</i>	<i>Middle 8 IE</i>

Switches and LEDs

To access the push-button switches and LEDs on the board, you need to know the following:

Port or Register	Address	Notes
PT1AD1	\$0279	Order: RYG ULDRM
DDR1AD1	\$027B	HIGH = Out, LOW = In
ATD1DIEN1	\$008D	HIGH = Input Enabled

The code snippet below shows what needs to be done to appropriately activate the part of PAD that's connected to the switches and LEDs, written as a subroutine:

```

;*****
;* SW_LED_Init:                                     *
;*                                                 *
;* Registers affected: none                         *
;* sets LEDs as outputs, switches as inputs, sets outputs to zero *
;*                                                 *
;*****
SW_LED_Init:
    CLR     PT1AD1                                ;make sure LEDs are off (no effect on switches)
    MOVB   #%11100000,DDR1AD1                    ;LEDs as outputs, switches as inputs
    MOVB   #%00011111,ATD1DIEN1                 ;enable switch inputs
    RTS

```

Notice that, since we want to directly manipulate the conditions of all eight bits in the three registers, we use "MOVB" instead of "BSET" and "BCLR". BSET and BCLR only affect the bits indicated in a *bit mask*, leaving the other bits unchanged.

Also, notice the use of "#", which tells the assembler to move the byte indicated into the associated register (immediate addressing mode).

Another thing to notice is that, before we turn on the output pins in the Data Direction Register, we initialize their values to prevent an unwanted condition when the pins become enabled as outputs. This is a wise thing to do whenever you control any port intended to be used as outputs. In the boot condition, all ports default to inputs, and often the default value for each pin is SET to 1. In the case of the LEDs and many other attached circuits, we don't want the LEDs or other circuitry to be initially on, even for a split second. (Imagine if the connected circuitry was the detonator for a rocket or explosive, or perhaps control for the two transistors in a CMOS motor controller!) If you watch the memory window as you step through the code above, you'll notice that the value written to PT1AD1 doesn't appear until after DDR1AD1 is changed – the microcontroller holds the value previously written to PT1AD1 in a buffer, and sends it out as soon as the pins are changed to outputs.

As we move through this course, we'll need to access a variety of other peripherals. What you have just learned about the switches and LEDs will serve as a guide to accessing and controlling each of these.

You'll notice that, as a microcontroller programmer, you need to know a lot about the hardware you're working with. That includes the microprocessor at the heart of the microcontroller, the built-in peripherals, and the electronic interface connected to the microcontroller. This information is typically available in Data Sheets and Schematic Diagrams.

Due to time limitations in this course, you will typically be directed to the appropriate information in these supporting documents. However, in real life, you will need to develop the skills required to find, interpret, and apply the necessary information.

Each microcontroller application will be a stand-alone system, typically different from any other system in the world. Searching the Internet will very likely not produce the answers you're looking for: you're on your own! Looking at this a different way, you are the one in control of the system you're designing, and that can be a very empowering experience.

S12XCPU Assembly Language and the S12XCPU Microprocessor Core

In a previous course, you should have been introduced to the S12XCPU Core and to programming in S12XCPU Assembly Language. The following will be a short refresher.

Most of the pertinent information you'll need for programming is in the "S12XCPUV1 Reference Manual", which is for the S12XCPU microprocessor that's at the core of our MC9S12XDP512 microcontroller. You should download this document. Here's the link to access it:

http://cache.freescale.com/files/microcontrollers/doc/ref_manual/S12XCPUV1.pdf

It is also available at the Moodle site for this course. The document is over 500 pages long, but if you want a ready reference in paper form, print Appendix A (Instruction Reference). It contains a summary of the various instructions and of the support documentation to help you use them.

The following is a clip from the first page of the appendix.

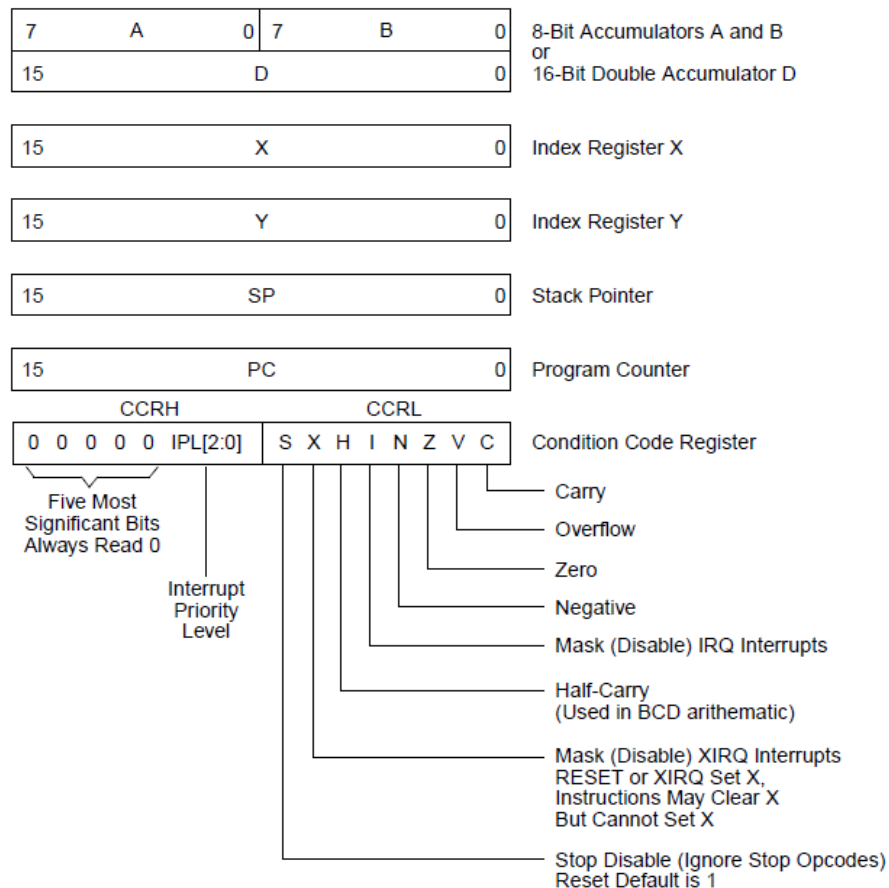


Figure A-1. Programming Model

The S12XCPU microprocessor is a complex array of digital logic gates, arranged to follow algorithms hard-coded into the logic. These algorithms need something to work on – that's where the "accumulators" and "registers" come in.

Although the microprocessor can perform some actions directly on memory locations (Direct Memory Access or DMA) and performs other actions internally ("inherent" commands), the vast majority of actions are performed on values loaded into the accumulators and registers in the device itself. The following is a brief description of each of these.

Accumulators and Registers

Register is the general name for a latch or buffer that holds a set of bits for the microprocessor or circuitry.

An **accumulator** is a special type of register that can be manipulated in a wide variety of ways. We can use accumulators for adding, subtracting, performing bit-wise Boolean logic operations such as ANDing, ORing, Exclusive ORing, complementing, counting up, counting down, shifting bits left or right with a variety of options, etc.

An **index register** is a more limited register that can primarily be used for counting up or counting down, and is called an "index" register because it can be used to locate addresses in memory as referenced to a specific starting location.

The **A and B accumulators** are the work-horses of our microprocessor. This microprocessor has a slight "personality disorder": it's not sure if it's 8-bit or 16-bit, so there are commands that work on 8-bit data (bytes) and commands that work on 16-bit data (words). To accommodate this, the eight-bit A and B accumulators can be combined into the 16-bit (i.e. "double") **D accumulator**. Please do not think of the D accumulator as being separate from the A and B accumulators! Anything you do to the D accumulator affects the A and B accumulators, and anything you do to the A or B accumulators affects the D accumulator. This is a limitation, but it also allows for creative manipulation of the parts of the D accumulator, a feature you can definitely use to your advantage.

The **X and Y registers** are 16-bit index registers. We'll frequently use them to point to locations in memory. They *must* be loaded with 16-bit data or addresses. You will be tempted to load them with 8-bit data, but the results will be highly unsatisfactory – you will get 16 bits, but the other 8 bits will come from a location adjacent to the one you are interested in, and will end up in the lower half of the register, which is probably where you wanted your data to be. In the S12XCPU microprocessor core, the X and Y index registers are capable of a number of operations that were not available in earlier versions, making them much more versatile.

During the running of a program, the **Program Counter Register** is constantly updated to keep the microprocessor moving through the program. Branches and jumps change the contents of the program counter so that it carries on from a newly-determined location.

The **Stack Point Register** is used something like an electronic "scratch pad" by the microprocessor. During operation, the microprocessor may temporarily store things like the contents of the various registers in a special location called the **Stack**. Each newly stored item is placed on the stack in the next available location (actually, the address just below the last one used, since the stack is designed to grow backwards from an endpoint), and the stack point register is adjusted to point to this new location. This is really useful when it comes to calling subroutines or responding to interrupt routines, as the microprocessor can quickly put the current conditions onto the stack, go off to perform a new function, then, by

“unstacking” in reverse order, can return to exactly the conditions that existed previously and continue on as if nothing had happened. We can, and will, deliberately place things on the stack – just remember to take them back off the stack, and in reverse order, or you will quickly fill up the stack to a point where it interferes with memory you’re using for other operations. This is called “stack overflow”, and usually results in very bizarre activity or a total crash.

The **Condition Code Register**, or **CCR**, continuously reports back on events that occur while the microprocessor is executing code. There are eight bits (flags) in the lower byte of the CCR (CCRL), which is the part we’re interested in:

- S, X, and I – these are bits we can deliberately manipulate to control the operation of the microprocessor. S allows the microprocessor to ignore or respond to “stop” commands in the program, X allows it to ignore or respond to certain interrupt requests, and I allows it to ignore or respond to a different set of interrupt requests. Interrupts will be discussed *much* later.
- C and V – the Carry and Overflow bits provide us with information in the event that some operation has produced a result that’s too big for the accumulator we’re working with. For example, if we end up with a result that’s one bit too big (like trying to display \$13B in an eight-bit register), the Carry flag will be set and the register will contain just the part that fits in the register (\$3B in the previous example). The Overflow bit indicates that a mathematical process involving signed numbers has produced a result where the sign bit is probably incorrect. For example, $100 + 100$ should be +200, but in binary this is $01100100 + 01100100 = 11001000$, which is -56. The Overflow bit would be set in this case.
- H – this is the Half-Carry flag. This flag is set whenever an operation results in a carry out of the *Lower nibble* of the manipulated register (in other words, when the result is greater than \$F). This is, believe it or not, quite a useful feature, particularly when it comes to doing math with Binary Coded Decimal (BCD) values.
- Z – the Zero flag is set when the result of an operation is 0.
- N – the Negative flag is set when the most significant bit in a register is 1, since, in 2’s complement notation, negative numbers always start with 1. This flag will be set even if you aren’t intending a value to be interpreted as a 2’s complement negative.

We can, and will, deliberately manipulate bits in the CCR, but usually we use these flags as set by the microprocessor to help us make decisions during the flow of the program. For example, let’s consider the “DBNE” command. This command means “Decrement, and Branch if Not Equal to zero”. If, during execution of the decrementing stage, the particular register results in a non-zero value, the Z flag will be cleared LOW, and the program counter will be loaded with the address of a new location in the program, to which the operation will now “branch”; if the result is zero, the Z flag will be set HIGH and the program will continue to the next address in sequence. (This is the machine language equivalent of an “IF” statement.) Unfortunately, we can’t observe the action of the Z flag for the DBNE or others of the special compound commands, as the micro restores the original flags before it completes these types of instructions. However, you can observe the flags when using commands such as BCS, BEQ, BNE, BLE, BMI, etc.

In Appendix A of the Reference Manual and other supporting documents, you will find that each operation uniquely controls the bits in the CCR, and we will need to pay attention to the results. Here's an example:

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12X	HCS12		
ABA	(A) + (B) ⇒ A Add Accumulators A and B	INH	18 06	00	00	--Δ-	Δ Δ Δ Δ

Adding Accumulators B and A could change one or more of the five flags: H, N, Z, V and C.

Memory

We also need to know the memory arrangement in the MC9S12XDP512 Microcontroller.

- The available memory on the microcontroller is somewhat limited. The 9S12XDP512, for example, only contains 32 kB of RAM – your PC at home could easily have more than 250 000 times as much RAM.
- The address space on the microcontroller is much smaller. "Address space" is the range of memory locations that a processor can access. The 9S12X has a 16-bit address bus, so it is only capable of accessing 2^{16} (65 536) memory locations directly. This is quite small when compared to a PC that has a 32-bit address bus capable of accessing 2^{32} (4 294 967 296) locations. The MC9S12XDP512 has more memory than can be accessed using the 16 bit address bus, but the extra memory is only accessible using "paging", in which an 8-bit register selects which piece, or page, of memory will be accessed using the address bus.

Consequently, there is only 12 kB of the available 32 kB of RAM directly accessible, and only 48 kB of the available 512 kB of Flash directly accessible. If you are interested, later on in the course or for your capstone project, the "MC9S12XDP512 Data Sheet" provides information for accessing the other pages of memory.

- There is no operating system nor are there hardware abstraction layers on the microcontroller – your code is the only code running on the device.
- The code you write for an embedded system is intended for a specific end-product device, with fixed hardware. This means the code *may* be created around many assumptions, including the assumption that ports not connected to any hardware and internal modules don't need to be addressed or dealt with in any way.

The huge document called the "Data Sheet" for the MC9S12XDP512 device contains memory maps for the device. The following link is in Moodle, as well:

http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

The "Global Memory Map" shown on the following page shows the entire memory space in this microcontroller, much of which is only accessible using paging, as previously described. There is an "EEPROM window", a "4K RAM window", and a "16K FLASH window". These windows are the access points for the memory selected using the paging registers. Notice that the addresses in the "Global Memory Map" are six nibbles long – the first two nibbles come from the paging registers, and the other four nibbles are from the 16-bit address bus.

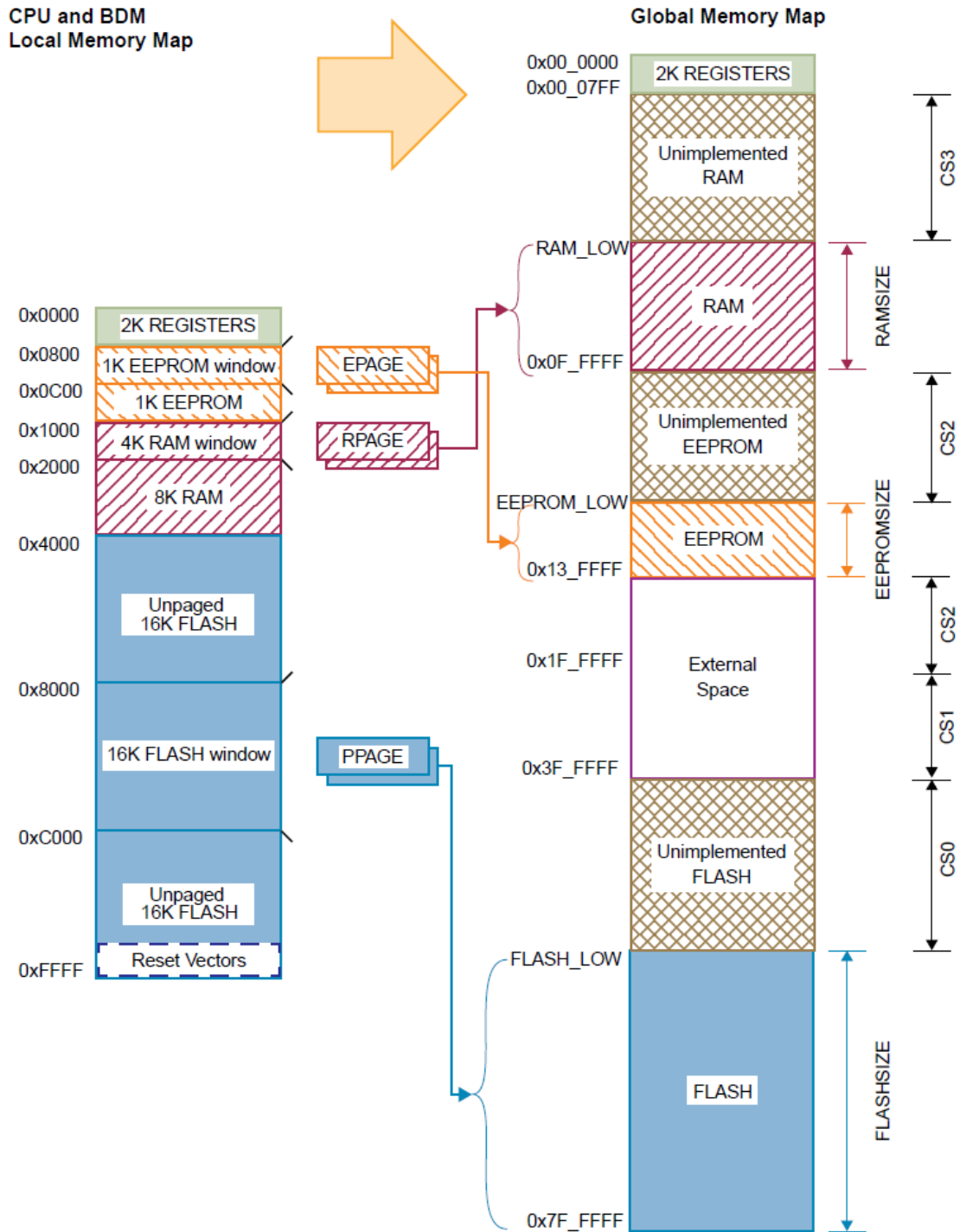


Figure 1-3. S12X CPU & BDM Global Address Mapping

MC9S12XDP512 Data Sheet, Rev. 2.21

Unless you have good reason to do so, you're probably wise to avoid the windowed areas of memory and work with the unpagged regions in the "Local Memory Map", which can be accessed without any extra page manipulation using the 16-bit address bus.

Memory Map

The "Local Memory Map", then, is what we will concentrate on, and contains the following regions of interest (this list doesn't include the paging windows):

- Chip Registers from \$0000 to \$07FF (2K)
- RAM from \$2000 to \$3FFF (8K)
- FLASH from \$4000 to \$7FFF (16K)
- FLASH from \$C000 to \$FEFF (16K minus 256 bytes)
- Vectors from \$FF00 to \$FFFF (256 bytes)

When you write your assembly language programs using the supplied skeleton file, you will put your code in the fixed FLASH block at \$4000, and will place working variables in RAM (\$2000). You will likely put constant data in FLASH at \$C000. The top of the RAM block will be used for stack space (more on that later). This implies that our programs will typically contain less than 16K of code, and less than 8K of combined variable and stack space. For us, this really isn't a limitation at all.

Topic 2 –Microcontroller Programming

Required supporting materials

- This Module and any supplementary material provided by the instructor
- Device documentation provided in the appendix of this CoursePack
- CNT MC9S12XDP512 Development Kit and 12 VDC Power Adapter
- USBDM Pod or BDM Pod and "A to B" USB Cable
- CodeWarrior

Rationale

Well-structured and documented code results in dependable and maintainable systems.

Expected Outcomes

The following course outcome will be addressed by this module:

Outcome #1: Develop and debug assembly language programs using an Integrated Development Environment (IDE).

Outcome #2: Create assembly language programs that manipulate data using operations and expressions.

Outcome #3: Interface with onboard, simple GPIO, and programmable devices.

As this course progresses, you will refine the basic skills and understanding of embedded systems through programming the 9S12X using S12XCPU Assembly Language and ANSI C.

Connection Activity

You have now learned enough about the 9S12X and its Assembly Language to create simple I/O tasks. More complex tasks may require careful pre-planning, more instructions, a clearer understanding of the ways in which address locations are accessed, and a better understanding of the ways in which program flow can be controlled. The more complex the software, the more careful you will need to be in structuring and documenting it. You will discover that certain tasks are used repetitively or have the potential to be used in different software routines – these should be stored in such a way that they can be accessed without needing to copy or re-enter the code. A well-structured program should be easily understood, easily operated, and easily maintained.

Assembly Language Fundamentals

When programming in S12XCPU Assembly Language there are two fundamental types of commands: Assembler Directives and Instructions.

Assembler Directives

Assembler directives are commands that control the development software on our computer, called the Assembler. Assembler Directives do not end up in the code that the microprocessor runs. Some of the more common Assembler Directives are:

- INCLUDE –tells the assembler to add the contents of an external file
- EQU –assigns a label to a particular address
- ORG –tells the assembler to move to an address location before continuing
- DS.B nn –defines storage space for nn Bytes (8 bits), and should be in RAM
- DS.W nn –defines storage space for nn Words (16 bits)
- DS.L nn –defines storage space for nn Longs (32 bits)
- DC.B *val(s)* –defines a constant Byte or Bytes (8 bits) and should be in ROM
- DC.W *val(s)* –defines a constant Word or Words (16 bits)
- DC.L *val(s)* –defines a constant Long or Longs (32 bits)

There are a lot of other Assembler Directives, which can be found in the “S12(X) Assembler Manual” from Freescale. Here’s the link:

http://cache.freescale.com/files/soft_dev_tools/doc/ref_manual/CW_Assembler_HC12_RM.pdf

There’s also a link to this 400 page document in Moodle. It should also be available in the lab in case you need it.

One useful feature of the S12(X) Assembler is its ability to do math on the fly. You can get it to calculate addresses or offsets while it is creating the machine code for the CPU, which can make your life a bit easier.

Instructions

Instructions, unlike assembler directives, are translated into machine language for the microprocessor to carry out.

A summary of the S12XCPU Assembly Language Instruction Set can be found in Appendix A of the Reference Manual, the link to which you’ve been given already. Let’s look at what we can learn about a particular instruction from this guide.

To understand the instruction set, we need to look at the explanatory notes that precede it, on pages 2 – 5 of the Guide. You’ll get to do that in the exercise that follows.

You also need to know a bit more about the terminology used in Assembly Language programming.

Op Code – Short for Operation Code, this refers to something the microprocessor will interpret as an instruction. Each version of each instruction will have a unique op code, which determines what else the microprocessor needs to look at in order to carry out the instruction.

Post Byte – Some op codes tell the microprocessor to read the next byte to get details on the operation to be carried out. In the Instruction Set, this will be indicated in the “Machine Coding” column as “eb”, “lb”, or “xb”, depending on the type of post byte.

Mnemonic – the Assembly Language Mnemonic is the pseudo-English abbreviation that represents a particular op code or set of related op codes and their post-bytes. Programming in Assembly Language involves getting to know the Mnemonics and figuring out what each of the variants for that instruction does and what it needs.

Operand – Some, but not all, instructions require something to work on. This could be actual data, it could be an address containing the necessary data, or it could be an offset from some other point of reference.

Here's an example from the Reference Guide: LDAA. Some pertinent points follow.

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C	
				HCS12X	HCS12			
LDAA #opr8i	(M) ⇒ A	IMM	86 ii	P		P	---	ΔΔ0-
LDAA opr8a	Load Accumulator A	DIR	96 dd	rPF		rPf		
LDAA opr16a		EXT	B6 hh ll	rPO		rPO		
LDAA oprx0_xyvsp		IDX	A6 xb	rPF		rPf		
LDAA oprx9_xyvsp		IDX1	A6 xb ff	rPO		rPO		
LDAA oprx16_xyvsp		IDX2	A6 xb ee ff	frPP		frPP		
LDAA [D_xyvsp]		[D,IDX]	A6 xb	fIfrPF		fIfrPF		
LDAA [oprnx16_xyvsp]		[IDX2]	A6 xb ee ff	fIfrPF		fIfrPF		

- There are eight different ways that "LDAA" can be interpreted by the Assembler. You can think of these as "overloads" in .net terminology.
- The first and simplest of these uses the "IMM" addressing mode. This means that the accumulator will be loaded with the contents of the address directly following the instruction. From the first column, you will notice that this requires a "#" sign in front of the next byte. Since A is an 8-bit register, it can only load 8-bit data.
- The source form "#opr8i" tells you that the instruction is made of a single-byte op-code followed by an 8-bit immediate value for an operand.
- "86 ii" shows that the actual op code is \$86, and "ii" means two nibbles (a byte) immediately following the op code.
- The "EXT" mode is used to access the 8-bit value contained at a particular 16-bit address. "opr16a" means that this version of the command has a single-byte op-code followed by a 16-bit address. "B6 hh ll" tells you that this version of the command has the op-code \$B6, and that the address will be two high nibbles followed by two low nibbles. This, by the way, indicates that this is a Motorola-type device, and uses "big-endian" address formats as opposed to Intel-type devices, which use "little-endian" addresses, read low-byte first followed by the high-byte.
- The "Access Detail" column tells you how many bus clock cycles this command takes, one cycle per letter code, and what happens for each clock cycle (something we usually don't need to know much about). Remember that the bus clock is half of the crystal frequency. Since the crystal on our board is 16 MHz, the bus clock is 8 MHz, with a period of 125 ns. So, the "IDX2" version of this command would take four clock cycles ("frPP") at 125 ns per cycle for a total of 500 ns.
- The last two columns tell us what to expect in the Condition Code Register. In this case, we should expect to see changes for the "negative" and "zero" flags, and the "overflow" flag will always be cleared to zero.

The following should be a review of work done in a previous course, but is included here as a reminder as to how to get started in CodeWarrior.

1. Start a project in CodeWarrior. (Typical settings shown in italics: Select the right microcontroller – *MC9S12XDP512*; select the right connection pod – *TBDML*; select the right core configuration – *Single Core*; select the right language – *Absolute assembly*; enter an appropriate project name and location – *Desktop->9S12X->Projects*.)
2. Skeleton.txt is a file that should be available in Moodle. Open and copy its contents (*Ctrl A – Ctrl C*); replace the text in main.asm with these contents (*Ctrl A – Ctrl V*).

3. Change the information in the file header to reflect who you are and the nature of the project.
4. Enter your code in ROM, which will be where the skeleton file says "Main:".

Here's a bit of code you can put into a project to practice with some of the concepts covered to this point. You might want to determine how long "kill some time" takes, based on what you now know about the timing of clock cycles and the size of the Y register. (You should come up with about a 25 ms delay.)

```

CLR    PT1AD1           ;initialize -- LEDs will be off
BSET   DDR1AD1,%11100000 ;make LED indicator pins outputs
Loop:  BSET   PT1AD1,%10000000 ;turn on red LED

LDY    #0
DBNE   Y,*             ;kill some time

BCLR   PT1AD1,%10000000 ;turn off red LED

LDY    #0
DBNE   Y,*             ;kill some more time

BRA    Loop            ;go again

```

Here's some code that performs exactly the same function as the code above. See if you can explain why.

```

CLR    PT1AD1           ;initialize -- LEDs will be off
MOVW   #%11100000,DDR1AD1 ;make LED indicator pins outputs, switches inputs
Loop:  LDAA   PT1AD1
EORA   #%10000000
STAA   PT1AD1           ;toggle red LED








LDY    #0
DBNE   Y,*             ;kill some time

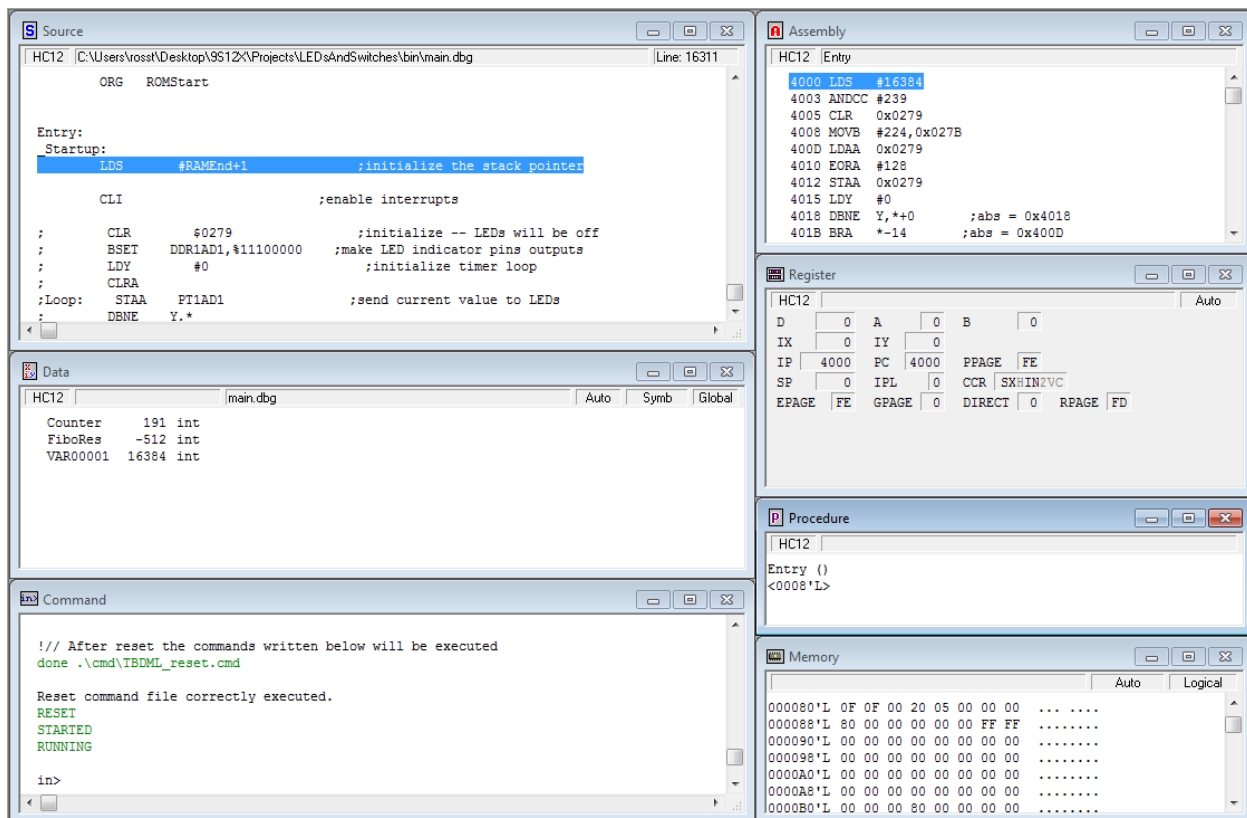
BRA    Loop            ;go again

```

Rudimentary Debugging Skills

In the Debug window, across the top tool bar, you will find the following set of buttons. You can mouse-click these or use the associated hot-keys, shown in the following table.

Button	Function	Hot Key
	Run	F5
	Single Step	F11
	Step Over (run subroutine)	F10
	Step Out (exit subroutine)	Shift+F11
	Assembly Step	Ctrl+F11
	Halt	F6
	Reset	Ctrl+R



The screenshot displays the debug environment with several windows open:

- Source:** Shows assembly code for 'main.dbg'. The current line is 'LDS #RAMEnd+1 ;initialize the stack pointer'.
- Assembly:** Shows the disassembled code for the current instruction, starting with '4000 LDS #16384'.
- Data:** Shows a list of variables: Counter (191 int), FibRes (-512 int), and VAR0001 (16384 int).
- Register:** Shows the contents of registers D, A, B, IX, IY, IP, PC, SP, EPAGE, GPAGE, CCR, DIRECT, and RPAGE.
- Procedure:** Shows the entry point 'Entry ()' at address '<0008'L'.
- Memory:** Shows a memory dump starting at address '000080'L'.
- Command:** Shows the command prompt with the output of the 'RESET' command: 'Reset command file correctly executed.', 'RESET', 'STARTED', 'RUNNING', and 'in>'.

- In the Debug environment, the "Source" and "Assembly" windows show you the code as typed by you and as interpreted by the Assembler.
- The "Data" window shows you the contents of variables and constants used in the program. These are updated whenever the program is halted or reaches a breakpoint.
- The "Memory" window shows the contents of any memory location, and highlights recent changes in red.
- The "Register" window shows the contents of all of the microprocessor's registers. A very good troubleshooting technique is to check the contents of the registers against what you think you're loading into them. This will help you determine if you've made

an error in loading something using Immediate addressing mode (#) or Extended addressing mode (contents of a memory location indicated by the address).

- In the "Source" window, you can right-click on a given line and set a breakpoint as a temporary stopping point in the program, allowing you to examine the contents of the registers, data, and memory.
- While the microprocessor's activity is halted, you can manually change the contents of the registers and memory, which will allow you to do "what if" scenarios or cut down the number of cycles in a long loop by changing the value of a register that's being used as a counter.

Once you compile and download code to your microcontroller, it will continue to run that code on start-up until you over-write it. You've burned your program into EEPROM on board, and, until you reprogram it, it will continue to run the same instructions faithfully.

Documentation and Comments

In a previous course, you used a "skeleton" file each time you started a new project. The following is a skeleton file modified from one used by Marc Anderson at NAIT. This is a good starting point – change the text and tabs, etc. to match your comfort zone, and save this as a simple .txt file for future use. It may also be provided by your instructor.

```

;*****
;* HC12 Program:      YourProg - MiniExplanation
;* Processor:        MC9S12XDP512
;* Xtal Speed:      16 MHz
;* Author:           P Ross Taylor
;* Date:             LatestRevisionDate
;*
;* Details: A more detailed explanation of the program is entered here
;*
;*****

;export symbols
;XDEF      Entry           ;export 'Entry' symbol
;ABSENTRY  Entry           ;for absolute assembly: app entry point

;include derivative specific macros
;INCLUDE 'derivative.inc'

;*****
;* Equates
;*****

;*****
;* Variables
;*****
;ORG      RAMStart        ;Address $2000

;*****
;* Code Section
;*****
;ORG      ROM_4000Start   ;Address $4000 (FLASH)
Entry:
;LDS      #RAMEnd+1      ;initialize the stack pointer

Main:

;*****
;* Subroutines
;*****

;*****
;* Interrupt Service Routines
;*****

;*****
;* Constants
;*****
;ORG      ROM_C000Start  ;second block of ROM

;*****
;* Look-Up Tables
;*****

;*****
;* SCI VT100 Strings
;*****

;*****
;* Absolute Library Includes
;*****
;INCLUDE "Your_Lib.inc"

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

```

Using the Skeleton.txt File

1. Start a project in CodeWarrior following steps you've previously used. (Typical settings shown in italics: Select the right microcontroller – *MC9S12XDP512*; select the right connection pod – *TBDML*; select the right core configuration – *Single Core*; select the right language – *Absolute assembly*; enter an appropriate project name and location – *Desktop->9S12X->Projects*.)
2. Open and copy the contents of Skeleton.txt (*Ctrl A – Ctrl C*); replace the text in main.asm with the copied contents (*Ctrl A – Ctrl V*).
3. Change the information in the file header to reflect who you are and the nature of the project.
4. To include a library, first right-click the "Includes" in the Browser and locate the file (*should be in Desktop->9S12X->Libraries*), then in main.asm insert an INCLUDE Assembler directive following the commented template shown.
5. Enter your code in ROM, which will be where the skeleton file says "Main:".
6. Declare any variables in RAM where the skeleton file indicates "Variables" using a "DS.x nn" Assembler directive.
7. Define any constants in ROM where the skeleton file indicates "Constants" or "Look-up Tables" or "SCI VT100 Strings" using "DC.x" and the actual constant data.
8. Put any locally-defined subroutines after the end of your main code loop, which will automatically happen if you use the skeleton area labelled "Subroutines".

When you write a subroutine, you should write a header that tells a programmer how to use the routine and what to expect of it. The following is an example.

```

;*****
;*                               *
;*           HexToBCD             *
;*                               *
;*                               *
;*Regs affected:  D (A and B)     *
;*                               *
;*                               *
;*A hexadecimal value arrives in Accumulator D and is converted *
;*to a 16-bit BCD, returned in D *
;*                               *
;*                               *
;*Maximum hexadecimal value allowed is $270F *
;*                               *
;*                               *
;*****

```

From this, the programmer knows that the D Accumulator must be loaded with the appropriate hexadecimal 16-bit word, and that, after a "JSR HexToBCD" the D Accumulator will contain the 16-bit BCD equivalent. You should also know, as a programmer, that since the contents of D are modified, the A and B registers will be modified by this subroutine.

You will be writing some subroutines that are specific to the task at hand – these will go in the "main.asm" file you're working on, usually close to the end of the code. You will also be writing subroutines that can be used in multiple projects. These you will collect into "libraries" of subroutines, which you will link to the main file using assembler directives. When you write a subroutine, then, you should determine whether it could be used by other programs and should be in a library or if it is unlikely to be used elsewhere and should therefore just be locally-accessible.

As you write code, get in the habit of writing comments as you go. Make your comments informative, not just a rewording of the Assembly instructions. In the following example, the first line is not informative; the second one is.

```

Bad:  LDAA  #$E0          ;load accumulator A with hex E0
Good: LDAA  #%11100000   ;ready to initialize Port AD Data Direction Register

```

Don't type pages of code and then try to go back and insert comments. The comments are there to help you keep track of what you're doing as you are coding just as much as to help win your instructor's favour!

Of course, if your Main code is set up as a carefully-planned sequence of calls to well-named subroutines (more on that later), comments would be redundant.

```
Start:      JSR    SwLED_Init
Flash:     JSR    Red_On
           JSR    lmsDelay
           JSR    GrnLEDon
           JSR    lmsDelay
           JSR    All_Off
           JSR    lmsDelay
           BRA    Flash
```

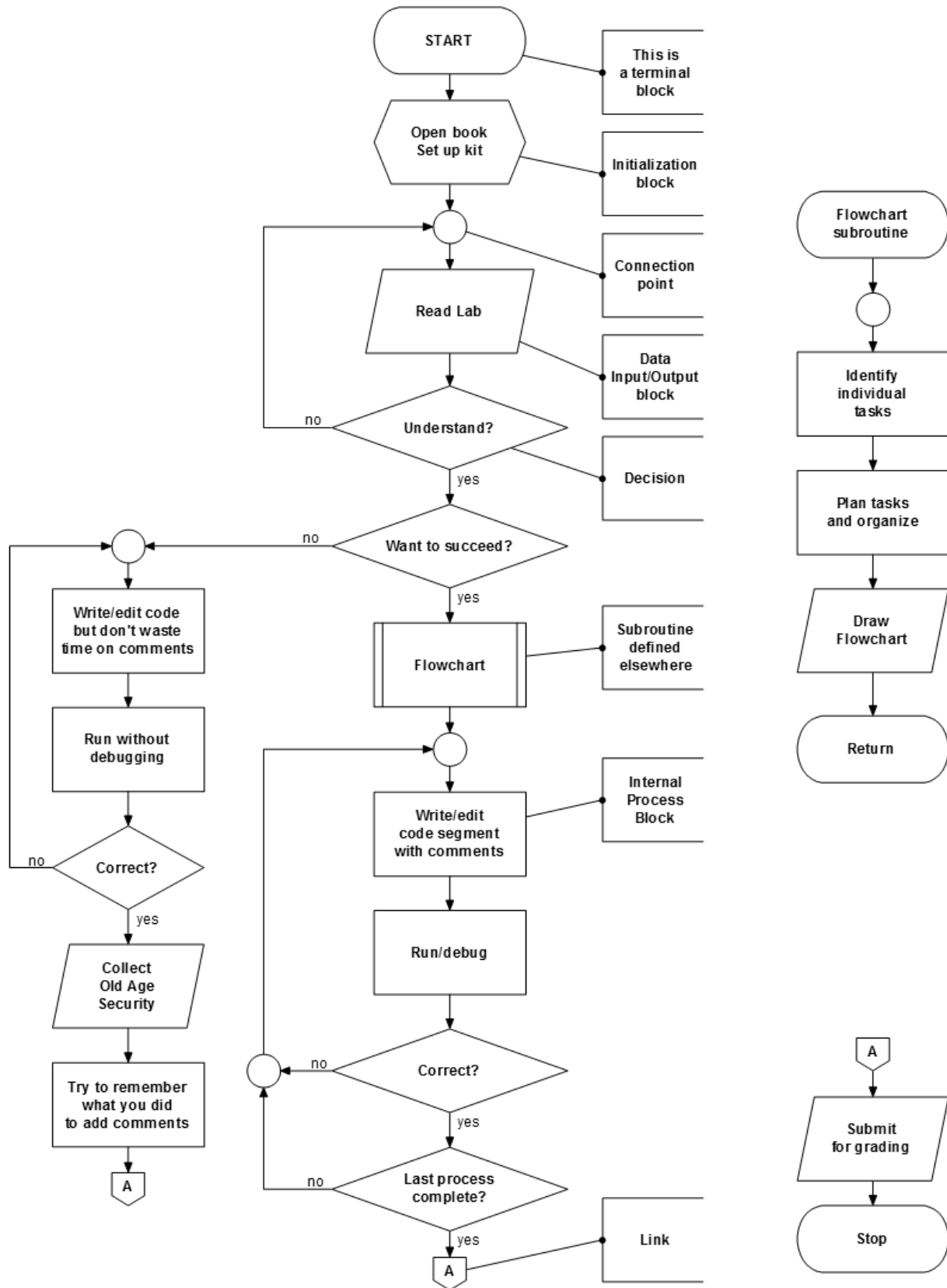
This kind of code wouldn't need comments, as it is self-commenting.

As your programs become more involved, you will need to do some pre-planning, as you would with any programming task in any language.

Some programmers are comfortable with writing pseudo-code as a guide to eventually developing proper code. Others prefer using flow-charts. Either way, a properly planned program will have cleaner code, will be more likely to run without errors, will be easier to troubleshoot, and will be easier to modify if the specifications change.

The following page points out some of the pitfalls of programming without proper planning.

Flowcharting



The flowcharting sequence and the clearly-recognizable blocks shown on the previous page provide a good way for you to organize your thoughts and your code.

When you attack a programming problem, one of the first things you should do is identify discrete tasks that need to be completed. Now, look over your list of tasks: are there any that could be made into general subroutines for other kinds of projects? If so, these should be written as generically as possible, for inclusion in libraries.

Consider writing your program so that the Main program is, for the most part, a sequence of calls to subroutines. Draw your flowchart to reflect this flow of events. You can put all the "special" subroutines (i.e. the ones not in libraries) below the Main code (well-marked and documented, of course).

Don't put "code snippets" into your flowchart – this should be understandable to someone who is knowledgeable about programming but doesn't necessarily know the language you're using. Instead, put descriptive terms or phrases in the program. For example, don't say "Carry Flag Set?" Instead, say what that carry flag means in terms of the program. It may mean "Data Ready?" or "Counter Max'd out?", or whatever your program is looking for.

Subroutines

Often when programming a microcontroller, you encounter pieces of code that are used in multiple places. Rather than doing the "cut'n'paste" routine, which results in very long and unreadable code, you can write subroutines (think "methods" in C#) which can be called from the Main program anytime you wish.

In fact, well-structured code should have a very simple Main program that calls well-named subroutines to do all the work. We'll get into proper code structuring later, after spending time writing useful subroutines. The following general pointers should help you immensely.

- A subroutine needs a unique label – use something informative, like "CheckLeftSw:". In this context, labels are followed by a colon.
- You must enter a subroutine using a **JSR** (ok, you could also use **BSR**, but it's not designed to jump more than 256 addresses from where you are, and takes no more effort or time than a JSR, so why would you bother?).
- You must exit a subroutine using **RTS**.

Important Note!!! You must not use any of the "branch on decision" instructions to enter or exit a subroutine. When you JSR into a subroutine, the microprocessor places the return address onto the stack. When you RTS from a subroutine, the microprocessor grabs the return address off of the stack and goes back to where it came from. If you don't have an RTS to match every JSR, you will mess up the stack. You will either add more and more stuff to the stack resulting in a stack overflow, or you will take too much stuff off the stack, straying into unknown territory in memory. Either way, your program will crash in microseconds.

If you use an accumulator or register within the subroutine, **PSH** it onto the stack before you use it, then **PUL** it back off the stack when you're done with it so that it's back to the condition it was in before you entered the subroutine. (That is, unless you want to use that register to return a value from the subroutine.)

Bottom line: make sure that you always unstack exactly the same number of items as you have stacked, and in reverse order.

- If you have subroutines within the file called Main, put them all below the actual Main program in a section clearly labelled "Subroutines".

- Make sure you have a descriptive header that explains what the subroutine does and what registers, if any, are modified by the subroutine. This way, when you decide to use the subroutine somewhere else, you'll know what it expects and what it returns.
- Where possible, try to make your subroutines, particularly ones used in libraries, broadly useful. Typically you wouldn't put commands into a subroutine that make it so it can only be used in one place in one program, unless it helps clarify the general operation of the program.
- If you need to change a subroutine, particularly one in a library, make sure the changes are backwards compatible so that previous programs that use these subroutines will still operate. If there is no way of keeping a subroutine backwards compatible, create a new subroutine with a different name for use in subsequent programs.
- In the context we've chosen for development (Absolute Assembly), you only have access to global variables. Where possible, try to make your subroutines work without using variables so that they are more portable. If you must use a variable or constant, make sure you put a note in the header reminding yourself or someone else using your subroutine that the variable or constant needs to be declared in the Main program.

Libraries of Subroutines

As previously mentioned, some subroutines should be made available so they can be used by other programs. These library files are simply text files containing the subroutines you want to include in them. The Assembler/Linker is designed to include any libraries you want to attach to your main code when you Run/Debug your program. Please note that everything in the library gets added into your assembled code, so you might want to plan your libraries so as to keep the amount of unused code in your assembled code to a reasonable amount.

As previously mentioned, you need to do the following two things to include a library file:

- Add the library to the "includes" folder in the Project window (right-click the folder and add the file when prompted).
- Put an "INCLUDE" Assembler directive **after** your code, so that the included subroutines will appear at the end of your code in ROM.

If you use the Skeleton file mentioned earlier, it has a pre-defined block for these INCLUDE statements.

One way to create a library is within the context of a Main program. Write all your subroutines below the Main program, as usual. After you have tested each of them and are convinced they do what you want them to do, move the subroutines into a separate text file and create an informative comment block at the top – it should list each of the subroutines contained in the library – and save the result as a ".inc" file. It's that easy!

Alternatively, you can start a new ".inc" file, and build it up alongside a "main.asm" file that checks each routine as you build it. You'll need to make the appropriate "includes" to make this work. This author recommends this method, because it reduces the number of surprises you might experience.

The more difficult part is deciding what you want to have in a library. You should collect together subroutines that are related, and are therefore likely to be used for a particular type of program. For example, you will be doing a lot of work with the Serial Communication Interface (SCI). It would make sense to have routines that send and

receive characters through the SCI in the same library. It doesn't make sense to have routines that turn on the LEDs in that library.

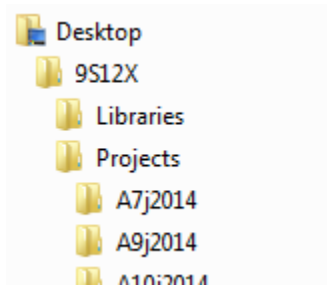
Assembly language libraries suited to the content of this course would include the following:

```
SwLED_Lib.inc
Misc_Lib.inc
SevSeg_Lib.inc
LCD_Lib.inc
SCI0_Lib.inc
PWM_Lib.inc
ATD0_Lib.inc
IIC0_Lib.inc
```

The focus of this course will shift to ANSI C programming before all of the material in this list of libraries has been covered, so you will likely not be required to create all of these libraries yourself. However, your instructor may provide you with these and other libraries as seems appropriate.

