Topic 1 – Embedded Systems Theory and the 9S12 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
- 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).

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 micro-controller 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 9S12, 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 9S12 Assembly Language, 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 9S12 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 MC9S12E128 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

Learning Exercise 1. Locate each of the following on the block diagram:

1. Microprocessor (CPU)
2. Read Only Memory (ROM)
3. Random Access Memory (RAM)
4. Six Serial Communication Interfaces (SCI)
5. Timer Module
6. Pulse Modulator
7. Two I^{2}C Bus Controllers (IIC)

Learning Exercise 2. Almost all of the interconnects to the modules can be redirected to the microprocessor as General Purpose Input/Output (GPIO) Ports by means of a Port Integration Module (PIM). Locate eleven 8-bit GPIO ports, one 7-bit GPIO port, and one 24-bit GPIO port.

Learning Exercise 3. Notice the "Notes" on the side. The development board we're using has a 112-pin microcontroller mounted on it. Find eight GPIOs that we can't use.

Learning Exercise 4. Almost all pins on the GPIO ports can be programmed to act as inputs and outputs. Locate two GPIO pins that can only be used as inputs.

Learning Exercise 5. Most pins can be programmed as either inputs or outputs, so they are controlled by Data Direction Registers (DDR). Locate these for each GPIO port.

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 MC9S12, as you have seen, most of the interface pins can be programmed independently to act as GPIO.

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 where more memory is required than what is available inside the microcontroller, which won’t be a problem for us in this course. In fact, most microcontrollers do most of their bus-work internally, taking away the complexity of design and programming. The 9S12 has an internal bus to interface 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 Interface (SCI) – 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), separate transmission lines are used for transmitting and receiving. In a system like USB 2.0 (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

The ports and control registers you identified earlier are 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}, 0278_{16} and 0279_{16}; the 24-bit Data Direction register for this port is at addresses 0273_{16}, $027A_{16}$ and $027B_{16}$. Another register required for the operation of this port is the 16-bit Port AD Input Enable Register, found at addresses $02CD_{16}$, $008C_{16}$, 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 start programming. 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.

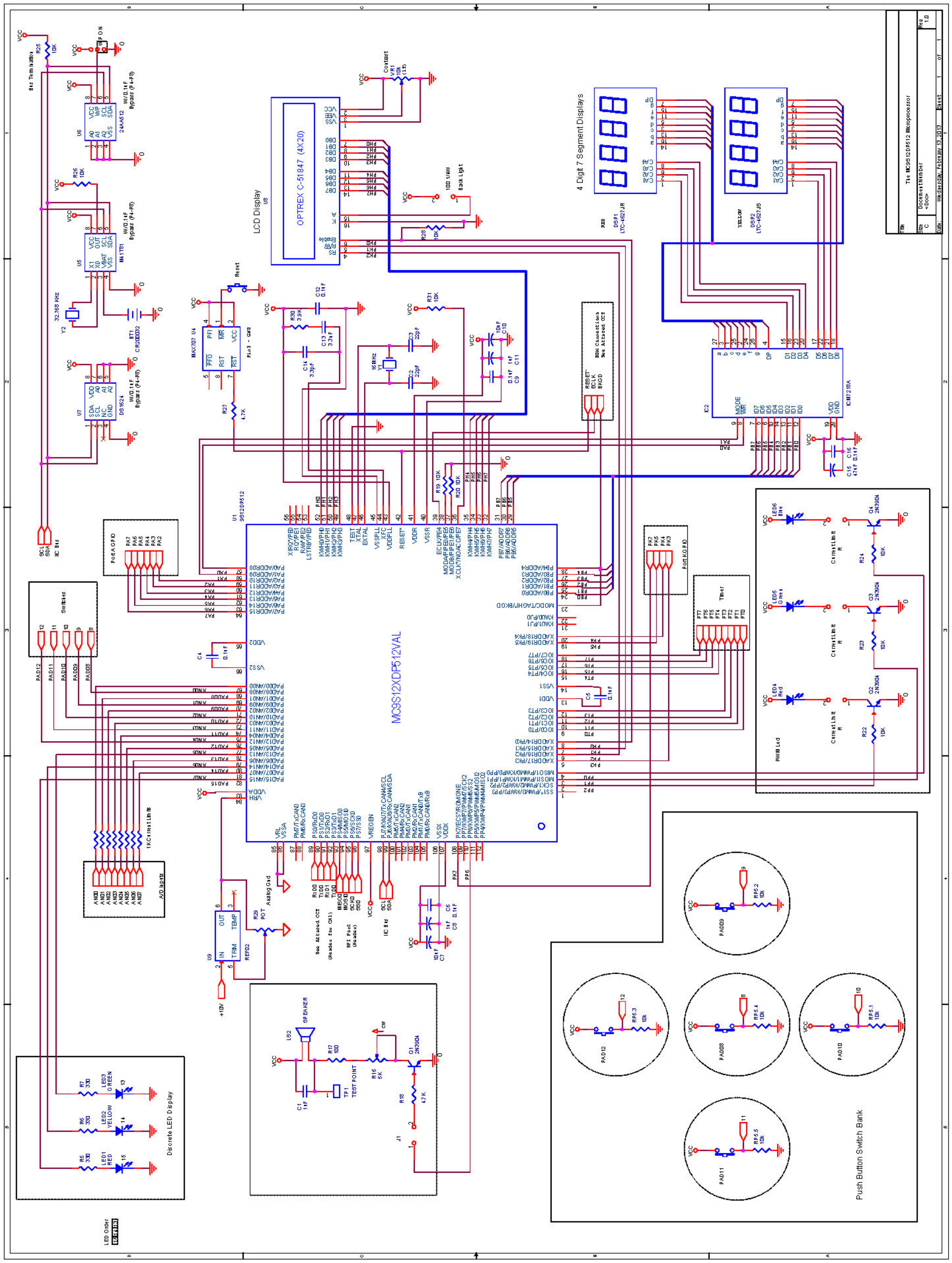
Here's a little table that summarizes what we've been saying about the 16-bit port, PAD. Notice that the addresses start with "$" – from now on, we'll use this notation to indicate numbers in hexadecimal instead of using $nnnn_{16}$.