Misc_Lib.inc will contain routines that could be used in a number of types of programs – things like HexToAsc, HexToBCD, etc.

If you haven't already done so, you should develop a useful file structure for your work, like the following:



Create this file structure, and simply copy the entire thing back and forth between the desktop of the computer you're working on and your file storage device.

You should probably store your skeleton file in the "9S12X" directory so you can access it every time you start a project.

During the creation of a new project, the CodeWarrior IDE will create the folder for that project and will build all the associated subfolders and files inside the project folder.

S12XCPU Addressing Modes

Addressing modes define what memory the instruction will operate on. Each instruction offers one or more addressing modes. The S12XCPU offers some very complex addressing modes, and we will not look at all of them. Some addressing modes lend themselves well to compiler output, so as humans, we aren't suitable candidates for their use.

The assembler selects the addressing mode, where appropriate, from the form of the source instruction. The text form of the instruction entered into the assembler is ultimately rendered into machine code – that which the CPU understands. Common omissions or 'trivial' mistakes in code entry can lead to incorrect values or incorrect addressing modes in machine code.

Inherent - INH

The simplest of all addressing modes is inherent (INH). Inherent instructions require no additional information to operate.

The following code snippet contains a number of Inherent commands. Notice how the Assembler interprets each – no reference to memory addresses.

The screenshot shows a source code window with the following text:

```

nop
aba
inx
mul
tfr x,y

```

The assembly window shows the following output:

Address	Hex	Instruction	Operands
4000	CF2000	LDS	#0x2000
4003	10EF	ANDCC	#0xEF
4005	A7	NOP	
4006	1806	ABA	
4008	08	INX	
4009	12	MUL	
400A	B756	TFR	X, Y

Immediate - IMM

The immediate addressing mode contains the required operands in the object code, meaning the required information is constant, user-defined, and part of the instruction.

To indicate the immediate addressing mode, these instructions must use a pound sign on the operand. This will differentiate the immediate form from the extended form, which we will look at next.

The following code snippet contains a number of commands in Immediate Addressing mode. Notice how the Assembler interprets each command, and what it will work on.

The screenshot shows a source code window with the following text:

```

ldaa #45 + $a
bitb ##10
ldy #Entry
sbc b #-1

```

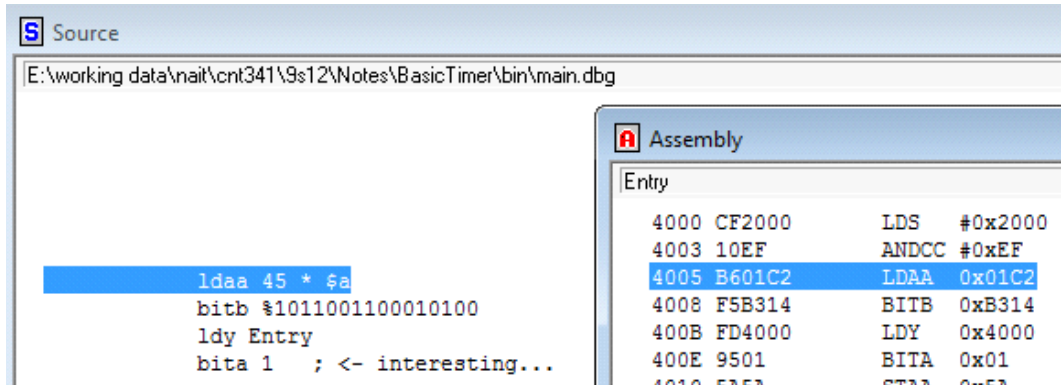
The assembly window shows the following output:

Address	Hex	Instruction	Operands
4000	CF2000	LDS	#0x2000
4003	10EF	ANDCC	#0xEF
4005	8637	LDAA	#0x37
4007	C502	BITB	#0x2
4009	CD4000	LDY	#0x4000
400C	C2FF	SBCB	#0xFF

Extended – EXT

The extended addressing mode requires a 16-bit address. The byte(s) at this address are used by the instruction. What happens to the byte(s) depends on the instruction.

The assembly language form for extended addressing requires no decoration, just anything that can be bent into an address:



We can use labels to define constants (in ROM) and variables (in RAM), then we can use various addressing modes, like "EXT", to access these. Here's an example.

```

Counter:   ORG    $2000                ;start of RAM
           DS.B   1                  ;one byte assigned as variable Counter
           ORG    $4000                ;start of FLASH for program
           MOVB  #$5A,Counter        ;place initial value into Counter

```

Direct – DIR

The direct addressing mode is used to operate on memory locations 0x0000 – 0x00FF. Syntactically this form is identical to extended, except this form requires one less byte of code, as the high byte of the address is assumed to be \$00. This addressing mode is useful when RAM is available in the first 256 bytes of the memory map, as it provides fast access for variables. Sadly, for the S12XCPU as configured in the 9S12XDP512, there is not a lot of call for direct addressing, since the "first page" of memory contains the microcontroller's internal peripheral module registers. The assembler will automatically detect and implement direct addressing instead of extended if the instruction supports it.

Relative – REL

The relative addressing mode is used principally in branching instructions. The S12XCPU supports long and short branching. The operand(s) in a branch instruction form a signed offset that participates in forming a new address for the program counter – in other words, the program execution moves to a new point, or branches away. The target address is found by adding the relative offset in the operand(s) to the address following the first (maybe only) offset operand. One of the things you should appreciate about the assembler is its ability to calculate relative offsets for you. You put in labels, it does the math.

Note: Some instructions like BRSET and BRCLR don't show the addressing mode as REL because they are performing two commands: a compare and a branch. However, they are using relative addressing, so you won't be able to use these commands to move more than -128 or +127 counts of the program counter from where you're at. This can be a serious limitation, and will probably catch you off guard: you'll have a program that's working

perfectly, then you'll add in a bit of code between one of these commands and its target. Suddenly, you'll get an "out of range" error, with no simple way of solving the problem.

The simple branching commands have a "long branch" option. By putting an "L" in front of the mnemonic, its range becomes $-32,768$ to $+32,767$. This isn't available for the complex instructions.

In the code snippet below, by comparing the "Source" window to the "Assembly" window, you can see how the Assembler has interpreted each of these instructions using Relative addressing. Take some time to get to know what the offsets are and how they are displayed in the disassembled code, in the comments, and in the machine code. Some of these are discussed below.

```

Source
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg

forever: bra *
brset $4000,#1,forever
backward: brn forward
forward: bra backward
lbra *

Assembly
Entry
4000 CF2000 LDS #0x2000
4003 10EF ANDCC #0xEF
4005 20FE BRA *+0x0 ;abs = 0x4005
4007 20FE BRA *+0x0 ;abs = 0x4007
4009 1E400001F9 BRSET 0x4000,#0x1,*0xFFFFFFFF ;abs = 0x4007
400E 2100 BRN *+0x2 ;abs = 0x4010
4010 20FC BRA *0xFFFFFFFF ;abs = 0x400E
4012 1820FFFC LBRA *+0x0 ;abs = 0x4012
4016 5A5A STAA 0x5A

```

Look at the "**bra ***" line. As written, this means we want the code to branch back to the beginning of the current line. In the disassembled code, this shows up as "BRA *+0x0", which again means the intent is to branch 0 positions from the beginning of this line. In the comments for the disassembled code, the absolute address is given as "0x4005", which, as you can see, is the beginning address of this particular line of code. The most important part, though, is to understand the machine code: "20FE". "20" means a short branch always. "FE", though, is -2 , and tells the microprocessor to subtract 2 from the current program counter. Since the program counter will have advanced by 2 while executing this command to address \$4007, it will be moved back to \$4005, where it will execute the command again ad infinitum.

Briefly look at the difference between the "**bra ***" line and the "**lbra ***" line. In this case, you'll notice that the relative offset is four nibbles: \$FFFC, or -4 , to cover the extra bytes required in this longer version of the command.

Look at the "**brset \$4000,#1,forever**" line. In this command, if the LSB of the value in address location \$4000 is set (the mask "#1" is the same as $\#00000001$), the program counter is supposed to go back to "forever", which is address \$4007. (Incidentally, address \$4000 contains \$CF as seen in the program listing, so the branching condition will be TRUE.) Notice what the machine code says: "1E400001F9". "1E" is the op code for BRSET. "4000" is the address we want to check the contents of. "01" is the mask we're comparing against. And now for the offset: "F9" is only 8-bit, so we can only do short branches with this command. "F9" is -7 , which takes the program counter back from \$400E where it's sitting to \$4007, the address of "forever".

Indexed – IDx, IDx1, IDx2, [IDx2], [D,IDX]

The S12XCPU chip has several forms of indexed addressing. The simplest form of indexed addressing uses X,Y,SP or PC ("x" in the form above) as a pointer. To this pointer (the value in the register) an offset is added. The offset is either a 5-bit signed offset (IDx), 9-bit signed offset (IDx1), 16-bit signed offset (IDx2), or the contents of accumulators A, B, or D. The assembler will automatically select the correct form of the instruction, as long as what you enter can be bashed into a valid instruction:

The screenshot shows a source code window with the following code:

```

;bra next

movw #$0102,$1000
movw #$0304,$1002

ldx #$1000
ldaa 0,x ; 5-bit
ldaa $20,x ; 9-bit
ldaa $2000,x ; 16-bit
ldaa a,x ; accumulator offset

```

The assembly window shows the corresponding assembly instructions:

```

Entry
4000 CF2000 LDS #0x2000
4003 10EF ANDCC #0xEF
4005 180301021000 MOVW #0x102,0x1000
400B 180303041002 MOVW #0x304,0x1002
4011 CE1000 LDX #0x1000
4014 A600 LDAA 0x0,X
4016 A6E020 LDAA 0x20,X
4019 A6E22000 LDAA 0x2000,X
401D A6E4 LDAA A,X
401F 5A5A STAA 0x5A

```

The indexed-indirect addressing mode ([IDx2] and [D,IDX]) allows either a 16-bit or D offset from X,Y,SP, or PC. In this mode the address formed from the offset is used as another address. The action of the instruction is on the target of this address:

The screenshot shows a source code window with the following code:

```

movw #$0700,$0500
movw #$1234,$0700

ldx #$0500
ldy [0,x] ; x+0 = $0500, so read addr from there
           ; address read from $0500 is $0700
           ; so load Y from $0700, Y = $1234

```

The assembly window shows the corresponding assembly instructions:

```

Entry
4000 CF2000 LDS #0x2000
4003 10EF ANDCC #0xEF
4005 180307000500 MOVW #0x700,0x0500
400B 180312340700 MOVW #0x1234,0x0700
4011 CE0500 LDX #0x500
4014 EDE30000 LDY [0x0000,X]
4018 5A5A STAA 0x5A
401A 5A5A STAA 0x5A
401C 5A5A STAA 0x5A

```

NOTE: You can live a long, happy life not using most of the indexed addressing modes. However, they're there if you need them.

The indexing modes also offer pre/post increment/decrement options for the indexed addressing modes. These are typically leveraged by compilers, but you are free to look up their operation, should you feel ambitious.

You should get to know the direct indexing modes (no square brackets) very well. Here's a bit of practice.

	LDX	#Table
	LDAA	2,X
	BRA	*
	ORG	\$\$C000
Table:	DC.B	\$1E, \$B6, \$2F, \$5A

After this code has run, X = \$C000, and A = \$2F. Make sure you can explain why before moving on.

Frequently-Used Instructions

In the Reference Guide, you will find a listing of all the possible instructions (the "Instruction Set") for the S12XCPU. You should look through this entire list to see what sorts of things you can do with this device.

Here are a few of the ones you will probably use extensively. As you go through this list, remember that references to "memory location" could refer to the byte or word directly following the Op Code (IMM mode) or a memory location elsewhere, accessed using any of the other addressing modes.

LDAA, LDAB, and LDx – (where **x** could be **D, X, Y, or S**) puts the contents of a memory location into the selected accumulator or register. Remember that 8-bit registers will load a single byte and 16-bit registers will load two bytes – always the one you point to and the one immediately following it – in order to get the full 16 bits.

STAA, STAB, and STx – (where **x** could be **D, X, Y, or S**) puts the contents of the selected accumulator or register into a memory location. Again, remember that 8-bit registers will store a single byte into the location you're pointing to, and 16-bit registers will store two bytes – one into the location you're pointing to and one into the one following it. If you forget this, you're in for a big surprise when you over-write a byte you didn't think you were going to affect.

CLR, CLRA, and CLRB – all bits cleared in the selected accumulator or memory location.

DEC, DECA, DECB, DEx – (where **x** can be **S, X, or Y**) subtracts one from a memory location or register.

INC, INCA, INCB, INx – (where **x** can be **S, X, or Y**) adds one to a memory location or register.

BCC and **BCS** – branch to a specified location, based on the condition of the Carry flag

BEQ and **BNE** – branch based on the condition of the Zero flag

BGE, BGT, BLE, BPL, BMI, and BLT – branching decisions based on the comparison of signed numbers.

BHI, BLO, BHS, and BLS – branching decisions based on the comparison of unsigned numbers.

DBEQ and **DBNE** – compound instructions that decrement a register or memory location, then make a decision based on whether or not the result is zero.

ADDx and **ADCx** – (where **x** could be **A, B, or D**) these add the contents of a memory location to the contents of the selected accumulator. If you use the "ADC" version, whatever is in the Carry flag of the CCR (0 or 1) will also be added in.

SUBx – (where **x** could be **A, B, or D**) subtracts the contents of a memory location from the contents of the selected accumulator.

MUL – multiplies A by B and dumps the result in D.

There are five different division routines: **FDIV, EDIV, EDIVS, IDIV, and IDIVS**. These are somewhat complicated to use, and will be explained when you need them.

ANDA and **ANDB** – these perform a bit-wise AND between the contents of the specified accumulator and the contents of a memory location (more later when we discuss masks).

ORAA and **ORAB** – these commands perform a bit-wise OR between the contents of the accumulator and the contents of a memory location.

EORA and **EORB** – performs a bit-wise Exclusive OR (XOR) between the accumulator and the contents of a memory location.

COM, **COMA**, and **COMB** – performs a 1's complement inversion of each bit.

NEG, **NEGA**, and **NEGB** – performs a 2's complement of the target value.

BITA and **BITB** – (bit test) performs an "AND" operation between the accumulator and the memory location, but doesn't affect the contents of either – only the Condition Code register is affected.

CBA – compares the A and B accumulators by subtracting B from A, and modifies the Condition Code Register accordingly – used to determine which is greater.

CMPA, **CMPB**, **CPx** – (where **x** can be **D**, **S**, **X**, or **Y**) compares the selected register to memory by subtracting the contents of memory from the register, but doesn't change anything except the CCR.

LSL and **LSLx** – (where **x** could be **A**, **B**, or **D**) performs a logical shift left, bringing in a "0" at the lowest bit and spitting the highest bit into the Carry flag of the CCR.

LSR and **LSRx** – (where **x** could be **A**, **B**, or **D**) performs a logical shift right, bringing in a "0" at the highest bit and spitting the lowest bit into the Carry flag.

ROL, **ROLA**, and **ROLB** – just like LSL, except that the contents of the Carry flag are brought in to the lowest bit instead of "0". Watch this command: it rolls 9 bits, not 8.

ROR, **RORA**, and **RORB** – just like LSR, except that the contents of the Carry flag are brought in to the highest bit. Again, this rolls 9 bits, not 8.

BCLR – clears bits (ensures that bits are "0") according to which bits are set in a mask. Other bits remain unchanged. This involves ANDing the bitwise complement of the mask.

BSET – sets bits (ensures that bits are "1") according to which bits are set in a mask. Other bits remain unchanged. This involves ORing the mask.

CLC – clears the Carry flag in the CCR.

CLI – clears the Interrupt bit in the CCR, thereby enabling interrupts.

EXG, **XGDX**, and **XGDY** – swaps the contents of two registers. Things get messy if you swap the contents of 8-bit and 16-bit registers!

TFR, **TAB**, **TBA**, **TAP**, **TPA**, **TSX**, **TXS**, **TSY**, and **TYS** – moves the contents of one register into another without changing the first register's contents. Again, transferring the contents from an 8-bit register to a 16-bit or *vice versa* can produce unexpected results!

MOVB and **MOVW** – moves a byte (8-bit) or a word (16-bit) from one memory location to another. Frequently used in IMM/EXT mode, (e.g. `MOVB #$3E,Table+1`) this can also be used in EXT/EXT mode (e.g. `MOVB Counter,Display`) or various indexed (IDX) modes. Note: Don't confuse the "B" for "byte" in `MOVB` with Accumulator B!

PSHx and **PULx** – (where **x** could be **A**, **B**, **C** (for CCR), **D**, **X**, or **Y**) places an item on the stack or takes it back off the stack, allowing you to use the stack as temporary storage.

JSR and **RTS** – jump to a subroutine and return from a subroutine. Be careful with these – they automatically involve placing the Program Counter (16-bit) on the stack and pulling it back off. If you follow a JSR with a branching statement instead of an RTS, you will quickly experience the pain of a stack overflow. Every JSR must have an RTS.

Masks and Bitwise Boolean Logic

Many of the instructions for the S12XCPU involve masks. A mask is an 8-bit or 16-bit binary pattern used to select individual bits in a register or memory location. In its original sense, the pattern "masked out" bits that weren't needed or wanted for a particular operation, leaving the significant ones "visible". We now use the term more generally for any pattern that allows us to operate on individual bits instead of the whole byte or word.

The various bitwise actions are as follows:

- SET the selected bits (i.e. make these bits HIGH, or logic 1).
- CLEAR the selected bits (i.e. make these bits LOW, or logic 0).
- TOGGLE the selected bits (i.e. LOW becomes HIGH, HIGH becomes LOW).
- BRANCH if the selected bit or bits is LOW.
- BRANCH if the selected bit or bits is HIGH.

It's important to distinguish between instructions or actions that affect an entire register or memory location and those that act on individual bits.

Commands affecting an entire register or memory location

LDAA/LDAB/LDD/LDX/LDY The register contents are replaced by the incoming data.

e.g. LDAA #%10100011 ;A ends up containing 10100011

STAA/STAB/STD/STX/STY The memory contents are replaced by the outgoing data.

e.g. STAA Counter ;Counter ends up containing the contents of A

MOVB/MOVW The memory contents are replaced by the indicated data.

e.g. MOVB #\$F2,Counter ;Counter ends up containing F2.

CLRA/CLRB/CLR Each bit in the register or memory location is cleared to 0.

e.g. CLR Counter ;Counter ends up containing 00.

COMA/COMB/COM Each bit in the register or memory location is toggled (complemented).

e.g. LDAA #%01011010
COMA ;A ends up containing 10100101

Commands affecting selected bits

ORAA/ORAB 1s in the mask SET the corresponding register bits; 0s have no effect.

e.g. LDAA #%01011010
ORAA #%11000000 ;A ends up containing 11011010

ANDA/ANDB 0s in the mask CLEAR the corresponding register bits; 1s have no effect.

e.g. LDAA #%01011010
ANDA #%11100111 ;A ends up containing 01000010

BSET 1s in the mask SET the corresponding bits in memory; 0s have no effect.

e.g. MOVB #%01100010,Counter
BSET Counter,%11000000 ;Counter ends up with 11100010

BCLR 1s in the mask CLEAR the corresponding bits in memory; 0s have no effect. This is essentially the same as ANDing with the complement of the mask.

e.g. MOVB #%01100010,Counter
BCLR Counter,%11000000 ;Counter ends up with 00100010

Commands responding to selected bits

BRSET If, in a memory location, a bit or all of the bits selected by 1s in the mask are HIGH, program execution will branch to the indicated address. The most straightforward way to use BRSET is in response to a single bit.

e.g. `BRSET PT1AD1,%00010000,UpSw ;Branches if the Up switch is pressed`

BRCLR If, in a memory location, a bit or all of the bits selected by 1s in the mask are LOW, program execution will branch to the indicated address. Again, this is easiest to use in response to the condition of a single bit.

e.g. `BRCLR PT1AD1,%00010000,NotUp ;Branches if the Up switch is not pressed`

Using Variables and Constants

As you code more complicated tasks, you will find it increasingly difficult to juggle the few CPU registers you have to work with. The use of RAM-based variables is an easy sell, and will help you with code management when there are multiple states to manage.

Variables need to be stored in RAM, or they won't be variable. Variables are created within ORG sections that place the program counter in RAM. The skeleton file you're working with has a header to show you where you should place your variables, which will be after an "ORG" that places the variables in RAM.

Variables are defined with a DS (Define Space) directive. There are three forms of the DS directive:

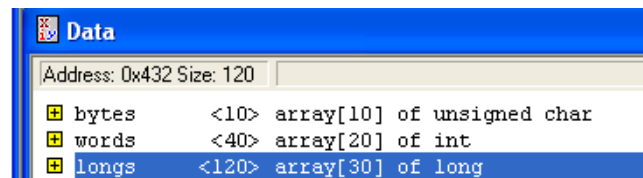
- DS.B reserve space for bytes (8-bits)
- DS.W reserve space for words (16-bits)
- DS.L reserve space for longs (32-bits)

The DS.x directive is followed by a count that indicates the number of elements to reserve. This number must range from 1 to 4096. The count is multiplied by the size of the type to determine the number of bytes that will be reserved for storage.

```

; variable/data section
                                ORG RAMStart
; Insert here your data definition.
bytes:    ds.b 10
words:    ds.w 20
longs:    ds.l 30

```



Reserved space is not initialized, and will typically contain garbage. It is your responsibility to initialize all reserved space if your code requires it.

You will usually include a label for each DS directive, although it's not required. The label and reserved space together are loosely referred to as a 'variable'.

NOTE: You've been told this before, but it doesn't hurt to say this again: Your library subroutines may use variables internally, but because of the layout of the projects we are creating, the variables must be created in the main program file. Use of variables in a subroutine in a library will require that you clearly document the required variable names and initial values in the subroutine block.

Constants are not something intended to change as the program runs. Consequently, they should be in ROM. For ease of debugging, we'll use the address space starting at \$C000.

Constants are defined with a DC (Define Constant) directive. There are three forms of the DC directive:

- DC.B byte-sized constants (8-bits)
- DC.W word-sized constants (16-bits)
- DC.L long constants (32-bits)

Since the data is constant, it must be defined at the time of assembly. Therefore, the DC command is followed by the data to be defined.

A typical application would be to store string data, as shown in the following example:

```
Name:    DC.B "P. Ross Taylor",0
```

This will create a 15-byte field with an ASCII character in each byte (the last one is NULL), where the address indicated by "Name" will be the address of the character "P".

Here are some important things to keep in mind when using variables and constants in your code.

1. The Assembler interprets your assignment of a constant or variable label as a 16-bit address, and will replace subsequent references to that label with the 16-bit address.
2. Storing a single byte into a multi-byte variable changes only the first byte.
3. Storing a 16-bit value to a single-byte variable will change the contents of that address and the address following it. If that address is another variable, you will have over-written its contents (affectionately known as clobbering the next variable).
4. Storing to a constant in ROM changes nothing – why else would we call it a “constant”?
5. Calling something in RAM “constant” is a lie – don’t do it! You could be fooled when debugging your program, as the Assembler will populate that location when you download your code. However, when you turn off the micro board, that information will be lost forever, and the code won’t run properly the next time.
6. The Assembler lets you enter bytes into a constant or variable using a lot of different formats. For example:

```
Str: dc.b    $48,'i', $64,"den", $20,"Me", %01110011, 115, $60+1, 'g', $67-2
```

...will contain the ASCII characters “*Hidden Message*”. See if you can determine how each of the characters is interpreted from what is given by the Assembler. You may find some of these techniques useful. For instance:

- Putting an ASCII character in single quotes (*'i'*) tells the Assembler to treat this as an ASCII character.
 - Putting multiple ASCII characters in double quotes (“*den*”) tells the Assembler to put each of the ASCII characters into sequential address locations.
 - Characters can be entered in binary, decimal, hexadecimal, or even octal form.
 - The Assembler will even do calculations “on the fly” to arrive at a value to store in an address location.
7. The X and Y registers are called “index registers” because you can use them as pointers to the beginning of a memory space (like a multi-byte constant or variable).
 8. Use a pound sign (#) to load the address of a variable or constant into a 16-bit register, usually (but not exclusively) an index register (X or Y).
 9. You can find a byte at a particular offset from the address contained in an index register. This is done using Indexed Addressing Mode, for example

```
LDX    #Str                ;X now points to the first address of Str
LDAA   2,X                ;A now holds the third char in Str (zero based)
```

10. The index registers can point to any memory location (not just the start of a variable), so you can crawl through a multi-byte variable or constant using INX or INY.

```
INX                                ;following the above code, X points to Str+1
```

11. You can’t load an address into an 8-bit register. You will only get half of the address, and certainly not the contents of that address, so trying to work with that will produce really bizarre results. So don’t try `LDAA #Str` – it won’t work.
12. If you don’t use a pound sign, you will load the contents of the memory location represented by the variable. If you load an 8-bit register, you will get the single byte from that memory location (`LDAA Str` puts ‘H’ into A in our previous example). If you load a 16-bit register, you will get two bytes: the one you pointed to and the one following it. (`LDD Str` puts ‘H’ into A and ‘i’ into B in our previous example).

Programming in C

Every time you've started a new project, you've unchecked "C" and checked "Absolute Assembly" instead. You probably don't even think about it anymore; but what if

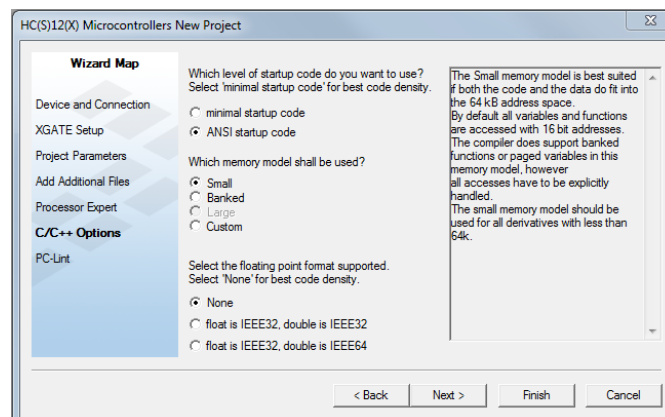
C is a much older language than the C# you've been working with. However, although Microsoft has tried to cloak the basics in Orwellian doublethink, you will probably find that there are some transferable concepts, fundamental operations, and syntax similarities that will make learning C for the 9S12X fairly easy. Unless your instructor has chosen to approach the order of this course differently, you should, at this point, have learned the fundamentals of programming in S12XCPU Assembly Language, which is tightly linked to an understanding of the operation of the microprocessor and its peripherals. Programming in C also requires a clear understanding of the operation of the microcontroller, particularly its registers; however, the C cross-compiler used by Code Warrior is capable of handling many things itself so you don't have to worry about all the details.

The following pages show how to start a C project, how to write and run simple code, how to write and use "Functions" (these are what C# architects decided to refer to as "Methods"), and how to write, include, and use uncompiled code libraries.

Setting Up an ANSI C Project

Start a new project, following the steps below:

1. Follow the usual steps to start a new project, including selecting the appropriate derivative of the 9S12X, the BDM interface you're using, and "Single Core".
2. Leave the "C" box checked, set the appropriate "Location" (your Projects folder), and give your project a name. Don't hit "Finish", as there are more screens coming.
3. When you get to the "C/C++ Options" page in the Wizard Map, select "ANSI startup code" and the "Small" memory model. If your program doesn't have to do anything mathematical, leave floating point format as "None". If, however, you want to do floating point math (i.e. fractional values) instead of just working with integers (not to be confused with *int* declarations), you should probably check "float is IEEE32, double is IEEE32". (This option will consume a lot more memory when generating code and will run more slowly, but you probably won't run out of room. Timing might be more of an issue.) Now you can click "Finish".



4. Open the "main.c" program. It will contain some basic code (a bit less annoying than the code they inserted in "main.asm", but still not entirely useful. Instead, create a "skel_C.txt" file like the one on the following page, replace the text in "main.c" with it, and tidy up the header information.

ANSI C Skeleton File

```

/*****
*HC12 Program:  YourProg - MiniExplanation
*Processor:    MC9S12XDP512
*Xtal Speed:   16 MHz
*Author:       This B. You
*Date:         LatestRevisionDate
*
*Details: A more detailed explanation of the program is entered here
*****/

#include <hidef.h>          // common defines and macros
#include <stdio.h>         // ANSI C Standard Input/Output functions
#include <math.h>          // ANSI C Mathematical functions
#include "derivative.h"    // derivative-specific definitions

/*****
*      Library includes
*****/

#include "Your_Lib.h"

/*****
*      Prototypes
*****/

/*****
*      Variables
*****/

/*****
*      Lookups
*****/

void main(void)           // main entry point
{
    _DISABLE_COP();

/*****
*      Initializations
*****/

    for (;;)              //endless program loop
    {
/*****
*      Main Program Code
*****/

    }
}

/*****
*      Functions
*****/

/*****
*      Interrupt Service Routines
*****/

/*****/

```

There are a number of ways to create an endless loop in ANSI C, including `while(1){code}`. However, we've chosen to use the endless `for (;;) {code}` loop as shown in the skeleton file above because it doesn't generate any compiler warnings.

Notice that *Variables* and *Lookups* appear to be in the same memory space, which would have to be in RAM. However, in ANSI C, both of these can be initialized or pre-loaded, which means their values would somehow have to be in ROM. In reality, the C compiler will store the initialization values in ROM, and will create a working copy in RAM on start-up so that the values can subsequently be changed under programmatic control. (The same can be done when programming in S12XCPU Assembly Language, but the process has to be written into the program, and is therefore much more involved.)

Switches and LEDs with ANSI C

Since you've already gained some experience working with the push-button switches and the LEDs connected to PT1AD1 of your microcontroller, this makes a good jumping point into writing programs in ANSI C.

In order to initialize PT1AD1, you need to set the LED-connected pins to OUTPUTS, set the Switch-connected pins to INPUTS, and digitally-enable the switch-connected pins. You probably also want to initialize the conditions of the LEDs to ALL OFF. Here's a bit of ANSI C code that performs these operations:

```
DDR1AD1=0b11100000; //LEDs as outputs, Switches as inputs
ATD1DIEN1=0b00011111; //Digitally-enable the Switch inputs
PT1AD1&=0b00011111; //Turn off all LEDs as initial condition
```

Notice that we can overwrite all eight bits in a register using the "=" operator. In the example above, the values have been entered as binary values, because that makes the best sense for bitwise operations. However, the first command could have been "DDR1AD1=0xE0" for hexadecimal or "DDR1AD1=0340" for octal or "DDR1AD1=224" for decimal – all of these options would have made the LED pins outputs and the Switch pins inputs. The cross-compiler is smart enough to convert any numeric representation into binary for the microcontroller, so pick the format that makes the most sense to you as a human-programmer. There are times when the decimal representation of a number makes most sense. For example, "NewDozen +=12" probably makes more sense to you than "NewDozen +=0b00001100", "NewDozen +=0x0C" or "NewDozen +=014".

If we only want to change some of the bits, we use bitwise operations like "&", "|", or "^" (AND, OR, or EOR). In the case above, we wanted to turn off the LEDs, so ANDing with 0 performs that function, whereas ANDing with 1 has no effect. If we wanted to turn on or set particular bits, we would OR the desired bits in the register with 1s, whereas ORing the other bits with 0 has no effect.

Functions

In ANSI C, a Function (a.k.a. Subroutine or Method in other languages) requires a "Prototype" or "Declaration", defined prior to the execution of any code. The prototype declares the *type* of what is returned from the function and the *types* of any parameters passed to the function. Although you can choose the variable names for parameters passed to the function in the prototype, this is not necessary. The prototype often looks like the first line in the actual function, just terminated with a semicolon. The following are some examples:

```
void SwLED_Init(void);
char SwCk(void);
void LEDOut(char LEDs);
void LEDOut2(char);
unsigned int TwoNumSum(unsigned char X, unsigned char Y);
```

The skeleton file provided has a section at the top for you to put your prototypes.

The function itself starts with a header much like the prototype, but with the variable names that will actually be used in the function, if these were not specified in the prototype. The "definition" of the function is contained between curly brackets {} (affectionately referred to as "chicken lips" in NAIT's CNT department).

If a value is to be returned from the function, a "return <value>" statement is required.

Libraries of Functions

As with your work with S12XCPU Assembly Language, you can create libraries of commonly-used functions. However, the process is quite different.

To begin with, you will need two files for each library: a header file (.h) and an uncompiled code library (.c). The header file contains all of the prototypes for the functions, with the definitions (actual code) appearing in the uncompiled code library.

When you create a new project, you will need to do three things:

- Include the ".c" file in the "Sources" section of the project browser window.
- Include the ".h" file in the "Includes" section.
- Add an #include "libname.h" line to the "Library includes" section of the skeleton.

The ".c" file itself needs to start with "include" statements. Here's a screen-shot of the beginning of this author's "SwLED_Lib.c" library:

```
//Switches and LEDs
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2014

#include <hidef.h>
#include "derivative.h"
#include "SwLED_Lib.h"
```

Notice the different punctuation: the <hidef.h> reference is to one of the standard ANSI C compiled libraries, whereas the libraries in quotes are uncompiled libraries. The "derivative.h" file points to the "mc9s12xdp512.h" file that contains all the definitions of the labels for the registers in the version of the 9S12 we're using.

You will be required to write a library to match the contents of the following "SwLED_Lib.h" header file:

```
//Switches and LEDs
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//June 2015

void SwLED_Init(void); //LEDs as outputs, Switches as inputs, dig in enabled
char Sw_Ck(void); //returns debounced condition of all switches in a byte, LED values = 0
void LED_On(char); //accepts R, G, Y, A (for all)
void LED_Off(char); //accepts R, G, Y, A (for all)
void LED_Tog(char); //accepts R, G, Y, A (for all), and toggles the condition of the LED(s) indicated
```

In the functions that accept a colour parameter, you will need to pass the value as an ASCII character, which requires the use of single quotes: e.g. 'R'.

To begin with, you won't need to build the "Sw_Ck()" function, as you need to learn a bit more about switch management before you can deal with that one.

Summary

You have now been provided with a bare minimum of what it takes to program in ANSI C. With your experience with other programming languages, particularly C#, you should be able to play around with ANSI C quite productively, and you will learn more of what the language looks like and what it can do as this course progresses.

Numeric Manipulation

A review of bit basics is prudent at this point, as an understanding of binary and hexadecimal will be assumed throughout the rest of this course.

Understanding Base 10

Base 10, or decimal, is a good radix to begin with, as you are familiar with it. We know that each digit contributes the digit value * 10^n , where n is the zero-based index of the digit, working right to left. Consider the number 343895_{10} :

Digit	3_{10}	4_{10}	3_{10}	8_{10}	9_{10}	5_{10}
Position Value	10^5	10^4	10^3	10^2	10^1	10^0
Digit Value	300000_{10}	40000_{10}	3000_{10}	800_{10}	90_{10}	5_{10}

$$343895_{10} = 300000_{10} + 40000_{10} + 3000_{10} + 800_{10} + 90_{10} + 5_{10}$$

$$343895_{10} = 343895_{10}$$

This pattern seems obvious for base 10, but works for base 2 (binary) and base 16 (hexadecimal) as well.

Converting Binary to Decimal

In binary the number is valued as the sum of each digit * 2^n , where n is the zero-based index of the digit, working right to left. Consider the number 100101_2 :

Digit	1_2	0_2	0_2	1_2	0_2	1_2
Position Value	2^5	2^4	2^3	2^2	2^1	2^0
Digit Value	32_{10}	0_{10}	0_{10}	4_{10}	0_{10}	1_{10}

$$100101_2 = 32_{10} + 4_{10} + 1_{10}$$

$$100101_2 = 37_{10}$$

The system shown above is called "Weighted Sum of Powers".

Converting Hexadecimal to Decimal

Hexadecimal is no different, other than including A-F as digits to allow each hex digit to represent one of 16 different values.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Consider the number $3D5F2A_{16}$, converted as shown below using weighted sum of powers:

Digit	3_{16}	D_{16}	5_{16}	F_{16}	2_{16}	A_{16}
Position Value	16^5	16^4	16^3	16^2	16^1	16^0
Digit Value	3145728_{10}	851968_{10}	20480_{10}	3840_{10}	32_{10}	10_{10}

$$3D5F2A_{16} = 3145728_{10} + 851968_{10} + 20480_{10} + 3840_{10} + 32_{10} + 10_{10}$$

$$3D5F2A_{16} = 4022058_{10}$$

Converting Hexadecimal to Binary

Converting hex numbers to binary and vice versa is nice and easy, as each hex digit can be converted to a nibble. You may use the lookup table above, or your brain, to do the conversion. Using this technology, the hex number above ($3D5F2A_{16}$) could easily be converted to binary. Always remember to work right to left, and strip any leading zeros on the result (unless you want to show a specified number of bits in your result, regardless of what they are):

Hex	3_{16}	D_{16}	5_{16}	F_{16}	2_{16}	A_{16}
Binary	0011	1101	0101	1111	0010	1010

$$3D5F2A_{16} = 001111010101111100101010_2$$

or

$$3D5F2A_{16} = 1111010101111100101010_2$$

Converting Binary to Hexadecimal

Converting binary to hex requires that you work right to left 'snapping' the binary digits into nibbles, padding the left-most digits with zeros to fill the final nibble, if necessary.

For example, convert 1101010010101011010111_2 to hexadecimal:

Binary	1101010010101011010111 ₂					
Nibbler	0011	0101	0010	1010	1101	0111
Hexadecimal	3 ₁₆	5 ₁₆	2 ₁₆	A ₁₆	D ₁₆	7 ₁₆

$1101010010101011010111_2 = 352AD7_{16}$

8 Bit Arithmetic

You will principally be concerned with 8 bit numbers while coding. Mastery of all that is 8 bit needs to become part of your mental fabric – stat. Typically when you look at a binary or hexadecimal number, you assume it is unsigned. The fact that binary and hexadecimal numbers have no sign notation for 'negative' contributes to this. There are times when numbers need to be interpreted as signed, and binary numbers may be 2's complement coded to manage a signed value. Please note, before we get too far here, that there is no way to determine if a binary number is intended to be signed or unsigned – it is entirely up to context and interpretation. The S12XCPU has instructions that will assume that an operand is signed, and will interpret the binary number that way. These instructions are *fairly* clearly marked.

Binary numbers that are interpreted as being signed consider the most significant bit as contributing a negative value. This means that for an 8 bit number, the most significant bit will contribute $-128 (-2^7)$, if it is set:

Bit Pattern	Hex Value	Unsigned Decimal Value	Signed Decimal Value
%00000000	\$00	0	0
%10000000	\$80	128	-128
%11111111	\$FF	255	-1
%01111111	\$7F	127	127

From this, we can glean a couple of important points:

- Representation of -0 is not possible
- The MSB directly represents the sign of the number (but not as a fundamental flag)

Topic 3 – Interfacing With Internal and External Devices

Required supporting materials

- This Module and any supplementary material provided by the instructor
- Device documentation provided in the appendix of this CoursePack
- CNT MC9S12XDP512 Development Kit and 12 VDC Power Adapter
- USBDM Pod or BDM Pod and “A to B” USB Cable
- CodeWarrior

Rationale

Well-structured and documented code results in dependable and maintainable systems.

Expected Outcomes

The following course outcome will be addressed by this module:

Outcome #3: Interface with onboard, simple GPIO, and programmable devices.

As this course progresses, you will refine the basic skills and understanding of embedded systems through programming the 9S12X using ANSI C.

Connection Activity

Some devices are relatively easy to access or control. These devices require no internal programming, and often only require communication in one direction with no feedback. You’ve already read from a bank of switches and written to three LEDs. Another device on your board, the ICM7218 LED Display Driver, is similar in that it receives simple instructions through a GPIO port, and provides no feedback to the microcontroller.

Many microcontroller devices need, at some point, to send meaningful information to a computer or other communications-enabled device and/or receive meaningful information from such a device. This kind of activity often uses Asynchronous Serial Communication.

Microprocessors can manage a large number of devices and can transfer a large amount of data quickly because they use parallel communication arrangements. Consequently, many devices – such as memory ICs, banks of LEDs, and pixel array displays, have been designed to operate using parallel communication. However, the microprocessors embedded in most microcontrollers do not directly provide access to the address bus, the data bus, and the various control lines. Instead, designers must “recreate” the necessary interface lines using the general purpose I/O bus pins available on the controller.

Disclaimer

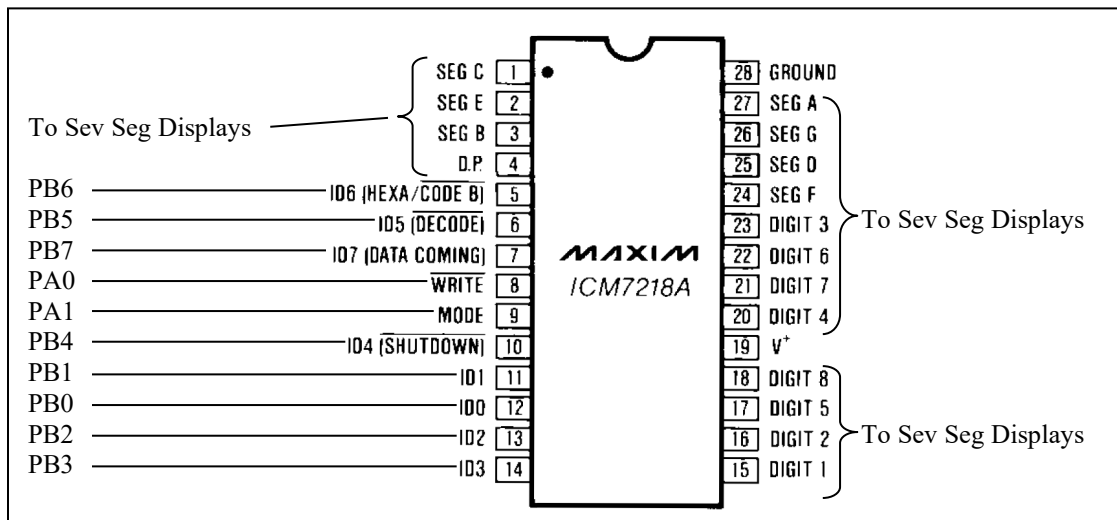
In earlier years in CNT, all of the programmatic control of the peripherals was done using variants of Assembly Language. Recently, the course has migrated to a heavier emphasis on ANSI C. Since the material developed in Assembly Language is still valid for the microcontroller you’re currently using, the associated material has not been removed. You may find it useful, either if you choose to write some code in Assembly Language or if you choose to use it as reference material to help you gain a broader understanding of what is being done in the ANSI C adaptations.

Interfacing the ICM7218A 8-Digit LED Display Driver

At this point it is difficult for you to display program output in a very meaningful way. The 8-digit LED seven-segment display driver would significantly add output capability to your programs. We will discuss this device now.

The ICM7218A is not a very complicated device, but it does require addressing and command codes to operate. This is more challenging than operating a simple indicator LED, but comes with the benefit of displaying far more useful information.

The first thing to note is how the device is connected to your 9S12X. The ICM7218A device requires 8 instruction/data connections, and 2 control signal connections. On your development board, the instruction/data connections have been tied to Port B, and the control signals have been tied to PA0 and PA1 of Port A. All communication to the ICM7218A device will occur through GPIO on these two ports.



The ICM7218A is able to interpret input in a number of ways, and may be commanded to update single digits or update multiple digits. The full operation of the device is beyond the scope of this document, but is something you are encouraged to investigate.

The easiest way to get output on the display is to write a command byte to the device (which contains a digit address) and then write out the data for the digit. Because the device is connected to the 9S12X through GPIO, you must manually produce the correct signals to have this happen. Before any signals may be generated, Ports A and B must first be configured correctly.

You will only be writing to (not reading from) the ICM7218A, so all port pins used should be configured as outputs. It is also important that the active low "/write" line of the device stay high until you actually write to the device. To protect against this, you should set the outputs on the port to HIGH before you set the data direction registers for the ports.

The code below is an S12XCPU Assembly Language initialization subroutine for the ICM7218A. To begin with, the two 9S12X ports attached to the device need to be set up as outputs. Good design procedure indicates that the data on the port should be set to a known condition that will have minimal effect on the connected device when the port is enabled as outputs, as shown in the following code.

```

;*****
;*          SevSeg_Init
;*
;*Regs affected:    none
;*
;*Sets up Port A for the Seven Segment Controller as control,
;*Sets up Port B for the Seven Segment Controller as data
;*  only b1 and b0 of Port A are used
;*  clears all eight digits using 8-digit sequential commands
;*
;*****

SevSeg_Init:
BSET    PORTA,%00000011    ;resting state:  mode and /write HIGH
BSET    DDRA, %00000011    ;make A1 (mode) and A0 (/write) outputs

MOVWB   #%11111111,DDRB    ;make all PORTB outputs

;JSR    SevSeg_BIA11

RTS

```

Note the use of “BSET” for DDRA and PORTA. By only affecting two bits in this register, the rest of PORTA is left untouched, which means it can be used for other purposes if desired.

This routine will correctly initialize the ports for communication with the ICM7218A device. However, there’s no guarantee as to what will appear on the display digits when the device is first accessed, so the last line in the header is a lie at this point – this line could be included in the actual code once a subroutine called “SevSeg_BIA11” was written.

The following pages will provide you with a working knowledge of the operation of the ICM7218A seven-segment display driver, so that you can write code to display values on the seven segment display array.

The Maxim ICM7218A data sheet can be found here:

<http://datasheets.maximintegrated.com/en/ds/ICM7218-ICM7228.pdf>

Here are the most important programming-related tables from this data sheet.

ICM7218A Programming Tables

8 Digit LED Display Driver

ICM7218/ICM7228

Table 1. Input Definitions, ICM7218A and ICM7218B

Note: Pin Configurations for the ICM7218A/B are shown on last page.

INPUT	PIN	STATE	FUNCTION
WRITE	8	High Low	Input Not Loaded Into Memory Input Loaded Into Memory
MODE	9	High Low	Loads Control Word on WR Loads Input Data on WR
ID0-ID2, DIGIT ADDRESS	12, 11, 13	High Low	Loads "one" Loads "zero"
ID3, BANK SELECT	14	High Low	Select RAM Bank A (Hex or Code B Select RAM Bank B Data only)
ID4, SHUTDOWN (MODE High)	10	High Low	Normal Operation Shutdown
ID5, DECODE/NO DECODE (MODE High)	6	High Low	No Decode Decode
ID6, HEX/CODE B (MODE High)	5	High Low	Hexadecimal Decoding Code B Decoding
ID7, DATA COMING (MODE High)	7	High Low	Data Coming (control word) No Data Coming (control word)
ID0-ID7, INPUT DATA (MODE Low)	5-7, 10-14	High Low	Loads "one" (Note 1) Loads "zero" (Note 1)

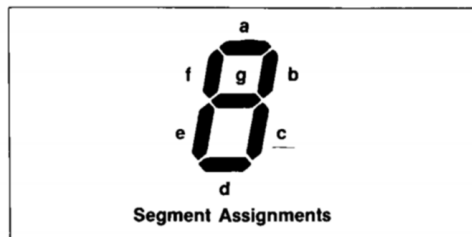
Note 1: A "zero" or low level on ID7 turns ON the decimal point. In the NO DECODE mode, a "one" or high input turns ON the corresponding segment, except for the decimal point which is turned OFF by a high level on ID7.

Getting the device to display information can be done in a variety of ways. You have the ability to turn on individual segments of any of the 8 digits ("No Decode" – the digits map as shown below), but the easiest thing to do is have the device decode the input as Hex ("Decode" -> "Hexadecimal Decoding"). Another option is "Decode"->"Code B Decoding", which produces a different set of characters including *H, E, L, P* and *Blank* but not the top six Hex numbers, *A - F*.

ICM7218/ICM7228

ID3	ID2	ID1	ID0	HEXADECIMAL	CODE B
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	A	-
1	0	1	1	b	E
1	1	0	0	C	H
1	1	0	1	d	L
1	1	1	0	E	P
1	1	1	1	F	(Blank)

Figure 7. Display Font



Microprocessor Interface, ICM7218C and ICM7218D

Data Input	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
Controlled Segment	Decimal Point	A	B	C	E	G	F	D

Sending Data to the ICM7218A

To get a digit on the display, you must send a control byte to the device that describes the digit address and mode of operation. After this you must send the byte value for the digit.

The device will require that the /write line to be lowered then raised to latch the data. This is referred to as "strobing" the /write line. The mode control signal will determine if the byte being written is a control byte or a data byte. Here are two timing diagrams from the datasheet that show how this works:

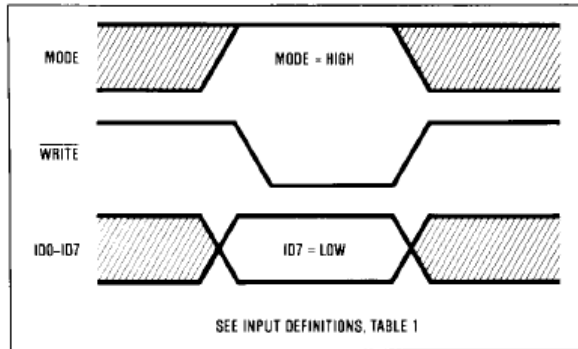


Figure 4. Control Word Update Timing—ICM7218A/B

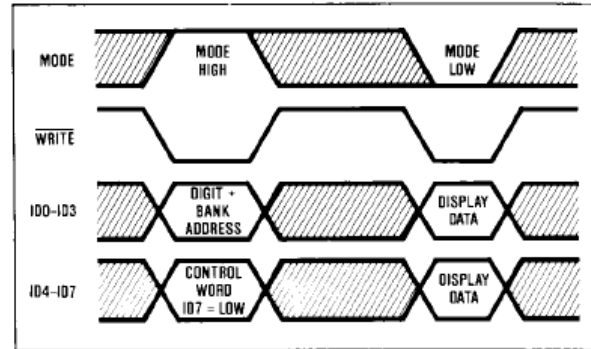


Figure 5. Single Digit Update Timing—ICM7218A/B

The procedure for sending a single control command (Figure 4) is as follows:

- Control:
 - Present control byte on GPIO data lines (Port B)
 - Set mode HIGH and write LOW (indicate that you are writing a control byte)
 - Set write HIGH (latches the control byte into the device)

The procedure for sending a single digit (Figure 5) requires both a control byte and a data byte, as follows:

- Control (containing the address and other control bits):
 - Present control byte on GPIO data lines (Port B)
 - Set mode HIGH and write LOW (indicate that you are writing a control byte)
 - Set write HIGH (latches the control byte into the device)
- Data:
 - Present data on GPIO data lines (Port B)
 - Set mode LOW and write LOW (indicate that you are writing a data byte)
 - Set write HIGH (latches the data byte into the device)

The bits in the control byte affect the behavior of the device, and, in the mode we're using, are also used to set the address of the digit being written to. (A note of caution: other manufacturers make versions of this controller that do not allow individual addressing of the digits – in these, all eight digits must be written in a single sequence each time the display is updated. The discussion in this course material is specific to the Maxim part.)

Locate the table of Input Definitions (shown previously, from page 5 in the current data sheet). For standard writing of a hex character, you are interested in Hex mode, Bank A, normal operation. Using the table of Input Definitions, verify each of the bits in the control byte shown in the routine on the following page. Add an ANSI C version of this routine to your library.

Seven Segment Display Library Components

The following S12XCPU Assembly Language version of the initialization routine is included as a reference that can be used as a jumping point for all of the library components you will need to write for your ANSI C library. One thing you may find particularly useful from this version of the routine is the system used for bitwise commenting the command byte.

```

;*****
;*      SevSeg_Char
;*
;* Regs affected: none
;*
;* Accepts a hex character in Accumulator A and
;* a location (0 to 7) in Accumulator B and places the character
;*
;* The routine expects the user to know the device limits,
;* and assumes that SevSeg_Init has been run already.
;*
;*****
SevSeg_Char:
    PSHA
    PSHB

    ANDB    #00000111    ;clean up address - only three bits valid
    ;-----"No data coming" means single digit mode
    ;-----Hexadecimal Decoding
    ;-----decode mode
    ;-----not shutdown
    ;-----memory bank A
    ;----->
    ;-----don't mess with the address
    ;-----
    ORAB    #01011000    ;add control: hex, decode, no SD, Bank A

    ANDA    #00001111    ;prep the data digit: clear upper nibble
    ORAA    #10000000    ;... no decimal point

    STAB    PORTB        ;control byte placed on the data bus
    BCLR    PORTA,%00000001 ;mode still high (control), /write low
    BSET    PORTA,%00000011 ;resting state: mode and /write HIGH

    STAA    PORTB        ;data byte placed on the data bus
    BCLR    PORTA,%00000011 ;mode low (data), /write low
    BSET    PORTA,%00000011 ;resting state: mode and /write HIGH

    PULB
    PULA
    RTS

```

Note how "BSET" and "BCLR" were used so as to preserve the state of the upper six bits of PORTA, just in case this port is being used for some other purpose in your program. In ANSI C, you will need to use AND or OR operators to achieve the same functionality.

Seven-segment Display Control Using ANSI C

You are going to need three program files in a new project: a library header file (SevSeg_Lib.h), an uncompiled C library file (SevSeg_Lib.c), and the main program file within the project itself (its default name will be main.c).

SevSeg_Lib.h

The header file will probably be supplied to you by your instructor.

```
//Seven Segment Display Controller Library
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//June 2015

void SevSeg_Init(void);
void SevSeg_Char(unsigned char, unsigned char); //digit address and character
void SevSeg_B1Char(unsigned char); //digit address
void SevSeg_B1All(void);
void SevSeg_dChar(unsigned char, unsigned char); //digit address and character
void SevSeg_Top4(unsigned int); //four chars compressed as four nibbles in an int
void SevSeg_Bot4(unsigned int); //four chars compressed as four nibbles in an int
void SevSeg_Cust(unsigned char, unsigned char); //digit address and selected segments
```

SevSeg_Lib.c

Now comes the task of creating the actual library and the functions to match the prototypes in the header file. The library file needs to be linked to the files it will be using, which include the ones set up when a project is created: the ANSI C library hidef.h and the derivative.h file set up to provide the labels for all the ports in our particular 9S12X microcontroller. Also, if any of the functions call other functions in the library, it needs to be linked to itself. Notice again the different syntax for including a standard compiled ANSI C library and for including an uncompiled library. As previously discussed, you will need to add both the .h and .c files for the uncompiled libraries to the project, whereas the compiler will find the standard library without our help.

```
//Seven Segment Display Controller Library
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//June 2015

#include <hidef.h>
#include "derivative.h"
#include "SevSeg_Lib.h"
```

The first function you'll need is an initialization routine that does what the Assembly Language version did. In this case, we can do practically a line-by-line translation into C. This will not always be the case when moving from Assembly Language to C, as the thought processes involved in programming for the two languages is fundamentally different. Sometimes, Assembly Language provides the most succinct and efficient result, whereas other times the structured nature of C will allow the programmer to easily control program flow in ways that would be difficult to achieve in Assembly Language. Here's a suitable initialization function, but with the screen blanking function disabled until we build it.

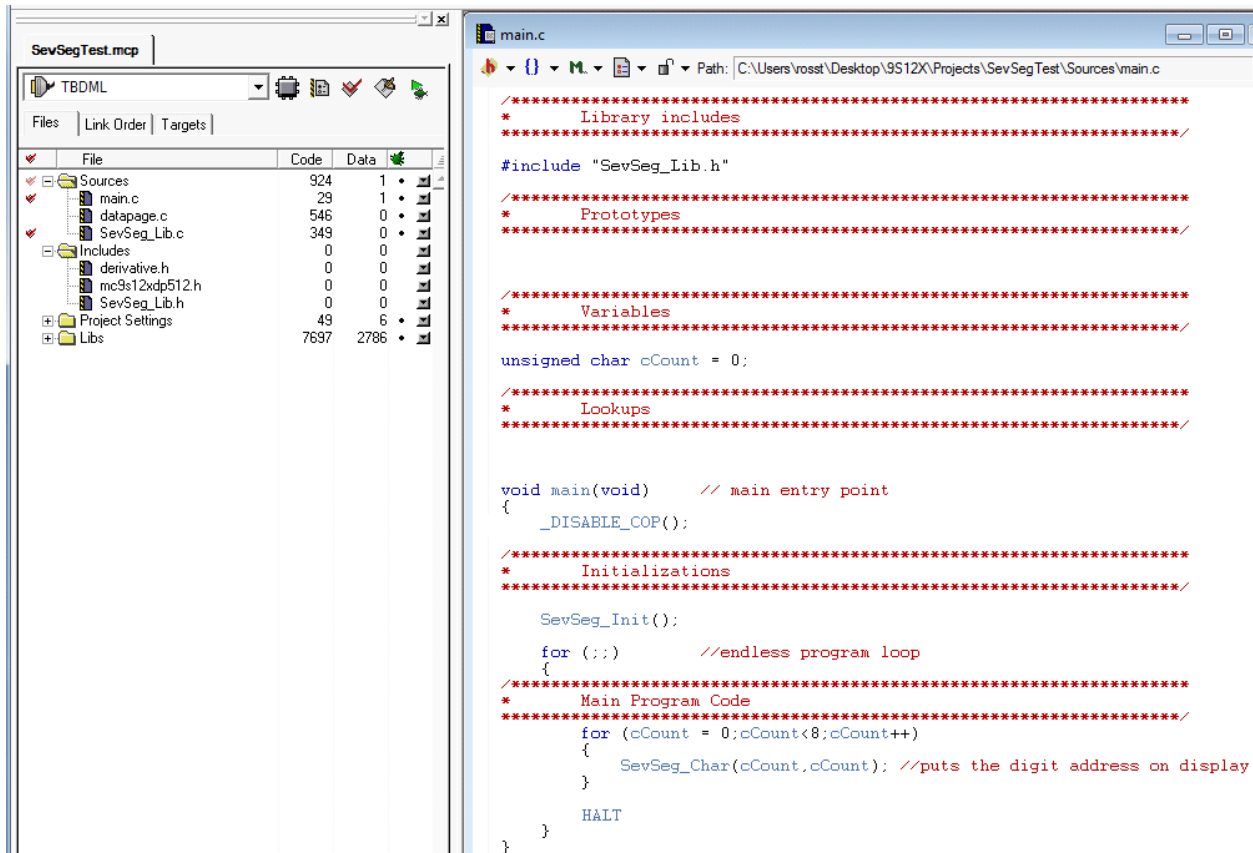
```
void SevSeg_Init(void)
{
    PORTA |= 0b00000011; //preset control lines high
    DDRA |= 0b00000011; //A0:1 outputs
    DDRB = 0b11111111; //all PORTB outputs

    // SevSeg_B1All();
}
```

Notice the use of bitwise OR commands for working with PORTA. Again, the upper six bits of this register are not being used for the seven-segment display, so, if we are careful not to mess with them in our routines, they will be available for other applications if so needed. If

we had written an entire byte to PORTA and DDRA (i.e. PORTA = 0b00000011 and DDRA = 0b00000011), we would have messed up any other activity on the upper six bits.

This is a good time to create a project that you can use for testing the library you are working on. Follow the steps outlined previously, and create a project called something like "SevSegTest". The following screen capture shows a sample "main.c" file, but it also shows what needs to be done to include the library you're working on. (You won't be able to run this code yet, as you will need to create SevSeg_Char first. At least try your initialization routine, which, without the blanking function, will probably display garbage.)



To include the library you're building you need to do three things:

1. In the browser to the left of the screen, search for and add the library's ".c" file under "Sources".
2. In the browser, add the library's ".h" file under "Includes".
3. In the main file, put in the #include statement as shown.

In the main file, notice that a counter variable has been declared and initialized in the "Variables" section.

SevSeg_Init() is called in the "Initializations" section, outside of the main loop for the program. You only want to initialize the ports once – not each time through the loop.

Since this entire code was only intended to be run through once, there's a HALT command to stop execution at the end. This is not usually a useful command in a microcontroller program, but helps us with testing or troubleshooting functions as we program.

It's up to you now to build the rest of the functions in this library. For each item, make sure you adhere to the prototype in the .h file. Here's a sneak peek at a suitable function for blanking a single character, passed as an address between 0 and 7.

```

//requires a digit address (0 to 7) to blank
void SevSeg_BlChar(unsigned char cDigit)
{
    cDigit &= 0b00000111; //clean up in case of error
    cDigit |= 0b01111000; //single digit, (don't care), no decode, no SD.
    PORTB = cDigit;
    PORTA &= 0b11111110; //Mode stays high, strobe /Write
    PORTA |= 0b00000011; //back to resting state - Mode and /Write high

    RTB = 0b10000000; //turn off all segments and dp
    RTA &= 0b11111100; //Mode low for data, strobe /Write
    RTA |= 0b00000011; //back to resting state - Mode and /Write high
}

```

Once you've created the above function for blanking a single character, you can use it to blank all eight digits. A good way to do this, using the ANSI C way of thinking, is shown below.

```

//blanks the display using the SevSeg_BlChar routine in an 8-cycle loop
void SevSeg_BlAll(void)
{
    unsigned char cCount;
    for(cCount=0;cCount<8;cCount++)
    {
        SevSeg_BlChar(cCount);
    }
}

```

Go back to your initialization function and enable the line that calls SevSeg_BlAll(). You should now be able to display a completely blank set of eight digits on the seven segment display. (I know, I know, it's pretty exciting to you, but the average person on the street won't understand, so don't rush out looking for someone to show this to!)

Once you've created a function for displaying a single digit at a specific location, you should be able to run the code shown on the previous page. Then you should be able to finish off the rest of the items required for this library, as shown in the header file.

Hopefully, you can see how what you've learned in your C# courses transfers to writing code in ANSI C, which was the original language from which all the various C-family derivatives grew. With a basic knowledge of the operation of the 9S12XDP512, a chance to work at the machine level using S12XCPU Assembly Language, and a fair bit of experience programming in C#, you should soon be able to make your microcontroller development kit carry out some fairly sophisticated activities.

Binary-Coded Decimal Representation and Manipulation

At this point, you're probably aware of two conflicting realities: Your microcontroller only talks binary, which we often compress into hexadecimal for easier viewing; and the bulk of humanity works with decimal numbers.

The cross-over between these two systems is something called Binary-Coded Decimal (BCD), a system that uses hexadecimal (actually binary) coding to represent decimal values. It's important to remember that BCD is a code, not a real number system. It's a way to use hexadecimal values to represent decimal numbers. Real math must be done by your microcontroller using hexadecimal ("real numbers"). BCD is only for display purposes.

In BCD, the upper six hexadecimal values (ABCDEF) are not be used, since they're not a part of the decimal number system. Instead, after 0123456789, the sequence must roll over to 10. The microprocessor will consider this to be 10₁₆ because it only does binary (shown here as hexadecimal). But it looks like 10₁₀ to the rest of us, and, properly used, would be the BCD representation of 10₁₀. For clarity, we'll use the notation 10_{BCD}.

BCD's only purpose is to display values in a form humans are comfortable with. Just because they could, the designers of microprocessors made by Freescale have included the DAA (Decimal Adjust Accumulator A) command that allows you to do simple addition of BCD

values. This author's recommendation is that you let your microcontroller do all its work in hexadecimal (true numeric values), then convert the results to BCD when needed for a human interface. Other instructors may feel differently – humour them if necessary.

Converting Hexadecimal Values to BCD

In a previous course, you learned the "Division with Remainders" method of converting from numbers of any base to decimal. Division with Remainders involves dividing the number repeatedly by 10 (i.e. A_{16}), each time concatenating the remainders together, starting at the right and moving left. Here's an example:

Convert $123F_{16}$ to decimal.

```

    0, R:4
  A) 4, R:6
    2E, R:7
  A) 1D3, R:1
  A) 123F
  
```

Final result: $123F_{16} = 4671_{10}$, or for human display, 4671_{BCD}

If we used a microcontroller to hold or display this value, it would hold it as 4671_{16} , which is certainly not 4671_{10} , nor is it intended to be thought of as 4671_{16} . That's why we use the notation 4671_{BCD} , and why we only use this to represent the value as a coded representation that humans are comfortable with.

Also in your previous course, you developed and used a Hexadecimal-to-BCD converter using S12XCPU Assembly Language. Here's one version of that routine:

```

*****
* Hex2BCD5
*
* 4-nibble hex to 5-digit BCD converter
*
* Hexadecimal value arrives in D. BCD returned in X and D
* with the most significant digit in X
*
* Registers affected: X and D only
*
*****

Hex2BCD5:
    PSHY                ;Save contents of Y for later
    LDY                #5                ;Downcounter for a 5-digit loop
Hex2BCD5Loop:
    LDX                #10              ;Division with remainders method: /10
    IDIV               ;Answer is in X, remainder in D (i.e. B)
    PSHB              ;Store remainder from "right to left"
    TFR X,D           ;Put answer in D for next division
    DBNE Y,Hex2BCD5Loop ;Decrement the counter
                        ;Not at the end? Loop again

    CLRB              ;now have 5 digits on the stack: need 6...
    PSHB              ;so put a 0 on the stack

    PULX              ;pull first two digits (0 and 10,000's)
    PULA              ;pull 1000's off stack...
    LSLA              ;shift it into the high nibble...
    LSLA
    LSLA
    LSLA
    LSLA
    ADDA 0,SP         ;...and add the 100's, so 1000's and 100's are in A

    INS               ;move up to beginning of 10's
    PULB              ;pull 10's off the stack...
    LSLB              ;shift it into the high nibble...
    LSLB
    LSLB
    LSLB
    ADDB 0,SP         ;...and add the 1's, so 10's and 1's are in B

    INS               ;increment the stack so it's ready for a pull
    PULY              ;restore original value of Y
    RTS               ;and go back to the main program with answer in X and D
  
```

Notice that this routine does the entire hexadecimal to BCD routine in the six lines of the "Hex2BCD5Loop": the rest of the subroutine is concerned with compressing the resulting BCD representation into the D Accumulator, with the fifth digit in the X register.

Using ANSI C, all of the details of packing and returning the coded characters can be handled by the cross-compiler. Also, since we're no longer directly concerned with the sizes of the accumulators, we can change our parameters.

In the Assembly Language version, we chose to return a five-character value because the largest number we could send, in a sixteen-bit register, was $FFFF_{16}$, or $65,535_{BCD}$, which is five characters. Of course, that's difficult to display on the seven-segment displays we've got, at least in the way they're configured. On your own, you could build a board that has the displays side-by-side, in which case displaying five digits would make sense.

Now that we're no longer directly tied to the size of the accumulators, we've got some decisions to make:

- The way our board is configured, it probably doesn't make sense for us to work with numbers larger than 9999_{BCD} , or $207F_{16}$, if the target is the seven-segment display.
- If we want a generally-useful hexadecimal-to-BCD converter, we could choose to work with a "Long" data type, in which case we could return eight characters and raise the size of the number we're working with to $99,999,999_{BCD}$, or $5F5E0FF_{16}$. Handling all the digits returned by the function would then be up to the programmer.
- If the target is some device that works in ASCII, such as the LCD display, a computer acting as a "dumb terminal", or a Raspberry Pi, you might want to display your values as floating-point numbers converted to ASCII strings, formatted using the "sprintf" function available in the ANSI C "stdio.h" library.

For simplicity, let's work first with simple 4-digit converters suited to use with the seven-segment display, which is looking for actual numbers (i.e. 0 to F in hexadecimal or 0 to 9 in BCD), not ASCII (which would be 0x30 to 0x39 for both hexadecimal and BCD, and 0x41 to 0x46 for the rest of the hexadecimal numbers).

You are going to need a library for various miscellaneous functions. Here's a header file for this library, called "Misc_Lib.h", and it shows you the functions you will eventually add to the source-code library. It's best that you comment out the prototypes that you haven't developed code for, and enable them when you're ready to use them.

Misc_Lib.h

```
//Miscellaneous generally-useful routines
//Processor: MC9S12XDF512
//Crystal: 16 MHz
//by P Ross Taylor
//September 2016

//Binary-Coded Decimal conversion routines

unsigned int HexToBCD(unsigned int); //integer math; result is BCD - not converted to ASCII; make it 4-digits for sev-seg display
unsigned int BCDToHex(unsigned int); //integer math; requires BCD - not ASCII equivalent; make it 4-digits to complement HexToBCD

//ASCII-Code handling routines

unsigned char ToUpper(unsigned char);
unsigned char ToLower(unsigned char);
unsigned char HexToASCII(unsigned char); //single character converter
unsigned char ASCIIToHex(unsigned char); //single character converter

//9S12X simple timer routines

void TimInit125ns(void);
void TimInit8us(void);
void Sleep_ms(unsigned int); //requires TimInit8us setup; blocking delay
```

Notice that the routines come in three groups: BCD, ASCII, and Timer. For now, we'll just do the simple BCD-related functions.

HexToBCD

In a source-code file named "Misc_Lib.c", you will want to create a function that does the Division with Remainders routine we used in the S12XCPU Assembly Language routine previously. The code snippet below shows the heart of this routine, done using division and modulus operations. Recall that the modulus operation (%) provides the remainder of an integer division, whereas the division operation provides the integer result with no rounding.

Notice that there are three local variables: *iBCDOut*, *cCount*, and *iPow*. Declare these and initialize them within the function, but ahead of any active code.

Prior to this code snippet, *iPow* was initialized to 1 and *iBCDOut* was initialized to 0. So, first time through, the remainder is multiplied by 1. The second time through, the remainder is multiplied by 16 and added to the previous remainder; in other words, it is put in the next most significant nibble of the result. The next time through, the remainder is multiplied by 256 and added to the result, and the last time through it is multiplied by 4096 and added to the result. Consequently, the BCD characters end up in the required four nibbles of the final result, starting with the least significant character and ending with the most significant character.

```
for (cCount=0;cCount<4;cCount++) //Four digits in a loop
{
    iBCDOut+=(iHexIn%10)*iPow; //Division-remainder conversion
    iHexIn/=10;
    iPow*=16; //16^cCount
}
```

In your function, you should also provide some sort of error trapping. If the number sent to the function is greater than 9,999₁₀, it's probably best to return some recognizable value that's out of range rather than some gibbled half-BCD/half-hex result. If we return FFFF, it should be clear that this represents an error, as BCD does not include the character 'F'.

BCDToHex

As with HexToBCD, we'll create a simple routine to convert integers up to 9,999_{BCD} to their equivalent true number values in hexadecimal so that the microcontroller can use them for numeric calculations. In your previous course, you worked with an S12XCPU version of this routine, as shown below. This routine is pretty complex, given that we needed to manipulate the digits individually, and, to avoid variables, we used the stack for storage. One item we won't be able to duplicate easily in C is indicating a valid value using CARRY.

```

*****
*      BCD2Hex4
*
*      4-digit BCD to hex converter with error checking: not BCD = 0000 and Carry
*
*      BCD value arrives in D, hex returned in D
*
*      Registers affected: D only
*
*****

BCD2Hex4:
    PSHX
    PSHY

    TFR    D,X          ;for now, hold BCD in X

BCDCheck:
    ANDB  #%00001111   ;check for valid 1's digit
    CMPB  #$09
    BHI   ErrorBCDtoHex4

    ANDA  #%00001111   ;check for valid 100's digit
    CMPA  #$09
    BHI   ErrorBCDtoHex4

    TFR    X,D          ;get original BCD again

    ANDB  #%11110000   ;check for valid 10's digit
    CMPB  #$90
    BHI   ErrorBCDtoHex4

    ANDA  #%11110000   ;check for valid 1000's digit
    CMPA  #$90
    BHI   ErrorBCDtoHex4
    BRA   BCDConvert   ;if all are valid, continue on ...

ErrorBCDtoHex4:
    LDD   #0            ;if not, return zero ...
    SEC   ;set Carry Flag as error indicator ...
    BRA   FinishBCDtoHex4 ;and leave

BCDConvert:
    TFR    X,D          ;select 1's digit
    ANDB  #%00001111   ;put it on the stack ...
    PSHB  ;and pad it with a zero for 16 bit add ...
    CLRB  ;on the stack

    TFR    X,D          ;grab the original ...
    LSRB  ;shift over the 10's digit ...
    LSRB
    LSRB
    LSRB
    PSHB  ;put it on the stack ...
    CLRB  ;and pad it with a zero for 16 bit add ...
    PSHB  ;on the stack

    TFR    X,D          ;grab the original again ...
    ANDA  #%00001111   ;get the 100's digit ...
    PSHA  ;put it on the stack ...
    CLRA  ;and pad it with a zero for 16 bit add ...
    PSHA  ;on the stack

    TFR    X,D          ;grab the original one last time ...
    LSRA  ;shift over the 1000's digit ...
    LSRA
    LSRA
    LSRA
    TAB  ;move it to the low byte of D ...
    CLRA  ;blank the high byte of D to start with D = 1000's digit

    LDX   #3           ;FOR loop downcounter for 3 passes
                    ;algorithm: hex=(((This*10)+Huns)*10)+Tens*10+Ones

WeightedLoop:
    LDY   #10          ;going to multiply current value by 10 ...
    EMUL ;useful part of answer is in D ...
    ADDD  0,SP         ;add in the next digit
    INS  ;move the stack pointer past the used digit
    INS
    DENE  X,WeightedLoop ;not the last digit? Loop again

    CLC   ;Carry Flag cleared for good answer

FinishBCDtoHex4:
    PULY
    PULX
    RTS

```

This routine doesn't need error trapping for the size of the number, because the hexadecimal value will always take the same or fewer nibbles. However, it does need a trap for invalid characters (A through F, since these are never valid in a BCD representation of a number). Since there's no particularly easy way, like checking the CARRY bit, to indicate an invalid BCD value, we'll simply return a value that is out of range and leave it up to the programmer to decide what to do with that.. Again, FFFF is a good choice, since the biggest return value is 270F. The programmer can be left to decide what to do with errors.

Since the logic and mathematics behind this routine can be a bit complicated, a fully-functional version of the code is shown below. Before you simply copy this routine, make sure you understand how it works. Other instructors may ask you to use or develop slightly different code to perform the same task – being nice to them will be to your advantage!

```
//BCDToHex handles numbers up to 4 digits (9999), returns Hex as 2 bytes (int)
//does not require the math.h C library
unsigned int BCDToHex(unsigned int iBCDIn)
{
    unsigned int iHexOut=0;
    unsigned char cDigit,cCount;
    unsigned int iPow=1;

    for (cCount=0;cCount<4;cCount++)
    {
        cDigit=iBCDIn%0x10;    //locate right-most digit with a MOD 16
        if (cDigit<10)        //not valid BCD (0-9)? might as well quit!
        {
            iHexOut+=cDigit*iPow; //multiply by correct power of 10 and add
            iBCDIn/=0x10;        //get ready for next digit by dividing by 16
            iPow*=10;            //10^bcount
        }else
        {
            iHexOut=0xFFFF;    //error? return FFFF
            break;              //... and break out of loop
        }
    }
    return iHexOut;
}
```

First of all, notice how much more compact the ANSI C version of this routine is than the S12XCPU version! To a great extent, that's because we can declare and use local variables, so we don't need to manipulate the stack. (Incidentally, the C compiler may ignore your declared variables and use the stack instead – you'll never know until you disassemble the code to see what it did or try to trace the variables, which won't be listed if the compiler chooses not to use them. Some of us old-guard programmers find that mildly disturbing.)

Notice that we use 16, (i.e. 0x10), in our division and modulus calculations. That's because the microprocessor only really works in binary (i.e. hexadecimal) values, so even if we picture a BCD value as a real number, the microprocessor thinks of it as hexadecimal. So, to correctly locate and identify the characters, we need to work with them in groups of four bits, or 16's, not 10's. For example, consider $1264_{\text{BCD}}: 0x1264 / 16 = 0x126$ with a remainder of 4. By doing so, we identify the lowest digit, and preserve the upper three digits for the next stage of the calculation. If we tried $0x1264 / 10$, we'd get $0x1D6$ with a remainder of 8, which is no use to us at all.

Once we identify the characters, we multiply them by powers of ten to make them into real numbers, which we add together to get the final true number. In the above example, we'd get $0x4F0$, which is – you guessed it – 1264_{10} .

Each incoming character is checked to see if it is valid for BCD. If any invalid character is encountered, we break out of the loop and return FFFF.

Switch Management

Switches, as user interface devices, have two complicating features:

- Long Activation Times
- Bounce

Consider a computer keyboard: When you press a key, the mechanical action of that key results in a number of connects and disconnects while the spring mechanism settles down. Once the key is pressed, your finger remains on that key for a certain period of time, which seems quite short to you but could be thousands, even millions of cycles of the computer's clock. How does the keyboard controller "know" that you only intended one instance of that particular keystroke, even though it's aware of multiple quick changes of state followed by thousands of readings of the switch's new condition? Let's work through these problems.

Detecting Switch Change of State

There are basically four ways to write a program to respond to switches.

- The first situation is one in which it doesn't matter if the switch condition is read thousands or millions of times – the output directly relates to the current condition of the switch at all times.

For example, you could have a program that turns the RED LED on as long as the LEFT switch button is pressed. "while (LEFT()) RedOn;" is pseudo-code for this.

The other three are ways to make it so that your program will respond just once to each button press or change of switch condition.

- One way is to have the program branch away from the routine that's checking for the switch as soon as the switch change is detected, thereby ignoring the condition of the switch until it is needed again. State Machines use this technique, staying in a particular state until a transition condition occurs to go to a new state. This works well in menu-driven applications, too, where selecting an item from a menu sends the microprocessor off on a particular task that doesn't check the switch again.

```
if((PT1AD1&0b00010000)!=0) //UP: redirects to another routine if pressed
{
    UpHandler();
}
```

- A second way is what's called a *blocking routine*, where the program is held up until the switch condition changes back to its original condition. For example, the switch may normally be open. When it is closed, the program executes the desired action, then enters a loop, waiting for the switch to be released before it continues on to other commands. This blocking action may or may not be an issue. If you have other things that the program should be doing, holding it up waiting for a switch to be released is a bad thing; but if your program has nothing better to do than wait for the switch to be released, blocking isn't a problem.

```
if(PT1AD1&0b00001000) //LEFT switch pressed?
{
    SevSeg_Top4(++iCount);
    while(PT1AD1&0b00001000); //wait for LEFT release: blocks program
}
```

A variant of this which is sometimes useful is to wait for the switch to be released before executing the code. For most applications, this feels odd, because the action doesn't happen when you press the button – only when you release it. You're familiar with one application of this: touch screen item selection. With a touch screen, you can put your finger on an icon or control on the screen, but it doesn't respond until you lift your finger. This allows you to change your mind – if you decide you don't want to do what you've just touched, you can move your finger away before lifting, and the originally-selected action doesn't happen. This would

not be a good thing to do with an emergency shutoff switch, though – you want the equipment to stop as soon as you press the switch, not when your unconscious body finally falls away, releasing the switch!

```
if(PT1AD1&0b00000010) //RIGHT switch pressed?
{
  while(PT1AD1&0b00000010); //wait for RIGHT release: blocks program
  SevSeg_Top4(++iCount);
}
```

Remember that either of these will “block”, or hold up the processor, which may not be acceptable. Choose wisely!

- The non-blocking way to handle this is to use memory (i.e. a variable) to keep track of the previous condition of the switch. This technique is the best for continuous loops that need to monitor a switch or a set of switches continuously, such as in a control system or in something like a keypad or keyboard entry system for a calculator or computer. Here’s a typical sequence:
 - With the switch open, the variable is cleared to indicate that the switch has not been pressed in the recent past.
 - If at some point the switch is closed, the program compares the current condition to the previous condition, detects a difference, records the new condition by setting the variable, and provides an indication to the main program that the switch condition has changed.
 - The next time through, if the switch is still closed, the current condition will be the same as the previous condition, so the routine will report no change, and therefore the program can ignore the switch.
 - Once the switch is released, the difference will be detected, the new condition will be stored (i.e. the variable will be cleared), and the main program will be notified that the switch has been released. The program can be set up either to respond to this change or to ignore it (which is probably the most likely situation).
 - Next time through, the variable and the condition of the switch will be the same (both cleared), so no change will be reported to the main program.

```

/*****
*   Variables
*****/

char cSwNew;
char cSwState=0;

/*****
*   Main Program Code
*****/

cSwNew=PT1AD1&0b00000100; //DOWN: read the current condition of just DOWN
if (cSwNew!=cSwState) //only enters if a change in DOWN happens
{
  cSwState=cSwNew; //store new DOWN switch condition
  if ((cSwState&0b00000100)!=0) //means change is a PRESS -- ignore RELEASE
  {
    SevSeg_Top4(++iCount);
  }
}
```

Note: If you need to keep track of the states of a number of switches, read them all at once and, if there’s a change, store all of them in the switch state variable. Your code for the above situation, then, would look like this:

```

/*****
*   Main Program Code
*****/

cSwNew=PT1AD1&0b00011111; //Read the current condition of all switches
if (cSwNew!=cSwState) //only enters if a change in the switches happens
{
  cSwState=cSwNew; //store new switch conditions
  if ((cSwState&0b00000100)!=0) //means change is DOWN PRESSED -- ignore RELEASE
  {
    SevSeg_Top4(++iCount);
  }
}
```

Other switches would then be checked using their own “if” statements (or a switch – case) inside the main “if” statement.

Debouncing

If you've entered and tried the previous code snippets, you've probably noticed that the displayed count sometimes skips ahead by one or two when the switch is pressed. This is due to switch bounce, which we will now address.

The simplest form of debouncing involves detecting a change of switch state, then ignoring all subsequent changes for a period of time that's long enough to pretty much ensure that the switch has reached a steady state.

A slightly more reliable debounce sequence involves waiting for a short period of time, then checking to see if the switch is still in the new condition. If not, it must be bouncing – store the condition, wait for another short period of time, and check again. Once the state is consistent from one loop to the next, assume that the switch is stable and continue on.

These two types of debouncing both require a blocking loop – the program is held up in a timing loop while we wait for the switch to settle down. It is possible to design a non-blocking debounce routine which continues to run the main program while it waits for the switch to stabilize, but we shouldn't need to get that complicated in this course. The amount of time we spend in the debounce routine is so small (on the order of 10 ms) that it probably won't affect the routines we're creating.

This is a good time to make a library of switch and LED-related functions to link into our program. The following is the contents of a header file that your instructor will probably make available to you in one form or another.

```
//Switches and LEDs
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//June 2015

void SwLED_Init(void); //LEDs as outputs, Switches as inputs, dig in enabled
void LED_On(char); //accepts R, G, Y, A (for all)
void LED_Off(char); //accepts R, G, Y, A (for all)
void LED_Tog(char); //accepts R, G, Y, A (for all), and toggles the condition of the LED(s) indicated
char Sw_Ck(void); //returns debounced condition of all switches in a byte, LED values = 0
```

All but the last item in this list should be relatively easy for you to create. (Your instructor will likely ask you to complete these items.) The **SwCk()** routine, which returns a debounced version of the present conditions of all five of the switches, is provided below.

Add comments!

SwCk() Debounced Switch Routine

```
char SwCk(void)
{
    char cSample1=1;
    char cSample2=0;

    while(cSample1!=cSample2)
    {
        cSample1=PT1AD1&0b00011111;
        asm LDX    #26667;        /* 26667 x 3 cycles x 125 ns = 10 ms */
        asm DBNE   X,*;
        cSample2=PT1AD1&0b00011111;
    }
    return cSample1;
}
```

In the "main" program, you will need a variable to keep track of the previous condition of the switches, as in the example on the previous page. In that example, reading PT1AD1 into cSwNew would be replaced by a call to SwCk().

In Moodle, there should be a document on switch management by Simon Walker, as well.

Parallel Interfaces: Get On the Bus

For high-speed communication over short distances, designers prefer to use parallel interfaces. These interfaces provide one conductor per bit, and deliver all bits in a particular piece of information simultaneously.

Early microprocessors had 4-bit busses, and could communicate the four bits in a nibble simultaneously. Later, 8-bit busses were introduced, transmitting whole bytes. Since then, microprocessors have gone to 16-bit busses, then 32-bit busses, and now to 64-bit busses in an attempt to keep up with the growing speed requirements in the computer market.

The microprocessor at the heart of the 9S12X microcontroller uses a 16-bit bus. In fact, it uses two 16-bit busses: one for data, and one for addresses.

Data Bus

The data bus carries information between two devices, and is typically bidirectional. In other words, data can be sent to the device and data can be received from the device. Some specialized devices require only one of these directions. All bussed devices in a piece of equipment will share the same bus, but only one device can talk at a time. If more than one device tries to talk, the results will be, at best, totally unintelligible, and at worst, damaging to one or more of the devices on the bus. To prevent this, bus interfaces on idle devices are put into a state called "High-Z", or high impedance, effectively disconnecting them from the bus so they won't interfere with other devices.

Address Bus

In order for the microprocessor at the heart of a bussed communication system to talk to the right devices at the right time, each device (and, almost always, each memory location within a device) will be given a unique address. So, for example, when you want to see if the switches on your board are pressed, you need to look at address 0x0270 in the memory space of the 9S12X micro – the address assigned to PTADHi. As you run your code, the Program Counter steps its way through the addresses in ROM where the bytes that make up the opcodes and operands in your assembled machine code reside.

The number of unique addresses depends on the number of address lines in the address bus. If the address bus is sixteen bits wide, as in the 9S12X, we can access 2^{16} unique addresses, or 65,536. Obviously, your home computer's address bus is a lot bigger than sixteen bits in order to access all the RAM and all the peripherals it's got.

Control Lines

So, in order to talk to a device at a particular address, we must put the correct address on the address bus. But there's more: the device needs to be activated (placed on the data bus), and it needs to know if data is coming to it or is required from it. Some devices need to notify the micro that they need to be serviced, and initiate an Interrupt Request (IRQ). Sometimes, a device also needs something to synchronize its internal activities with the microprocessor's bus clock. All these activities are managed by a separate set of control lines. The following are typical for Motorola-based microprocessor bus devices:

/EN – when LOW, this line takes the device out of High-Z mode and "places it on the bus".

R/W – when HIGH, the microprocessor READS from the device; when LOW, the microprocessor WRITES to the device. (Some non-Motorola-based devices require separate /READ and /WRITE lines – watch out for these if you end up doing design work!)

PH2 or ECLK – this is a clock line that lags the bus clock by 90°. It is used by devices that require a bit of time to respond or that need to know when data on the bus is truly valid.

LCD Displays Using the Hitachi HD44780U Controller

A particular LCD controller IC is almost ubiquitous: almost any small character array LCD will have one of the variants of the Hitachi HD44780 as its brains. In fact, Wikipedia says “An **HD44780 Character LCD** is a de facto industry standard liquid crystal display (LCD) display device designed for interfacing with embedded systems.” – lousy English, but true. The 4 row by 20 character LCD display on your development kit is driven by one of these.

The HD44780 is, itself, an embedded microcontroller. So, in effect, your development board an example of parallel processing – two microcontrollers running separate processes, but communicating with each other to produce coordinated results.

The HD44780 is designed to operate within a bussed, or parallel, interconnect system. It has eight data lines, requires a single address line to select between two internal registers, has an active HIGH enable line (that’s unusual – “enable” is usually LOW), and a R/W line.

This controller is quite flexible. The full details of its capabilities are listed in the data sheet, available in Moodle, with some key parts appearing as needed in this topic. Here are some of its capabilities:

- Can be used with a variety of LCD displays, ranging from 1 line x 8 characters to 2 lines x 40 characters or 4 lines x 20 characters.
- Can be used on an eight-bit bus or, by multiplexing data lines, on a four-bit bus.
- Can print stationary characters from left to right or right to left, or can scroll characters to the left or right.
- Can produce characters in a 5 x 8 dot matrix or in a 5 x 10 dot matrix.
- Can display standard ASCII characters or use extended character sets of symbols from different languages.
- Can be used to display up to 8 user-defined special characters.
- Can control the cursor in a variety of ways.

Upon start-up, the HD44780 has no idea what it’s connected to, on either side: It doesn’t know whether it’s on a 4-bit or 8-bit bus on the micro side, and it doesn’t know what LCD it’s connected to on the device side, so it doesn’t know whether to produce 5 x 8 or 5 x 10 characters, or how many rows and characters per row it should be producing. You are responsible for telling it everything it needs to know, and you can only do that by communicating with it through the 9S12X.

The HD44780-controlled LCD on the 9S12X Development Kit

If you check out the schematic for your development kit, you’ll discover the following set of interconnections between the 9S12X and the HD44780.

Port H is used to create an eight-bit data bus, using PH0 through PH7 to map to b0 through b7, respectively. Port K is used for the address line and the two control lines:

PK2 => RS (internal address select)
PK1 => R/W
PK0 => Enable (active HIGH)

Operation

The LCD controller is able to read *instructions* and *data*. The device uses a separate address line (RS for Register Select) to differentiate between the two. Address 0 accesses the Instruction Register (IR) and provides control of the device. Address 1 accesses the Data Register (DR) and provides information to and from the device.

As previously mentioned, the HC44780 LCD Controller is a microcontroller designed to drive a number of different LCD displays. In our application, it needs to drive a 4-line x 20-character display, with characters built using a 5 x 8 matrix. The 9S12X Development Kit is also designed for operation using the 8-bit interface described above.

Inside the controller, the display is actually two lines of 40 characters per line, each in a unique memory location. On our 4-line display, the first "line" of 40 characters actually appears on lines 1 and 3, and the second "line" appears on lines 2 and 4.

The addresses for the various character locations are as follows:

Line on screen	Address (decimal)	Address (hexadecimal)
First	0 to 19	\$00 to \$13
Second	64 to 83	\$40 to \$53
Third	20 to 39	\$14 to \$27
Fourth	84 to 103	\$54 to \$67

Note that the display memory addresses for the lower "line" have bit 6 turned on, which may be useful if you want to switch between lines. The display memory addresses from \$28 to \$3F (40 to 63) are not to be used, and may be mirrors of other display address locations, resulting in unpredictable behaviour.

You may write instructions to the LCD to shift the display position. This means something different for different displays – on a two-line display, you can bring "hidden" characters in from the part of the internal line that are outside of the window. On our four-line display, the characters roll between lines 1 and 3, and between lines 2 and 4, which isn't usually desirable.

The LCD features a cursor. The cursor is configurable for appearance and behavior. The cursor is usually set to automatically advance to the next location after a display write (i.e. to the right of the previous character), but you may change this.

The LCD internally keeps track of the display data address (i.e. the character location in the display, also known as DDRAM). When you write display data to the device, it goes into the memory location specified by the current display data address. The controller may be configured to increase or decrease the display data address after a write (i.e. move right or move left). The display may also be set to shift after a write, providing a scrolling effect – again, either to the right or to the left. With the four-line display, this means switching to the alternate line when the DDRAM address gets to the end (or beginning) of the addresses for the current visible line – again, probably not what you were hoping for.

In this course, you're expected to have and use the functions shown in the following header file:

```
//Hitachi 44780 initialization and commands
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2015

void LCD_Init(void); //8-bit, 2-line, 5x8 chars, disp on, curs on, blink off, inc curs mode, no shift, clear, home
void LCD_Ctrl(unsigned char);
unsigned char LCD_Busy(void);
void LCD_Char(unsigned char);
void LCD_Addr(unsigned char); //raw LCD DDRAM address -- requires knowledge of device
void LCD_Pos(unsigned char,unsigned char); //Row and Column, zero based; out of range values go to home location
void LCD_String(char *); //requires a NULL-terminated string of ASCII characters in main program
```

Unfortunately, the _Init routine requires _Ctrl and _Busy, which complicates things. Your instructor may also ask you to develop functions to generate special characters later.

HD44780 Instructions

In the LCD instructions, the first bit that's set (HIGH) in the instruction byte determines the group of instructions to choose from. These instructions are found in the data sheet for the HD44780, for which a link has been provided in Moodle, and are shown below:

HITACHI										HD44780U		
Table 6 Instructions												
Instruction	RS	R/W	Code								Description	Execution Time (max) (when t_{cp} or f_{osc} is 270 kHz)
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	1	I/D	S		Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 μ s
Display on/off control	0	0	0	0	0	1	D	C	B		Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 μ s
Cursor or display shift	0	0	0	0	1	S/C	R/L	—	—		Moves cursor and shifts display without changing DDRAM contents.	37 μ s
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 μ s
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 μ s
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 μ s
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 μ s
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 μ s $t_{ADD} = 4 \mu s^*$
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 μ s $t_{ADD} = 4 \mu s^*$
		I/D = 1:	Increment								DDRAM: Display data RAM	Execution time changes when frequency changes Example: When t_{cp} or f_{osc} is 250 kHz, $37 \mu s \times \frac{270}{250} = 40 \mu s$
		I/D = 0:	Decrement								CGRAM: Character generator RAM	
		S = 1:	Accompanies display shift								ACG: CGRAM address	
		S/C = 1:	Display shift								ADD: DDRAM address	
		S/C = 0:	Cursor move								(corresponds to cursor address)	
		R/L = 1:	Shift to the right								AC: Address counter used for both DD and CGRAM addresses	
		R/L = 0:	Shift to the left									
		DL = 1:	8 bits, DL = 0: 4 bits									
		N = 1:	2 lines, N = 0: 1 line									
		F = 1:	5 x 10 dots, F = 0: 5 x 8 dots									
		BF = 1:	Internally operating									
		BF = 0:	Instructions acceptable									

Note: — indicates no effect.

* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10, t_{ADD} is the time elapsed after the busy flag turns off until the address counter is updated.

The default condition of the controller is as follows: 8-bit mode, 1-line display, 5 x 8 matrix, display off, cursor off, blink mode off, increment cursor position (move to the right), with shifting turned off (display doesn't scroll). We need to initialize the controller to make it match our hardware.

LCD Controller Initialization

Timing is critical in all communications with this controller, and particularly so in the initialization of the device. To begin with, the Busy flag is not active until a particular sequence of commands has been executed. In addition, data needs to be present 60 ns prior to an Enable pulse, and the Enable pulse must be HIGH for at least 500 ns followed by at least 500 ns LOW. Due to internal activity in the HC44780U, at least 40 ms must be allowed following power-up. After the first command is sent to the controller, at least 4.1 ms must be allowed before the second command is sent, then 100 μs must be allowed before the third command is sent. After the third command, the Busy flag becomes available, and can thereafter be used to monitor the controller's activity.