Port (Block Diagram)	Port Name (mc9s12xdp512.inc)	Addresses	Function
AD			24-bit GPIO port
AD0	PT1AD0	$0271	Lowest 8 bits
AD1	PT01AD1	$0278	Upper 16 bits
	PT0AD1	$0278	Highest 8 bits (n/a)
	PT1AD1	$0279	Middle 8 bits
DDRAD0	DDR1AD0	$0273	Lowest 8 Data Direction
DDRAD1	DDR01AD1	$027A	Upper 16 Data Direction
	DDR0AD1	$027A	Highest 8 DDR (n/a)
	DDR1AD1	$027B	Middle 8 DDR
(not shown)	ATD0DIEN	$02CD	Lowest 8 Input Enable
	ATD1DIEN	$008C	Upper 16 Input Enable
	ATD1DIEN0	$008C	Highest 8 IE (n/a)
	ATD1DIEN1	$008D	Middle 8 IE

The CNT MC9S12XDP512 I/O Board

Why this fascination with Port AD? Take a look at the schematic for the CNT MC9S12XDP512 I/O Board, shown on the following page.

Learning Exercise 6. Identify all the things you see connected to Port AD. Which parts of Port AD are these connected to?


The MC9S12DP512 Microprocessor
LCD Display
OPTREX C-51847 (4X20)
4 Digit 7 Segment Displays
MC9S12XDP512VAL
ICM7218A
MAX707 U4
24AA512
DS1624
Switches
Timer
Port AGPIO
Port KGPIO
Discrete LED Display
Push Button Switch Bank
A/D Inputs
Wednesday, February 13, 2013

Let's put all of this together, while simultaneously learning a bit about CodeWarrior, the programming environment we'll be using to develop Assembly Language code for the 9S12.

Learning Exercise 7. Do the following at a computer terminal running CodeWarrior.

1. Connect the 12 V_{DC} supply to your board, but leave the power switch OFF.
2. Connect the BDM Pod to the BDM header on your board. Your instructor will show you the proper orientation of the cable.
3. Use the USB "A to B" cable to connect the BDM Pod to your computer.
4. Turn on the power switch on your board. The LEDs on the BDM Pod should show some activity, indicating that it is communicating with your board and with your computer.
5. Locate and run Freescale CodeWarrior. You will be asked to do a number of things, which you will learn more about later. For now, do the following when prompted:
    a. Click "Create New Project".
    b. In the first window, select the MC9S12XDP512, if it hasn't been automatically selected. (Select carefully! Every character counts!)
    c. You should still be in the first window. Select "TBDML" under "Connections" before you click "Next>".
    d. In the next window, select "Single Core" if it hasn't already been selected.
    e. In the next window, uncheck "C" and check "Absolute assembly".
    f. Name your project something informative (e.g. LEDsAndSwitches.mcp) and pick a useful location (For now, just use the desktop – you can browse by clicking the "Set…" button). Click "Finish>".
    g. In the top left corner of the Integrated Development Environment (IDE) that appears, you'll find a navigation window. If you look in the "includes", you'll see the *mc9s12xdp512.inc* file we've been talking about.
    h. You'll also find a "*main.asm*" file. This is where we will eventually enter some Assembler code, but for now we want to delete everything between the line that starts with "CLI" and the box with "Interrupt Vectors" in it.
    i. Press the green arrow with an insect beside it ("Run/Debug"). Many things will happen, as the compiler and linker try to create a program from the nothingness on your screen. Don't worry; you don't need any code yet.
    j. You will probably get a screen called "USBDM Configuration – HCS12". As long as it shows something other than "No devices found", you should be ok. You don't want to see this window every time you try to run a project, so before you move on, check "Don't show this dialogue in future".
    k. You will probably get a "Loader Warning". Check the "Do not display …" box so you don't keep getting this message for this project in the future, and press "OK".
    l. Locate the "Memory" window in the "True-Time Simulator & Real-Time Debugger" window (hereinafter called the Debugger). This is where you will be working for this exercise.