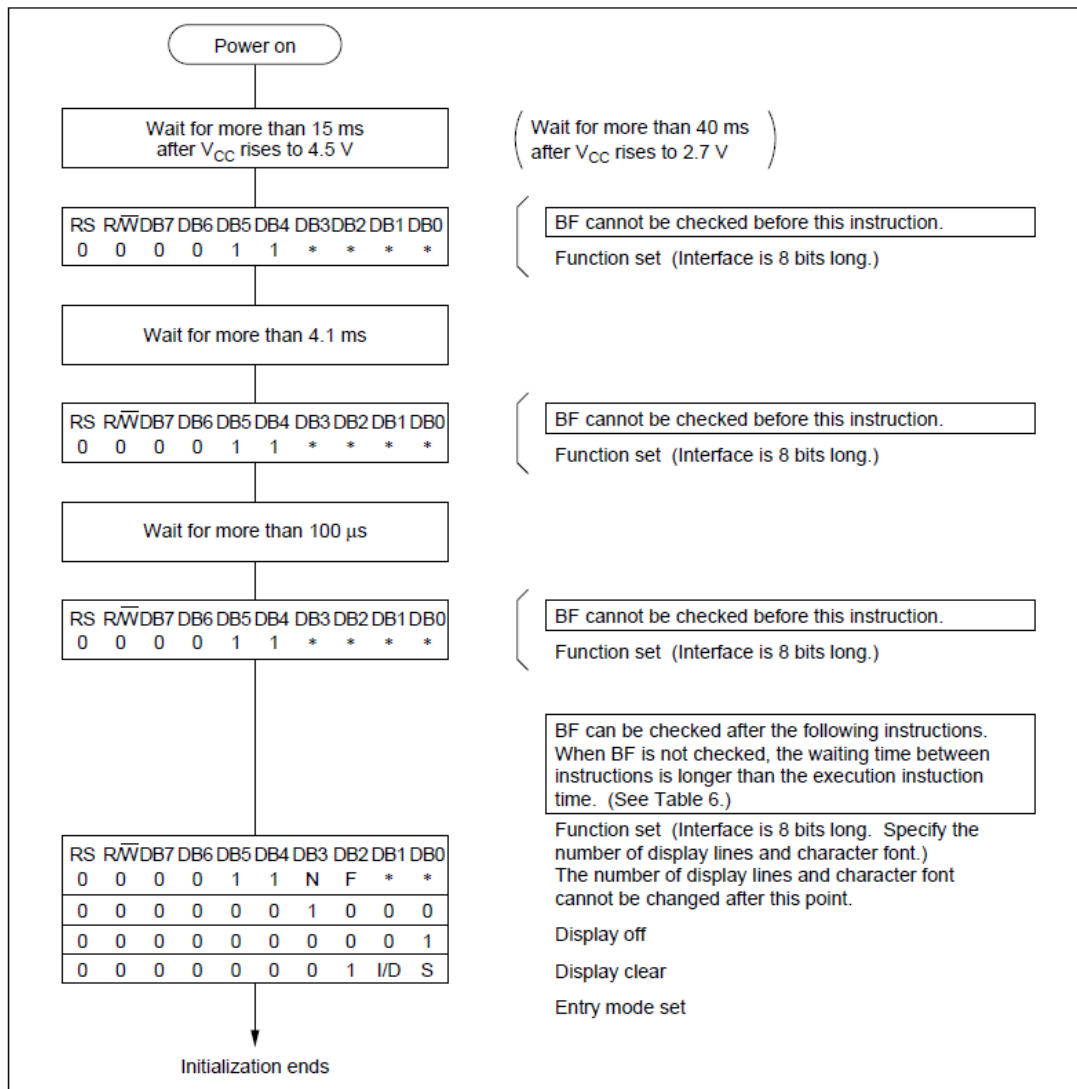


Figure 23 8-Bit Interface



LCD_Init

For this discussion, the initialization flowchart from the previous page will be used as a template for the code developed.

Before we can do anything at all, we need to set up our simple parallel bus interface between the microcontroller and the LCD controller. Checking back to the schematic for our board, we see that Port H (PTH) is being used as the eight-bit data bus and the lower three bits of Port K (PORTK) are being used as the control lines:

PORTK bits:

- 7 – x
- 6 – x
- 5 – x
- 4 – x
- 3 – x
- 2 – RS (register select: LOW for Control, HIGH for Data)
- 1 – R/W (HIGH for READ, LOW for WRITE)
- 0 – EN (chip enable: HIGH for Enable)

Most of the time, we will be writing to the LCD controller: we will write control bytes to it to tell it how we want it to look and respond; we will write data bytes to it, primarily providing it with the ASCII codes we want to display on the screen. So, it makes sense for us to set the default condition for the data bus, PTH, as outputs for all eight bits using the Port H Data Direction Register (DDRH).

Although it doesn't really matter what's on the bus when we enable it, good programming practice suggests we should write something innocuous to the bus, so clearing all eight bits before we change the pins to outputs is a good idea.

Since we're only using three of the eight bits in PORTK, we should leave the other five bits alone in case there's some other possible use for those bits. So, instead of writing an entire byte to the Port K Data Direction Register (DDRK), we'll OR the three bits we need with 1s to make them into outputs, while leaving the other five alone.

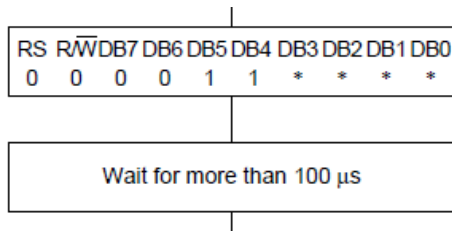
Before we do that, however, we should set the bits in PORTK to the condition we want them to be in when the port pins are enabled. The resting state that makes sense for us is to have all three controls lines LOW – RS set for control, R/W set to Write, and EN low so that the chip is not being addressed. The code below also shows the beginning of the LCD_Lib.c file that you will be building to match the header file shown previously.

```
//Hitachi 44780 initialization and commands
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2015

#include <hidef.h>
#include "derivative.h"
#include "LCD_Lib.h"

void LCD_Init(void)
{
    PTH = 0b00000000;
    DDRH = 0b11111111; //data bus as outputs for write
    PORTK&=0b11111000; //preset RS low, R/W low, EN low
    DDRK |= 0b000000111; //activate three control lines
```


Now for the second attempt: Since the control byte is still sitting on the bus, all we need to do is strobe the control lines again, then wait the required time.



```

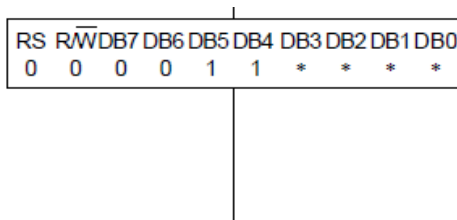
PORTK|=0b00000001; //RS low, R/W low, EN high to write a control
PORTK&=0b11111000; //resting state

asm LDD #267; //need 100 us delay
asm DBNE D,*;

```

Check the timing: $267 \times 3 \text{ cycles} \times 125 \text{ ns/cycle} = 100 \mu\text{s}$

Third time from the flowchart. Oddly, for this, no delay time is indicated. However, Our microcontroller is faster than the LCD controller, so we'll insert a delay, anyway. We've already got a 100 μ s delay calculated, so we'll use it.



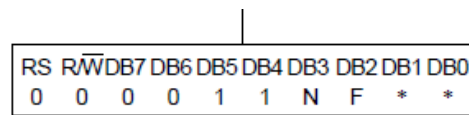
```

PORTK|=0b00000001; //RS low, R/W low, EN high to write a control
PORTK&=0b11111000; //resting state

asm LDD #267; //need 100 us delay
asm DBNE D,*;

```

According to the notes, the HD44780 LCD controller should be working properly, and its "Busy Flag" should be available for further instructions. And, although it seems from experience that the HD44780 has been properly configured at this point, the flowchart says "do it again", so who are we to argue?



This time, we'll rely on the busy flag, and will use the `LCD_Ctrl()` command. Oh, wait a second, we haven't written that yet! We'll code in the function call, then come back to writing it later.

```

LCD_Ctrl(0b00111000); //same as above, but using LCD_Ctrl (Busy is active)

```



```

unsigned char LCD_Busy(void)
{
    unsigned char cBusy;

    DDRH = 0b00000000;    //data bus as inputs for read

    PORTK|=0b000000011;
    /*
        ||_____EN (Enable: HIGH to enable)
        ||_____R/W (HIGH for READ)
        ||_____RS (Register Select: LOW for Status)
    */
    PORTK&=0b11111000;    //resting state

    cBusy=PTH&0b10000000; //Busy Flag is the MSB of the Status Register

    DDRH = 0b11111111;    //data bus returned to outputs for next write

    return cBusy;
}

```

As always, make sure you understand how this routine works before you start using it.

LCD_Char

Up to this point, all we've done is get the LCD controller set up and ready to work for us. The LCD is now sitting there doing nothing until we send it ASCII characters to display. *LCD_Char()* is the name of the function we'll create to carry out this task.

The only differences between *LCD_Char()* and *LCD_Ctrl()* are the type of data sent and the internal register it's sent to.

- The data sent will be a single ASCII character. So, if you want to send a number, you'll have to do a Hex to ASCII conversion first.
- The target in the LCD controller is the Data Register, not the Control Register. That means that you will need to SET RS when you strobe the control lines.

That should be enough information for you to create and test *LCD_Char()*. If you need further help, your instructor can provide it.

LCD_String

Quite often, you'll want to send multiple characters to the LCD (or to other peripherals or equipment that's looking for ASCII characters). The best way to send longer strings of characters is by using a **null-terminated string** of ASCII characters. The *LCD_String* function you will be creating can handle strings of any length (up to the length of a row on the LCD display, or 20 characters), since it's not looking for a particular length, but is expecting the string to end with the *NULL* character (ASCII code 0). The routine transmits each character, then checks to see if the character was a *NULL*. If it is a *NULL*, program execution exits the function.

You will be using *LCD_Char()* as the working block of *LCD_String(*)*, a routine that sends a null-terminated string, from a memory location specified in your program, to the LCD.

In ANSI C, the starting point of a string or array of bytes is referenced using *pointers*. The following screen clip of the author's *LCD_String* function shows how to get the contents of an address which is being pointed to using an asterisk (*). *cString* is the label of the string you want to transmit, and is actually initially the address of the first character in the string.

```

void LCD_String(char * cString)
{
    while(*cString!=0)    //watch for NULL terminator
        LCD_Char(*cString++); //send next character
}

```

Take some time to make sense of how the pointer is being used in the `LCD_String()` function. In this course, there won't be much need to learn more about pointers than you see in this example. If you need to know more, the Internet is an endless source of wisdom and hilarity regarding pointers, along with the dereferencing "*" operator and the "&" address operator used for pointer management.

LCD_Addr

This is intended to be a simple way to locate a particular position on the LCD display. All it does is to take the address provided, convert it into a DDRAM Address control byte, and send that command to the LCD controller. The cursor will move to that location, ready for you to send a character to that spot. Here's the code for this function:

```
void LCD_Addr(unsigned char cAddr)
{
    cAddr|=0b10000000; //assumes programmer understands raw addresses
    LCD_Ctrl(cAddr); //add command bit for "Set DDRAM Address"
}
```

Compare back to the table of instructions for the HD44780 controller to see why the MSB needed to be changed to 1 before sending the address to the control register.

LCD_Pos

In order to use `LCD_Addr` effectively, the programmer needs to know what the valid addresses are and how they are arranged on the display. The following table appeared earlier in this document, but has been duplicated here for convenience:

Line on screen	Address (decimal)	Address (hexadecimal)
First	0 to 19	\$00 to \$13
Second	64 to 83	\$40 to \$53
Third	20 to 39	\$14 to \$27
Fourth	84 to 103	\$54 to \$67

`LCD_Pos()` is intended to simplify the process of moving to a particular location on the display. The basic idea is to pass two numbers to the function: a ROW and a COLUMN, and let the function generate the correct address to send to the controller using `LCD_Addr()`.

For consistency between classes, the instructors for this course have settled on zero-based row and column addressing, so the available rows are 0 through 3 and the available columns are 0 through 19. Part of the necessary code for this routine is shown below; you are expected to complete the function so that it works satisfactorily. This code demonstrates an application for the `switch->case` operation in C.

```

void LCD_Pos(unsigned char cRow, unsigned char cCol)
{
    if (cRow>3 || cCol>19) //zero-based for both row and column
    {
        LCD_Addr(0);
    }
    else
    {
        switch (cRow)
        {
            case 0: //first row is 0 to 19
                LCD_Addr(cCol);
                break;

            case 1: //second row is 64 to 83
                LCD_Addr(cCol+64);
                break;

            case 2: //third row
                LCD_Addr(cCol+20);
        }
    }
}

```

At this point, you will have at your disposal all of the LCD functions deemed necessary by all of your instructors. Your instructor, however, may want you to generate a collection of other functions to enhance your use of the LCD on your board. Some of these may simply be *LCD_Ctrl()* calls that do things like turn the cursor on or off, blink on or off, clear the display, etc. without requiring you to constantly look up the instructions on the table.

Character Generation

One set of optional functions you can try out allows you to access the LCD's capacity for generating characters other than the ones in the ASCII set contained in its memory. If your instructor deems this extraneous, you can skip the next few pages.

Your LCD is able to display 8 user-defined characters, which you design pixel by pixel within the 5 x 8 pixel matrix. The memory in the device that holds the pixel pattern is known as character generator RAM (CGRAM).

These user-defined characters will take the place of the first eight ASCII characters. In the ASCII table, these are non-printable characters, so Hitachi engineers decided to re-define them as the spaces available for your custom characters. So, to access the characters you generate, simply reference ASCII characters 0x00 to 0x07.

To get your user-defined character patterns into the device, you must program them, row by row, for each character. The device must also be instructed to accept CG data. The "Set CGRAM Address" instruction does this. Here it is from the data sheet's table of instructions:

Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 μ s
-------------------	---	---	---	---	-----	-----	-----	-----	-----	-----	---	------------

This instruction is written to the LCD Instruction Register and tells the LCD that all subsequent data written to the Data Register will be CG (Character Generation) data. The instruction includes the address of CGRAM to start at for a given character. Only 6 bits are required, since only 64 bytes are needed to represent the eight 5 x 8 characters – each row of pixels in a character requires one byte, so each of the eight characters requires eight bytes.

The top of ASCII character \$00 is at CG address \$00, and extends to address \$07. Note that the first 3 bits (most significant) are ignored, as the characters are only 5 pixels wide.

In the current version of the data sheet, Table 5 shows you how to build the bitmap for special characters in CGRAM, as shown on the next page:

HD44780U

Table 5 Relationship between CGRAM Addresses, Character Codes (DDRAM) and Character Patterns (CGRAM Data)

For 5 × 8 dot character patterns

Character Codes (DDRAM data)								CGRAM Address				Character Patterns (CGRAM data)																																																																																			
7	6	5	4	3	2	1	0	5	4	3	2	1	0	7	6	5	4	3	2	1	0																																																																										
High				Low				High		Low		High				Low																																																																															
0 0 0 0 * 0 0 0								0 0 0				<table border="1"> <tr><td>*</td><td>*</td><td>*</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>*</td><td>*</td><td>*</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>								*	*	*	1	1	1	1	0				1	0	0	0	1				1	0	0	0	1				1	1	1	1	0				1	0	1	0	0				1	0	0	1	0				1	0	0	0	1				1	0	0	0	0	*	*	*	0	0	0	0	0	Character pattern (1)			
*	*	*	1	1	1	1	0																																																																																								
			1	0	0	0	1																																																																																								
			1	0	0	0	1																																																																																								
			1	1	1	1	0																																																																																								
			1	0	1	0	0																																																																																								
			1	0	0	1	0																																																																																								
			1	0	0	0	1																																																																																								
			1	0	0	0	0																																																																																								
*	*	*	0	0	0	0	0																																																																																								
0 0 0 0 * 0 0 1								0 0 1				<table border="1"> <tr><td>*</td><td>*</td><td>*</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td></td><td></td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>*</td><td>*</td><td>*</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>								*	*	*	1	0	0	0	1				0	1	0	1	0				1	1	1	1	1				0	0	1	0	0				1	1	1	1	1				0	0	1	0	0				0	0	1	0	0				0	0	0	0	0	*	*	*	0	0	0	0	0	Character pattern (2)			
*	*	*	1	0	0	0	1																																																																																								
			0	1	0	1	0																																																																																								
			1	1	1	1	1																																																																																								
			0	0	1	0	0																																																																																								
			1	1	1	1	1																																																																																								
			0	0	1	0	0																																																																																								
			0	0	1	0	0																																																																																								
			0	0	0	0	0																																																																																								
*	*	*	0	0	0	0	0																																																																																								
0 0 0 0 * 1 1 1								1 1 1				<table border="1"> <tr><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>0</td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>1</td><td>0</td><td>1</td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>1</td><td>1</td><td>0</td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td><td></td><td></td></tr> <tr><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td><td></td><td></td></tr> </table>								*	*	*									1	0	0						1	0	1						1	1	0						1	1	1			*	*	*						Cursor position																											
*	*	*																																																																																													
			1	0	0																																																																																										
			1	0	1																																																																																										
			1	1	0																																																																																										
			1	1	1																																																																																										
*	*	*																																																																																													

CG data may be programmed at any time, including when characters for that type are currently being displayed. Some unusual animation effects may be generated by re-programming the characters that are currently being displayed.

Once the programming of CG data is complete, the device should be set back to display data (DDRAM). The "Set DDRAM Address" instruction does this. Go to DDRAM address 0, the home position on the display, as a good place to start. This instruction sets the LCD to accept DD information from the DR for all subsequent writes.

Remember that you can create up to eight custom characters. The following two functions can be used to create a single character at a given location or to create all eight available characters from a table of 64 row-definition bytes defining all eight characters.

- *LCD_CharGen()* is a routine that builds a single custom character for the ASCII code passed to it as a parameter (0x00 to 0x07), with the character definition pattern beginning at a location pointed to as a parameter.
- *LCD_CharGen8()* is a routine that builds eight custom characters, with their definition patterns beginning at a location pointed to as a parameter. (If you don't need all eight characters, just fill the unused row bytes with nulls to produce blank characters.) This function doesn't need an ASCII code, since it fills 0x00 to 0x07.

LCD_CharGen Example

```

/*****
*   Variables
*****/

char cNameString[14] = "P Ross Taylor";

char ASCII_1[8] =
{
    0b00000100,
    0b00001110,
    0b00011011,
    0b00010001,
    0b00011011,
    0b00011011,
    0b00011011,
    0b00001110,
    0b00000100
};

/*****
*   Lookups
*****/

void main(void)    // main entry point
{
    _DISABLE_COP();

/*****
*   Initializations
*****/

    LCD_Init();
    LCD_Ctrl(0b00001100); //Cursor off

    LCD_CharGen(1,ASCII_1); //Generate a single ASCII Char as 0x01

    for (;;)        //endless program loop
    {
/*****
*   Main Program Code
*****/

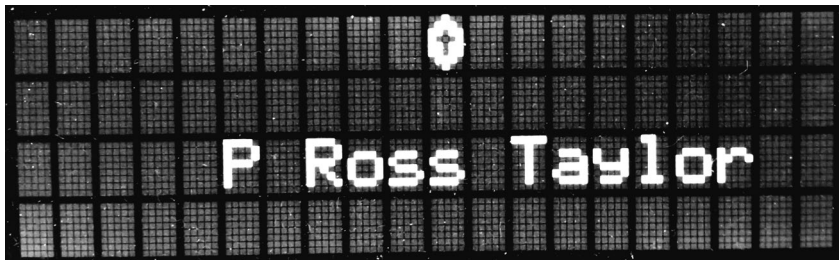
        LCD_Pos(2,5);           //Row 3, Column 6
        LCD_String(cNameString); //Display name

        LCD_Pos(0,10);          //Row 1, Column 11
        LCD_Char(1);            //Display custom character

        HALT;                    //Halt program in endless loop
    }
}

```

Notice that `LCD_CharGen()` is run only once, in the initializations. That's where ASCII character 0x01 is created. To use this character, we simply display ASCII character 0x01 by using our `LCD_Char()` function. The results are seen below. Notice how the pattern in the first row relates to the pixel map shown in the *Variables* space above.



LCD_CharGen8 Example

Here's the author's version of `LCD_CharGen8()`. If you're an enterprising type, you will probably realize that you don't need to modify this much to make `LCD_CharGen()`. The eight-character version writes 64 bytes into CGRAM, starting with address 0x00 and going to address 0x3F.

```
void LCD_CharGen8(char *cDefs)
{
    unsigned char cLine;

    LCD_Ctrl(0b01000000); //start of ASCII(0)

    for(cLine=0;cLine<64;cLine++)
    {
        LCD_Char(*cDefs++);
    }

    LCD_Ctrl(0b10000000); //back to DDRAM, home location
}
```

To keep the entire useful code together, the following images have been shrunk probably beyond your ability to read them on paper. However, you can zoom in on the electronic version to see more detail.

```

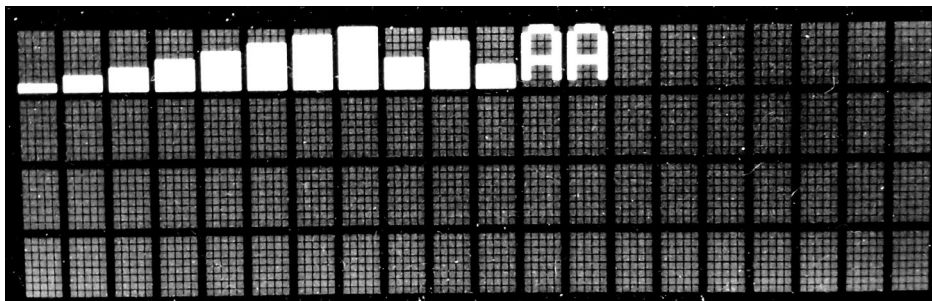
/*****
 * Variables
 *****/
unsigned char cCount;
char ASCII_0to7[64] =
{
    //ASCII char 0
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    //ASCII char 1
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    //ASCII char 2
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    //ASCII char 3
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    //ASCII char 4
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    //ASCII char 5
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    //ASCII char 6
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    //ASCII char 7
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
};

void main(void) // main entry point
{
    _DISABLE_COP();

    /*****
     * Initializations
     *****/
    LCD_Init();
    LCD_Ctrl(0b00001100); //Cursor off
    LCD_CharGen8(ASCII_0to7); //Generate all eight custom characters
    for (;;) //endless program loop
    {
        /*****
         * Main Program Code
         *****/
        LCD_Pos(0,0); //Home position
        for(cCount=0;cCount<8;cCount++)
        {
            LCD_Char(cCount); //Display chars in order
        }
        LCD_Char(3); //Display three randomly chosen
        LCD_Char(5);
        LCD_Char(2);
        LCD_Char(0x41); //Same procedure for normal ASCII
        LCD_Char('A'); //Alternate way to display ASCII
    }
    HALT; //Halt program in endless loop
}

```

Notice again that `LCD_CharGen8()` is only run once in the initializations, and creates all eight custom ASCII characters: 0x00 through 0x07. Once created, these characters can be displayed in the same way as any of the other pre-defined ASCII characters.



ASCII Code Manipulation

The LCD display and the Serial Communication Interface (still to come) work primarily with ASCII values. The following table shows the standard 7-bit ASCII codes.

ASCII Table

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(NUL)	0	0x00	(SP)	32	0x20	@	64	0x40	`	96	0x60
(SOH)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(STX)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62
(ETX)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(EOT)	4	0x04	\$	36	0x24	D	68	0x44	d	100	0x64
(ENQ)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ACK)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
(BEL)	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
(BS)	8	0x08	(40	0x28	H	72	0x48	h	104	0x68
(HT)	9	0x09)	41	0x29	I	73	0x49	i	105	0x69
(NL)	10	0x0a	*	42	0x2a	J	74	0x4a	j	106	0x6a
(VT)	11	0x0b	+	43	0x2b	K	75	0x4b	k	107	0x6b
(NP)	12	0x0c	,	44	0x2c	L	76	0x4c	l	108	0x6c
(CR)	13	0x0d	-	45	0x2d	M	77	0x4d	m	109	0x6d
(SO)	14	0x0e	.	46	0x2e	N	78	0x4e	n	110	0x6e
(SI)	15	0x0f	/	47	0x2f	O	79	0x4f	o	111	0x6f
(DLE)	16	0x10	0	48	0x30	P	80	0x50	p	112	0x70
(DC1)	17	0x11	1	49	0x31	Q	81	0x51	q	113	0x71
(DC2)	18	0x12	2	50	0x32	R	82	0x52	r	114	0x72
(DC3)	19	0x13	3	51	0x33	S	83	0x53	s	115	0x73
(DC4)	20	0x14	4	52	0x34	T	84	0x54	t	116	0x74
(NAK)	21	0x15	5	53	0x35	U	85	0x55	u	117	0x75
(SYN)	22	0x16	6	54	0x36	V	86	0x56	v	118	0x76
(ETB)	23	0x17	7	55	0x37	W	87	0x57	w	119	0x77
(CAN)	24	0x18	8	56	0x38	X	88	0x58	x	120	0x78
(EM)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(SUB)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(ESC)	27	0x1b	;	59	0x3b	[91	0x5b	{	123	0x7b
(FS)	28	0x1c	<	60	0x3c	\	92	0x5c		124	0x7c
(GS)	29	0x1d	=	61	0x3d]	93	0x5d	}	125	0x7d
(RS)	30	0x1e	>	62	0x3e	^	94	0x5e	~	126	0x7e
(US)	31	0x1f	?	63	0x3f	_	95	0x5f	(DEL)	127	0x7f

Some of these characters (everything less than \$20) have special meaning – BD, LF, FF, CR, BEL, etc. Eight-bit ASCII includes another 128 characters (\$80 to \$FF) which are called “extended ASCII” and are not standardized. Trying them out will produce different results on different displays and terminals, so you’re welcome to play around with them if you have a fairly high tolerance for frustration. For the Hitachi 44780 display, the following table from the datasheet shows the characters that can be displayed:

HD44780U

Table 4 Correspondence between Character Codes and Character Patterns (ROM Code: A00)

Lower 4 Bits	Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	P				-	夕	ミ	α	ρ	
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	厶	ä	g	
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	×	β	θ	
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	ε	ε	∞	
xxxx0100	(5)		\$	4	D	T	d	t			、	イ	ト	ト	μ	Ω	
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	1	σ	Ü	
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ	
xxxx0111	(8)		'	7	G	W	g	w			ア	キ	ヌ	ウ	g	π	
xxxx1000	(1)		(8	H	X	h	x			イ	ク	ネ	リ	フ	×	
xxxx1001	(2))	9	I	Y	i	y			ウ	ケ	ル	ル	、	γ	
xxxx1010	(3)		*	:	J	Z	j	z			エ	コ	ン	レ	j	〒	
xxxx1011	(4)		+	;	K	L	k	l			オ	サ	ヒ	ロ	*	斤	
xxxx1100	(5)		,	<	L	¥	l	l			カ	シ	フ	フ	φ	円	
xxxx1101	(6)		-	=	M	J	m	}			ユ	ヌ	ハ	ン	も	÷	
xxxx1110	(7)		.	>	N	^	n	+			ヨ	セ	ホ	°	ん		
xxxx1111	(8)		/	?	O	_	o	←			ツ	ソ	マ	°	ö		

Note: The user can specify any pattern for character-generator RAM.

The note at the bottom reiterates the fact that you can create your own characters for ASCII codes 0 through 7. Apparently, they can also be accessed using ASCII codes 8 through F, but they would be the same characters as 0 through 7.

Upper and Lower Case ASCII Codes

Look at the table of ASCII characters to see what's different between uppercase and lowercase letters. (Hint: write them out as binary representations.) You should discover that there is only one bit different. This makes manipulation of uppercase and lowercase values pretty simple. The only thing you need to watch out for is that you only want to change this bit if the value is a valid alphabet character – otherwise, you could be seriously messing up a number or punctuation mark!

In your *Misc_Lib.h* header file, you'll notice that prototypes have been included for two routines designed to handle changing the case of an ASCII character:

- `ToUpper()`
- `ToLower()`

Your instructor will want you to have these routines completed and working, not only for the LCD, but also for subsequent use with dumb terminals attached to an SCI comm port.

Hexadecimal to ASCII conversion

Look back at the table of ASCII characters. Notice that the ASCII character codes for numeric digits (0123456789) and the hexadecimal extensions (ABCDEF) do not match their actual value. In other words, if you want to display "7" on a terminal emulator, sending the ASCII code "7" will make the terminal beep instead. What you need to do is send "\$37" in order to display "7" on the screen. It's a code!

Converting regular digits (0123456789) to ASCII is easy – just add 0x30 to the digit or OR the digit with 0b00110000.

Converting the hexadecimal values ABCDEF to ASCII is similar, but with a different offset. For these, you need to add 0x37.

In your *Misc_Lib*, you will want to write `HexToASCII()` so that you can convert individual numeric digits to ASCII code in order to send the results to the LCD or other equipment that displays ASCII codes.

Also in your *Misc_Lib*, you will want to write `ASCIIToHex()` that takes the codes for valid ASCII codes (0x30 – 0x39 and 0x41 – 0x46) and converts them to real numbers (0 – 9 and A – F). Again, don't mess with any values outside of these ranges.

Make sure that your function only converts true numerals (0 – 9 and A – F or a – f), and that it can't be broken if a two-nibble or two-digit value is sent to it. Incidentally, you can simplify the handling of A – F and a – f by using `ToUpper()` inside your `ASCIIToHex()` routine.

Your `ASCIIToHex()` function should return 0 if non-valid (i.e. non-numeric) or double-digit values are passed to it. It will then be up to your handling of the returned value in the `main()` program to determine what to do with a returned 0. You may choose to work with the returned zero, or you may want to set up a trapping routine that determines when zero represents an invalid response and when it actually means zero.

The Serial Communications Interface

Your 9S12X chip contains SCI (Serial Communication Interface) modules for asynchronous serial communications. You will use one of the SCI modules to communicate with a PC running "terminal emulation" software over a standard RS-232 connection.

Using the PC as a terminal allows you to interact with a color display and a keyboard. This will bring improved I/O to your programs.

Because you will be reading bytes from and writing bytes to the serial port, the SCI module acts as a parallel-to-serial and serial-to-parallel converter. There is an external chip on your microcontroller board that level shifts the signals from the 9S12X (TTL levels) to RS-232 levels (typically around $\pm 10V$).

In asynchronous communications, the transmitter may begin a data send to the receiver at any time. Once started, a complete block of data (known as a data character) must be completely transmitted. The delay between data characters may be any length. Transmission of the individual bits in the data character is driven by a clock. The transmitter and receiver must use a clock rate that is approximately equal in order to correctly exchange data. The term "asynchronous" refers to the fact that the clocks in the two pieces of equipment are independent, and communication can be initiated at any time.

The RS-232C standard for serial communication allows for a wide range of signaling characteristics. Here are a few of them:

- Transmission rates vary from 75 baud to 115 200 baud (these must be at clearly-specified speeds only, like 9600, but not 10 000, for example)
- Data can be sent as 7-bit standard ASCII characters, 8-bit extended ASCII characters, or binary data
- Simple error checking, in the form of a Parity Bit, may or may not be activated
- The Parity Bit, if present, can be Even, Odd, always 1, or always 0
- The minimum rest time ("stop bits") between data characters can be adjusted
- "Handshaking" for setting up and maintaining sessions can be configured or ignored

The 9S12X SCI modules are able to send 8-bit or 9-bit data payloads. This provides a fair bit of flexibility:

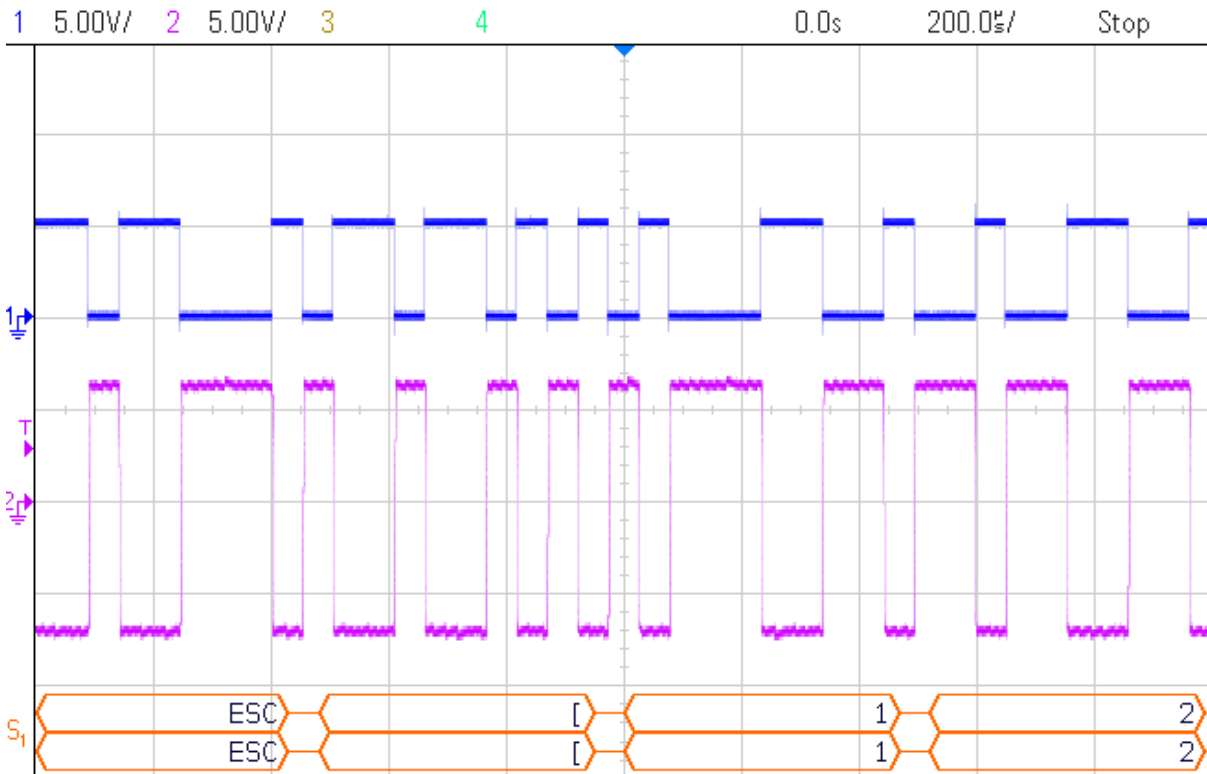
- 9-bit mode provides for 9 actual data bits (very rarely used) or 8 data bits and a parity bit for error checking
- 8-bit mode provides for 8 actual data bits or 7 data bits and a parity bit

Since the 9-bit configurations require us to check two data registers (eight bits in one and the ninth in another), we'll restrict our work to one of the 8-bit modes: 8 actual bits with no parity. The simple error checking made available by the parity bit isn't something we need to concern ourselves with, as, in the lab, we'll be within two metres of the computer we're using as a terminal. If you find yourself in a situation involving greater distance or an electrically-noisy environment, you might consider enabling and checking parity.

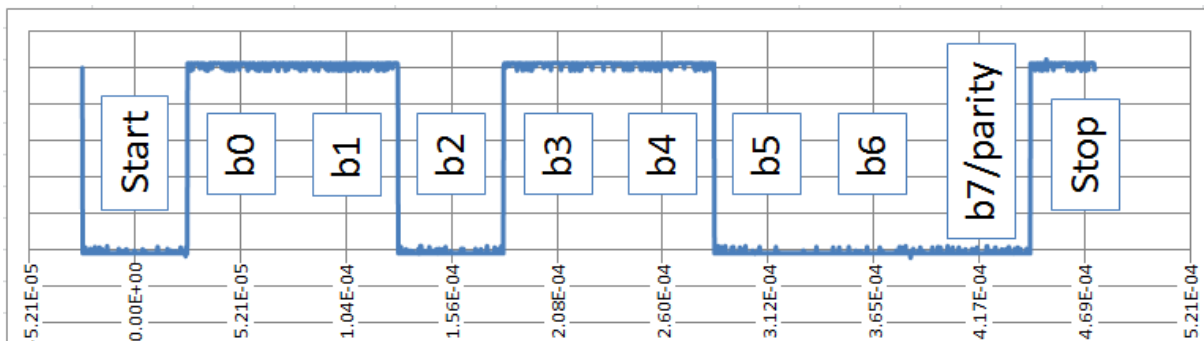
Also, we will be working with the simplest electrical connection possible between our board and the terminal – three wires only: Ground, Transmit Data, and Receive Data. This means that we will not be using the bundle of handshaking wires made available in the RS-232C standard.

Note: When you set up your terminal, select "flow control: none".

The data character sent in this configuration will consist of 1 start bit, 8 data bits (sent LSB first), and 1 stop bit. In terminal language, this is referred to as "8N1" communication – 8 bits, no parity, one stop bit. The start bit signals the start of a data character. The data bits are the data payload. The stop bit signals the end of the data character and is the minimum delay required between data characters. With early communication equipment, the stop bit gave the receiver time to process the received data – this is typically a non-issue these days. In many pieces of equipment, this wait time can be set to 1 bit length, 1.5 bit lengths, or 2 bit lengths. The 9S12X's SCI port only offers 1 bit length – that's another thing to remember when you're setting up your terminal. So, once your SCI port is set up to match the conditions above, you will see something like the top trace on the TX pin from the microcontroller, and the bottom trace on the TX pin of the Comm Port.



The following shows the formatting and order of bits, as seen at the microcontroller output.



Note: The TTL level for a mark (logic 1) is +5 V, and 0 V for a space (logic 0). However, these values are approximately -7 V (mark), and +7 V (space) when level-translated to non-return-to-zero RS-232 levels.

The transmitter clocks out serial data at the transmission rate. The receiver samples the line at intervals determined by that clock rate to receive the data. It is critical that the sending and receiving clocks are at the same rate, otherwise the receiver will be sampling the line at the wrong times. In actuality the receiver typically samples the line at a much higher rate and considers multiple samples per bit time to determine the state of each received bit. The 9S12X SCI modules have a sampling rate that is 16 times the bit rate.

The resting state between characters is called "mark idle", and is a continuation of the stop bit. Therefore, the start bit is always a space, to let the equipment know data is coming.

The number of bits transmitted per second is known as the baud rate. The data rate is actually less, since the framing start and stop bits and the error-checking parity bit, if used, do not contribute to the data payload.

Baud rates for serial communications are relatively slow by today's standards. The following is a fairly comprehensive list of available baud rates:

- 75
- 110
- 300
- 600
- 1 200
- 2 400
- 4 800
- 9 600
- 14 400
- 19 200
- 38 400
- 57 600
- 115 200

Initializing the Serial Communications Interface

To activate an SCI module on the 9S12X, you typically need to configure just three registers. A full description of the activities of these registers is found in chapter 11 of the 9S12X "Data Sheet". Look these up if you want more a more detailed understanding of the operation of these registers than is provided below.

The first register, SCIBD, is a sixteen-bit register (SCIBDH/SCIBDL) that controls the baud rate for the module, and requires a 13-bit value as a clock divisor.

11.3.2.1 SCI Baud Rate Registers (SCIBDH, SCIBDL)

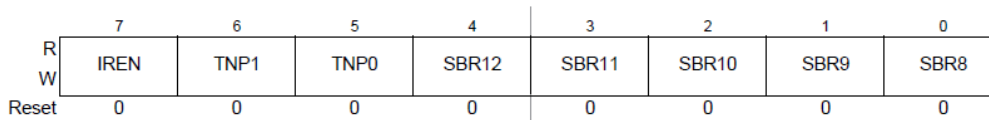


Figure 11-3. SCI Baud Rate Register (SCIBDH)

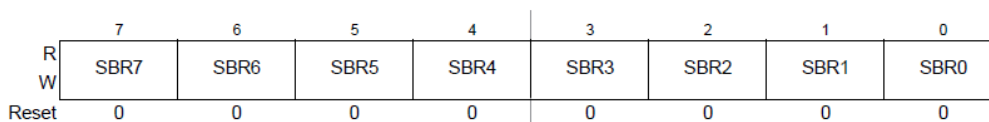


Figure 11-4. SCI Baud Rate Register (SCIBDL)

Read: Anytime, if AMAP = 0. If only SCIBDH is written to, a read will not return the correct data until SCIBDL is written to as well, following a write to SCIBDH.

Write: Anytime, if AMAP = 0.

NOTE

Those two registers are only visible in the memory map if AMAP = 0 (reset condition).

The SCI baud rate register is used by to determine the baud rate of the SCI, and to control the infrared modulation/demodulation submodule.

Table 11-1. SCIBDH and SCIBDL Field Descriptions

Field	Description
7 IREN	Infrared Enable Bit — This bit enables/disables the infrared modulation/demodulation submodule. 0 IR disabled 1 IR enabled
6:5 TNP[1:0]	Transmitter Narrow Pulse Bits — These bits enable whether the SCI transmits a 1/16, 3/16, 1/32 or 1/4 narrow pulse. See Table 11-2.
4:0 7:0 SBR[12:0]	SCI Baud Rate Bits — The baud rate for the SCI is determined by the bits in this register. The baud rate is calculated two different ways depending on the state of the IREN bit. The formulas for calculating the baud rate are: When IREN = 0 then, SCI baud rate = SCI bus clock / (16 x SBR[12:0]) When IREN = 1 then, SCI baud rate = SCI bus clock / (32 x SBR[12:1]) Note: The baud rate generator is disabled after reset and not started until the TE bit or the RE bit is set for the first time. The baud rate generator is disabled when (SBR[12:0] = 0 and IREN = 0) or (SBR[12:1] = 0 and IREN = 1). Note: Writing to SCIBDH has no effect without writing to SCIBDL, because writing to SCIBDH puts the data in a temporary location until SCIBDL is written to.

The three most significant bits of this 16-bit register can be 0, as you will not be using the infrared configuration for this course. The actual baud rate is the bus frequency (8 MHz on your board) divided by 16 divided by the 13-bit value provided in SCIBDH:SCIBDL. By the way, you can simply write a sixteen-bit value to SCIBD. Since there are a number of SCI ports available, the prototype file distinguishes between them by inserting a number into the register name. The one we want is SCI0BD. This port is connected to the 9-pin RS-232 connector on your 9S12X board.

If you want to access the other SCI ports, SCI1 is connected to the infrared hardware on your 9S12X board, and the other ports are available at the break-out headers – just look up the appropriate pin numbers for TX and RX for the channel you're interested in. There are six SCI ports available in total!

If you decide to use the infrared channel for wireless point-of-sight communication, you'll also need to manage the upper three bits of SCI1BDH. For standard wired communication, these can be all be cleared to zero.

Because integer division might make it impossible to hit the desired baud rate exactly, you will need to select a value that makes the baud rate as close as possible to the target rate. For example, if you wanted to generate a 19 200 baud rate, what value would you put in SCI0BD?

$$8000000 / 16 / x = 19200$$

$$x = 26.0417$$

We can't place a value of 26.0417 into the baud register. A value of 26 will provide a baud rate of 19230.8 baud. The oversampling mechanism used by the receiver compensates to some extent for baud rate mismatch – but it has its limits.

As it turns out, the SCI modules are somewhat tolerant of clock slippage. The Data Sheet indicates that slow data tolerance (characters arriving slower than expected) is 4.63% and fast data tolerance (characters arriving faster than expected) is 3.75%.

It is suggested that your baud rates not deviate by more than 2%, as the other side of the connection will likely have tolerances to deal with as well.

The next configuration register to consider is the SCI Control Register 1, shown on the following page.

11.3.2.2 SCI Control Register 1 (SCICR1)

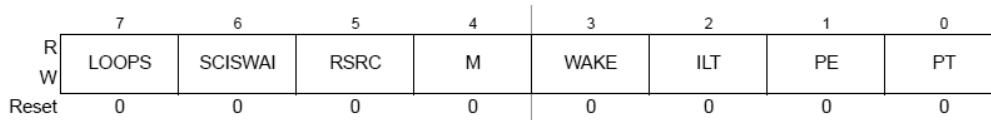


Figure 11-5. SCI Control Register 1 (SCICR1)

Read: Anytime, if AMAP = 0.

Write: Anytime, if AMAP = 0.

NOTE

This register is only visible in the memory map if AMAP = 0 (reset condition).

Table 11-3. SCICR1 Field Descriptions

Field	Description
7 LOOPS	Loop Select Bit — LOOPS enables loop operation. In loop operation, the RXD pin is disconnected from the SCI and the transmitter output is internally connected to the receiver input. Both the transmitter and the receiver must be enabled to use the loop function. 0 Normal operation enabled 1 Loop operation enabled The receiver input is determined by the RSRC bit.
6 SCISWAI	SCI Stop in Wait Mode Bit — SCISWAI disables the SCI in wait mode. 0 SCI enabled in wait mode 1 SCI disabled in wait mode
5 RSRC	Receiver Source Bit — When LOOPS = 1, the RSRC bit determines the source for the receiver shift register input. See Table 11-4. 0 Receiver input internally connected to transmitter output 1 Receiver input connected externally to transmitter
4 M	Data Format Mode Bit — MODE determines whether data characters are eight or nine bits long. 0 One start bit, eight data bits, one stop bit 1 One start bit, nine data bits, one stop bit
3 WAKE	Wakeup Condition Bit — WAKE determines which condition wakes up the SCI: a logic 1 (address mark) in the most significant bit position of a received data character or an idle condition on the RXD pin. 0 Idle line wakeup 1 Address mark wakeup
2 ILT	Idle Line Type Bit — ILT determines when the receiver starts counting logic 1s as idle character bits. The counting begins either after the start bit or after the stop bit. If the count begins after the start bit, then a string of logic 1s preceding the stop bit may cause false recognition of an idle character. Beginning the count after the stop bit avoids false idle character recognition, but requires properly synchronized transmissions. 0 Idle character bit count begins after start bit 1 Idle character bit count begins after stop bit
1 PE	Parity Enable Bit — PE enables the parity function. When enabled, the parity function inserts a parity bit in the most significant bit position. 0 Parity function disabled 1 Parity function enabled
0 PT	Parity Type Bit — PT determines whether the SCI generates and checks for even parity or odd parity. With even parity, an even number of 1s clears the parity bit and an odd number of 1s sets the parity bit. With odd parity, an odd number of 1s clears the parity bit and an even number of 1s sets the parity bit. 0 Even parity 1 Odd parity

This register controls the main communications behaviours of the module. For example, we probably don't want loopback mode, we want the SCI to be enabled in Wait Mode, we want 8 bit data, the device should wake up even on an idle line following a start bit, and we don't want parity checking.

So, we probably want **SCI0CR1 = 0b00000000!**

(Note the typo in the description of "Parity Type Bit". I guess errors are to be expected in a 1300 page document!)

The final configuration register to consider is the SCI Control Register 2.

11.3.2.6 SCI Control Register 2 (SCICR2)

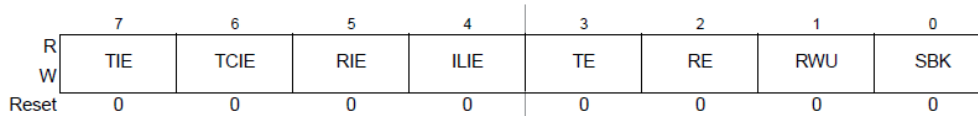


Figure 11-9. SCI Control Register 2 (SCICR2)

Read: Anytime

Write: Anytime

Table 11-9. SCICR2 Field Descriptions

Field	Description
7 TIE	Transmitter Interrupt Enable Bit — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	Transmission Complete Interrupt Enable Bit — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	Receiver Full Interrupt Enable Bit — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled
4 ILIE	Idle Line Interrupt Enable Bit — ILIE enables the idle line flag, IDLE, to generate interrupt requests. 0 IDLE interrupt requests disabled 1 IDLE interrupt requests enabled
3 TE	Transmitter Enable Bit — TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble. 0 Transmitter disabled 1 Transmitter enabled
2 RE	Receiver Enable Bit — RE enables the SCI receiver. 0 Receiver disabled 1 Receiver enabled
1 RWU	Receiver Wakeup Bit — Standby state 0 Normal operation. 1 RWU enables the wakeup function and inhibits further receiver interrupt requests. Normally, hardware wakes the receiver by automatically clearing RWU.
0 SBK	Send Break Bit — Toggling SBK sends one break character (10 or 11 logic 0s, respectively 13 or 14 logics 0s if BRK13 is set). Toggling implies clearing the SBK bit before the break character has finished transmitting. As long as SBK is set, the transmitter continues to send complete break characters (10 or 11 bits, respectively 13 or 14 bits). 0 No break characters 1 Transmit break characters

MC9S12XDP512 Data Sheet, Rev. 2.21

488

Freescale Semiconductor

This register configures power state and interrupts for the module. We aren't interested (yet) in interrupts, but do want the transmitter and receiver turned on.

So, for now, the best choice for configuration is **SCICR2 = 0b00001100**.

SCI0 Library

The following is the SCI0_Lib.h header file that will probably be provided to you by your instructor.

```
//Com port SCI0 initialization and commands
//Processor: MC9S12KDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2015

void SCI0_Init(unsigned long); //any valid baud rate can be passed to this; 8-bit, 1 Stop, No parity, no interrupts
void SCI0_Init9600(void); //8-bit, 1 Stop, No parity, no interrupts
void SCI0_Init19200(void); //8-bit, 1 stop, No parity, no interrupts
void SCI0_TxChar(unsigned char);
unsigned char SCI0_RxChar(void); //Non-blocking; returns NULL if no new valid character is available
void SCI0_TxString(char *); //Requires a NULL-terminated ASCII string in the main program
```

To begin with, you should create the appropriate functions in your SCI0_Lib.c to match the prototypes for the following two initialization routines. Once these are working, you should develop the first routine, *SCI0_Init(IBaud)*, which provides you the flexibility of operating in any of the valid baud rates, but requires that you know what these valid rates are.

- SCI0_Init9600
- SCI0_Init19200

Unfortunately, you won't be able to verify these initialization routines until you've created functions to communicate through the SCI0 port. That's our next item of discussion.

Communicating through the Serial Communications Interface

Characters are transmitted when written to a register called SCIDRL (SCI Data Register, Low byte). There are low and high SCIDR registers, but the high register is only used for 9-bit data formats. Since you will only be using 8-bit data transfers in this course, you will only need to write to the low data register.

Care must be taken to write data to the SCIDRL only when the module is ready. The SCI Status Register 1 (SCISR1) indicates the current status of the port. The following selection from the data sheet has been abbreviated to discuss the only two bits that are significant to us at this point. If you need enhanced error handling, consult the full discussion in the data sheet.

Chapter 11 Serial Communication Interface (S12SCIV5)

11.3.2.7 SCI Status Register 1 (SCISR1)

The SCISR1 and SCISR2 registers provides inputs to the MCU for generation of SCI interrupts. Also, these registers can be polled by the MCU to check the status of these bits. The flag-clearing procedures require that the status register be read followed by a read or write to the SCI data register. It is permissible to execute other instructions between the two steps as long as it does not compromise the handling of I/O, but the order of operations is important for flag clearing.

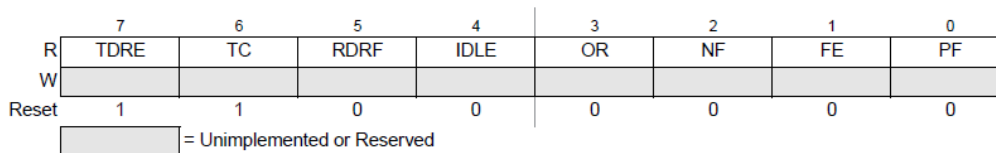


Figure 11-10. SCI Status Register 1 (SCISR1)

Read: Anytime

Write: Has no meaning or effect

Table 11-10. SCISR1 Field Descriptions

Field	Description
7 TDRE	Transmit Data Register Empty Flag — TDRE is set when the transmit shift register receives a byte from the SCI data register. When TDRE is 1, the transmit data register (SCIDRH/L) is empty and can receive a new value to transmit. Clear TDRE by reading SCI status register 1 (SCISR1), with TDRE set and then writing to SCI data register low (SCIDRL). 0 No byte transferred to transmit shift register 1 Byte transferred to transmit shift register; transmit data register empty
5 RDRF	Receive Data Register Full Flag — RDRF is set when the data in the receive shift register transfers to the SCI data register. Clear RDRF by reading SCI status register 1 (SCISR1) with RDRF set and then reading SCI data register low (SCIDRL). 0 Data not available in SCI data register 1 Received data available in SCI data register

When transmitting, check for availability of the port using the Transmit Data Register Empty Flag (TDRE) in bit 7 before attempting to write to the port. If TDRE is a '1', then it is OK to write a new byte to the SCIDRL. NOTE: TDRE does not indicate that transmission is complete – it only indicates that the transmit data register is empty, which is good enough for us. Bit 6 (missing from the above clip) indicates when transmission is actually complete.

To read data from the SCI module you need to read the SCIDRL register. This is a bidirectional register, as writing to it transmits data and reading from it fetches received data.

There's no point in reading the register to get a received byte until a valid new one has actually been received. You can check to see if a byte has been received since the last *read* by looking at the Receive Data Register Full flag (RDRF), which is bit 5 in the SCISR1

register. This flag will be set when a new byte of data has been received by the module. So, if this flag is a 1, you should read from SCIDRL to extract the received byte.

NOTE: The SCI module is only able to buffer one received byte. If you fail to extract a received byte before another is received, a buffer overrun condition occurs and the previous data character will be lost.

When you create a routine to receive a byte of data from the SCI, you don't necessarily want the subroutine to block, waiting for a byte to be received. In actual operation, the byte may never be sent due to a failure in communication, so the subroutine could block forever. As a general rule, you want to avoid creating routines that could block indefinitely.

A better approach would be to check to see if a byte has been received and is waiting to be read. If so, return it; if not return from the subroutine, but indicate that a byte was not available. In an Assembly Language routine, it would be typical to use a condition-code register bit, such as *Carry*, to indicate whether or not a new byte has been received. However, in *C*, that isn't a workable plan. Instead, you may want to return a *NULL* character (ASCII code 0), as that's a very unlikely character to have appearing in a transmitted file. (In some protocols, the actual transmission of a delimiting character like this is indicated by sending the character twice; the programmer would need to do an error-trapping routine that would recognize this condition and handle the character as a special case.) In our case, we'll simply write our *main* program so that it treats any *NULL* returned as an indication that no valid character is present, so we'll ignore that result.

At this point, you will want to write the following functions indicated in your library header file:

- `SCIO_TxChar`
- `SCIO_RxChar`

Once you've written these, you can do a loop-back test by running `SCIO_RxChar` to receive a character from your computer's keyboard, then sending that character by running `SCIO_TxChar` to send the character back to your computer, operating as a "dumb terminal". The software you will likely be asked to use is called Tera Term Pro.

Terminal Emulation

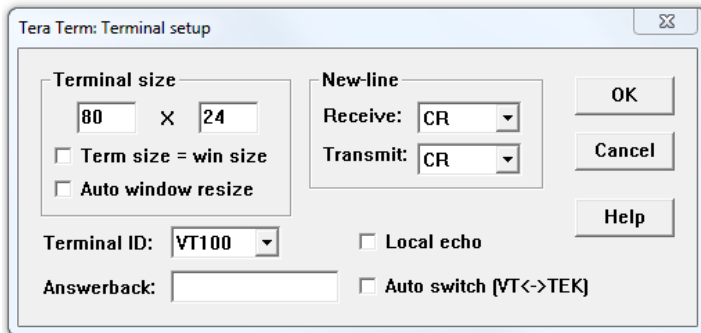
This is your chance to make a highly-advanced and super-fast computer act like a 1960's remote terminal, designed to allow mere mortals to communicate with main-frame computers like the UNIVAC! Here are a couple of pictures of real terminals, which you are going to emulate. The one on the left uses paper instead of a monitor!



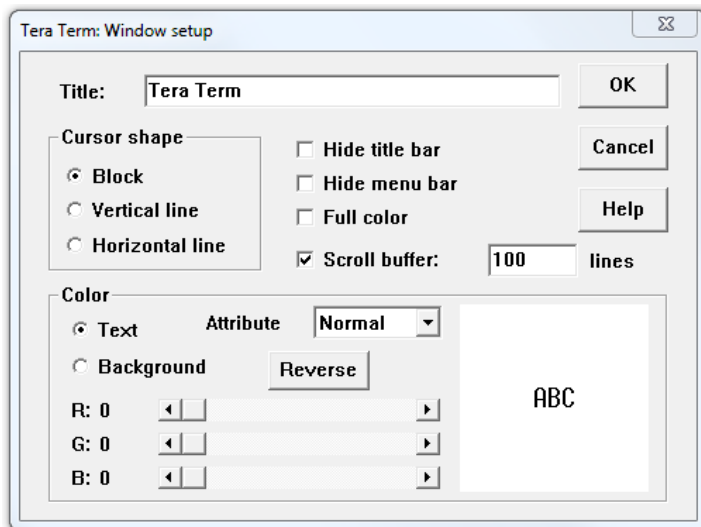
The term “dumb terminal” was coined to indicate that, in this configuration, your computer will be doing nothing other than sending and receiving ASCII characters, just like the terminals shown on the previous page. Of course, since our computers are multi-tasking devices, they will actually be doing a gazillion things in the background; however, the terminal emulator window will not be doing anything other than acting as a dumb terminal.

Inside Tera Term, you will need to set up a number of characteristics so that the dumb terminal can communicate with your 9S12 development kit. These are found in the “Setup” menu.

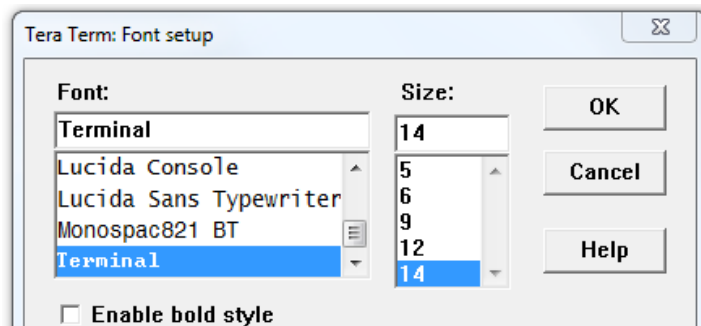
First, you need to make Tera Term into the right dumb terminal. Here’s the “Terminal” setup you want:



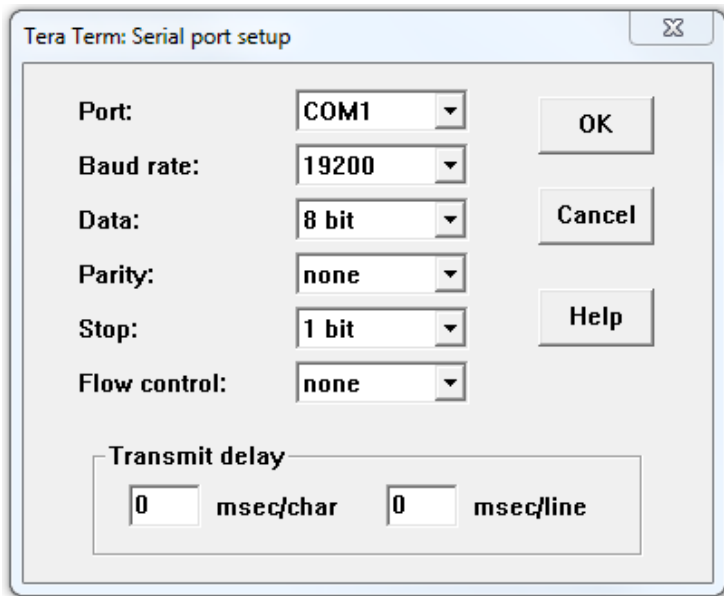
Next, you’ll probably want your “Window” setup to look like the following:



The following “Font” setup makes your display fairly readable:

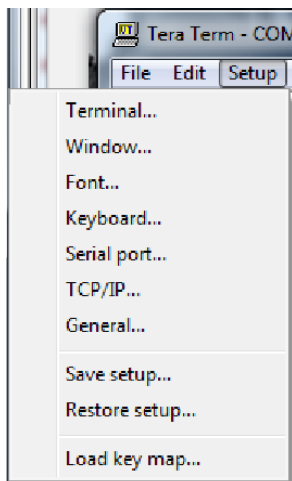


Last but certainly not least, you need to do the following "Serial Port" setup:



This setup matches the settings for the *SCIO_Init19200* initialization routine in your library. If you want to communicate at a different baud rate, this is where you would select that.

Before you leave Tera Term Pro, you will want to save these configuration changes so that you don't have to do this every time you run the dumb terminal.



From the "Setup" menu, select "Save setup..." and simply accept the defaults. This will overwrite the default setup file with your new settings, and Tera Term will start up the way you want it to for the rest of the semester (unless your computer gets re-Ghosted).

Once you've done this configuration, have written a program that initializes the 9S12X, have connected your 9S12X board to the computer using a 9-pin Comm cable, and have written a small program to receive a character from the keyboard and transmit it back to the dumb terminal, you should be able to verify that your functions are working properly (or not). Have fun!

SCIO_TxString

Quite often, you'll want to send multiple characters to the dumb terminal (or to other peripherals or equipment that's looking for ASCII characters). For example, you probably don't want to send the characters for the previously-listed escape sequences one-by-one every time you want to control the terminal. As with the LCD, the best way to send these sequences, as well as longer strings of characters, is by using a **null-terminated string** of ASCII characters. The *SCIO_TxString* function you will be creating can handle strings of any length, since it is expecting a string that ends with the *NULL* character (ASCII code 0). The routine transmits each character, then checks to see if the character was a *NULL*. If it is a *NULL*, program execution exits the function.

You should be able to generalize what you did with *LCD_String(*)* to develop *SCIO_TxString(*)*. If you need further assistance, your instructor can give you direction.

The simple string management program shown on the next page should be a good reference for your work with the SCI port. Here are some pertinent things to notice:

1. The author's version of *SCIO_Init19200* is shown, from which you can build the other initialization routines required. Although it's probably not necessary for most of what you'll be doing, you'll notice that the contents of the upper three bits of the sixteen-bit baud rate register *SCIOBD* have been preserved, in case there's a chance that the infrared settings might be significant. They really shouldn't be, because this SCI port is connected to a wired comm port.
2. Notice that you need to calculate the value for the baud rate register, *SCIOBD*.
3. Note that there was no need to add a *NULL* character to the end of the string – the IDE does that automatically. However, in the declaration of the array size, room had to be provided for the *NULL*.
4. The undeclared *for (;;) {}* loop provides an endless loop, which shows up on the screen as an endless printout of the string.
5. Notice that occasionally, there's an unexpected letter appearing on screen. These characters have been entered from the keyboard using *SCIO_RxByte()*. Note that we read the SCI port, check to see if it contains a valid character from the keyboard, and if it does, we print it.

Note: The header information shown on the next page hasn't been updated to show who wrote it, when, and what it does. As Stan and Jan Berenstain once famously said "That is what you should not do. So let that be a lesson to you." ([The Bike Lesson](#) by Stan and Jan Berenstain, 1964 Random House, Inc.)

The VT100/VT52 Terminal

The terminal program on the PC side of the connection should now be set up to emulate a VT100 terminal. These terminals were able to display received 7-bit ASCII characters, manage a cursor, and send characters from a keyboard through an RS-232 connection.

Escape Sequences

Some sequences of multiple characters are interpreted in a special way by the terminal, and are not displayed. Instead, these sequences trigger a change in the terminal. They may alter the cursor position, character or background color, and other terminal settings.

The character sequences the VT100 terminal recognizes are typically 'escape sequences'. The name comes from the fact that these character sequences begin with an escape character. You will use several different escape sequences to control the terminal.

The following small table shows a few of the sequences of interest. A more complete, but not entirely trustworthy, set is available in Moodle.

A word of caution: Not all terminal emulators produce the same results from the escape sequences, even if they claim to emulate the same terminals (e.g. VT100). You will probably soon find that a number of the sequences you try within Tera Term don't do what you want them to do. HyperTerminal will produce yet other results. Trial and error will, hopefully, bring you satisfactory performance.

Escape Sequence (<esc> means escape character, or 0x1B)	Function
<esc>[2K	Erase Line
<esc>[y;xH or <esc>[y;xf	Set Cursor Position (y = row, x = column)
<esc>[31m	Set text red
<esc>[?25l (that's lowercase L for LOW)	Cursor off
<esc>[?25h (that's lowercase H for HIGH)	Cursor on

Previously, you created ToUpper() and ToLower() routines. One example of using these routines would be in handling the response to a "Y/N" question, received using SCIO_RxChar(). The operator could just as easily enter 'y' instead of 'Y', or 'n' instead of 'N', so your program should respond to either case. This is easily done by simply running the response through ToUpper() and responding to the uppercase value returned.

The next page is a screen capture of a simple “adder” that demonstrates, among other things, the way in which ASCII values from the keyboard need to be manipulated in order to do simple math, and how the results need to be manipulated back to ASCII codes in order to display them in a manner that is meaningful to human operators. Here are some pertinent things to notice:

1. In the string declaration, the escape characters `“\r\n”` are interpreted as a carriage return and line feed respectively, moving the text display on screen down to the beginning of the next line.
2. The string array size is declared to be one larger than the number of characters, to allow for the *NULL* terminator.
3. Single characters can be sent to the terminal by putting single quotes around them.
4. Special characters can be sent to the terminal as hexadecimal numbers, or, for that matter, binary, octal, or decimal numbers. `0x0d` and `0x0a` are the ASCII codes for carriage return and line feed respectively.
5. In order to add two numbers together, they must be true numbers, not ASCII codes; In order to display a true number, it must be converted to ASCII for the terminal.
6. This simple program doesn’t check for non-valid (i.e. non-numeric) entries. A better program would reject these and would wait for a valid input.
7. Note that the larger hex values (A – F) can be entered as either lowercase or uppercase, and are interpreted correctly.
8. This simple routine can’t handle results that are larger than a single digit (i.e. `0x10` up to `0x1E`, which is `0x0F + 0x0F`). Only the lower digit is displayed. A better program would send two characters, or would at least trap the error.

```

main.c
Path: C:\Users\vosst\Desktop\9512X\Projects\AdderOnSCI\Sources\main.c

/*****
*HC12 Program: Adds two single digits, displays one digit
*Processor: MC9S12XDP512
*Xtal Speed: 16 MHz
*Author: P Ross Taylor
*Date: September 2015
*
*Details: Doesn't handle results bigger than 0x0f
*****/

#include <hidef.h> // common defines and macros
#include <stdio.h> // ANSI C Standard Input/Output functions
#include <math.h> // ANSI C Mathematical functions
#include "derivative.h" // derivative-specific definitions

/*****
* Library includes
*****/

#include "SCI0_Lib.h"
#include "Misc_Lib.h"

/*****
* Prototypes
*****/

/*****
* Variables
*****/

char cString[36] = "Press two numbers to get the sum:\r\nf";
char cASCII; //for input and output of ASCII character
char cNum1;
char cNum2;

/*****
* Lookups
*****/

void main(void) // main entry point
{
_DISABLE_COP();

/*****
* Initializations
*****/

SCI0_Init19200();

for (;;) //endless program loop
{
/*****
* Main Program Code
*****/

SCI0_TxString(cString);
cASCII=0; //start with "no input"
while (cASCII==0) //wait for input from keyboard
{
cASCII = SCI0_RxByte();
}

SCI0_TxByte(cASCII); //echo input to screen
SCI0_TxByte('+'); //display a plus sign
cNum1=ASCIIToHex(ToUpper(cASCII)); //idiot-proof number recognition

cASCII=0; //as above
while (cASCII==0)
{
cASCII = SCI0_RxByte();
}
SCI0_TxByte(cASCII);
SCI0_TxByte('='); //display equals sign

cNum2=ASCIIToHex(ToUpper(cASCII)); //number recognition
cNum2+=cNum1; //perform the addition

cASCII=HexToASCII(cNum2); //single-digit ASCII - no error trap

SCI0_TxByte(cASCII); //transmit the sum to screen
SCI0_TxByte(0x0d); //carriage return
SCI0_TxByte(0x0a); //line feed
}
}

```

```

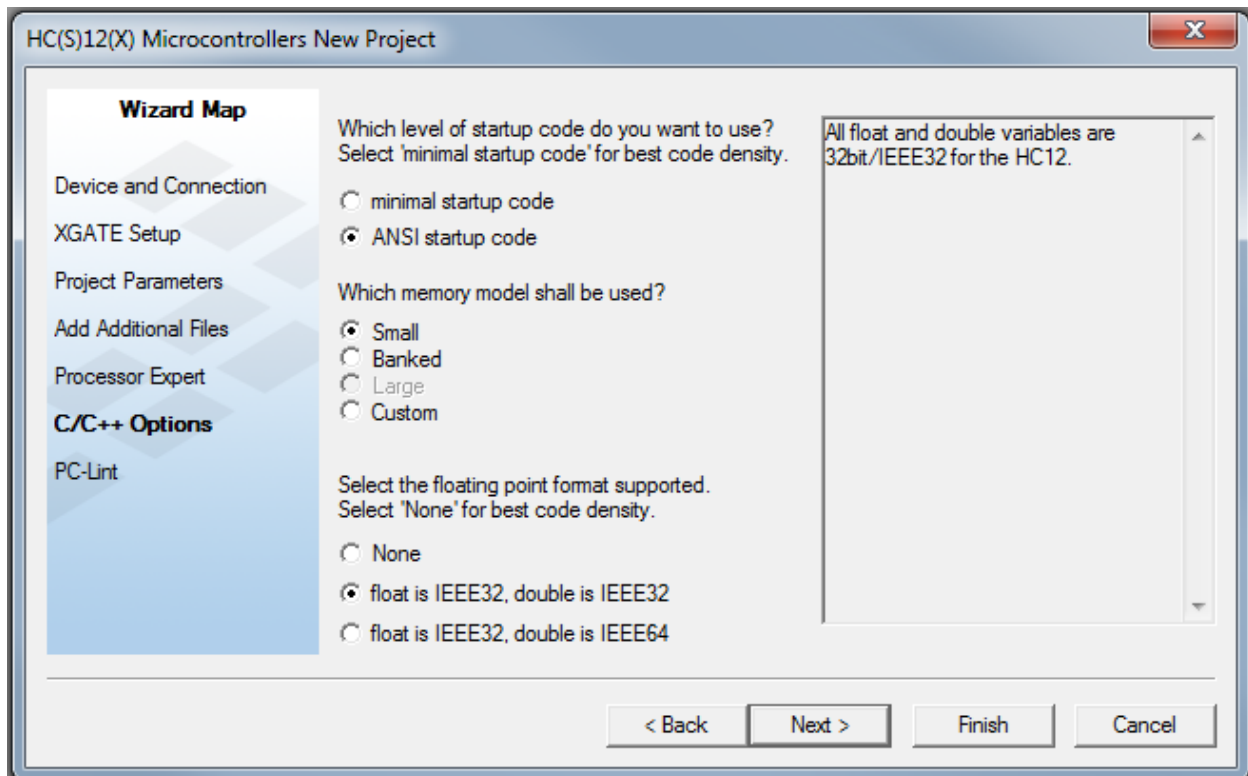
Tera Term - COM1 VT
File Edit Setup Control Window Help
Press two numbers to get the sum:
1+2=3
Press two numbers to get the sum:
0+0=0
Press two numbers to get the sum:
q+t=0
Press two numbers to get the sum:
a+2=C
Press two numbers to get the sum:
A+2=C
Press two numbers to get the sum:
a+b=5
Press two numbers to get the sum:
9+8=1
Press two numbers to get the sum:

```

Floating-Point Math in ANSI C

Up until now, we've been doing mathematical operations using integers, which have no fractional component. The 9S12, operating in ANSI C, can do floating point math, as well. One reason we've been avoiding this until now is that floating point math requires fairly large amounts of memory, takes a lot of clock cycles, and involves some fairly complex functions. However, there are times when it just makes sense to work with real numbers rather than integers, as long as you are aware that download times will increase, you might run out of memory, and most everything is going to slow down.

To begin with, when you are setting up a new project, you need to specify that you are going to be using floating point math, and in which format. One of the screens you've been going to in order to select the memory model also gives you the option of working in floating point:



From experience, your instructors recommend choosing "float is IEEE32, double is IEEE32", as shown above: the other option has been problematic.

Also, in the skeleton file used when creating a new project, you will probably need to invoke the ANSI C `<stdio.h>` library, which, among other things, helps with formatting strings, and, depending on what mathematical functions you intend to use, you may also need to invoke the `<math.h>` library.

<stdio.h>

Here are the functions available in <stdio.h>. You can find it at
C:\Program Files (x86)\Freescale\CWS12v5.1\lib\hc12c\include\math.h.

fopen	fclose	fgetpos
fsetpos	freopen	fseek
scanf	remove	ftell
sscanf	rename	rewind
vsscanf	tmpfile	fgetc
puts	tmpnam	fread
printf	fflush	fwrite
fprintf	setbuf	fgets
vfprintf	setvbuf	fputs
sprintf		fscanf
vsprintf		ungetc
set_printf		gets
vprintf		

Since there's no file structure, or even mass storage device, on your microcontroller board, none of the file-related commands are of any direct use to you, and since your microcontroller board doesn't have a keyboard or monitor, none of the standard user-interface device (console) commands do anything directly, either. So, what use is this library to you? You could potentially set up your board with some "Standard I/O" devices (keyboard interface, video interface, external memory), and then you would have to define the parameters of these devices so the board knew what to access using the console commands. This would be a significant challenge (beyond this course), but not impossible.

However, there are other functions in this library that are of direct use to you. Commands related to string manipulation fit this category; "sprintf", for example, is a function you can use to format strings, particularly those with floating-point numbers in them.

Here's an informative example from a program that has previously received a 12-bit 2's complement value from the X channel of an accelerometer, placed into the variable "iX":

```
if(sprintf(DispString,"X = %+4.3f g      ",iX/1000.0)>0)
{
    LCD_Pos(0,0);
    LCD_String(DispString);
}
```

Note the following:

- All sprintf returns is a flag to indicate a valid result – hence the "if" statement.
- The variable "DispString" was declared previously in the setup for this program as a 21-byte char array, and is used as the target for sprintf. (21 provides enough room for the 20 characters on a line of the display, followed by a NULL terminator.)
- The contents of the string are contained inside double quotes.
- The "%" symbol indicates that formatting commands for a fillable field follow.
- "+" indicates that the sign of the number will be shown, both positive and negative.
- "4.3f" indicates that we want to display a floating point number made up of four digits, of which three will follow the decimal point.
- In the calculation of the value to be placed in the field, we're dividing by 1000.0 (not just 1000) to do an implicit cast of the result into floating point format.

Go here for a complete description of the "sprintf" command:

http://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm

<math.h>

This library has a lot of useful functions in it, as shown below.

frexp	pow	asinf
ldexp	sqrt	acosf
modf	ceil	atanf
frexpf	floor	atan2f
ldexpf	fabs	log10f
modff	fmod	expf
cos	sincos	logf
cosh	sncsh	powf
sin	sqrt_r	sqrtf
sinh	pow_i	ceilf
tan	exp_r	floorf
tanh	log_r	fabsf
asin	cosf	fmodf
acos	coshf	sincosf
atan	sinf	sncshf
atan2	sinhf	sqrtf_r
log10	tanf	powf_i
exp	tanhf	expf_r
log		logf_r

Again, a lot of information about the use and formatting of each of these is available on the Internet when you need to use them.

Of all the functions in this library, you're most likely to use the trigonometry and exponential groups. It should come as no surprise that the trig functions are radians-based, so, if you want to work in degrees, you'll need to do the appropriate conversions.

$$\text{Degrees} = \text{Radians} \left(\frac{180}{\pi} \right)$$

The math.h library provides a value for π , but it's a bit awkward: `_M_PI`. You might want to assign that to a slightly more workable variable name – just make sure it's a *float*. There are other constants available, too. A complete listing can be found in the header file itself, under C:\Program Files (x86)\Freescale\CWS12v5.1\lib\hc12c\include\math.h.

Interrupts

Up to this point, you've been using a technique called "polling" when designing your software for the 9S12X. In this system, you check each peripheral on a regular basis to see if it needs servicing. So, when you use your switches, you check to see if any of them are pressed as part of your program; when you are connected to a dumb terminal through the SCI port, you check to see if a key has been pressed each time you go through that part of your program.

Interrupt programming unleashes a level of power you've experienced in your C# programming – the ability to have one process running and having other processes temporarily take control when another event occurs, such as the click of a mouse button.

Here's an analogy to show you the difference between polling and interrupts. In a classroom, an instructor can constantly walk around the lab benches asking each student, one at a time, if they need help: That's polling, and it keeps the instructor busy all class period long. Or, the instructor can sit at his desk getting caught up on *marking*, while students put up their hands to call him over when they need help: That's using interrupts.

With polling, there's no problem figuring out where to go next in the program: everything is linear, and if the routine that requires your attention is in a subroutine, the program always leaves from a defined point to go to the subroutine, and always returns to where it left.

However, an interrupt can happen at any time and can be completely unpredictable. The program control must be able to leave what it's doing, service the interrupt, then pick up where it was at as if nothing had happened in between.

Interrupts in S12XCPU Assembly Language

Although in this course we won't spend more time programming in S12XCPU Assembly Language, we will revisit it here in order to gain an understanding of how interrupts work.

This is a good time to compare what's required, in S12XCPU Assembly Language, for *branches*, *subroutines*, and *interrupts*.

Action	Going To	Returning From
<i>Branch</i>	Jump to branch address	N/A
<i>Subroutine</i>	Stack the return point address Jump to subroutine address	Retrieve return point from stack (pushes and pulls handled in code)
<i>Interrupt</i>	Stack everything – return address, Y, X, A, B, CCR	Retrieve everything – program continues as if nothing happened

It's also important to know how to get to and return from these types of routines:

Action	Going To	Returning From
<i>Branch</i>	JMP, BRA, LBRA, BRSET, BRCLR, BEQ, BNE, BCC, BCS, BVC, BVS, BGE, BGT, BLE, BPL, BMI, BLT, BHI, BLO, BHS, BLS, DBEQ, DBNE, IBEQ, IBNE	N/A
<i>Subroutine</i>	BSR, JSR	RTS
<i>Interrupt</i>	Vector table	RTI

Note: Since everything is retrieved from the stack when returning from an interrupt, you can't use A, B, D, X, Y, or any of the condition code register bits to return information from an interrupt. You must place information in global variables instead.

So, how do interrupts work? Each item that can be used as an interrupt will have an interrupt enable, an interrupt flag, and an interrupt vector address associated with it. The interrupt vector is a hard-coded address that the microprocessor uses to determine where to transfer control to when a particular interrupt flag is set. The interrupt vector must be programmed with the address of the desired routine (called an interrupt service routine or ISR).

Here's how to set up a program to use an interrupt:

1. Write the interrupt service routine (ISR), preferably collected with other ISRs under a header that sets them apart from the rest of the code.
2. Start the ISR with an informative label that the Assembler will interpret as the entry address.
3. Inside the ISR, make sure you clear the interrupt flag that brought you here. That usually means writing a "1" to that flag in the associated flag register.
4. Make sure you exit the ISR with RTI – never try to exit any other way!
5. Associate the starting address of the ISR with the appropriate Interrupt Vector.
6. In the initialization of your program, enable the specific interrupt for this routine.
7. Enable the maskable interrupts with CLI. (Incidentally, SEI turns interrupts off.)

The following code snippets show how to set up and use the SCI Receive interrupt to read a character from the keyboard and echo it to the screen, while spending the remainder of the time in a power-down WAIT condition.

```

;Initializations
JSR   SCI0Init192

      BSET   SCI0SR1,%00100000   ;clear receive flag
      BSET   SCI0CR2,%00100000   ;enable RIE interrupt

      CLI                               ;enable interrupts

Main:

RegularLoop:
      WAI
      BRA   RegularLoop

;*****
;*   SCI_Echo_ISR
;*Receives a character from the keyboard and echoes it to terminal
;*Requires an initialization routine for the SCI port
;*****

SCI_Echo_ISR:
      BSET   SCI0SR1,%00100000   ;clear interrupt flag
      LDAA  SCI0DRL              ;receive character from keyboard
      JSR   SCI0TxByte           ;echo to screen
      RTI

;*****
;*   Interrupt Vectors
;*****
      ORG   $FFD6                ;SCI0 SCI1CR2
      DC.W SCI_Echo_ISR

```

The screen capture below shows just the top of a three-page listing of interrupt vectors, enough to show where the microcontroller jumps to if an interrupt occurs on Timer Channel 0, which we're using in this example:

Table 1-12. Interrupt Vector Locations (Sheet 1 of 3)

Vector Address ¹	XGATE Channel ID ²	Interrupt Source	CCR Mask	Local Enable
\$FFFE	—	System reset or illegal access reset	None	None
\$FFFC	—	Clock monitor reset	None	PLLCTL (CME, SCME)
\$FFFA	—	COP watchdog reset	None	COP rate select
Vector base + \$F8	—	Unimplemented instruction trap	None	None
Vector base + \$F6	—	SWI	None	None
Vector base + \$F4	—	\overline{XIRQ}	X Bit	None
Vector base + \$F2	—	\overline{IRQ}	I bit	IRQCR (IRQEN)
Vector base + \$F0	\$78	Real time interrupt	I bit	CRGINT (RTIE)
Vector base + \$EE	\$77	Enhanced capture timer channel 0	I bit	TIE (C0I)
Vector base + \$EC	\$76	Enhanced capture timer channel 1	I bit	TIE (C1I)
Vector base + \$EA	\$75	Enhanced capture timer channel 2	I bit	TIE (C2I)

The default Vector base is \$FF00, so you would add this to the Vector Address column value.

It's possible to have multiple Interrupt Service Routines running simultaneously. The code shown on the following page counts up endlessly on the bottom four digits of the seven-segment display, maintains a timer, grabs characters from console when interrupted by the SCI0 Receive Interrupt, and clears the display in response to an interrupt generated on a switch connected to bit 0 of PortJ.

```

;*****
;* HC12 Program:  Creates an upcounter based on an interrupt timer *
;* Processor:    MC9S12XDP512 *
;* Xtal Speed:   16 MHz *
;* Author:       F Ross Taylor *
;* Date:         April 2014 *
;* Details: Also responds to interrupts from the SCI Rx (keyboard) *
;* *
;*****

;export symbols
XDEF      Entry           ;export 'Entry' symbol
ABSENTRY  Entry           ;for absolute assembly: app entry point

;include derivative specific macros
INCLUDE   'derivative.inc'

;*****
;* Equates *
;*****

;*****
;* Variables *
;*****
TimCounter: DC.W 1 ;Address $2000

;*****
;* Code Section *
;*****
Entry:      ORG      ROM_4000Start ;Address $4000 (FLASH)
LDS        #RAMEnd+1 ;initialize the stack pointer

Main:
;Initializations
JSR        SevSeg_Init
JSR        SCI0Init192

        MOVW    #%10000000,TSCR1 ;enable timer module 0
        LDAB   #%00000111 ;2^7 prescaler
        STAB   TSCR2 ;prescaler set to Bus/(2^B)
        MOVW    #%00000001,TIOS ;set IOS0 for output compare
        MOVW    #%00000001,TCTL2 ;toggle mode for PT0
        LDD    #6250 ;100 ms
        ADDD   TCNT ;set new event timer value based on clock
        STD    TCO

        BSET   TFLG1,%00000001 ;clear flag
        BSET   TIE,%00000001 ;enable OC0 interrupt

        BSET   SCIOSR1,%00100000 ;clear receive flag
        BSET   SCIOCR2,%00100000 ;enable RIE interrupt

        BCLR   DDRJ,%00000001 ;ensure input at PortJ0
        BSET   PPSJ,%00000001 ;set polarity for rising edge (key press)
        BSET   PIFJ,%00000001 ;clear PortJ0 flag
        BSET   PIEJ,%00000001 ;enable PortJ0 interrupt

        CLI    ;enable interrupts

RegularLoop:
LDD    #0
LDX    #0
DBNE  X,*
ADDD  #1
JSR   SevSeg_Low4
BRA  RegularLoop

;*****
;* Subroutines *
;*****

;*****
;* Interrupt Service Routines *
;*****
Timer_ISR:
        BSET   TFLG1,%00000001 ;clear interrupt flag
        LDD    #6250 ;place counter delay value in D
        ADDD   TCO ;add old event target
        STD    TCO ;new event target
        LDD    TimCounter ;increment the counter variable
        ADDD   #1
        STD    TimCounter
        JSR   SevSeg_Top4 ;display the new value
        RTI

SCI_Echo_ISR:
        BSET   SCIOSR1,%00100000 ;clear interrupt flag
        LDAA  SCIODR1 ;receive character
        JSR   SCIOtxByte
        RTI

Switch_ISR:
        BSET   PIFJ,%00000001 ;clear interrupt flag
        LDD    #0 ;ready to blank top display
        STD    TimCounter
        RTI

;*****
;* Constants *
;*****
ORG      ROM_C000Start ;second block of ROM

;*****
;* Lock-Up Tables *
;*****

;*****
;* SCI VTI00 Strings *
;*****

;*****
;* Absolute Library Includes *
;*****
INCLUDE "Misc_Lib.inc"
INCLUDE "SCI0_Lib.inc"
INCLUDE "SevSeg_Lib.inc"

;*****
;* Interrupt Vectors *
;*****
ORG      $FFD6 ;SCI0 SCII1CR2
DC.W    SCI_Echo_ISR

ORG      $FFCE ;Port J
DC.W    Switch_ISR

ORG      $FFEE ;TIE C0I
DC.W    Timer_ISR

ORG      $FFFE ;Reset Vector
DC.W    Entry

```

Interrupts using ANSI C

Interrupt handling in ANSI C is different from handling interrupts in Assembly Language, as we rely on something called pragma interrupt handling. Also, since we're not allowed to pass parameters to or from an interrupt (remember that everything gets stacked upon entry and then everything gets pulled back off the stack at the end of the ISR), we'll have to use global variables for anything we want to send to or receive from an interrupt routine.

Recall that there are seven things we need to do to handle an interrupt properly:

1. Write the interrupt service routine (ISR), preferably collected with other ISRs under a header that sets them apart from the rest of the code.
2. Start the ISR with an informative label that the compiler will interpret as the entry address.
3. Inside the ISR, make sure you clear the interrupt flag that brought you here. That usually means writing a "1" to that flag in the associated flag register.
4. Make sure you exit the ISR properly (not so hard in C).
5. Associate the ISR with the appropriate Interrupt Vector.
6. In the initialization of your program, enable the specific interrupt for this routine.
7. Enable the maskable interrupts.

For the SCI port, the most useful interrupt is the Receive Data Register Full (RDRF), which we have used in our polling routines. Since people type pretty slowly, compared to the processing rate of a microcontroller, and since people tend to take relatively long breaks to think about what they are typing or to get a coffee, leaving the microcontroller in a blocking routine while it waits for a new character is an incredible waste of time and computing power.

If you don't want to try and find the Vector Handlers in the 1300 page data sheet, you can open up the "mc9s12xdp512.h" file that always shows up in your projects when you make the correct initial selections. Here's a snippet out of that file, which should be generally useful for the interrupt-driven parts of this course.

```
#define VectorNumber_Vtimch0 200
#define VectorNumber_Vporth 250
#define VectorNumber_Vportj 240
#define VectorNumber_Vatd1 230
#define VectorNumber_Vatd0 220
#define VectorNumber_Vscil 210
#define VectorNumber_Vsci0 200
#define VectorNumber_Vspi0 190
#define VectorNumber_Vtimpaie 180
#define VectorNumber_Vtimpaovf 170
#define VectorNumber_Vtimovf 160
#define VectorNumber_Vtimch7 150
#define VectorNumber_Vtimch6 140
#define VectorNumber_Vtimch5 130
#define VectorNumber_Vtimch4 120
#define VectorNumber_Vtimch3 110
#define VectorNumber_Vtimch2 100
#define VectorNumber_Vtimch1 90
#define VectorNumber_Vtimch0 80
#define VectorNumber_Vrti 70
```

Here are some important facts to use in setting up an SCI0 Receive interrupt routine:

- SCI0 interrupts are handled by "VectorNumber_Vsci0". (Be careful, as the cross-compiler is case sensitive.)
- RDRF is bit5 of SCI0SR1.
- Flags are cleared by writing a "1" to them.
- The receive interrupt enable, RIE, is bit5 of SCI0CR2.
- The interrupts you've selected are ultimately enabled using "EnableInterrupts", which is the equivalent of "CLI".
- You can use "asm WAI" to put the microcontroller to sleep, waiting for an interrupt.

Points 2 to 5 of the things that are needed for an interrupt-driven program are done for you in the following screen capture. Notice, in particular, the syntax of the declaration of the vector handler and its associated interrupt service routine – the top line of this code. The function is void(void) because everything is placed on the stack and retrieved from the stack. If you want to have the ISR work on values or provide results, use global variables as the memory access points.

```
/******  
*   Interrupt Service Routines  
*****/  
  
interrupt VectorNumber_Vsci0 void SCI_Echo(void)  
{  
    SCIOSR1|=0b00100000;    //clear the flag  
    //put your code here  
}  
  
/******
```

At this point, you should be able to write an ANSI C equivalent of the “receive and echo” program shown earlier in S12XCPU Assembly Language.

By the way, you may have noticed that, in one of the ISRs the flag was cleared at the beginning of the routine and in the other the flag was cleared at the end. This doesn’t matter with Motorola-based microcontrollers, as the interrupt is automatically disabled as long as the interrupt is being serviced.

Don’t forget points 6 and 7! They need to be done outside of the ISR.

Input-Driven Interrupt

Often, we want our microcontroller to respond to simple external events, such as can be detected by a logic change on an input.

On our board, Port J can be used to generate interrupts in response to a change in the signals connected to it.

The microcontroller board has PortJ0 and PortJ1 connected to push-button switches. Of course, you could also wire one or both of these to a circuit of your own design, or an add-on peripheral, that generates binary logic levels as external interrupts.

The following is from page 819 of the "Data Sheet":

0x0268	Port J Data Register (PTJ)	Read / Write ¹
0x0269	Port J Input Register (PTIJ)	Read
0x026A	Port J Data Direction Register (DDRJ)	Read / Write ¹
0x026B	Port J Reduced Drive Register (RDRJ)	Read / Write ¹
0x026C	Port J Pull Device Enable Register (PERJ)	Read / Write ¹
0x026D	Port J Polarity Select Register (PPSJ)	Read / Write ¹
0x026E	Port J Interrupt Enable Register (PIEJ)	Read / Write ¹
0x026F	Port J Interrupt Flag Register (PIFJ)	Read / Write ¹

In order to use PortJ0 and/or PortJ1, we need to define them as inputs, using DDRJ.

Also, we need to specify if we want interrupts to be generated on a rising or falling edge of the input signal using PPSJ, where 0 is for a falling edge and 1 is for a rising edge.

Once the port is set up, we can enable the interrupt for our particular channel using PIEJ.

Interrupt events will be reported using PIFJ, which will have to be cleared before further interrupts can be detected.

The Interrupt Vector for PortJ is \$FFCE, and the Interrupt Vector Handler for ANSI C is VectorNumber_Vportj.

There are some distinct advantages to using an interrupt service routine for binary input signals, particularly if they are generated by switches.

For one, we don't need to worry about long activation time for the switch, because a single event gets us into the service routine, and further events won't have any effect until we clear the interrupt flag and exit the ISR. Holding the switch down doesn't generate any more edges, so no further action is detected until the switch is reactivated. Also, events are only initiated by the edge we've chosen, so, for example, if we've chosen a rising edge to detect switch press, the falling edge for the switch release condition will be ignored.

Switch bounce is also significantly reduced, as the switch will probably stabilize during the operation of the interrupt service routine. If your switch bounces upon release, however, you may get an unwanted event after the service routine has finished its operation, which you may need to find a solution for. This will probably vary from one application to another.

Accurate Timing

Up to this point, you've created simple but not necessarily very accurate timing delays using counters and loops. There are better ways to do this!

Periodic Interrupt Timer (PIT)

If all you're looking for is something to clock a repetitive sequence very accurately, the Periodic Interrupt Timer is a good choice. As the name indicates, this timer runs on interrupts, and, when set up properly, will run an interrupt service routine at exact, periodic intervals. The specification document for the 9S12XDP512 provides this block diagram:

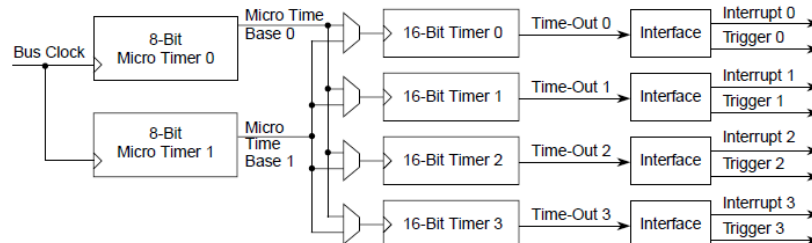


Figure 13-1. PIT Block Diagram

From this, we can see that there are two 8-bit counters ("Micro Timers") running from the 8 MHz bus clock. Each of these can be loaded with a different countdown value using registers PITMTLD0 and PITMTLD1. These counters can be connected to any of four 16-bit timers using a multiplexer register (PITMUX), and these counters further divide down the clock signals, based on values loaded into PITLDx, where "x" is 0 through 3. When these counters run down, they generate interrupts that can be detected by the 9S12 core.

To determine the period of a particular PIT timer, use the following formula:

$$T = (PITMTLD_y + 1) * (PITLD_x + 1) / f_{bus}$$

The frequency, if you want it, is just the inverse.

We won't use any more than one interrupt timer at any given time in this course, because, the way the 9S12XDP512 is set up, it can't respond to more than one interrupt at a time, and it's quite difficult to set up two timers that don't occasionally fire on the same clock cycle, in which case the lower priority interrupt will be missed. (If you're interested, the priority of interrupts and ways to handle nested interrupts are discussed in the specification document and *ad nauseum* online.)

The registers you will need to deal with for simple operation of a single channel, in this case, channel 0 based on microtimer 0, are as follows:

PITCE	four lower bits enable corresponding channels
PITMUX	a 0 or 1 in the bit matching your channel connects to microtimer 0 or 1
PITCFLMT	PIT control register (upper three bits enable, control debug activity)
PITINTE	1 in lower four bits enables corresponding channel
PITMTLD0	microtimer count down value
PITLD0	timer count down value

The interrupt vector for Channel 0 is VectorNumber_VPit0.

The next page shows a 1.0 s periodic interrupt timer running a BCD counter on the seven segment display. Just remember, when setting up your values, that the microtimer is 8-bit (0 to 255) and the timer is 16-bit (0 to 65,535).

```

/*****
*   Variables
*****/
unsigned int CountT = 0;

void main(void)    // main entry point
{
    _DISABLE_COP();
/*****
*   Initializations
*****/
    SevSeg_Init();

    PITCE |= 0b00000001;    //enable PIT0 0channel
    PITMUX &= 0b11111110;    //connect PIT0 to microtimer 0
    PITCFLMT = 0b10100000;    //Enable, /StopOnWait, StallInFreeze, xxx00 (don't force load)
    PITINTE |= 0b00000001;    //enable PIT0 interrupt
    PITMTLDO = 159;    //microtimer 0 set to 20 us (159+1)*125 ns
    PITLDO = 49999;    //top display updates every 1000 ms (49999+1)*20 us

    EnableInterrupts;

    for (;;)    //endless program loop
    {
/*****
*   Main Program Code
*****/
        asm WAI;    //wait for interrupt
        CountT++;    //increment counter
        if(CountT>9999) CountT=0;    //roll over at 9999
        SevSeg_Top4(HexToBCD(CountT));    //display value
    }
/*****
*   Interrupt Service Routines
*****/
interrupt VectorNumber_Vpit0 void PIT0Int (void)
{
    PITTF |= 0b00000001;    //clear PIT0 interrupt flag
}

```

Enhanced Capture Timer

For more control (and the possibility of connecting timer channels to output pins), the 9S12XDP512 chip contains a built-in enhanced timer module, with a great deal of functionality and flexibility. The block diagram for the timer module is in the 9S12X data sheet. In the current version, this is on page 310. Here it is for quick reference:

Chapter 7 Enhanced Capture Timer (S12ECT16B8CV2)

7.1.3 Block Diagram

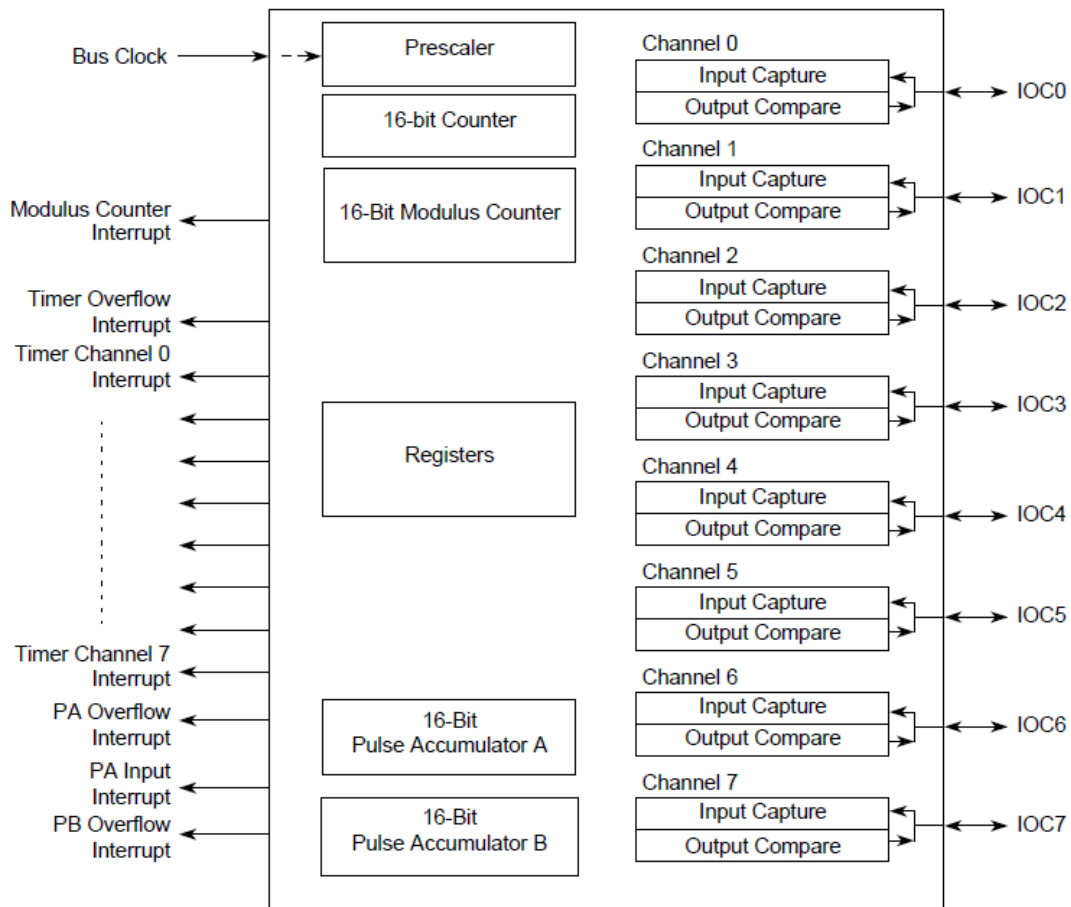


Figure 7-1. ECT Block Diagram

In other versions of the 9S12 microcontroller, there are multiple enhanced capture timers available. However, in the MC9S12XDP512, the manufacturers made room for multiple CAN Bus modules by removing all but one of the timer modules. In those versions of the controller, you need to specify which module you want (e.g. TIM0_?????). We don't have to do that with this version of the controller.

At the core of the timer module exists a 16-bit counter, the current count value being available at the 16-bit location TCNT (for us, this is at addresses 0x0044 and 0x0045). This counter simply counts up as long as the timer module is enabled. The points of interest here are that this is a 16-bit register (occupies two bytes), and a 16-bit read takes an instantaneous snapshot of the register's contents, while the counter itself runs on.

There are a number of registers that need to either be written to for control or read from to determine the current status of the timer module. Here's a screen-shot showing all of the available registers. Pretty daunting!

Chapter 7 Enhanced Capture Timer (S12ECT16B8CV2)

7.3 Memory Map and Register Definition

This section provides a detailed description of all memory and registers.

7.3.1 Module Memory Map

The memory map for the ECT module is given below in Table 7-1. The address listed for each register is the address offset. The total address for each register is the sum of the base address for the ECT module and the address offset for each register.

Table 7-1. ECT Memory Map

Address Offset	Register	Access
0x0000	Timer Input Capture/Output Compare Select (TIOS)	R/W
0x0001	Timer Compare Force Register (CFORC)	R/W ¹
0x0002	Output Compare 7 Mask Register (OC7M)	R/W
0x0003	Output Compare 7 Data Register (OC7D)	R/W
0x0004	Timer Count Register High (TCNT)	R/W ²
0x0005	Timer Count Register Low (TCNT)	R/W ²
0x0006	Timer System Control Register 1 (TSCR1)	R/W
0x0007	Timer Toggle Overflow Register (TTOV)	R/W
0x0008	Timer Control Register 1 (TCTL1)	R/W
0x0009	Timer Control Register 2 (TCTL2)	R/W
0x000A	Timer Control Register 3 (TCTL3)	R/W
0x000B	Timer Control Register 4 (TCTL4)	R/W
0x000C	Timer Interrupt Enable Register (TIE)	R/W
0x000D	Timer System Control Register 2 (TSCR2)	R/W
0x000E	Main Timer Interrupt Flag 1 (TFLG1)	R/W
0x000F	Main Timer Interrupt Flag 2 (TFLG2)	R/W
0x0010	Timer Input Capture/Output Compare Register 0 High (TC0)	R/W ³
0x0011	Timer Input Capture/Output Compare Register 0 Low (TC0)	R/W ³
0x0012	Timer Input Capture/Output Compare Register 1 High (TC1)	R/W ³
0x0013	Timer Input Capture/Output Compare Register 1 Low (TC1)	R/W ³
0x0014	Timer Input Capture/Output Compare Register 2 High (TC2)	R/W ³
0x0015	Timer Input Capture/Output Compare Register 2 Low (TC2)	R/W ³
0x0016	Timer Input Capture/Output Compare Register 3 High (TC3)	R/W ³
0x0017	Timer Input Capture/Output Compare Register 3 Low (TC3)	R/W ³
0x0018	Timer Input Capture/Output Compare Register 4 High (TC4)	R/W ³
0x0019	Timer Input Capture/Output Compare Register 4 Low (TC4)	R/W ³
0x001A	Timer Input Capture/Output Compare Register 5 High (TC5)	R/W ³
0x001B	Timer Input Capture/Output Compare Register 5 Low (TC5)	R/W ³
0x001C	Timer Input Capture/Output Compare Register 6 High (TC6)	R/W ³
0x001D	Timer Input Capture/Output Compare Register 6 Low (TC6)	R/W ³

MC9S12XDP512 Data Sheet, Rev. 2.21

Freescale Semiconductor

312

Chapter 7 Enhanced Capture Timer (S12ECT16B8CV2)

Table 7-1. ECT Memory Map (continued)

Address Offset	Register	Access
0x001E	Timer Input Capture/Output Compare Register 7 High (TC7)	R/W ³
0x001F	Timer Input Capture/Output Compare Register 7 Low (TC7)	R/W ³
0x0020	16-Bit Pulse Accumulator A Control Register (PACTL)	R/W
0x0021	Pulse Accumulator A Flag Register (PAFLG)	R/W
0x0022	Pulse Accumulator Count Register 3 (PACN3)	R/W
0x0023	Pulse Accumulator Count Register 2 (PACN2)	R/W
0x0024	Pulse Accumulator Count Register 1 (PACN1)	R/W
0x0025	Pulse Accumulator Count Register 0 (PACN0)	R/W
0x0026	16-Bit Modulus Down Counter Register (MCCTL)	R/W
0x0027	16-Bit Modulus Down Counter Flag Register (MCFLG)	R/W
0x0028	Input Control Pulse Accumulator Register (ICPAR)	R/W
0x0029	Delay Counter Control Register (DLYCT)	R/W
0x002A	Input Control Overwrite Register (ICOWW)	R/W
0x002B	Input Control System Control Register (ICSYS)	R/W ⁴
0x002C	Reserved	--
0x002D	Timer Test Register (TIMST)	R/W ⁵
0x002E	Precision Timer Prescaler Select Register (PTPSR)	R/W
0x002F	Precision Timer Modulus Counter Prescaler Select Register (PTMCPSPR)	R/W
0x0030	16-Bit Pulse Accumulator B Control Register (PBCTL)	R/W
0x0031	16-Bit Pulse Accumulator B Flag Register (PBFLG)	R/W
0x0032	8-Bit Pulse Accumulator Holding Register 3 (PA3H)	R/W ⁵
0x0033	8-Bit Pulse Accumulator Holding Register 2 (PA2H)	R/W ⁵
0x0034	8-Bit Pulse Accumulator Holding Register 1 (PA1H)	R/W ⁵
0x0035	8-Bit Pulse Accumulator Holding Register 0 (PA0H)	R/W ⁵
0x0036	Modulus Down-Counter Count Register High (MCCNT)	R/W
0x0037	Modulus Down-Counter Count Register Low (MCCNT)	R/W
0x0038	Timer Input Capture Holding Register 0 High (TC0H)	R/W ⁵
0x0039	Timer Input Capture Holding Register 0 Low (TC0H)	R/W ⁵
0x003A	Timer Input Capture Holding Register 1 High (TC1H)	R/W ⁵
0x003B	Timer Input Capture Holding Register 1 Low (TC1H)	R/W ⁵
0x003C	Timer Input Capture Holding Register 2 High (TC2H)	R/W ⁵
0x003D	Timer Input Capture Holding Register 2 Low (TC2H)	R/W ⁵
0x003E	Timer Input Capture Holding Register 3 High (TC3H)	R/W ⁵
0x003F	Timer Input Capture Holding Register 3 Low (TC3H)	R/W ⁵

¹ Always read 0x0000.² Only writable in special modes (test_mode = 1).³ Writes to these registers have no meaning or effect during input capture.⁴ May be written once when not in test00mode but writes are always permitted when test00mode is enabled.⁵ Writes have no effect.

MC9S12XDP512 Data Sheet, Rev. 2.21

Freescale Semiconductor

313

Clearly, we can only touch on a small subset of all the capabilities of this very important module, so let's get started.

Initially, we will be working with seven of the registers:

TSCR1	Timer System Control Register 1
TSCR2	Timer System Control Register 2
TIOS	Timer Input Capture/Output Compare Select
TCTL2	Timer Control Register 2 (This is half of a 16-bit TCTL register)
TCNT	Timer Count Register
TC0	Timer Input Capture/Output Compare Register 0
TFLG1	Main Timer Interrupt Flag 1

Once we have the timer working, you might be interested in checking out the functionality of Input Capture, which requires TCTL3 and TCTL4, and reports the clock value when an external event happens. You could also try Pulse Accumulation, which requires PACTL and the 16-bit Pulse Accumulator Count register PACN32, and counts the number of external events detected during a period of time. Input capture can be used to determine the **period** of a periodic waveform, whereas pulse accumulation can be used to determine the **frequency** of a periodic waveform. Of course, there are a lot of other applications for these, too, such as determining revolutions per minute (RPM) of a rotating shaft or the time between two external events such as the time between front tires and back tires of a car crossing a sensor to determine its speed.

Timer Initialization

To initialize the timer, we need to do the following:

1. Enable the timer module.
2. Determine the rate at which we want the timer to count up, based on a prescaler from the main bus clock.
3. Set up the timer to operate in "Output Compare" mode for one of the 8 channels – for now, we'll use IOC0.
4. Connect an output signal to an external pin available on the microcontroller kit.
5. Clear the Output Compare Flag so the timer is ready to announce the first Output Compare instance.

Here are the registers we'll be working with in the initialization for Output Compare on Channel IOC0:

TSCR1	R	TEN	TSWAI	TSFRZ	TFFCA	PRNT	0	0	0
	W								
TSCR2	R	TOI	0	0	0	TCRE	PR2	PR1	PR0
	W								
TIOS	R	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
	W								
TCTL1	R	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
	W								
TCTL2	R	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
	W								
TCNT (High)	R	TCNT15	TCNT14	TCNT13	TCNT12	TCNT11	TCNT10	TCNT9	TCNT8
	W								
TCNT (Low)	R	TCNT7	TCNT6	TCNT5	TCNT4	TCNT3	TCNT2	TCNT1	TCNT0
	W								
TC0 (High)	R	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
	W								
TC0 (Low)	R	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	W								
TFLG1	R	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
	W								

For our basic initialization routine, we want to do the following:

1. Turn on the timer.
2. Set the prescaler so that the timer's count interval is 8 μ s (to begin with).
3. Set up Channel 0 for Output Compare.
4. Connect Channel 0 to its corresponding pin in PortT, PT0, in Toggle Mode.
5. Clear the capture flag for Channel 0.

In your "Misc_Lib.c" library, begin a function to match the following header which should be in the corresponding "Misc_Lib.h" file:

```
void TimInit8us(void);
```

1. Turn on the Timer using TSCR1, but don't mess up the contents of the other bits in this register. (You'll need to OR or AND the contents of the register for this).
2. In TSCR2, you'll notice that there are only three bits available for the prescaler. These bits represent the power of 2 that you want to divide the clock frequency by. Your options, then, are 1, 2, 4, 8, 16, 32, 64, or 128. Dividing the clock frequency means the same as multiplying the period, so this is also the power of 2 that you multiply the 125 ns clock period by. So, for the desired 8 μs, we want:

$$\frac{8\mu s}{125ns} = 64 = 2^6$$

...so the prescaler should be 6, or 110₂.

You will need to create this prescaler without messing up the other bits, so "OR" in the 1s, and "AND" out the 0s.

Alternatively, you could store the entire thing temporarily, clear out the prescaler bits in the temporary copy, write the desired values into the prescaler, then OR the original bits back into the register. You'll probably find the first way of doing this to be the easier of the two.

3. Set up TIOS to enable Output Compares on Channel 0. Consistent with other operations in this microcontroller, a 1 represents output and a 0 represents input. Again, don't mess up the other bits in this register.
4. Now to connect Channel 0's Output Compare events to a pin that we can monitor with an oscilloscope. Notice that the TCTL register is 16-bit, even though there are only 8 channels. That's because each channel has four options available to it, hence the need for 2 bits. Here's the table that explains this operation:

Table 7-10. Compare Result Output Action

OMx	OLx	Action
0	0	Timer disconnected from output pin logic
0	1	Toggle OCx output line
1	0	Clear OCx output line to zero
1	1	Set OCx output line to one

We want to toggle PT0 in response to events on Channel 0, so we want to use 01₂ for the bottom two bits of TCTL2. Again, we don't want to meddle with the settings of the other bits in TCTL2, so this will require AND-ing and OR-ing.

5. Finally, we want to clear the interrupt flag associated with Channel 0. For many peripherals, a flag is "cleared" by "setting" it. In other words, to make a flag go to 0, you write a 1 to it. Do this to the appropriate bit in TFLG1 using OR.

If you followed the steps above, you should have ended up with a function that looks very much like the following:

```
void TimInit8us(void)
{
    TSCR1 |= 0b10000000;    //enable timer module
    TSCR2 &= 0b11111000;    //set prescale to Bus/64 (8 us per tick)
    TSCR2 |= 0b00000110;    //...continued
    TIOS  |= 0b00000001;    //set IOS0 to output compare
    TCTL2 &= 0b11111100;    //set PT0 to toggle mode
    TCTL2 |= 0b00000001;    //...continued
    TFLG1 |= 0b00000001;    //clear flag
}
```

Hopefully, you took the time to go through the preceding steps instead of looking ahead and just putting this code into your library, because in doing so, you will have learned quite a bit about the operation of the timer.

At this point, you should be able to use an oscilloscope to see if your initialization routine worked, because PT0 will be changing states once for each complete cycle of the count in TCNT.

Why? Here's a quick explanation. A register called TC0 is compared to TCNT each tick to determine if an output compare event has occurred on this channel. Since the TC0 register will be some unspecified value upon startup (most probably \$0000), an output compare event will occur naturally each time TCNT wraps around to that value. This happens every 65536 ticks at 8 μ s sec/tick, or every 0.524288 s. So, without us manipulating the value in TC0, the board will produce a square wave with a period of 1.048576 s, or a frequency of 0.954 Hz. Check to see if that's what's happening at PT0, by observing the signal at Pin 9 of the microcontroller using your oscilloscope.

We will typically manipulate the amount of time between output compare events by changing the contents of TC0 for every desired time period.

If your 8 μ s initialization routine worked, add an initialization routine called "TimInit1us". This should be very similar to the one you've just finished, except for the timing. Make appropriate changes to match the label. This one should produce a free-running frequency of 7.63 Hz.

Add another routine called "TimInit125ns". This one should free-run at about 61 Hz.

Setting the Timer Compare Event Duration

Every time TCNT matches TC0, the corresponding flag is set in TFLG1. If you want to know when an output compare event has occurred, poll for the event flag in the TFLG1 register. Another way to handle this, which we will soon investigate, is to enable interrupts and allow the timer to interrupt the main program whenever the flag is set.

The simplest way to handle an output compare is to repeatedly check for the flag of the corresponding channel in your code (a blocking delay). Once the event is detected, you must write a 1 to the corresponding bit to acknowledge it and reset the flag.

For more useful systems, you should check the flag once each time through a loop that allows you to carry out other functions, such as checking switches or controlling LEDs. Alternatively, you could use timer interrupts, which we will soon address.

Delays vs. Intervals

We can use the timer in two slightly different ways: individual delays or regular intervals.

Here's a real-life analogy. You may want to sleep an extra ten minutes before getting up in the morning. To do this, you check the time, add ten minutes to it, and set your alarm for the new time. When the alarm goes off, you've experienced a ten-minute delay. After yawning and stretching, you may feel like another ten-minute delay, so you check the clock again, add ten minutes, and set the alarm to the new alarm time. As a result, your alarm will go off a bit more than twenty minutes after the original wake-up time. If, however, you've got kids playing in the back yard, you may want to check on them every ten minutes. In this case, you check the initial time on the clock and add ten minutes to it to set the alarm. When the alarm goes, you set the alarm to ten minutes past the old alarm setting (even if you get distracted in between), and as a result you end up checking on the kids exactly six times for every hour, since the alarm goes off in ten-minute intervals regardless of how long it takes you to get around to resetting the alarm (assuming you get to it within ten minutes, that is). We'll get back to the difference between these two ways of handling a timer, but we need to know a couple more things first.

Delays

Specific timing delays can be generated by looking at the current TCNT value and adding an offset that matches the desired delay time. If you write this value into the TC0 register, the event will occur at precisely the desired delay time. With a delay, the amount of time required to set up the delay and access it would be additional to the time spent waiting for the timer, but, unless the delay is extremely short, this is probably insignificant.

Intervals

First, we get the initial value in the counter to set a new target. Then, each time we detect an output compare, we add the value to the previous *target event* value instead of to the current *timer* value. Since we add the count interval to the previous event value and not to the current clock value, our interval will be accurate even if we don't service it immediately.

With the timer, we can perform other tasks of varying length while we wait for the output compare event to occur. This becomes especially useful when use Interrupt Service Routines. Since interrupts and interval timing work so well together, we'll leave a full discussion of interval timing until later in the course.

Delay Function for Misc_Lib

In your "Misc_Lib.h" header file, you will find the following prototype:

```
void Sleep_ms(unsigned int); //requires TimInit8us setup; blocking delay
```

This function is appropriately called "Sleep", because it will block and wait until the time interval has elapsed – nothing else, other than interrupt-driven behaviour, will happen once this function has been called, until the timer runs out. Unfortunately, the micro doesn't actually go into low power "sleep" mode, because it's madly checking the timer module!

Here's some information necessary to completing this function:

- 1 ms represents 125 8 μ s timer ticks.
- Since we don't know what value is present in the counter (TCNT) when we access this routine, we will need to read that and add one millisecond (125 counts) to it for the first interval.
- Once the first interval is set up, we can enter a *for* loop to handle the rest of the milliseconds, in which we will clear the flag, wait for it to trigger, and add 125 to the previous target present in the timer compare register (TC0).

Again, you can check your work against the following solution:

```
void Sleep_ms(unsigned int iTime) //requires TimInit8us()
{
    unsigned int iCount;
    TC0=TCNT+125; //first target -- 125 counts at 8 us = 1 ms
    for (iCount = 1;iCount<=iTime;iCount++)
    {
        TFLG1 |= 0b00000001; //clear flag
        while((TFLG1&0b00000001)==0); //BLOCKING wait for flag
        TC0+=125; //next target -- based on previous target for accuracy
    }
}
```

A good test of this routine would be to set up the program to toggle all three LEDs on the board after a delay. With an oscilloscope, you could probe the control line for one of the LEDs to see if the timing is what you expect it to be. Try values like 1 ms, 2 ms, 10 ms, 100 ms, 1000 ms, and 65000 ms (if you can wait that long!). Your results should be accurate to within the limitations of your oscilloscope.

Notice in the Sleep_ms() routine that the first value for TC0 comes from TCNT, because when you enter a Delay, you don't know what the starting time is. Consequently, there will be a small period of time lost between when you call this delay and when it actually starts, so repetitive calls of this routine will run a bit more slowly than a true interval timer.

However, notice that inside the loop, the subsequent values are based on the previous contents of TC0. That makes these true Intervals, because they are not affected by the processing time for managing the loop.

Interrupt-Driven Timer

Having an interval timer running on interrupts is a great idea. Your program can carry out any number of tasks, either dependent on or independent of the timer, without needing to poll the timer module to see if the interval is over.

The following example creates an interrupt-driven timer on Timer Channel 0. For an S12XCPU Assembly Language version of this code, look back to the code following the introduction to interrupts for the SCI receive operation.

Here are steps 1 to 5, written into the part of the skeleton file dedicated to ISRs:

```
/******
//          Interrupt Service Routines
/******

interrupt VectorNumber_Vtimch0 void TimerInterval(void)
{
    TC0 =(int)(iTimeVal+TC0); // next time
    TFLG1 |= 0b00000001; // acknowledge interrupt
}
```

Notice the declaration statement in the top line: "VectorNumber_Vtimch0" is interpreted by the compiler using the MC9S12SDP512.c file to point to the correct interrupt vector for Timer Channel 0. "TimerInterval" is the name of our ISR. Notice it is "void (void)", since we can't pass parameters to or from it.

We tell the compiler to cast the result of "iTimeVal+TC0" to *int*, because the compiler knows the result could be bigger than two bytes, and will give us a warning otherwise.

After clearing the flag, we simply end with a curly bracket, and the compiler knows to use RTI to end the routine. So, that's five out of the seven requirements.

Back in the main program, we need to enable the interrupts (steps six and seven), and we also need to make sure the `iTimeVal` variable is global.

```
TIE |= 0b00000001;    /*enable channel 0 interrupts*/
EnableInterrupts;
```

Now for the `iTimeVal` variable: As it sits, it is a global variable, so we should be able to use it without any changes. However, in case there's a chance it could be changed by the program during operation, it might be wise to put "volatile" in front of "int" to prevent it from being mangled by an interrupt occurring while it is being changed.

The following endless loop watches for the press of the MID switch, one check per timer cycle, as established in the ISR on the previous page:

```
    for (;;)
    {
//main program loop
        if(SwCk()==0b00000001)
        {
            PT1AD1&=0b00011111;
        }
        else
        {
            PT1AD1+=0b00100000;
        }
        asm WAI;
    }
```

The "WAI" command puts the micro to sleep, waiting for the next interrupt from the timer. Unlike an endless loop, the WAI command actually puts the microcontroller into a low-power state, so it conserves energy, which is particularly important for a battery-operated application.

The following page shows the code snippets discussed above all together as a single printout.

```

#include <hdef.h>          /* common defines and macros      */
#include "derivative.h"   /* derivative-specific definitions */
/*****
//      Library includes
*****/

#include "SW_LED_Lib.h"

/*****
//      Prototypes
*****/

void TimerSetup(int iTimeVal,byte bPrescaler);

/*****
//      Variables
*****/

volatile int    iTimeVal = 62500;
byte           bPrescaler = 0b00000101;

/*****
//      Lookups
*****/

void main(void)
{
// main entry point
_DISABLE_COP();

/*****
// initializations
*****/

    SW_LED_Init();

    TimerSetup(iTimeVal,bPrescaler);
    EnableInterrupts;

    for (;;)
    {
        //main program loop

        if(SwCk()==0b00000001) PT1AD1&=0b00011111; /*clear LEDs on MID press*/
        else PT1AD1 += 0b00100000; /*count up on the LEDs*/

        asm WAI;
    }
}

/*****
//      Functions
*****/

void TimerSetup(int iTimeVal,byte bPrescaler)
{
    TSCR1 = 0b10000000; /*turn on timer module*/
    TSCR2 &= 0b11111000; /*start by clearing the prescaler bits*/
    TSCR2 |= bPrescaler; /*now set the prescaler*/
    TIOS |= 0b00000001; /*IOS0 set to output compare*/
    TCTL2 &= 0b11111100; /*start by clearing bits for output to PT0*/
    TCTL2 |= 0b00000001; /*set low bit for TCO for toggle (01) on PT0*/
    TCO = TCNT + iTimeVal; /*first interval set up*/
    TIE |= 0b00000001; /*enable channel 0 interrupts*/
    TFLG1 |= 0b00000001; /*clear TCO flag*/
}

/*****
//      Interrupt Service Routines
*****/

interrupt VectorNumber_Vtimch0 void TimerInterval(void)
{
    TCO += iTimeVal; /* next time*/
    TFLG1 |= 0b00000001; /* acknowledge the interrupt*/
}

```

Real-Time Loop

(*Optional topic*) One formalized use of interrupts is Real-Time Loop Multiplexing. Unless your instructor has extra time available, you probably won't do an exercise on this yet – wait until the end of the semester, when you have more peripherals to work with. However, you should know how this is intended to work.

The idea is to have just one interrupt – a regular timer that establishes the loop interval. During each interval, a set number of tasks are handled in order, then the microcontroller is put into a power-down WAIT condition until the interval timer's interrupt occurs. Here's an example, shown first in S12XCPU Assembly Language, then in ANSI C.

```
RTL:
    JSR      SevSegTask
    JSR      SecTask
    JSR      ADCDACTask
    JSR      VoutTask

    WAI

    BRA     RTL

;*****
;*          Timer Interrupt Service Request          *
;*****
TIM_ISR:
    LDD     #1250                ;10 ms interval
    ADDD   TC0
    STD     TC0                ;new interval
    BSET   TFLG1,%00000001    ;clear interrupt
    RTI
```

The four tasks are in carefully-designed subroutines, and are accessed during each interval. The "WAI" command puts the microcontroller into a low-power sleep mode, but it still responds to the timer compare interrupt, which wakes it up and sends it to the beginning of the real-time loop.

```
for(;;)
{
    SevSegTask();
    SecTask();
    ADCDACTask();
    VoutTask();

    asm WAI;
}

//*****
//*          Timer Interrupt Service Request          *
//*****
interrupt VectorNumber_Vtimch0 void TimerInterval(void)
{
    TC0 += 1250;                // next time
    TFLG1 |= 0b00000001;      // acknowledge the interrupt
}
```

Of course, for this system to work, the total time taken by the task subroutines must be less than the time interval.

This can be handled two ways.

- One is to make the interval long enough to handle the maximum time required by all the tasks. Sometimes this is unrealistic, and results in jittery code management.
- The other is to find ways to break up tasks that occasionally have long bursts of activity into smaller pieces. For example, if one task occasionally sends a long string of text to a dumb terminal through the SCI port, consider sending the string one character at a time, or maybe no more than 10 characters at a time, until the entire string is sent. This would require putting the string into a buffer and keeping track of the current location in the buffer.

A well-planned real-time loop multiplexing system is the perfect application for a microcontroller acting as the brains for a repetitive system, such as the "computer controls" in an automotive fuel injection and ignition system.

Input Capture and Pulse Accumulation

(Optional topic) The timer pins can be used to provide timing information from outside events to the microcontroller. There are two ways we might want the microcontroller to respond to outside events: We might want to know how much time has elapsed since the last event (Input Capture) or we might want to know how many events have occurred over a set period of time (Pulse Accumulation).

Input Capture

Input Capture is a very simple procedure. Once a channel is configured as an Input Capture pin, each time an electrical event occurs on that pin, the current value of the internal clock is stored in the 16-bit Timer Compare (TCx) register associated with that pin. The usual initialization steps are required – setting up the clock speed and enabling the clock. In addition, we need to set up the channel we're using for input capture. This involves TIOS, which we previously used when we set up our timer channel for Output Compare.

TIOS	R								
	W	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0

This time, we want Channel 7 to be an "Input Compare", so it should be a 0.

Another parameter that should be controlled is the Input Capture Edge – sometimes you want to count when the signal goes from LOW to HIGH (rising edge) or when the signal goes from HIGH to LOW (falling edge). Sometimes, you might want to detect all changes, rising or falling (incidentally, this would double the frequency of a square wave). This involves Timer Control Registers 3 and 4 (TCTL3 and TCTL4).

TCTL3	R							
	W	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B
TCTL4	R							
	W	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B

There are two bits associated with each channel, since there are four possible options.

Table 7-12. Edge Detector Circuit Configuration

EDGxB	EDGxA	Configuration
0	0	Capture disabled
0	1	Capture on rising edges only
1	0	Capture on falling edges only
1	1	Capture on any edge (rising or falling)

The Input Capture channel indicates that an event has occurred by setting the corresponding bit in the Flag register, TFLG1. As usual, you will need to clear this flag before you can wait for it to appear again.

The following page shows a code snippet that displays the period, in microseconds, of a signal connected to PT7 (pin 18 of the microcontroller).

```
TimInitlus();
SevSeg_Init();

TIOS &=0b01111111; //put channel 7 into Input Compare mode
TCTL3&=0b01111111; //first part of making channel 7 rising edge
TCTL3|=0b01000000; //second part of making channel 7 rising edge

TFLG1|=0b10000000; //clear the flag
iStart=TCNT; //first timer reading

for (;;) //endless program loop
{
/*****
* Main Program Code
*****/

while((TFLG1&&0b10000000)==0); //wait for rising edge on channel 7
iDiff= TC7-iStart;
iStart=TC7;
SevSeg_Top4(HexToBCD(iDiff));
TFLG1|=0b10000000; //clear the flag
}
```

Pulse Accumulation

Pulse accumulation means to count the number of incoming events that occur over a time interval. As you may have deduced from the previous exercise, Input Capture provides information that directly relates to a signal's period. Pulse accumulation is the inverse: it tells us information that directly relates to a signal's frequency.

To do a pulse accumulation, you will need two timer channels: one to set up the time period over which you wish to count events, and another to count the events that occur in that time period.

We'll just use routines we developed previously to set up the required time period, so that means Timer Channel 0 will be used for that.

For the Pulse Accumulator, there are a number of options. The MC9S12XDP512 has four 8-bit Pulse Accumulators connected to Timer Channels 3 through 0, or, in a different mode, it has two 16-bit Pulse Accumulators connected to Channels 7 and 0. The easiest one of these to work with, and the one that doesn't interfere with our time period counter, is the 16-bit Pulse Accumulator A, connected to PT7. Here is its control register, PACTL:

PACTL	R	0	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI
	W								

Table 7-18. PACTL Field Descriptions

Field	Description
6 PAEN	<p>Pulse Accumulator A System Enable — PAEN is independent from TEN. With timer disabled, the pulse accumulator can still function unless pulse accumulator is disabled.</p> <p>0 16-Bit Pulse Accumulator A system disabled. 8-bit PAC3 and PAC2 can be enabled when their related enable bits in ICPAR are set. Pulse Accumulator Input Edge Flag (PAIF) function is disabled.</p> <p>1 16-Bit Pulse Accumulator A system enabled. The two 8-bit pulse accumulators PAC3 and PAC2 are cascaded to form the PACA 16-bit pulse accumulator. When PACA is enabled, the PACN3 and PACN2 registers contents are respectively the high and low byte of the PACA. PA3EN and PA2EN control bits in ICPAR have no effect. Pulse Accumulator Input Edge Flag (PAIF) function is enabled. The PACA shares the input pin with IC7.</p>
5 PAMOD	<p>Pulse Accumulator Mode — This bit is active only when the Pulse Accumulator A is enabled (PAEN = 1).</p> <p>0 Event counter mode</p> <p>1 Gated time accumulation mode</p>
4 PEDGE	<p>Pulse Accumulator Edge Control — This bit is active only when the Pulse Accumulator A is enabled (PAEN = 1). Refer to Table 7-19.</p> <p>For PAMOD bit = 0 (event counter mode).</p> <p>0 Falling edges on PT7 pin cause the count to be incremented</p> <p>1 Rising edges on PT7 pin cause the count to be incremented</p> <p>For PAMOD bit = 1 (gated time accumulation mode).</p> <p>0 PT7 input pin high enables bus clock divided by 64 to Pulse Accumulator and the trailing falling edge on PT7 sets the PAIF flag.</p> <p>1 PT7 input pin low enables bus clock divided by 64 to Pulse Accumulator and the trailing rising edge on PT7 sets the PAIF flag.</p> <p>If the timer is not active (TEN = 0 in TSCR1), there is no divide-by-64 since the +64 clock is generated by the timer prescaler.</p>
3:2 CLK[1:0]	<p>Clock Select Bits — For the description of PACLK please refer to Figure 7-70.</p> <p>If the pulse accumulator is disabled (PAEN = 0), the prescaler clock from the timer is always used as an input clock to the timer counter. The change from one selected clock to the other happens immediately after these bits are written. Refer to Table 7-20.</p>
2 PAOVI	<p>Pulse Accumulator A Overflow Interrupt Enable</p> <p>0 Interrupt inhibited</p> <p>1 Interrupt requested if PAOVF is set</p>
0 PAI	<p>Pulse Accumulator Input Interrupt Enable</p> <p>0 Interrupt inhibited</p> <p>1 Interrupt requested if PAIF is set</p>

We will need to enable Pulse Accumulator A. We also typically need to tell it that we're going to use it as an Event Counter. We also need to indicate whether it will respond to rising edges or falling edges, and how we will use the clock source.

Table 7-19. Pin Action

PAMOD	PEDGE	Pin Action
0	0	Falling edge
0	1	Rising edge
1	0	Divide by 64 clock enabled with pin high level
1	1	Divide by 64 clock enabled with pin low level

Table 7-20. Clock Selection

CLK1	CLK0	Clock Source
0	0	Use timer prescaler clock as timer counter clock
0	1	Use PCLK as input to timer counter clock
1	0	Use PCLK/256 as timer counter clock frequency
1	1	Use PCLK/65536 as timer counter clock frequency

It's fairly obvious that this pulse accumulator can be used in a lot of different ways. We will just use it in its simplest mode: responding to a rising edge, using the timer prescaler as the counter clock. At this point, we aren't using any interrupts, so we'll inhibit the two interrupts. Hopefully, you've determined that we want to put the value 0b01010000 into PACTL (the first bit isn't used, and is always 0).

Once the Pulse Accumulator is set up, we need to set up our regular clock, clear the contents of the Pulse Accumulator register, and whenever the clock indicates that the time period is up, we read the Pulse Accumulator Count Register (16-bit) and reset it to zero for the next count. The Pulse Accumulator Count register is the 16-bit combination of PACN3 and PACN2. In the mc9s12xdp512.inc file, they provide us with the option of doing a 16-bit read of PACN3 or, with the same functionality, PACN32, an alternate name that probably helps you remember it's a 16-bit register. Here's a bit of code that counts events on PT7 (pin 18) for a full second, then displays the frequency, in Hz, on the seven segment display.

```

/*****
*      Initializations
*****/
SevSeg_Init();
TimInit8us();

PACTL=0b01010000; //rising edge event counter using timer prescaler

for (;;) //endless program loop
{
/*****
*      Main Program Code
*****/

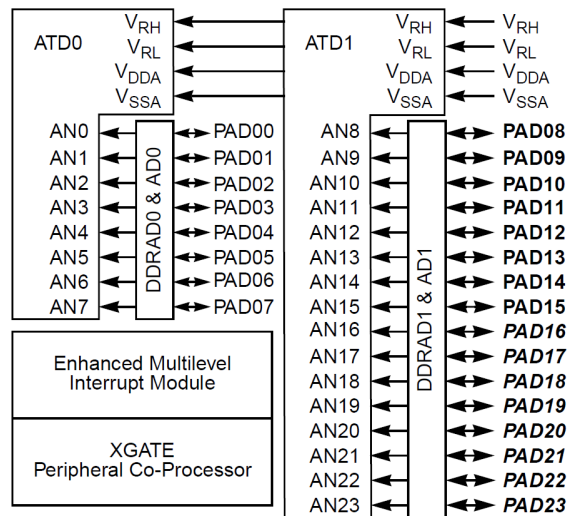
PACN32=0; //clear counter
Sleep_ms(1000);
SevSeg_Top4(HexToBCD(PACN32));

}

```

A To D Conversion

The 9S12XDP512 has two fairly complex Analog to Digital Conversion blocks. Here's the top left corner of the block diagram, which we've looked at previously.



Notice that PAD00 to PAD07 are associated with ATD0, and PAD08 to PAD15, the pins that are connected on our board to the LEDs and Switches, could also be attached to ATD1. So, of the two available converters, we're going to be using ATD0.

This peripheral is highly configurable, and can do a lot of things. Again, we'll just scratch the surface of its capabilities.

Here are some of its features:

- It can run in single input or multiplexed input mode. In other words, it could be measuring up to 8 external voltages simultaneously.
- It can sample on command or it can operate in continuous scan mode. This means you can either ask for a sample and wait for it, or you can have samples available all the time for faster reading.
- You can ask for multiple samples from one channel, one sample from each of the channels, or a number of samples from a number of channels (within limits!).
- You can choose where the results end up, since the result registers aren't directly tied to the input channels. In one mode (FIFO) it continuously wraps through the channels placing the next available value in the next output register; in another mode, it always starts at result channel 0 and runs until it gets to the last result you've asked it for; it can also start filling at a result register of your choice, wrapping around until it gets to the last result you've asked for.
- Sampling can either be clock driven or initiated by external trigger events.
- The results can be either 10-bit or 8-bit. 10-bit is better: just remember to read a two-byte word to get the result!
- The reference voltages, both top and bottom, can be set using external circuitry. In our case, V_{RL} is grounded, and V_{RH} is connected to the output of a REF02 that has a trimmer potentiometer connected to it. We'll set this to 5.120 V_{DC} to provide a nice step size.
- The sample rate is selectable. Fast sample rates allow for high-speed signals, but slow sample rates are more accurate.
- The input buffering of the signal is configurable.
- The data format is configurable. You can select unsigned or signed values (Single quadrant or 4-quadrant), and left or right justified values.

The block diagram for ATD0 shows some of the capabilities of this converter, along with the internal devices that make possible these capabilities.

Chapter 5 Analog-to-Digital Converter (S12ATD10B8CV2)

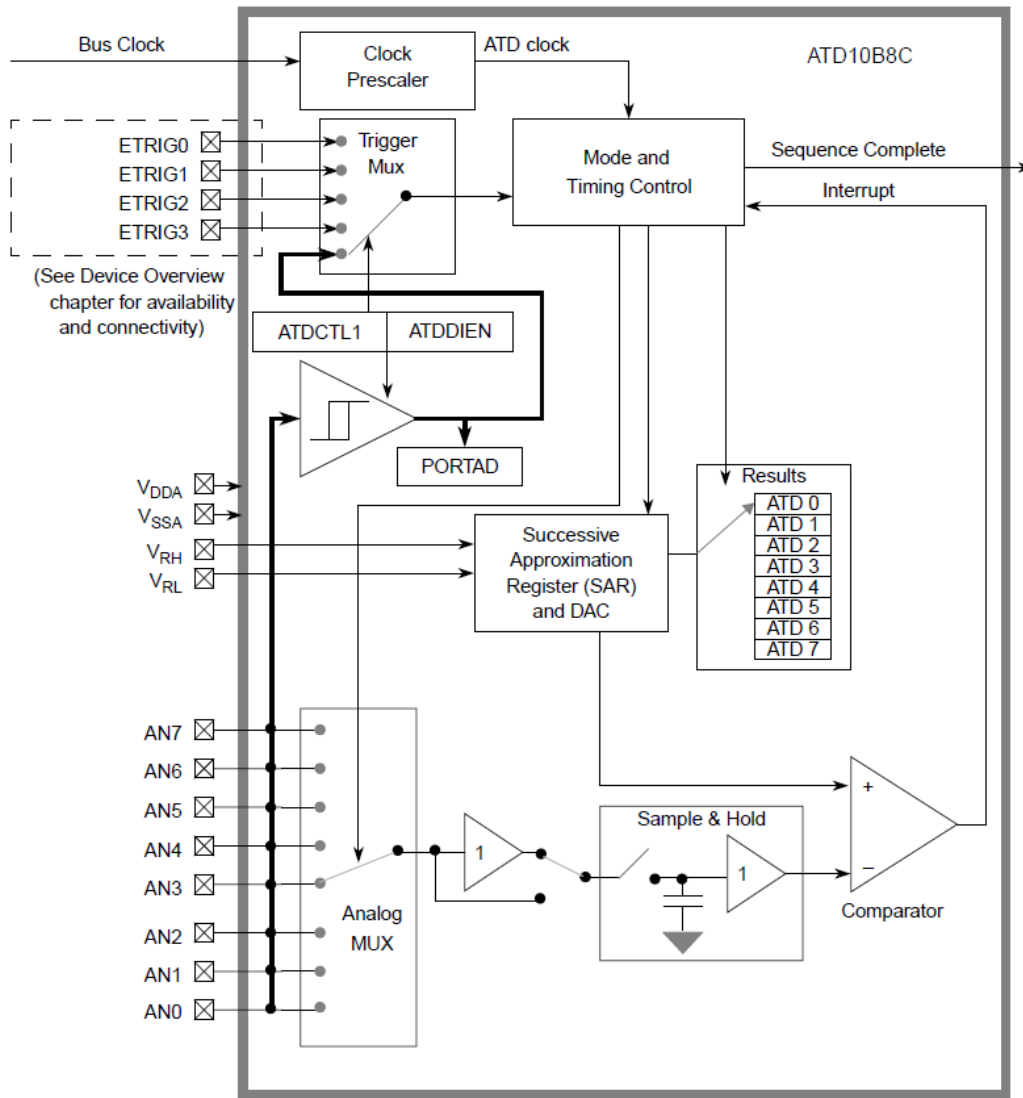


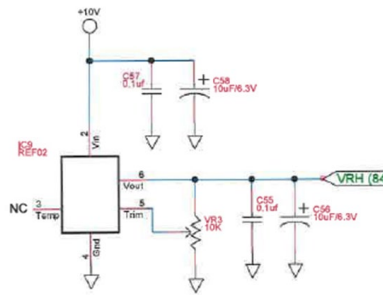
Figure 5-1. ATD Block Diagram

Setting up V_{RH}

In other courses, you've come to think of the REF02 integrated circuit as a fixed 5.0 V reference. The designers of the IC knew that it would be very difficult to make all of their products produce exactly 5.0 V, so they included a *TRIM* pin that can be used to adjust the output slightly. We have taken advantage of this feature, and have incorporated an output trimming circuit that can be used to set the REF02 output to 5.120 V_{DC}, which gives us a useful step size. With a 10-bit A to D converter, the step size is

$$\text{StepSize} = \frac{V_{ref}}{2^n} = \frac{5.120}{2^{10}} = 5.000\text{mV} / \text{step}$$

Here’s the circuitry involved. The capacitors are there to filter out noise and variations in the power supplies. What’s important to our discussion is *VR3*, the trim potentiometer.



On your board, you will need to connect a digital voltmeter between ground and *VREF*, and adjust *VR3* to set *VREF* to 5.120 *V_{DC}*.

Configuring ATD0

The full details for the registers we’re using are found in Chapter 5 of the “Data Sheet”. Here’s a summary that includes the ones we’ll be using.

Register Name	Bit 7	6	5	4	3	2	1	Bit 0
ATDCTL0	R 0	0	0	0	0	WRAP2	WRAP1	WRAP0
ATDCTL1	R ETRIGSEL	0	0	0	0	ETRIGCH2	ETRIGCH1	ETRIGCH0
ATDCTL2	R ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
ATDCTL3	R 0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
ATDCTL4	R SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
ATDCTL5	R DJM	DSGN	SCAN	MULT	0	CC	CB	CA
ATDSTAT0	R SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0
ATDDR0H	10-BIT 0	0	0	0	0	0	BIT 9 MSB	BIT 8
	8-BIT 0	0	0	0	0	0	0	0
ATDDR0L	10-BIT BIT 7 MSB	BIT 6 BIT 6	BIT 5 BIT 5	BIT 4 BIT 4	BIT 3 BIT 3	BIT 2 BIT 2	BIT 1 BIT 1	BIT 0 BIT 0
ATDDIEN	R IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0

In the datasheet, the registers are named “ATDCTLx”, etc. But since there are two A to D converters in this controller, we need to insert a “0” after “ATD” in each case to specify that we’re using ATD0 (not ATD1, which is connected to the switches and LEDs).

Remember DDR1AD1 and ATD1DIEN1? We needed to digitally-enable the inputs in order to get a digital signal into them. However, to receive analog signals, we need to have DDRAD and ATDDIEN or, in our case, DDR1AD0 and ATD0DIEN, cleared to LOW for the pins to be enabled as analog inputs instead. Since the unit defaults to 0, we shouldn’t have to worry about DDR1AD0 and ATD0DIEN, unless somewhere else in software we’ve set these bits to “1”. In any case, it’s good practice to make sure the pins are configured correctly.

Your instructor will probably want you to create a library for A to D conversion, called “ATD0_Lib.c”, with its prototype header file “ATD0_Lib.h”. In this library, the first entry should be an initialization routine, ATD0_Init().

Notice in the list of registers that there are six control registers: ATDCTL0 through ATDCTL5. The first two are for modes we're not going to use. So, let's start with ATD0CTL2 (notice the "0" inserted in the register name). You should probably refer to the register descriptions in Chapter 5 of the "Data Sheet" to determine what each bit should be in the following registers, if they aren't obvious.

- For ATD0CTL2, we want to power up the A to D converter, run in fast flag mode (no need to write to the flag to clear it), run in wait mode, operate without external triggering, and turn off interrupts.
- In the "Data Sheet", it says that, once ATD0CTL2 has been set up and the A to D Converter has been powered up, we need to wait at least 50 μ s before anything else can be configured. You probably don't want to be forced to include your Misc_Lib every time you run the A to D converter, so it makes sense to create a simple delay using a clock-cycle-based loop. We've done this before using "asm" commands to: 1) load an accumulator with a desired number of loops, then 2) execute the "DBNE" Assembly Language command (three clock cycles) until the counter runs out.
- For ATD0CTL3, we want 8 conversions per sequence, we want the converter to start at our selected register (which we will soon set to 0) rather than being "first-in first-out", and we want the A to D converter to finish conversions before freezing on a break. This one probably requires a look at the description in the "Data Sheet":

Table 5-8. Conversion Sequence Length Coding

S8C	S4C	S2C	S1C	Number of Conversions per Sequence
0	0	0	0	8
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	X	X	X	8

Table 5-9. ATD Behavior in Freeze Mode (Breakpoint)

FRZ1	FRZ0	Behavior in Freeze Mode
0	0	Continue conversion
0	1	Reserved
1	0	Finish current conversion, then freeze
1	1	Freeze Immediately

(continued)

- For ATD0CTL4, we want to run this as a 10-bit converter, we'll use four clocks per sample, and we want each sample to be at least 7 μs. This one definitely needs some information from the "Data Sheet":

Table 5-10. ATDCTL4 Field Descriptions

Field	Description
7 SRES8	A/D Resolution Select — This bit selects the resolution of A/D conversion results as either 8 or 10 bits. The A/D converter has an accuracy of 10 bits; however, if low resolution is required, the conversion can be speeded up by selecting 8-bit resolution. 0 10-bit resolution 1 8-bit resolution
6–5 SMP[1:0]	Sample Time Select — These two bits select the length of the second phase of the sample time in units of ATD conversion clock cycles. Note that the ATD conversion clock period is itself a function of the prescaler value (bits PRS4–0). The sample time consists of two phases. The first phase is two ATD conversion clock cycles long and transfers the sample quickly (via the buffer amplifier) onto the A/D machine's storage node. The second phase attaches the external analog signal directly to the storage node for final charging and high accuracy. Table 5-11 lists the lengths available for the second sample phase.
4–0 PRS[4:0]	ATD Clock Prescaler — These 5 bits are the binary value prescaler value PRS. The ATD conversion clock frequency is calculated as follows: $ATDclock = \frac{[BusClock]}{[PRS + 1]} \times 0.5$ Note: The maximum ATD conversion clock frequency is half the bus clock. The default (after reset) prescaler value is 5 which results in a default ATD conversion clock frequency that is bus clock divided by 12. Table 5-12 illustrates the divide-by operation and the appropriate range of the bus clock.

Table 5-11. Sample Time Select

SMP1	SMP0	Length of 2nd Phase of Sample Time
0	0	2 A/D conversion clock periods
0	1	4 A/D conversion clock periods
1	0	8 A/D conversion clock periods
1	1	16 A/D conversion clock periods

Table 5-12. Clock Prescaler Values

Prescale Value	Total Divisor Value	Max. Bus Clock ¹	Min. Bus Clock ²
00000	Divide by 2	4 MHz	1 MHz
00001	Divide by 4	8 MHz	2 MHz
00010	Divide by 6	12 MHz	3 MHz
00011	Divide by 8	16 MHz	4 MHz
00100	Divide by 10	20 MHz	5 MHz
00101	Divide by 12	24 MHz	6 MHz
00110	Divide by 14	28 MHz	7 MHz
00111	Divide by 16	32 MHz	8 MHz
01000	Divide by 18	36 MHz	9 MHz
01001	Divide by 20	40 MHz	10 MHz
01010	Divide by 22	44 MHz	11 MHz
01011	Divide by 24	48 MHz	12 MHz
01100	Divide by 26	52 MHz	13 MHz
01101	Divide by 28	56 MHz	14 MHz
01110	Divide by 30	60 MHz	15 MHz
01111	Divide by 32	64 MHz	16 MHz
10000	Divide by 34	68 MHz	17 MHz
10001	Divide by 36	72 MHz	18 MHz
10010	Divide by 38	76 MHz	19 MHz
10011	Divide by 40	80 MHz	20 MHz
10100	Divide by 42	84 MHz	21 MHz
10101	Divide by 44	88 MHz	22 MHz
10110	Divide by 46	92 MHz	23 MHz
10111	Divide by 48	96 MHz	24 MHz
11000	Divide by 50	100 MHz	25 MHz
11001	Divide by 52	104 MHz	26 MHz
11010	Divide by 54	108 MHz	27 MHz
11011	Divide by 56	112 MHz	28 MHz
11100	Divide by 58	116 MHz	29 MHz
11101	Divide by 60	120 MHz	30 MHz
11110	Divide by 62	124 MHz	31 MHz
11111	Divide by 64	128 MHz	32 MHz

¹ Maximum ATD conversion clock frequency is 2 MHz. The maximum allowed bus clock frequency is shown in this column.

² Minimum ATD conversion clock frequency is 500 kHz. The minimum allowed bus clock frequency is shown in this column.

- For ATD0CTL5, we want our results to be right-justified unsigned values, we'll run in continuous scan mode sampling all eight channels, and we want to have the results of each conversion sequence start at result register 0 so that the result register will match the input channel.

Pulse-Width Modulation

When the first personal computers hit the market, the best audio they could manage was a collection of annoying squeaks and beeps. Creative individuals figured out how to make these squeaks and beeps sound like badly recorded voices and music by varying the frequencies and duty cycles of the waveforms in a technique called pulse-width modulation. With the advent of sound cards, those days are now in the past. However, it might surprise you to know that many of our low-power audio devices (and expensive high-powered audio amps, as well) use pulse-width modulation with slightly more sophisticated integration and filtering circuitry to produce high-fidelity sound. This is called Class-D power amplification.

Also, in the early days of remote-controlled toys, motors would either be turned fully on or off, resulting in jack-rabbit starts and stops, and crazy all-or-nothing turns. Pulse-width modulation now allows for much smoother motor control, not only for remote-controlled toys but for industrial processes, automotive devices, etc. Many microcontrollers have sophisticated pulse-width modulators to allow for programmable control of such devices. The 9S12X has a highly-configurable eight-channel PWM module. We will only begin to scratch the surface of the capabilities of this module.

The PWM module is used to create waveforms with programmable period and duty. There are a number of uses for programmable waveforms, most residing well outside the scope of this course.

For fun, we've connected a speaker to one of the PWM channels of your board, channel 6, with a jumper to enable you to disable the speaker when you see an angry hulk approaching. We also have three channels of the PWM (channels 0, 1, and 4) wired to an RGB LED to allow you to control the resulting colour and brightness of this LED, and we have channel 3 wired to the backlight of the LCD display to allow you to control that, as well. The other three channels are available at the general breakout headers on the board.

PWM Channel	Function
0	RGB Blue
1	RGB Green
4	RGB Red
3	Backlight
6	Speaker

Using the PWM channel connected to the speaker, you can create waveforms of the correct period and duty to generate amusing sounds on your speaker. You can use these sounds to add useful enhancements to your code (key clicks, alarms, start-up sounds, etc.), create cheap '80s style music, or generally drive your lab mates crazy.

The PWM subsystem is fairly easy to get along with, and relies heavily on a series of clocks. As with most modules on the 9S12X, the PWM subsystem is configured through a series of registers, shown in a screen capture on the next page to give you a sense of the complexity of this module. We'll learn about the registers of interest to you as you need them.

The next characteristics of the waveform all relate to timing. You’ve learned in other courses that the most basic aspects of waveform timing are frequency and its inverse, the period. The timing characteristics are based on clocks internal to the PWM module.

There are four clocks that are available to drive the PWM rates, and all are based on the bus clock. The clocks are called A, SA (scaled A), B, and SB (scaled B). Either the A/SA or B/SB clock pairs are available for the PWM you’re working with, defined in the hardware of the device. For each PWM channel, you must decide whether you want to use the basic clock that’s available (A or B) or whether you want to use the scaled clock (SA or SB). This is done through the **PWMCLK** register.

PWMCLK	R								
	W	PCLK7	PCLK6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0

Table 8-3. PWMCLK Field Descriptions

Field	Description	Field	Description
7 PCLK7	Pulse Width Channel 7 Clock Select 0 Clock B is the clock source for PWM channel 7. 1 Clock SB is the clock source for PWM channel 7.	3 PCLK3	Pulse Width Channel 3 Clock Select 0 Clock B is the clock source for PWM channel 3. 1 Clock SB is the clock source for PWM channel 3.
6 PCLK6	Pulse Width Channel 6 Clock Select 0 Clock B is the clock source for PWM channel 6. 1 Clock SB is the clock source for PWM channel 6.	2 PCLK2	Pulse Width Channel 2 Clock Select 0 Clock B is the clock source for PWM channel 2. 1 Clock SB is the clock source for PWM channel 2.
5 PCLK5	Pulse Width Channel 5 Clock Select 0 Clock A is the clock source for PWM channel 5. 1 Clock SA is the clock source for PWM channel 5.	1 PCLK1	Pulse Width Channel 1 Clock Select 0 Clock A is the clock source for PWM channel 1. 1 Clock SA is the clock source for PWM channel 1.
4 PCLK4	Pulse Width Channel 4 Clock Select 0 Clock A is the clock source for PWM channel 4. 1 Clock SA is the clock source for PWM channel 4.	0 PCLK0	Pulse Width Channel 0 Clock Select 0 Clock A is the clock source for PWM channel 0. 1 Clock SA is the clock source for PWM channel 0.

It’s important to note which basic clock is being used, A or B, and that ‘0’ selects the basic clock whereas ‘1’ selects the prescaled version of that basic clock.

The next register to configure is **PWMPRCLK**. This register determines how clock A and/or clock B is divided down from the bus clock. PWMPRCLK affects the basic clock speeds, and so affects the frequency of all waveforms produced by the PWM module. Notice that both prescalers are contained in the same register, but at different bit locations. Be careful with this! Also, notice that there are only three bits associated with each prescaler. That’s because these bits represent the power of 2 for the prescaler. We’ll refer to the prescaler as 2^{PREx} , where ‘x’ is either A or B.

PWMPRCLK	R	0				0		
	W		PCKB2	PCKB1	PCKB0		PCKA2	PCKA1

If you have chosen to use the scaled version of the clock, SA or SB, it is necessary to use the **PWMSCLA** or **PWMSCLB** register, as well. We'll refer to these as PWMSCLx, where 'x' is either A or B. This provides the scaling factor, allowing for fine control over the settings chosen for the A or B clock previously. The information for PWMSCLA has been provided; PWMSCLB behaves identically, but for clock SB.

PWMSCLA divides down Clock A to generate Clock SA, and PWMSCLB divides down Clock B to generate Clock SB.

8.3.2.9 PWM Scale A Register (PWMSCLA)

PWMSCLA is the programmable scale value used in scaling clock A to generate clock SA. Clock SA is generated by taking clock A, dividing it by the value in the PWMSCLA register and dividing that by two.

$$\text{Clock SA} = \text{Clock A} / (2 * \text{PWMSCLA})$$

NOTE

When PWMSCLA = \$00, PWMSCLA value is considered a full scale value of 256. Clock A is thus divided by 512.

Any value written to this register will cause the scale counter to load the new scale value (PWMSCLA).

	7	6	5	4	3	2	1	0
R	Bit 7	6	5	4	3	2	1	Bit 0
W								
Reset	0	0	0	0	0	0	0	0

Figure 8-11. PWM Scale A Register (PWMSCLA)

Read: Anytime

Write: Anytime (causes the scale counter to load the PWMSCLA value)

Be aware that the clock is divided by TWICE the value you choose for the scaling value, not the scaling value itself.

There's a "NOTE" on the data sheet indicating that 0x00 means 256. This does not seem to be true. The biggest divisor available seems to be $2 \times 255 = 510$.

In addition to the clock rates chosen, the waveforms you will be generating are managed by byte-sized period and duty values, controlled using two more registers: **PWMPERn** and **PWMDTYn**, where "n" is the number of our selected channel. Channel 0 is shown below:

PWMPER0	R	Bit 7	6	5	4	3	2	1	Bit 0
	W								
PWMDTY0	R	Bit 7	6	5	4	3	2	1	Bit 0
	W								

The period and duty values are expressed as numbers of clock cycles. This means that the shortest period would be 2 cycles of the clock, (up for one cycle, down for one) and the longest would be 255 cycles of the clock. For this shortest period, the duty could only be 1, since either 0 or 2 would produce a DC signal.

Typically, the duty cycle of a signal is defined as a ratio or percentage, as shown below:

$$d = \frac{t_p}{T} \quad \text{or} \quad d = \frac{t_p}{T} \times 100\%$$

So, the duty cycle for the PWM module, as a ratio, would be

$$d = \frac{PWMDTYn}{PWMPERn}$$

A very common duty cycle is 50%, which represents a true square wave. For this, PWMDTYn would be half of PWMPERn.

There are several strategies you may use to determine clock scaling values, the simplest being to pick a frequency and a fixed duty cycle, then work backwards to solve for the required clock pre-scalers, period value, and duty value. Enter the number of clock cycles for the period into PWMPERn, then enter the number of clock cycles (less than PWMPERn for obvious reasons) into PWMDTYn.

The frequency, and consequently the period, of the output signal are, therefore, controlled by two variables if the prescaled clocks are not used: The A or B clock pre-scaler from PWMPRCLK and the number of PWM clock cycles per period in the PWMPERn register.

$$f = \frac{8MHz}{2^{PREx} \times PWMPERn} \quad \text{and} \quad T = 125ns \times 2^{PREx} \times PWMPERn$$

If the prescaled clocks are used, three variables control the resulting frequency: PWMPRCLK, the SA or SB clock scale register (PWMSCLx), and the number of PWM clock cycles per period in the PWMPERn register.

$$f = \frac{8MHz}{2^{PREx} \times 2 \times PWMSCLx \times PWMPERn} \quad \text{and} \quad T = 125ns \times 2^{PREx} \times 2 \times PWMSCLx \times PWMPERn$$

For reasonably accurate frequencies, you should try to keep the value of PWMPERn large – close to 255. For example, if you are off by a cycle, one cycle out of 255 is much less significant than one out of three!

The last thing to do is actually turn on the channel, which is done by setting the corresponding bit in the **PWME** register:

PWME	R	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0
	W								

We've left discussion of this register to the end because, unlike many of the modules we've worked with to this point, we sometimes don't want to have the PWM channel turned on all the time. Manipulating this register allows you to turn your signals on and off under software control.

The code snippet on the following page turns a 1 kHz square wave, sent to the speaker, on and off once per second. If you build this code, you can monitor the signal by probing the left side of JP1 on the board with an oscilloscope. Also, with JP1 installed and VR2 turned up, you should be able to hear the tone.

```

/*****
*       Initializations
*****/

TimInit8us();

PWMPOL = 0b11111111; //positive polarity, all channels
PWMCLK|=0b01000000; //Use SB as the clock for channel 6 (speaker)
/* 1 kHz = 8000000/(2^2 x 2 x 4 x 250) */
PWMPRCLK &=0b00001111; //clear PRE-B before setting
PWMPRCLK |=0b00100000; //PRE-B = 2^2
PWMSCLB = 4; //Scale = 2 x 4
PWMPER6 = 250; //keep the period number large for accuracy
PWMDTY6 = 125; //50% duty cycle

for (;;) //endless program loop
{
/*****
*       Main Program Code
*****/

Sleep_ms(500); //half second on, half second off
PWME^=0b01000000; //...by toggling PWME6

}

```

True Pulse-Width Modulation

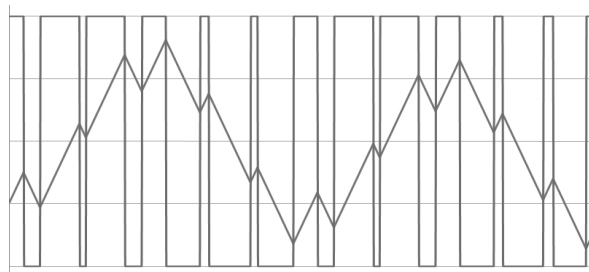
For many applications, including motor control, a pulse-width modulator is set to operate at a fairly high but constant frequency (called the “chop frequency”), and the pulse width, or duty cycle, is modified (modulated) to produce a varying average voltage. On our microcontroller development kits, the most obvious application of this is in controlling the brightness of the LED back light and/or the brightness of the three colour elements of the Red/Green/Blue (RGB) LED. Since our eyes are typically not sensitive to changes faster than about 100 per second, we can set the frequency to something fairly slow, as far as the PWM module is concerned – on the order of 100 to 200 Hz. Once the frequency is established and the desired channels are turned on, we simply vary the duty cycles to vary the brightness.

The LED backlight (channel 3) operates over a reasonable range of brightness, so we can comfortably vary the duty cycle from 0% (full OFF) to 100% (full ON, or DC). However, the RGB LED is intensely bright, so we usually limit the duty cycles of its elements (channels 0, 1, and 4) to less than 25%.

In both cases, since the duty cycle is of primary importance to us, we will set its range to something easily-manipulated by using a PERIOD value like 100 or 200. With a range like that, 100% is either a duty of 100 or 200, making the steps easy to work with – either 1% per step or 0.5% per step.

(Optional topic) True Pulse Width Modulation is also used in Class-D audio amplifiers, which are used in most low-power audio devices (cell phones, mp3 players, tablets, laptops, etc.); and it is also used in FET power amplifiers used for public address systems and many home audio systems and high power sound systems like those used in live music venues.

In this case, the chop frequency is set to double the desired audio range or often much higher (double meets the requirements of Nyquist’s sampling rate). The duty cycle is modulated to follow the constantly-varying amplitude of the desired signal. The resulting series of pulses is fed into an integrator, which produces a signal averaged, or smoothed, over time. The figure below shows the modulated pulses (full scale) and the output of the integrator, which is roughly sinusoidal in this instance.



The “jagged sinusoid” shown above would be the result of simple integration of the PWM pulses. Note that, when the duty cycle is greater than 50%, the integrated signal rises; when the duty cycle is less than 50%, the integrated signal falls.

In reality, the difference in frequency between the PWM signal (chop frequency) and the output of the integrator (the demodulated signal) would be so great that the “jaggedness” would be greatly reduced. Additional filtering, following the integrator, would be used to further remove unwanted high frequency components.

The following page shows a sample section of code, with the resulting waveforms seen at the output of the PWM (channel 1) and after the integrator (channel 2). Note the following:

1. The values for the duty cycles in the lookup table "cWave[180]" were generated using the formula shown in the "fx" line of the Excel screen shot below, applied to angles, in steps of 2 degrees, from 0 to 358:

B2		fx =INT(63*SIN(RADIANS(A2))+64)					
	A	B	C	D	E	F	G
1	Angle	Value					
2	0	64					
3	2	66					
4	4	68					
5	6	70					

Excel works in radians only, hence the conversion of angles to radians; the resulting values will range from $63+64 = 127$ to $-63+64 = 1$, giving us all positive values in a sine wave with an offset of 64; we want the nearest integer, since the 9S12X microcontroller isn't set up at this time to use floating-point decimals; we don't include 360° , as that's the same as 0° in our repeating sine wave.

2. Both the PWM module and the Timer are running as fast as possible – the Bus clock.
3. The PWM module is set to run at 62.5 kHz, well above the audible range.
4. Using a timer interval of 44 counts, each 125 ns long, it takes 990 μ s to run through all 180 values for the duty cycle in the table, for a frequency of 1.01 kHz on the resulting sine wave. The endless "for" loop starts at the top of the table again.
5. The PWM frequency is constant – only the duty cycle changes sinusoidally.
6. The speaker is driven by a transistor switch, which is an inverter, so the sine wave rises when the duty cycle is close to $1/128$ and falls when it is close to $127/128$.
7. The jagged edge of the integrator can be seen on the sine wave.
8. Channel 2 is AC coupled to block the DC offset generated by the transistor switch.
9. The speaker volume control is set to maximum to drive the integrator appropriately.

```

/*****
*   Variables
*****/
unsigned char cCounter;

/*****
*   Lookups
*****/
unsigned char cWave[180]=
{
    64,66,68,70,72,74,77,79,81,83,85,87,89,91,93,95,
    97,99,101,102,104,106,107,109,110,112,113,114,116,117,118,119,
    120,121,122,123,123,124,125,125,126,126,126,126,126,127,126,126,
    126,126,126,125,125,124,123,123,122,121,120,119,118,117,116,114,
    113,112,110,109,107,106,104,102,101,99,97,95,93,91,89,87,
    85,83,81,79,77,74,72,70,68,66,64,61,59,57,55,53,
    50,48,46,44,42,40,38,36,34,32,30,28,26,25,23,21,
    20,18,17,15,14,13,11,10,9,8,7,6,5,4,4,3,
    2,2,1,1,1,1,1,1,1,1,1,1,1,2,2,3,
    4,4,5,6,7,8,9,10,11,13,14,15,17,18,20,21,
    23,25,26,28,30,32,34,36,38,40,42,44,46,48,50,53,
    55,57,59,61
};

void main(void)    // main entry point
{
    _DISABLE_COP();

/*****
*   Initializations
*****/

    TimInit125ns();    //set up Timer Channel 0 for bus clock speed

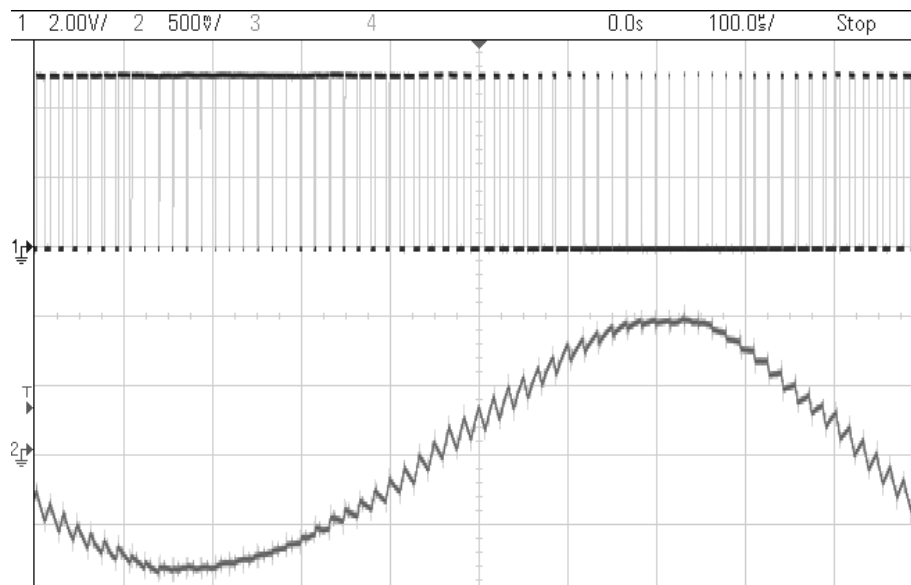
    PWMCLK   &=0b10111111;    //Use B Clock directly
    PWMPRCLK &=0b10001111;    //Set B clock to Bus Clock (8 MHz)
    PWMPER6  = 128;           //128 clock cycles = 62.5 kHz
    PWMDTY6  = 64;            //initially 50% duty cycle
    PWME     |=0b01000000;    //turn on channel 6 (speaker)

    TC0=TCNT+44;            //first timer interval: 44 x 180 x 125 ns = 990 us
                            //for a 1.01 kHz sine wave

    for (;;)    //endless program loop
    {
/*****
*   Main Program Code
*****/

        for(cCounter = 0;cCounter<180;cCounter++)    //endlessly loops through
        {
            PWMDTY6= cWave[cCounter];    //the entire set of 180 values
            while((TFLG1&0b00000001)==0);    //get the current duty value
            TC0+=44;    //wait for the timer flag
            TFLG1|=0b00000001;    //set up the next interval
        }    //clear the flag
    }
}

```



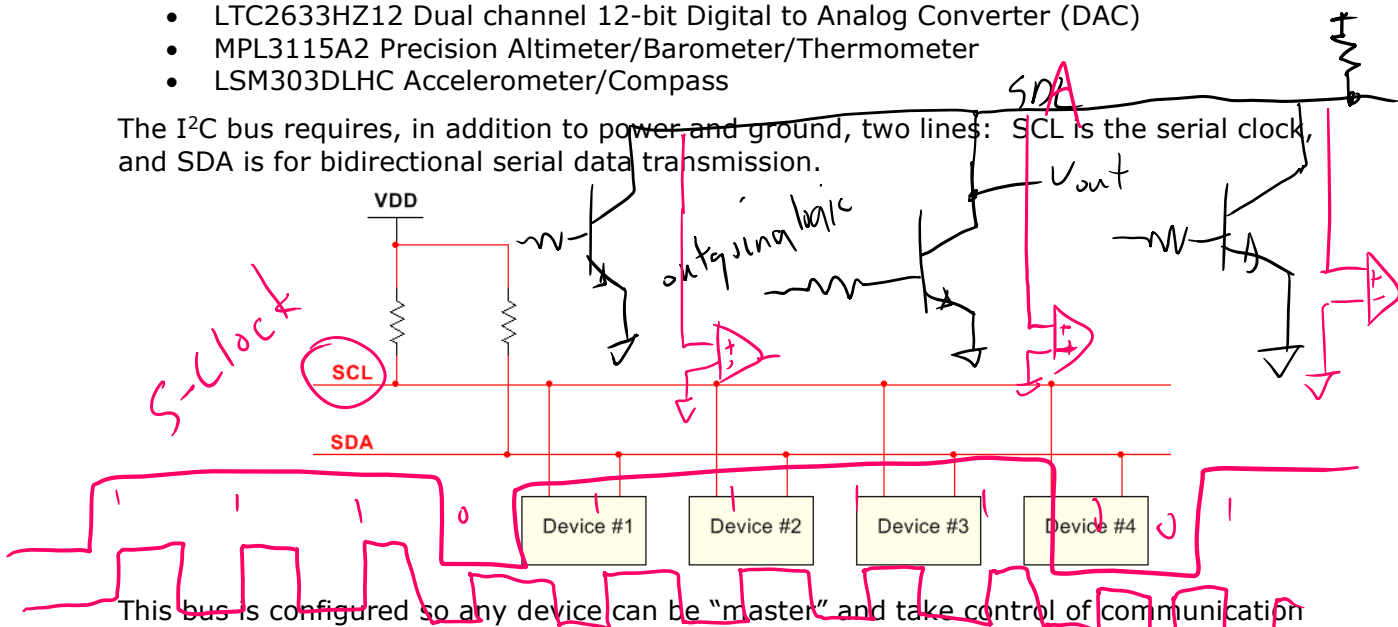
I²C Bus

There are a number of serial communication systems that can be used to communicate between devices on a printed circuit board or over short distances. One reason for doing this is to reduce the number of interconnections required between devices. We discussed this earlier, when we compared using more than 64 parallel wires (and, incidentally, more than 64 pins per device) with using RS-232 as a serial communication system that needed, as a minimum, three wires: transmit, receive, and ground. The 9S12XDP512 also provides other serial communication options: Serial Peripheral Interface, or SPI, Controller Area Network, or CAN Bus (typically used in automotive applications), and Inter-Integrated Circuit, or I²C Bus. Different peripherals are available for these different busses, depending on the desired application. What they share in common is the ability to put multiple devices on a single bus, which dramatically simplifies the hardware component, but complicates the software component somewhat.

In previous versions of this course, we chose to work with the SPI bus, as it is an old workhorse that's not likely to go away anytime soon. However, when the current version of the microcontroller board was designed, we focused on the I²C bus. On your board, you will find the following I²C devices (they're tiny, so you might have to search for them):

- M41T81 Real-time Clock (with battery backup – the battery is under the LCD display)
- 24AA512 Memory – 512kB of EEPROM
- LTC2633HZ12 Dual channel 12-bit Digital to Analog Converter (DAC)
- MPL3115A2 Precision Altimeter/Barometer/Thermometer
- LSM303DLHC Accelerometer/Compass

The I²C bus requires, in addition to power and ground, two lines: SCL is the serial clock, and SDA is for bidirectional serial data transmission.



This bus is configured so any device can be "master" and take control of communication with any "slave" it chooses to talk to. Since only one device can talk at a time, this becomes a logistical issue: devices that aren't talking must not have any effect on the bus, or they will prevent other devices from communicating or will introduce errors.

Multi-drop communication is achieved by making all of the connections to the bus **open drain** or **open collector**. In your semiconductors course, you learned about at least one such device – the dedicated comparator. For these devices to work, an external pull-up resistor is needed. Internally, each device has a FET or BJT wired as a switch, but the switches lack an R_D or R_C , which we must provide externally. With the I²C bus, all the devices use the same pull-up resistor, which makes them act as **wired-OR** devices. When their transistors are turned OFF, the pull-up resistor pulls the line up to a logic HIGH. When any one of the devices turns on its transistor, the line pulls down to a logic LOW. So, as long as all devices rest with their transistors off, any single device can talk without interference from the rest.

Another feature of this system is its ability to communicate at different speeds for different devices. The device talking at any point in time generates the clock, which synchronizes communication with the receiving device. The typical maximum speed for the I²C bus is 100 kb/s, but some devices have been designed to handle up to 400 kb/s.

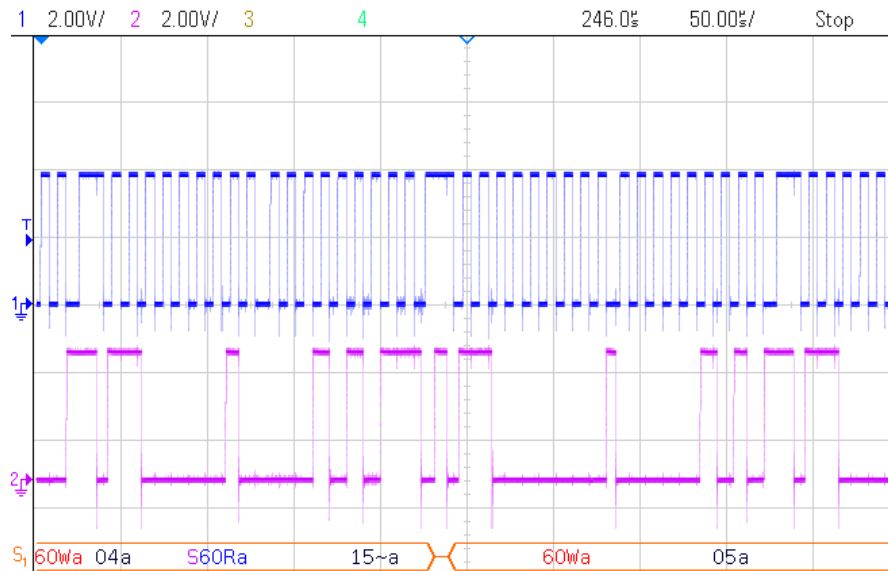
How do the devices know who's talking and who's listening? Each device has a unique 7-bit **address**, partly built into the device and partly hard-wired when the board is built. For example, the LTC2633 DAC has, as the top bits of its address, 00100xxr. There's one pin that makes possible three distinct addresses by controlling the bottom two bits. Here's how that works:

left justify

CA0 Condition	Lower two bits
Ground	00
Floating	01
V _{DD}	10

This means that you could have three LTC2633 DACs on the same bus with different addresses: 0010000r, 0010001r, and 0010010r. Now for one of the things you need to know as a programmer: These seven bits are at the TOP of the address, and the least significant bit is used as a Read/Write line. So, to talk to these devices, you would need to treat these three addresses as 0x20, 0x22, and 0x24. However, as is made obvious by the oscilloscope in this author's cubicle, they are officially only 7-bit addresses with the low bit missing, and should be thought of as 0x10, 0x11, and 0x12!

Here's a screen shot from the author's oscilloscope of the 9S12X talking to a device with the seven-bit address 1100000r which we would have to treat as 0xC0, but is officially 0x60.



The top trace is SCL. Notice that it starts and stops, depending on how the bus is being used. The lower trace is SDA, and contains communication both from the 9S12X, acting as a "master", and the device at address 0b1100000r, acting as a slave. The line at the bottom shows that the master told the device it was going to Write something to it "60W", sent 0x04 "04" to it, indicated a restart "S", told the device it was going to Read from it "60R", so the device put 0x15 "15" on the bus. (The "a" and "~a" are ACK and NACK handshaking tools – each master Command expects an ACK (acknowledge), but communication back from the slave is followed by a NACK (not acknowledge) instead.)

Basic I²C Communication Using the 9S12X

On our board, the 9S12X will always be the Master, and the peripherals on the board will be Slaves. Here's a table of the addresses for the available devices:

Part	Part Number	I²C Address (7-bit)	I²C Address (with R/W as r)
Real Time Clock	M41T81	0x68	0b1101000r
512 kbit Memory (64kB)	24AA512	0x50	0b1010000r
Dual 12-bit DAC	LTC2633HZ12	0x10	0b0010000r
Pressure/Temperature	MPL3115A2	0x60	0b1100000r
Accelerometer	LSM303DLHC	0x19	0b0011001r
Compass	LSM303DLHC	0x1E	0b0011110r

To use the bus, our micro has to do the following:

1. Check to see if the bus is available by checking the "IBB" bit (b5) of the status register (IBSR). This is the "Bus Busy" bit, and is SET when the bus is being used.
2. If the bus is available, issue a "Start" to take control of the bus. (Incidentally, "Start" involves a negative-going transition on SDA while SCL is HIGH. This is generated in our I²C controller by setting the Master and Tx bits in the control register, IBCR.)
3. Notify the desired slave by its address, while at the same time indicating, typically, that we're going to write to it.
4. Send a byte that contains the internal address of the register we want to put something into or take something out of.

Now, things go either of two ways, depending on whether we're writing or reading.

Writing

5. Send the data.
6. Indicate a "Stop" to free up the bus for another device to take control (of course, on our board there's no one else to take control, because there's just one device that has the brains to be Master, and that's our 9S12X). A "Stop" involves a positive-going transition on SDA while SCL is HIGH. (In our I²C module, clear the Tx bit in the IBCR.)

Reading

5. Indicate a "Restart", which will allow us to keep control of the bus, and allows us to issue a new command to the slave of our choice. (There's a Restart bit in IBCR.)
6. Send the slave's address again, but this time indicating we're going to Read from it. (Usually, the contents of the register we indicated in step #4 will be waiting for us.)
7. Receive the data byte.
8. Indicate a "Stop" to free up the bus.

You'll have to also check to see if data is actually available, and wait until communication is complete, etc., but for now that's the basic process.

Variations on the theme:

- If you need to write more than one byte to a device that knows how to do that (e.g. one that auto-increments the internal address), you can just keep writing data bytes until you're done, then indicate a Stop. One example of this is for memory devices that require a 16-bit address, like the 24AA512. For these, we have to send two bytes to establish the starting address of the internal memory location we're

interested in. Then, we can keep sending sequential bytes until we've stored everything we intended to store.

- If you need to read more than one byte from a device that knows how to do that, you can just keep reading data bytes until you're done before indicating a Stop. One example of this is the MPL3115A2, which can send its information in five consecutive bytes. Other devices need to transmit 16-bit values in two bytes.
- ... There seem to be an almost-infinite number of variations on the theme. The datasheets for each I²C device will provide necessary information and timing diagrams to help you establish a working relationship with that device.

You will want to start a new library and library header called "IIC0_Lib". The IIC0 part is because there are two I²C busses on our controller, and the devices on board are wired up to I²C-0. These are the primary functions you'll eventually have in it:

```
void IIC0_Init(void);
void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand, unsigned int iData);
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData);
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg);
```

In the header file, you may choose not to specify the names of the parameters passed, to provide you with more flexibility. Suitable names have been provided above to indicate what the various parameters do.

The Init routine sets up the I²C-0 port in the 9S12XDP512. In the "Data Sheet", this is discussed in detail in Chapter 9, parts of which are included in the discussion below for convenience.

There are five registers used by the I²C controller. To specify which I²C module we're interacting with, we need to tack IIC0_ in front of the register names.

Chapter 9 Inter-Integrated Circuit (IICV2) Block Description

9.3.2 Register Descriptions

This section consists of register descriptions in address order. Each description includes a standard register diagram with an associated figure number. Details of register bit and field function follow the register diagrams, in bit order.

Register Name	Bit 7	6	5	4	3	2	1	Bit 0
IBAD	R	ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1
	W							0
IBFD	R	IBC7	IBC6	IBC5	IBC4	IBC3	IBC2	IBC1
	W							IBC0
IBCR	R	IBEN	IBIE	MS/SL	Tx/Rx	TXAK	0	0
	W						RSTA	IBSWAI
IBSR	R	TCF	IAAS	IBB	IBAL	0	SRW	IBIF
	W							
IBDR	R	D7	D6	D5	D4	D3	D2	D1
	W							D0

□ = Unimplemented or Reserved

Figure 9-2. IIC Register Summary

IIC0_IBAD – This is the I²C slave address assigned to the 9S12. We don't have to worry about this one, as we'll always be the Master, not the Slave.

IIC0_IBFD – This is the Frequency Divider Register to set up the communication rate. To set it up, you need to know what the requirements for the slowest device on the bus will be, then pick a value that matches. This register sets up the clock speed and how many clock cycles will be used for SDA Hold, SCL Hold for Start, and SCL Hold for Stop. There's a divider and a multiplier and a complicated formula, all of which can be bypassed by using

their lookup table, Table 9-5 in the current version of the Data Sheet, which is five pages long! Here's a piece of it that will help us figure out what we need to put into IIC0_IBFD:

Chapter 9 Inter-Integrated Circuit (IICV2) Block Description

Table 9-5. IIC Divider and Hold Values (Sheet 2 of 5)

IBC[7:0] (hex)	SCL Divider (clocks)	SDA Hold (clocks)	SCL Hold (start)	SCL Hold (stop)
3E	3072	513	1534	1537
3F	3840	513	1918	1921
MUL=2				
40	40	14	12	22
41	44	14	14	24
42	48	16	16	26
43	52	16	18	28
44	56	18	20	30
45	60	18	22	32
46	68	20	26	36
47	80	20	32	42
48	56	14	20	30
49	64	14	24	34
4A	72	18	28	38
4B	80	18	32	42
4C	88	22	36	46
4D	96	22	40	50
4E	112	26	48	58

Given the peripherals installed on our board, your instructors (primarily Simon Walker) have determined that we want to operate at 100 kHz, with 20 cycles for SDA Hold, 32 cycles for SCL Hold for Start, and 42 cycles for SCL Hold for Stop. (All of that information is found in the data sheets for the various devices, and values have to be chosen for the slowest device on the bus.) Given that we have an 8 MHz bus clock, you should be able to determine that the appropriate value for IIC0_IBFD is 0x47.

IIC0_IBCR – We want to enable I²C, turn off interrupts, and operate normally in WAIT mode. The rest of the bits can be 0 for now. One hitch: The Data Sheet says that I²C must be enabled before changing any of the other bits in this register, so we have to turn that bit ON first by itself, then make sure the interrupts and WAIT mode bits are turned OFF after that – two writes to this register.

If you've been following this discussion, you should be able to verify the IIC0_Init() routine shown in the start to the IIC0_Lib.c file, shown below.

```
//IIC0 Library Files
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2014

#include <hidef.h>
#include "derivative.h"
#include "IIC0_Lib.h"

void IIC0_Init(void)
{
    IIC0_IBFD=0x47;           //100 kHz, SDA Hold = 20 cks, SCL Hold Start = 32 SCL Hold Stop = 42
    IIC0_IBCR|=0b10000000;   //enable the bus - must be done first
    IIC0_IBCR&=0b10111110;  //no interrupts, normal WAI
}
```

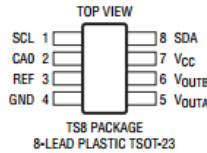
Two bits are important in the Status Register, IIC0_IBSR: *b5*, IBB, is the I²C Bus Busy flag, and *b1*, IBIF, the I²C Interrupt Flag, which is set whenever a transfer is complete (whether or not we have interrupts enabled). The IBIF flag needs to be cleared by writing a 1 to it. For some devices, such as the 24AA412 EEPROM, you will also need to monitor RXAK, the "receive acknowledge" bit if you want to access more than one byte of memory at a time.

LTC2633HZ12 I²C DAC – 16-bit Data Writes

We’re going to start with a device that “breaks the rules”, because it’s a fairly easy device to work with. It has no registers that we need to read, so communication is one-way. However, it’s a 12-bit DAC, so it needs a dedicated 16-bit “write” function so you can get going with the I²C bus, then later we’ll write the standard 8-bit “write” and “read” functions.

The LTC2633HZ12 DAC that’s been added to your board is a dual 12-bit DAC, set up so that its internal reference is effectively 4.096 V. DACA and DACB, the outputs of the two internal DACs, are available to the right of the DAC module on your microcontroller kit, with a ground pin positioned between them.

The pinout of the IC is as follows:



On the board, “CA0” is connected to ground to make the device’s slave address end in “00” – i.e. 0x10 or 0b0010000r. (Check back a few pages to recall why this is the case.) If you want, you can add up to two more LTC2633HZ12 modules: leaving CA0 floating makes the address 0x11 or 0b0010001r, and connecting CA0 to +5 V makes it 0x12 or 0b0010010r.

The data sheet for this device goes into a lot of detail about how to use it, but the following snippet is particularly informative for us in terms of how to send information to the DAC.

LTC2633

OPERATION

The format of the three data bytes is shown in Figure 3. The first byte of the input word consists of the 4-bit command, followed by the 4-bit DAC address. The next two bytes contain the 16-bit data word, which consists of the 12-, 10- or 8-bit input code, MSB to LSB, followed by 4, 6 or 8 don’t-care bits (LTC2633-12, LTC2633-10 and LTC2633-8 respectively). A typical LTC2633 write transaction is shown in Figure 4.

The command bit assignments (C3-C0) and address (A3-A0) assignments are shown in Tables 3 and 4. The first four commands in the table consist of write and update operations. A write operation loads a 16-bit data word from the 32-bit shift register into the input register. In an update operation, the data word is copied from the input register to the DAC register. Once copied into the DAC register, the data word becomes the active 12-, 10-, or 8-bit input code, and is converted to an analog voltage at the DAC output. Write to and update combines the first two commands. The update operation also powers up the DAC if it had been in power-down mode. The data path and registers are shown in the Block Diagram.

Table 3. Command Codes

COMMAND*				
C3	C2	C1	C0	
0	0	0	0	Write to Input Register n
0	0	0	1	Update (Power-Up) DAC Register n
0	0	1	0	Write to Input Register n, Update (Power-Up) All
0	0	1	1	Write to and Update (Power-Up) DAC Register n
0	1	0	0	Power-Down n
0	1	0	1	Power-Down Chip (All DAC’s and Reference)
0	1	1	0	Select Internal Reference (Power-Up Reference)
0	1	1	1	Select External Reference (Power-Down Internal Reference)
1	1	1	1	No Operation

*Command codes not shown are reserved and should not be used.

Table 4. Address Codes

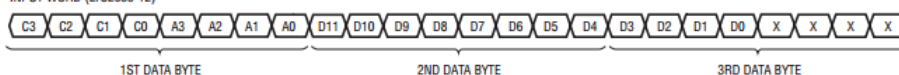
ADDRESS (n)*				
A3	A2	A1	A0	
0	0	0	0	DAC A
0	0	0	1	DAC B
1	1	1	1	All DACs

* Address codes not shown are reserved and should not be used.

WRITE WORD PROTOCOL LTC2633



INPUT WORD (LTC2633-12)



Since it's a 12-bit device, it's obviously going to need two bytes of data sent to it. In fact, as an I²C device, it's going to need three bytes total: A Command byte and two bytes of data, as shown on the previous page. Of course, this comes after identifying the DAC itself on the I²C bus using its address.

We'll send the address as a parameter each time to be consistent with other functions in our I²C library, although it should always be 0x20 – based on the seven-bit I²C address of 0x10, which translates to 0b0010000r, where 'r' is the Read/Write bit. If, at a later date, you add another DAC or two to your board, you will be able to use this function for them, as well – just wire them up for the other two addressing options, and send to 0x21 or 0x22.

From the data sheet, the device defaults to the mode in which the DAC reference is 4.096 V. Also, the command we're going to use to write to the device includes a power-up, so we don't need to do any initializing.

The Command byte is actually made up of one nibble for the command and one nibble to specify which DAC channel or channels you're addressing. The instructors in this course have played with this device a fair bit, and have determined that the simplest way to send a value to one of the DAC channels so that it appears instantly is to use the "Write to and update DAC register n" command, 0b0011. Then, the lower nibble will be 0b0000 for DAC A, 0b0001 for DAC B, 0r 0b1111 to set both DACs to the same value.

Like a number of I²C devices (another example is the MPL3115A2 barometer), this device expects its data to arrive "left-justified", meaning that the 12 bits it's expecting are the upper 12, not the lower 12 of the 16 bits in an *unsigned int*. (Incidentally, this is to keep it compatible with other members of the family that have more bits, which improve the resolution by using the lower (i.e. finer resolution) bits.) So we need to send a byte that contains the 8 upper bits, and a byte that contains the 4 lower bits followed by four zeros.

We need to write a new version of the I²C "Write" command that fits the following header:

```
void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand, int iData);
```

We also need to know what the step size is for this DAC.

$$StepSize = \frac{V_{ref}}{2^n} = \frac{4.096}{2^{12}} = 1mV / step$$

When we send a numeric value to the DAC, it will be a number representing the voltage in millivolts, and it will be in the format we're used to: right-justified hexadecimal. So, the first thing we need to do in our function for writing to the DAC is to convert the incoming value to left-justified format, which simply means moving it from the lower 12 bits to the upper 12 bits of a 16-bit value. Once that's been done, we need to send the result out as two 8-bit bytes, since we can only send 8 bits at a time on the I²C bus.

Based on the timing diagram on the previous page, the three bytes can be sent one after the other, as long as we wait for the I²C flag to indicate that the device is ready for another byte. (Incidentally, this is the sequential write process discussed previously, and can be used for other devices we've touched on that have this as one of their modes of operation.)

Again, if you've been following the previous discussion, you should come up with something like the code on the following page. Don't just copy this: Work it through to make sure you understand what it does. You may also want to come up with more sophisticated ways of moving the data to the right place in the *int* variable and parsing out the two bytes, such as using regular division and MOD division.

```

void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand, int iData)
{
    iData*=16;                //move data into the upper 12 bits

    while(IIC0_IBSR & 0b00100000); //wait for not busy flag
    IIC0_IBCR |= 0b00110000;      //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110; //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010;      //clear flag

    IIC0_IBDR = cCommand;        //send the desired command
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010;      //clear flag

    IIC0_IBDR = (unsigned char)(iData/256); //send first unsigned char of the data
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010;      //clear flag

    IIC0_IBDR = (unsigned char)(iData&0b0000000011111111); //send second unsigned char of the data
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11001111;      //stop transmitting, exit Master mode
    IIC0_IBSR |= 0b00000010;      //clear flag
}

```

The following code snippet shows a very simple implementation of the preceding function that generates two 4,095-step ramp waves: a rising ramp on DAC A and a falling ramp on DAC B. You could use this to check your IIC0_WriteDAC() function. You will need to declare an *unsigned int* variable called iDataOut.

```

IIC0_Init();

for (;;)
{
    IIC0_WriteDAC(0x20, 0b00110000, iDataOut);
    IIC0_WriteDAC(0x20, 0b00110001, 0x0FFF-iDataOut++);
}
}

```


Standard Eight-Bit Writing

The standard 8-bit write routine is more generally useful for other I²C peripherals. Here's the prototype for this Write command from the header file:

```
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData);
```

The Write function needs to be supplied with a device address, an internal address for the register we're interested in, and a byte of data to put in that register. Here's the procedure:

1. Watch the Status Register (IIC0_IBSR) to see when the bus is Not Busy, as indicated by a LOW on the IBB bit, b5.
2. Once the bus is free, change the micro to "Master" mode, set to "Transmit". These bits are in the Control Register, IIC0_IBCR.
3. Place the device address on the bus with the LSB set to "Write" mode.
4. Wait for the Byte Transfer Complete process, as indicated on the IBIF flag of the Status Register.
5. Clear the IBIF flag by writing a "1" to it.
6. Repeat the last three steps, but this time with the internal address.
7. Repeat, but this time with the data byte, and don't clear the IBIF flag yet.
8. Stop transmitting and exit "Master" mode, using the Control Register.
9. Finally, clear the IBIF flag.

Take some time to see how the above discussion is handled in the following function.

```
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData)
{
    while(IIC0_IBSR & 0b00100000); //wait for not busy flag
    IIC0_IBCR |= 0b00110000; //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110; //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cReg; //locate desired register
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cData; //send data
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11001111; //stop transmitting, exit Master mode
    IIC0_IBSR |= 0b00000010; //clear flag
}
```

Note: This routine, and the others in this set of Course Notes, are handled as simply as possible, and do not provide means of escape if something goes wrong with communication – the system may simply freeze, waiting for a flag that never comes up. If you have an application where the system needs to be essentially fail-proof, you will need to incorporate ways of handling various unexpected exceptions, particularly by avoiding blocking loops (of which there are four in the previous code alone!). Your instructor or someone with a lot of experience with this (*i.e.* Simon Walker) might be willing to help you with this.

Reading

As you can see from the header, the 8-bit "Read" routine needs to be supplied with a device address and an internal address for the register we're interested in.

```
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg);
```

The contents of that register are returned to the main program as a byte. Here's the procedure, the first six parts of which were also part of the "Write" routine:

1. Watch the Status Register (IIC0_IBSR) to see when the bus is Not Busy.
2. Once the bus is free, change the micro to "Master" mode, set to "Transmit".
3. Place the device address on the bus with the LSB in "Write" mode.
4. Wait for the IBIF flag of the Status Register.
5. Clear the IBIF flag.
6. Repeat the last three steps, but this time with the internal address.
7. Now, using the Control Register, issue a "Restart" command.
8. Place the device address on the bus with the LSB in "Read" mode.
9. Wait for the IBIF flag of the Status Register.
10. Clear the IBIF flag.
11. Using the Control Register, get ready to Receive a byte. The last byte received from a device is supposed to have a NACK following it, so we need to indicate that no ACK is required. Since you need to SET one bit and CLEAR another bit, this will take two steps.
12. Here's a curious fact: in order to initiate a Read, you need to read the I²C Data Register (IIC0_IBDR) once, which will generate garbage, before you move on.
13. Next, you wait for the IBIF flag, but you don't clear it yet.
14. Instead, you "Stop" by exiting "Master" mode, using the Control Register.
15. Now, clear the IBIF flag.
16. Finally, you can read the real data out of the I²C Data Register and return it to the main program.

Take some time to see how the above discussion is handled in the function shown below.

```
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg)
{
    unsigned char cData;

    while(IIC0_IBSR & 0b00100000); //wait for not busy flag
    IIC0_IBCR |= 0b00110000; //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110; //place address on bus with Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cReg; //locate desired register
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR |= 0b00000100; //restart

    IIC0_IBDR = (cAddr | 0b00000001); //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR |= 0b00001000; //reading 1 unsigned char only
    IIC0_IBCR &= 0b11101111; //receive unsigned char
    cData = IIC0_IBDR; //not actually -- starts the process
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11011111; //stop
    IIC0_IBSR |= 0b00000010; //clear flag
    cData = IIC0_IBDR; //for real this time

    return cData;
}
```

MPL3115A2: Standard 8-bit Reads and Writes

The MPL3115A2 "Precision Altimeter" uses the standard 8-bit Read and Write functions discussed previously, since its internal registers are few enough to have only 8-bit addresses, and the data appears in 8-bit registers.

The following is a flowchart from the MPL3115A2 data sheet that shows how to set up the device and how to read the internal registers. For this course, we'll just be using the "Polling" side of the flowchart, so it's not as bad as it seems at first glance. The discussion on the following pages will follow the flowchart, with some changes to the data sent to make the device do what we specifically want it to do.

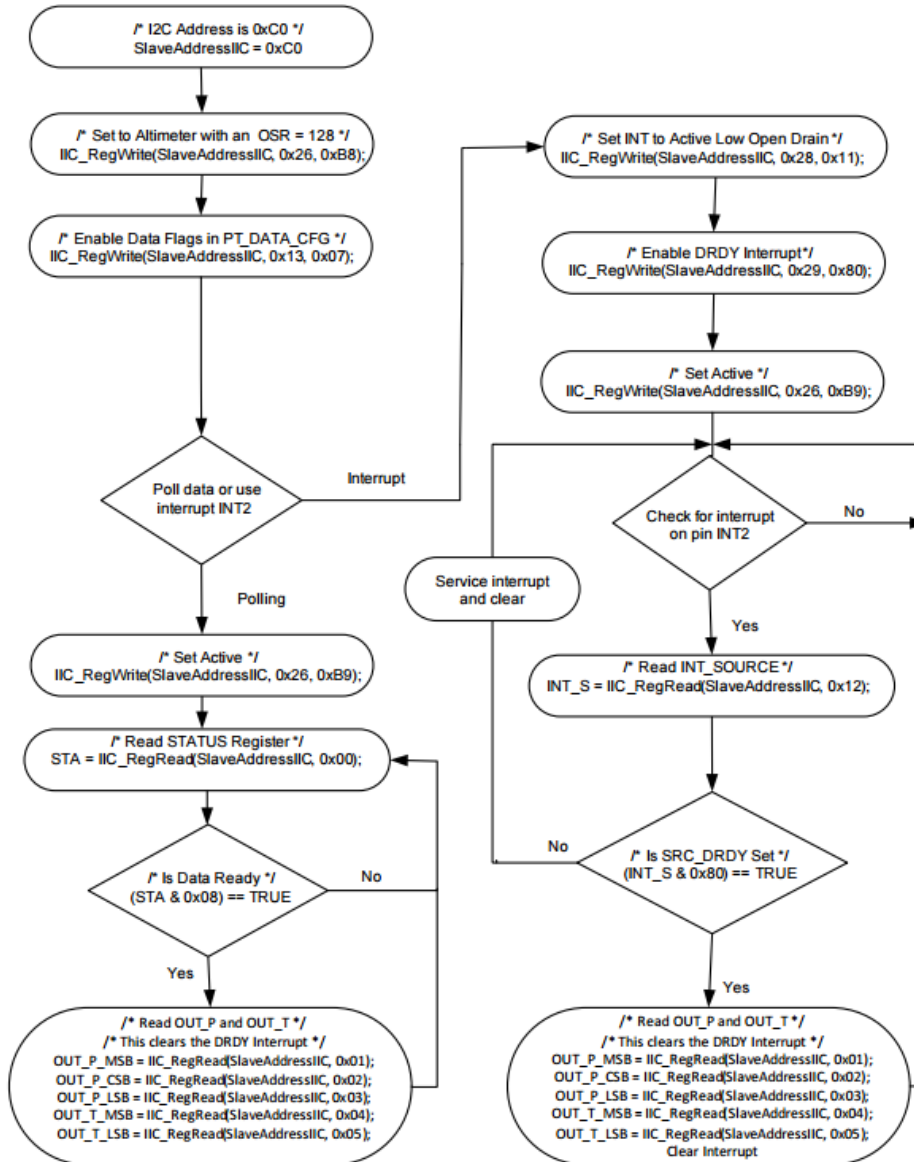


Figure 5. Polling or Interrupt - No FIFO

MPL3115A2

The registers we need to use are summarized below:

7 Register Descriptions

Table 9. Register Address Map

Register Address	Name	Reset	Reset when STBY to Active	Type	Auto-Increment Address		Comment	
0x00	Sensor Status Register (STATUS) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x01		Alias for DR_STATUS or F_STATUS	
0x01	Pressure Data Out MSB (OUT_P_MSB) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x02	0x01	Bits 12-19 of 20-bit real-time Pressure sample.	Root pointer to Pressure and Temperature FIFO data.
0x02	Pressure Data Out CSB (OUT_P_CSB) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x03		Bits 4-11 of 20-bit real-time Pressure sample	
0x03	Pressure Data Out LSB (OUT_P_LSB) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x04		Bits 0-3 of 20-bit real-time Pressure sample	
0x04	Temperature Data Out MSB (OUT_T_MSB) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x05		Bits 4-11 of 12-bit real-time Temperature sample	
0x05	Temperature Data Out LSB (OUT_T_LSB) ⁽¹⁾⁽²⁾	0x00	Yes	R	0x00		Bits 0-3 of 12-bit real-time Temperature sample	
0x13	PT Data Configuration Register (PT_DATA_CFG) ⁽¹⁾⁽³⁾	0x00	No	R/W	0x14		Data event flag configuration	
0x26	Control Register 1 (CTRL_REG1) ⁽¹⁾⁽⁴⁾	0x00	No	R/W	0x27		Modes, Oversampling	
0x27	Control Register 2 (CTRL_REG2) ⁽¹⁾⁽⁴⁾	0x00	No	R/W	0x28		Acquisition time step	
0x28	Control Register 3 (CTRL_REG3) ⁽¹⁾⁽⁴⁾	0x00	No	R/W	0x29		Interrupt pin configuration	
0x29	Control Register 4 (CTRL_REG4) ⁽¹⁾⁽⁴⁾	0x00	No	R/W	0x2A		Interrupt enables	
0x2A	Control Register 5 (CTRL_REG5) ⁽¹⁾⁽⁴⁾	0x00	No	R/W	0x2B		Interrupt output pin assignment	

Here are some points about the device, the flowchart, and the registers involved.

- The device’s I2C address is 0x60 (i.e. 0b1100000r), so we need to communicate with it using 0xC0. Instead of making a variable to hold the slave address as shown in the flowchart, we can just put 0xC0 in the slave address parameter field.
- We’ll be using our function “IIC0_Write” in place of their “IIC_RegWrite” function.
- During configuration, we need to set up Control Register 1 (0x26). The other registers default to conditions that are acceptable for us at this point. Here are the bits of CTRL_REG1:

Table 57. CTRL_REG1 Register

	7	6	5	4	3	2	1	0
R						0		
W	ALT	RAW	OS2	OS1	OS0	RST	OST	SBYB
Reset	0	0	0	0	0	0	0	0

Following the flowchart, we want to set this up for an “over-sampling rate” of 128 by setting the OSn bits to produce 2⁷. Also, the flowchart suggests setting b7, which puts the device into “Altimeter” mode instead of “Barometer” mode. This results in the value 0b10111000, or, as indicated in the flowchart, 0xB8. We actually want to operate in “Barometer” mode, so we’ll send 0b00111000 instead.

- The flowchart then sends 0b00000111 (0x07) to the PT_DATA_CFG register to enable data flags so that pressure events (PDEFE) or temperature events (TDEFE) are available, using Data Ready Event Mode (DREM).

Table 37. PT_DATA_CFG Register

	7	6	5	4	3	2	1	0
R	0	0	0	0	0	DREM	PDEFE	TDEFE
W								
Reset	0	0	0	0	0	0	0	0

- The last step in the initialization part of the flowchart takes us back to CTRL_REG1, where we take it out of standby (SBYB) by putting a 1 in *b0*. Since there isn't a good way of just use OR to change a single bit, we overwrite the register with the new value: 0b00111001. This is shown in the flowchart as 0xB9, but that was for Altimeter mode, so, for Barometer mode, we don't want the MSB set.
- Simon Walker indicates that, for reliable operation, we now need to read the Status register once before entering the main loop that repeatedly checks the status register, then reads the data registers when valid data is indicated.
- Inside the loop, as indicated in the flowchart, we wait until the MPL3115A2 reports the availability of good data, then we read the registers we're interested in. For pressure information, these would be 0x01, 0x02, and 0x03. For temperature information, these would be 0x04 and 0x05.
- The bits of the status register (0x00) are as follows:

Table 12. DR_STATUS Register

	7	6	5	4	3	2	1	0
R	PTOW	POW	TOW	0	PTDR	PDR	TDR	0
W								
Reset	0	0	0	0	0	0	0	0

The only bit we need to concern ourselves with is *b3*: PTDR stands for "Pressure and Temperature Data Ready". We have to wait for this bit to be SET before we can read valid data from the other registers. (If you wanted, you could watch for "PDR" or "TDR" instead, if you only needed pressure or temperature data.)

The values in the data registers are presented in a slightly complicated format, in that they are left-justified (occupying the upper bits and leaving the least significant bits as zeros), and they have a fractional component.

The pressure data is in Pascals (Pa), and is provided as 20-bits in Q18.2 format. This means that the most significant 18 bits are the hexadecimal value of the pressure in Pascals, and the other two bits are, in order, $\frac{1}{2}$ (2^{-1}) and $\frac{1}{4}$ (2^{-2}) Pascal weightings.

For example, the three-byte value for pressure can be interpreted as follows:

0x01 = 0b0110 1010

0x02 = 0b0111 0001

0x03 = 0b1001 0000

Pressure = 0b011010100111000110.01 = 108,998.25 Pa

The temperature is even more complicated, as it is returned as 2's complement negative 12-bit fractional data in °C, provided in Q8.4 format (although the data sheet says Q12.4, but that would be 16 bits). The whole-degrees portion is in the first byte (Register 0x04), and the fractional part is the high nibble of Register 0x05, accurate to $1/16^{\text{th}}$ of a degree (2^{-4}).

If you just want to display whole degrees, you can ignore Register 0x05 and just use Register 0x04. To display negative temperatures in a form suitable for human consumption (pretty important in Alberta!), you would need to perform a 2's complement conversion on the negative values to find their magnitude, then simply insert a "negative" sign in front of the value.

A simpler way to handle both of these sensor values is to use "sprintf" and put the formatted results into printable strings – see the discussion at the end of using the SCI.

M41T81 Real-Time Clock – Standard 8-bit Reads and Writes

Back to I²C devices: The M41T81 Real-Time Clock is designed to do either multi-byte communication (like the EEPROM), or individual register reads to get the information we're looking for, and individual register writes to change the time, date, or control bits.

As with the other devices we've covered in this course, there are a lot of features we don't have time to cover in this brief overview. However, the full details can be found in the data sheet for this device, available from the Internet or in Moodle for this course.

The designers of this device chose to present the data in BCD, which is quite helpful, in that we don't need to do conversions before we display its results.

Since the results are in BCD, you need to work with the two nibbles in a byte to get the full number (e.g.) to get 35 s, you read register 0x01, in which the upper nibble will contain 3 (once ST is masked off) and the lower nibble will be 5.

The M41T81 has likely been running on your board ever since it was assembled, and it contains a lot of information, which, at this point, is almost guaranteed to be incorrect. This device has its own internal 32.768 kHz crystal, so it doesn't rely on the bus clock or an external crystal oscillator. It has a backup battery with a circuit that detects when the main board power is turned off, so it continues to maintain the time and any settings when the board is turned off. It keeps track of time and date in hundredths of seconds, seconds, minutes, hours, day of week, date, months, years, and even centuries (although just the twenty-first and twenty-second centuries!), with leap years built in, so it is a true calendar as well as a clock.

Along with current time and date, the device has the capability of being used as an alarm, as a square wave generator, and as a microcontroller monitoring device called a "watchdog". Here's a list of its internal registers, along with a description of the labels used for control bits that are scattered throughout the registers:

M41T81

Clock operation

Table 2. Clock register map⁽¹⁾

Addr									Function/range BCD format	
	D7	D6	D5	D4	D3	D2	D1	D0		
00h	0.1 seconds				0.01 seconds				Seconds	00-99
01h	ST	10 seconds			Seconds				Seconds	00-59
02h	0	10 minutes			Minutes				Minutes	00-59
03h	CEB	CB	10 hours		Hours (24 hour format)				Century/hours	0-1/00-23
04h	0	0	0	0	0	Day of week			Day	01-7
05h	0	0	10 date		Date: day of month				Date	01-31
06h	0	0	0	10M	Month				Month	01-12
07h	10 years				Year				Year	00-99
08h	OUT	FT	S	Calibration				Control		
09h	0	BMB4	BMB3	BMB2	BMB1	BMB0	RB1	RB0	Watchdog	
0Ah	AFE	SQWE	ABE	AI 10M	Alarm month				AI month	01-12
0Bh	RPT4	RPT5	AI 10 date		Alarm date				AI date	01-31
0Ch	RPT3	HT	AI 10 hour		Alarm hour				AI hour	00-23
0Dh	RPT2	Alarm 10 minutes			Alarm minutes				AI min	00-59
0Eh	RPT1	Alarm 10 seconds			Alarm seconds				AI sec	00-59
0Fh	WDF	AF	0	0	0	0	0	0	Flags	
10h	0	0	0	0	0	0	0	0	Reserved	
11h	0	0	0	0	0	0	0	0	Reserved	
12h	0	0	0	0	0	0	0	0	Reserved	
13h	RS3	RS2	RS1	RS0	0	0	0	0	SQW	

Keys:
 S = Sign bit
 FT = Frequency test bit
 ST = Stop bit
 0 = Must be set to '0'
 BMB0-BMB4 = Watchdog multiplier bits
 CEB = Century enable bit
 CB = Century bit
 OUT = Output level
 ABE = Alarm in battery backup mode enable bit
 AFE = Alarm flag enable flag
 RB0-RB1 = Watchdog resolution bits
 RPT1-RPT5 = Alarm repeat mode bits
 WDF = Watchdog flag (read only)
 AF = Alarm flag (read only)
 SQWE = Square wave enable
 RS0-RS3 = SQW frequency
 HT = Halt update bit

Notice again that the data is presented in Binary-Coded Decimal (BCD), using two nibbles for each item. The number of bits required for the most significant digit depends on the size of the particular digit, so, for example, the month only needs a zero or a one for its first digit, the day of the month requires two bits for its first digit to cover the possibilities of 0, 1, 2, or 3, and the year requires all four bits to represent values from 0 to 9.

Since the other bits aren't needed for the data, some of them get used for control and reporting functions. A very important one is "ST", which is the MSB of the "seconds" register. This bit stops the clock crystal, and the registers hold the last available data.

Another bit that's very important is *b6* of the Alarm Hour register: "HT". When the power on the board goes down and the Real Time Clock switches to battery, this bit is set to halt the updating of the registers, while the clock continues to run in the background. This allows the user to write code to read the registers on power-up to find out when the power failed, before reactivating the normal operation in which the proper time will be reported.

So, if you want to know when the power went down, you can read the time registers before clearing the HT bit to get the power-down information; in any case, you will need to clear the "HT" bit to let the clock report the current time.

The "ST" bit must be cleared to allow the clock crystal to run. Once this bit has been cleared, it will stay cleared until it is deliberately set, so it's best to check to see if it needs to be cleared before doing anything to it. If you go through the process of clearing it unnecessarily, you may lose the occasional second on the clock, since this bit is in the "seconds" register. Why? Because to clear the ST bit, we need to read the seconds register, clear the ST bit in the read-in value, then write the resulting value back into the seconds register. If the seconds register updates during this process, the value written back in will be the old value, which is one second behind. Also, if you clear the ST bit before you clear the HT bit, you will be reading in the seconds value that was held for reporting the power-down, and will write that back in, overwriting the current seconds value with an old (and incorrect) value.

Unlike interrupt flags, these bits are cleared by writing '0' to them.

The Real-Time Clock's I²C address is 0x68, which translates into 0b1101000r for our purposes. The code snippet on the following page shows how to start the clock if it isn't running, how to allow it to report current data, and how to read in current data from the registers associated with seconds, minutes, hours, date, month, and year. The actual process of displaying the data is not shown, as that would be dependent on the display device chosen.

```
HasHT=IIC0_Read(0xD0,0x0C);
IIC0_Write(0xD0,0x0C,(HasHT&0b10111111)); //clear the Halt bit

Sec=IIC0_Read(0xD0,0x01); //if set, clear the ST bit
if((Sec&0b10000000)!=0) IIC0_Write(0xD0,0x01,(Sec&0b01111111));

//SetTime(0x30,0x25,0x10,0x02,0x07,0x12,0x15);

for (;;) //endless program loop
{
/*****
* Main Program Code
*****/

    Sec=IIC0_Read(0xD0,0x01)&0b01111111;
    Min=IIC0_Read(0xD0,0x02)&0b01111111;
    Hr=IIC0_Read(0xD0,0x03)&0b00111111;
    Date=IIC0_Read(0xD0,0x05)&0b00111111;
    Mth=IIC0_Read(0xD0,0x06)&0b00011111;
    Year=IIC0_Read(0xD0,0x07);

    UpdateDisplay();
}
```

In this code snippet, notice there's a "SetTime" function call that has been commented out. The code was run once with that line included, then the version of the code with it excluded was down-loaded to the microcontroller so that further resets or power-ups will not set the time back to the numbers hard-coded into this routine. Clearly, a more sophisticated means of setting the clock would be useful – for example, a function that responds to a switch press if the user wants to set the time.

Notice that all of the values sent to the "SetTime" function are indicated as hexadecimal: that's because BCD, which is what the clock is expecting, isn't decimal – it's a binary (or hex) code used to represent decimal values. So, 0x31 represents 31 minutes in the second byte of the function call.

Position Information with the LSM303DLHC – Standard 8-bit Reads and Writes

(*Optional topic*) Another device that can be controlled and accessed using eight-bit address and data reads and writes is the LSM303DLHC eCompass Module. (It can also be accessed using multi-byte sequences, but we'll stick with the easier approach.) This unit contains a three-axis accelerometer, a three-axis magnetometer, and, probably because everyone else is doing it, another temperature sensor. Here's a clip from the datasheet.

LSM303DLHC

Block diagram and pin description

1.2 Pin description

Figure 2. Pin connections

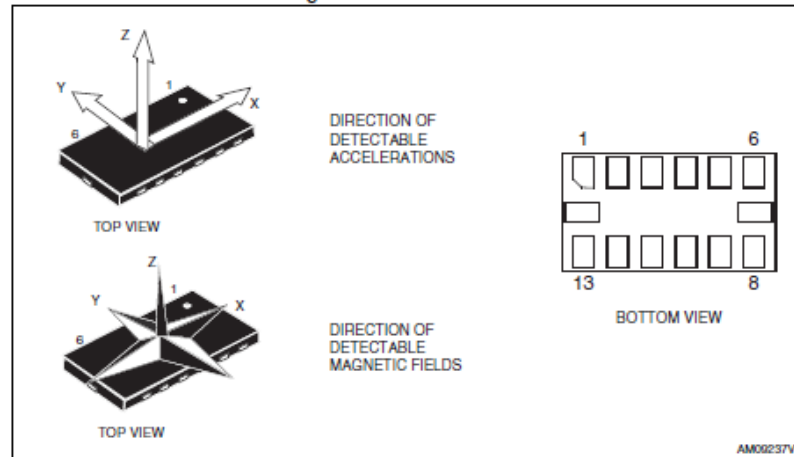


Table 2. Pin description

Pin#	Name	Function
1	Vdd_IO	Power supply for I/O pins
2	SCL	Signal interface I ² C serial clock (SCL)
3	SDA	Signal interface I ² C serial data (SDA)
4	INT2	Inertial interrupt 2
5	INT1	Inertial interrupt 1
6	C1	Reserved capacitor connection (C1)
7	GND	0 V supply
8	Reserved	Leave unconnected
9	DRDY	Data ready
10	Reserved	Connect to GND
11	Reserved	Connect to GND
12	SETP	S/R capacitor connection (C2)
13	SETC	S/R capacitor connection (C2)
14	Vdd	Power supply

3-Axis Accelerometer

You should recall, from earlier physics-related science courses, that objects near the earth's surface accelerate at a rate of approximately 9.81 m/s^2 if allowed to fall freely. This is referred to as 1.0 g (not to be confused with the SI unit for grams – we just ran out of letters!). The accelerometers in the LSM303DLHC report acceleration in milli-g's, which shows up in the datasheet as mg , again, not to be confused with milligrams.

The seven-bit I²C address for the accelerometer is $0x19$, so the eight-bit representation of this is $0b0011001r$.

There are a bunch of internal registers associated with the accelerometer:

Name	Type	Register	Name	Type	Register
		Hex			Hex
Reserved (do not modify)		00 - 1F	INT1_CFG_A	rw	30
CTRL_REG1_A	rw	20	INT1_SRC_A	r	31
CTRL_REG2_A	rw	21	INT1_THS_A	rw	32
CTRL_REG3_A	rw	22	INT1_DURATION_A	rw	33
CTRL_REG4_A	rw	23	INT2_CFG_A	rw	34
CTRL_REG5_A	rw	24	INT2_SRC_A	r	35
CTRL_REG6_A	rw	25	INT2_THS_A	rw	36
REFERENCE_A	rw	26	INT2_DURATION_A	rw	37
STATUS_REG_A	r	27	CLICK_CFG_A	rw	38
OUT_X_L_A	r	28	CLICK_SRC_A	rw	39
OUT_X_H_A	r	29	CLICK_THS_A	rw	3A
OUT_Y_L_A	r	2A	TIME_LIMIT_A	rw	3B
OUT_Y_H_A	r	2B	TIME_LATENCY_A	rw	3C
OUT_Z_L_A	r	2C	TIME_WINDOW_A	rw	3D
OUT_Z_H_A	r	2D	Reserved (do not modify)		3E-3F
FIFO_CTRL_REG_A	rw	2E			
FIFO_SRC_REG_A	r	2F			
INT1_CFG_A	rw	30			
INT1_SRC_A	r	31			
INT1_THS_A	rw	32			
INT1_DURATION_A	rw	33			

That's a lot of registers! At least they still fit in an 8-bit internal address space, so we don't need a new routine to set up this IC. The registers on the following pages the ones that are significant to us at this point, but you may find you can make use of some of the more esoteric features of this chip, such as the free-fall sensors (possibly used to park a hard-drive on a falling laptop), or the "click" sensor (possibly used to determine if someone is tapping an interactive display).

7.1.1 CTRL_REG1_A (20h)

Table 18. CTRL_REG1_A register

ODR3	ODR2	ODR1	ODR0	LPen	Zen	Yen	Xen
------	------	------	------	------	-----	-----	-----

Table 19. CTRL_REG1_A description

ODR[3:0]	Data rate selection. Default value: 0000 (0000: power-down, others: refer to Table 20)
LPen	Low-power mode enable. Default value: 0 (0: normal mode, 1: low-power mode)
Zen	Z-axis enable. Default value: 1 (0: Z-axis disabled, 1: Z-axis enabled)
Yen	Y-axis enable. Default value: 1 (0: Y-axis disabled, 1: Y-axis enabled)
Xen	X-axis enable. Default value: 1 (0: X-axis disabled, 1: X-axis enabled)

ODR[3:0] is used to set the power mode and ODR selection. In the following table bit selection of ODR [3:0] for all frequencies is shown.

Table 20. Data rate configuration

ODR3	ODR2	ODR1	ODR0	Power mode and ODR selection
0	0	0	0	Power-down mode
0	0	0	1	Normal / low-power mode (1 Hz)
0	0	1	0	Normal / low-power mode (10 Hz)
0	0	1	1	Normal / low-power mode (25 Hz)
0	1	0	0	Normal / low-power mode (50 Hz)
0	1	0	1	Normal / low-power mode (100 Hz)
0	1	1	0	Normal / low-power mode (200 Hz)
0	1	1	1	Normal / low-power mode (400 Hz)
1	0	0	0	Low-power mode (1.620 kHz)
1	0	0	1	Normal (1.344 kHz) / low-power mode (5.376 kHz)

From this set of tables, we can determine the values needed to enable the three axis sensors, turn on the accelerometer, and set up its refresh rate. Note that the default condition, 0b00000111 (found in the Register address map table of the data sheet), disables the accelerometer, so we have to deal with this register. For our purposes, a speed of 100 Hz in normal mode, with all three axes enabled is suitable: 0b01010111.

For now, we'll leave control registers 2_A, 3_A, 5_A, and 6_A as they are – they deal with interrupts and some of the features that are less useful to us at this point.

Table 21. CTRL_REG2_A register

HPM1	HPM0	HPCF2	HPCF1	FDS	HPCLICK	HPIS2	HPIS1
------	------	-------	-------	-----	---------	-------	-------

Default 0b00000000

Table 24. CTRL_REG3_A register

I1_CLICK	I1_AOI1	I1_AOI2	I1_DRDY1	I1_DRDY2	I1_WTM	I1_OVERRUN	--
----------	---------	---------	----------	----------	--------	------------	----

Default 0b00000000

Table 28. CTRL_REG5_A register

BOOT	FIFO_EN	--	--	LIR_INT1	D4D_INT1	LIR_INT2	D4D_INT2
------	---------	----	----	----------	----------	----------	----------

Default 0b00000000

Table 30. CTRL_REG6_A register

I2_CLICKen	I2_INT1	I2_INT2	BOOT_I1	P2_ACT	--	H_LACTIVE	--
------------	---------	---------	---------	--------	----	-----------	----

Default 0b00000000

7.1.4 CTRL_REG4_A (23h)

Table 26. CTRL_REG4_A register

BDU	BLE	FS1	FS0	HR	0 ⁽¹⁾	0 ⁽¹⁾	SIM
-----	-----	-----	-----	----	------------------	------------------	-----

1. This bit must be set to '0' for correct operation of the device.

Table 27. CTRL_REG4_A description

BDU	Block data update. Default value: 0 (0: continuous update, 1: output registers not updated until MSB and LSB have been read)
BLE	Big/little endian data selection. Default value 0. (0: data LSB @ lower address, 1: data MSB @ lower address)
FS[1:0]	Full-scale selection. Default value: 00 (00: ±2 g, 01: ±4 g, 10: ±8 g, 11: ±16 g)
HR	High-resolution output mode: Default value: 0 (0: high-resolution disable, 1: high-resolution enable)
SIM	SPI serial interface mode selection. Default value: 0 (0: 4-wire interface, 1: 3-wire interface).

For this, you need to know something you may have learned earlier: the difference between Motorola-type and Intel-type microprocessors. Motorola-type processors are referred to as "Big-Endian", as sixteen-bit values are accessed MSbyte first, LSbyte last; Intel-type processors are referred to as "Little-Endian", as sixteen-bit values are accessed LSbyte first, MSbyte last. Although the register tables tell us that the MSbyte for each accelerometer appears at the lower of the two addresses, that isn't necessarily the case: in Little-Endian mode (the default), the lower of the two addresses is actually the LSbyte, which can be very confusing. So, we want to put this device into Big-Endian mode.

Also, we need to determine the full-scale readings for the accelerometer. This is a good place to note that the values are returned as **16-bit 2's complement** signed integers – but we'll be reading them as two eight-bit values. More on that later. The FS bits determine the range that can be covered by the device. For high sensitivity, we'll choose the ±2 g scale (00). Given that this is a 12-bit device, the step size is

$$stepsize = \frac{4g}{2^{12} - 1}, \text{ or } 0.977 \text{ mg/step (Don't believe this value too quickly!)}$$

0.977 looks eerily close to 1 mg/step. In fact, in the datasheet, Table 3, the sensitivity is shown as below:

LA_So	Linear acceleration sensitivity	FS bit set to 00		1	mg/LSB
		FS bit set to 01		2	
		FS bit set to 10		4	
		FS bit set to 11		12	

The datasheet doesn't indicate which of these values is correct – it could be that the "±2 g" is an approximate value and the sensitivity is actually 1 mg/step for a true scale of ±2.047 g, or it could be that the full scale is accurate, and the sensitivity is rounded. We'll assume that the 1 mg/step is correct, as this seems to be verified empirically.

A suitable entry for CTRL_REG4_A is 0b01001000.

The data shows up in the following registers:

- 7.1.9 **OUT_X_L_A (28h), OUT_X_H_A (29h)**
X-axis acceleration data. The value is expressed in two's complement.
- 7.1.10 **OUT_Y_L_A (2Ah), OUT_Y_H_A (2Bh)**
Y-axis acceleration data. The value is expressed in two's complement.
- 7.1.11 **OUT_Z_L_A (2Ch), OUT_Z_H_A (2Dh)**
Z-axis acceleration data. The value is expressed in two's complement.
- 7.1.12 **FIFO_CTRL_REG_A (2Eh)**

Data is available as indicated by the status register:

- 7.1.8 **STATUS_REG_A (27h)**

Table 34. STATUS_A register

ZYXOR	ZOR	YOR	XOR	ZYXDA	ZDA	YDA	XDA
-------	-----	-----	-----	-------	-----	-----	-----

The bit we're interested in is **ZYXDA**, which tells us that all three values are available. For simplicity, we'll typically work with a blocking loop that waits for this flag to come TRUE. However, this could result in the program hanging if something is wrong with the I²C bus or the accelerometer IC.

Since the I²C bus only handles eight-bit values, we'll need to read two bytes to get a complete value. We can either do that by using our existing `IIC0_Read()` routine twice, or we can make a new routine that reads the two bytes and combines them into a single sixteen-bit value. If you want to go that route, here's one version of a working routine.

```
int IIC0_ReadD16(unsigned char cAddr, unsigned char cReg)
{
    int iData;
    char cRead;

    while(IIC0_IBSR & 0b00100000); //wait for not busy flag
    IIC0_IBCR |= 0b00110000; //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110; //place address on bus with Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cReg; //locate desired register
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR |= 0b00000100; //restart

    IIC0_IBDR = (cAddr | 0b00000001); //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR &= 0b11100111; //read byte with ACK
    cRead = IIC0_IBDR; //fake read
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag
    iData = IIC0_IBDR*256; //read first byte

    IIC0_IBCR |= 0b00001000; //read byte with NAK
    cRead = IIC0_IBDR; //fake read
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11011111; //stop
    IIC0_IBSR |= 0b00000010; //clear flag
    iData += IIC0_IBDR; //second byte

    return iData;
}
```

The data you read back will be formatted at 16-bit 2's complement, but it's actually 12-bit 2's complement left justified. In other words, the bits you're interested in are in the upper three nibbles. A quick way to fix this is to divide by 16, which will do an arithmetic shift left by 4, keeping track of the sign of the number.

3-Axis Magnetometer and Temperature Sensor

(Optional topic) Working with the earth's magnetic field is a surprisingly complex problem, as it is three-dimensional, and quite weak compared to the magnetic fields produced by electrical currents and magnets in equipment and the residual magnetism in metal used in a building or its furnishings. If you need to make a proper compass, you will find it not to be a trivial exercise. In this course, we will simply look at what you need to do to get the values from the magnetometer – it's up to you as to what you want to do with them!

The magnetometer is a separate device inside the LSM303DLHC, at a different I²C address and with different configurations. For example, there's no "Big-Endian/Little-Endian" issue: the registers are just MSbyte-LSbyte, in that order.

The magnetometer (and the temperature sensor) address is 0x1E, or 0b0011110r.

There are three control registers, all of which need our attention.

7.2.1 CRA_REG_M (00h)

Table 70. CRA_REG_M register

TEMP_EN	0 ⁽¹⁾	0 ⁽¹⁾	DO2	DO1	DO0	0 ⁽¹⁾	0 ⁽¹⁾
---------	------------------	------------------	-----	-----	-----	------------------	------------------

1. This bit must be set to '0' for correct operation of the device.

Table 71. CRA_REG_M description

TEMP_EN	Temperature sensor enable. 0: temperature sensor disabled (default), 1: temperature sensor enabled
DO[2:0]	Data output rate bits. These bits set the rate at which data is written to all three data output registers (refer to Table 72). Default value: 100

Table 72. Data rate configurations

DO2	DO1	DO0	Minimum data output rate (Hz)
0	0	0	0.75
0	0	1	1.5
0	1	0	3.0
0	1	1	7.5
1	0	0	15
1	0	1	30
1	1	0	75
1	1	1	220

A value of 0b00011000 in CRA sets the device up for a 75 Hz refresh rate, with no temperature sensor. 0b10011000 enables the temperature sensor, if you want it.

7.2.2 CRB_REG_M (01h)

Table 73. CRB_REG_M register

GN2	GN1	GN0	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾
-----	-----	-----	------------------	------------------	------------------	------------------	------------------

1. This bit must be set to '0' for correct operation of the device.

Table 74. CRB_REG_M description

GN[2:0]	Gain configuration bits. The gain configuration is common for all channels (refer to Table 75)
---------	--

Table 75. Gain setting

GN2	GN1	GN0	Sensor input field range [Gauss]	Gain X, Y, and Z [LSB/Gauss]	Gain Z [LSB/Gauss]	Output range
0	0	1	±1.3	1100	980	0xF800–0x07FF (-2048 to +2047)
0	1	0	±1.9	855	760	
0	1	1	±2.5	670	600	
1	0	0	±4.0	450	400	
1	0	1	±4.7	400	355	
1	1	0	±5.6	330	295	
1	1	1	±8.1	230	205	

The CRB register sets up the sensitivity of the magnetometer. Although the middle column claims to set the gains equally for all three channels, the next column over indicates that the Z channel has a different sensitivity. We'll go with the assumption that the Z column was put in the datasheet for a purpose, so the middle column must only apply to X and Y.

The units are unusual: LSB/gauss. This is the inverse of the step size, so the bigger the number, the more sensitive the device, as shown in the fourth column. Another thing that's unusual is the use of "gauss", a holdover from an old measurement system based on centimetres/grams/seconds (CGS) rather than the SI system's metre/kilogram/seconds (MKS) standard. In the MKS system, the tesla is used, and is 10,000 times bigger than a gauss.

For greatest sensitivity, we'll use a value of 0b00100000 for CRB. This makes the step-size for the X and Y channels 0.909 mG/step, or 90.9 nT/step. The Z channel sensitivity is 1.02 mG/step, or 102 nT/step.

7.2.3 MR_REG_M (02h)

Table 76. MR_REG_M register

0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	0 ⁽¹⁾	MD1	MD0
------------------	------------------	------------------	------------------	------------------	------------------	-----	-----

1. This bit must be set to '0' for correct operation of the device.

The MR register defaults to 0b00000011, which puts the magnetometers into sleep mode. Values of 00 are needed in the MD1 and MD0 bits to put the device into "Continuous conversion" mode.

Once all of that is set up, the data can be read when the LSB of the status register goes HIGH:

7.2.7 SR_REG_M (09h)

Table 79. SR_REG_M register

--	--	--	--	--	--	LOCK	DRDY
----	----	----	----	----	----	------	------

The data is available as shown below:

7.2.4 OUT_X_H_M (03), OUT_X_L_M (04h)

X-axis magnetic field data. The value is expressed as two's complement.

7.2.5 OUT_Z_H_M (05), OUT_Z_L_M (06h)

Z-axis magnetic field data. The value is expressed as two's complement.

7.2.6 OUT_Y_H_M (07), OUT_Y_L_M (08h)

Y-axis magnetic field data. The value is expressed as two's complement.

This time, the data is right-justified! This means there's no need to shift the data to the right – it arrives as a proper 2's complement signed number, with the upper four bits stuffed appropriately.

If you want to use the temperature sensor and you've enabled it earlier, its values are available as shown below:

7.2.9 TEMP_OUT_H_M (31h), TEMP_OUT_L_M (32h)

Table 84. TEMP_OUT_H_M register

TEMP11	TEMP10	TEMP9	TEMP8	TEMP7	TEMP6	TEMP5	TEMP4
--------	--------	-------	-------	-------	-------	-------	-------

Table 85. TEMP_OUT_L_M register

TEMP3	TEMP2	TEMP1	TEMP0	-	-	-	-
-------	-------	-------	-------	---	---	---	---

Table 86. TEMP_OUT resolution

TEMP[11:0]	Temperature data (8 LSB/deg - 12-bit resolution). The value is expressed as two's complement.
------------	---

Note that this is left-justified, so you'll need to divide by 16 to move it into proper position. The value is 2's complement signed, and has a resolution of "8 LSB/deg", or a step size of 0.125 °C/step. That means that the three LSB's are fractional: 1/2, 1/4, and 1/8.

Device with 16-bit Internal Addresses (e.g. EEPROM) – Write and Read Functions
(*Optional topic*) In order to use the EEPROM on your board, you need 16-bit address versions of these two routines, shown below:

```

void IIC0_WriteA16(unsigned char cAddr, int iAddr, unsigned char cData)
{
    unsigned char cUpper = (unsigned char)(iAddr/256);
    unsigned char cLower = (unsigned char)(iAddr&0b0000000011111111);
    while((IIC0_IBSR & 0b00100000)); //wait for not busy flag
    IIC0_IBCR |= 0b00110000; //micro as master, start transmitting

    IIC0_IBDR = (cAddr & 0b11111110); //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cUpper; //upper unsigned char of address
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cLower; //lower unsigned char of address
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cData; //send data
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11001111; //stop transmitting, exit Master mode
    IIC0_IBSR |= 0b00000010; //clear flag
}

unsigned char IIC0_ReadA16(unsigned char cAddr, int iAddr)
{
    unsigned char cData;
    unsigned char cUpper = (unsigned char)(iAddr/256);
    unsigned char cLower = (unsigned char)(iAddr&0b0000000011111111);

    while((IIC0_IBSR & 0b00100000)); //wait for not busy flag
    IIC0_IBCR |= 0b00110000; //micro as master, start transmitting

    IIC0_IBDR = (cAddr & 0b11111110); //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cUpper; //upper unsigned char of address
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBDR = cLower; //lower unsigned char of address
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR |= 0b00000100; //restart

    IIC0_IBDR = (cAddr | 0b00000001); //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBSR |= 0b00000010; //clear flag

    IIC0_IBCR |= 0b00001000; //reading 1 unsigned char only
    IIC0_IBCR &= 0b11101111; //receive unsigned char
    cData = IIC0_IBDR; //not actually -- starts the process
    while(!(IIC0_IBSR & 0b00000010)); //wait for flag
    IIC0_IBCR &= 0b11011111; //stop
    IIC0_IBSR |= 0b00000010; //clear flag
    cData = IIC0_IBDR; //for real this time

    return cData;
}

```

If you want to transfer an entire array of bytes between the micro and the EEPROM, you would want to make versions of these routines that can write or read a bunch of bytes sequentially. The EEPROM is designed to operate in a special paging mode, in which a full “page” of 128 bytes is read from or written to the device. This requires starting at a page-delineated address, and also involves pointers to string arrays for the microcontroller. The information as to how to manage “Page Write” or “Page Read” transfers is available in the data sheet for the 24AA512 EEPROM. This topic goes beyond the scope of this course, but the following gives you a starting point. Multiple reads and writes involve issuing an I²C ACK signal between each byte transferred, with a NAK signal at the end; unlike normal single byte transfers, which only issue NAK signals. With the 24AA512, since it operates more slowly than the I²C bus, we need to wait until it’s ready for the next byte. The only way to handle this is to send the byte and see if it’s acknowledged; if it isn’t, we send the

byte again and keep doing so until it is acknowledged. With all of this hand-shaking, the possibility of hanging up the program waiting for a flag, an ACK or a NAK looms large. It's best to write software that will only wait so long, then returns an error code to indicate that the system has failed. As previously mentioned, Simon Walker has written an extensive set of I²C library components that handle multiple reads and writes, along with page reads and writes, all of which have escape routes in case of failure. If you find yourself using I²C devices on a regular basis, you should talk him about how to use his library components.

I²C Reliability Measures

As indicated in the previous discussion, you probably discovered, with the simple routines created for your library, that the I²C bus sometimes goes insane (mostly when you're troubleshooting, as it's pretty dependable in normal operation), and your program will hang up waiting for a flag, often in IIC0_IBSR. A partial solution to this, which you would see implemented in Simon Walker's library, is to put a counter into the `while(!(IIC0_IBSR&0b00000010))` loop so that after a certain number of tries, say 5000 or so, you exit the loop and return an error code. A typical error code is 0b11111111 (i.e. 0xFF), which, as a signed number, is -1, and as a Boolean value, is ~0. You may also want to come up with more sophisticated "try-catch" routines that allow your program to continue operating when the I²C bus goes down, including prompting an operator to cycle the power on the board, if necessary.

Speaking of Simon Walker, he's come up with library components that allow for a greater layer of abstraction while, at the same time, allowing for the implementation of lower-level I²C commands and management of error conditions. You may be provided with instruction related to this approach to the I²C bus.

Parting Words

You have now touched on some of the capabilities of a very powerful microcontroller and a selection of associated peripherals that were built into your microcontroller kit. You've learned, with varying levels of proficiency, how to use a fairly wide range of peripherals, both internal to the microcontroller, and external, connected through a number of different interfaces. In addition, you've learned how to program the device in its native Assembly Language and in C. You know enough about electricity and electronics to be dangerous. With a bit of ingenuity, you could do some serious design work. Go forth and build things!