6. Turning on the three LEDs and reading the five push-button switches connected to Port AD:
   a. Consult the table of addresses to find out the address of the part of Port AD that's connected to the switches and LEDs.

   b. In the Memory window, locate and try to edit this address by double-clicking it and typing in a suitable value, followed by "Enter". (You can find locations in memory by right-clicking in the Memory window and selecting "Address…", at which point you can enter the address you're looking for. The 'L stands for Logical memory –you don't have to type it in.
   c. Describe what happens when you try to edit this address both in the Memory window and on the board (don't get your hopes too high yet).

   d. Hold down one of the switches, and refresh the memory display of the address you're playing with by right-clicking it and selecting "Refresh". Do you see any change? (Be prepared for disappointment.)

   e. It's time to get to know the Data Direction Register for this port. It controls, bit by bit, whether the port pins will be inputs (0) or outputs (1). Consult the table of addresses, then locate and examine the appropriate DDR. What binary data does it contain?

   f. Consult the schematic for the board, and determine what hex value needs to be written into this DDR in order to make the LED lines into outputs and the switch lines into inputs.

   g. In the Memory window, locate and edit this DDR. (You should finally feel like you've accomplished something, however trivial it may seem at this point.)
   h. Now edit the appropriate part of Port A to a value that should turn on just the Red LED. Record what you see on the board and in the Memory window.

   i. Try other values for to manipulate the states of the three LEDs.

   j. The reset state for the input pins on PORT AD is "High Impedance", meaning there is an open switch between each pin and the controller. Setting the bits in the appropriate IEN register (IEN = Input Enable) connects the pin to the input port. Consult the table of addresses to find the right IEN register.

   k. Consult the schematic to determine what hex value needs to be written into this IEN register to enable the switch inputs, and edit it appropriately.

   l. Hold down one of the switches, then locate and refresh PT1AD1. Record what you see. Does it make sense?

   m. Try different switches/combinations of switches, with different LEDs on as well. How impressive is that?!?

In “real life”, we’ll be writing programs to do what you’ve just done manually. In order to write these programs, you’ll need to know a bit more about the microprocessor in the 9S12 microcontroller, and some fundamental commands in the Assembly Language for this micro.

CPU12 Assembly Language Programming

In other courses, you’ve written programs in “high level languages”, like C#. These languages are written so as to relate fairly closely to the way in which people think, and involve software compilers that produce programs in “machine language”, the lowest possible level of programming. As an exploration into the operation of our microcontroller, let’s write a “machine language” program to operate the LEDs on your board.

Learning Exercise 8. Do the following at a computer loaded with CodeWarrior.

1. In the “Memory” window of the Debugger, go to memory address $2000 (more on that later). You can get there by right-clicking in the white-space in the right side of the Memory window and entering “2000” in the dialog box for “Address …”.
2. Carefully type the following hex codes into the addresses from $2000 to $2018. (Pressing the “space bar” moves to the next location.)
```
CF 40 00 10 EF 79 02 79
1C 02 7B E0 CD 00 00 87
7A 02 79 04 36 FD 42 20
F7
```
3. In the “Register” window of the Debugger, you’ll find something called “PC”. This is the program counter, which the microprocessor uses to keep track of where it is in the program. Change the value of PC to “2000”.
4. Along the top bar, you’ll find a green arrow. Hover over this for a moment, and it will tell you that it’s the “Start/Continue” button, and can be accessed using the F5 hotkey. Use this to start your program. What do you see on the board?
5. Clearly, the microprocessor knows what you typed, but at this point you probably have no clue as to why this meant anything at all. The Debugger has a disassembler that might help us. Locate the “Assembly” window, and locate address $2000.
6. To help with our analysis, let’s make numbers in the Assembly window appear in hexadecimal: Right-click in the Assembly window and change the “Format” to “Hex”.
7. Compare the contents of the Assembly and Memory windows. You should be able to determine what “englishy words” (actually, mnemonics) represent some of the hex values you typed in. Reverse-engineer the following into assembly mnemonics:
   a. $CF
   b. $10
   c. $79
   d. $1C
   e. $CD
   f. $87
   g. $7A
   h. $04 $36 (this one takes two command bytes – an op code and a post byte)
   i. $42
   j. $20
8. Locate and write down any addresses you recognize from what we were doing before.

9. Now, in the main.asm file, enter the following code between the CLI and the Interrupt Vectors section:

```
        CLR     PT1AD1              ;initialize -- LEDs will be off
        BSET    DDR1AD1,%11100000   ;make LED indicator pins outputs
        LDY     #0                  ;initialize timer loop
        CLRA
Loop:   STAA    PT1AD1              ;send current value to LEDs
        DBNE    Y,*
        INCA
        BRA     Loop
```

10. Once you have entered the code, press the green arrow with the bug, or hit “F5” to assemble and download what you have written. In the Debugger, run your program to see if it does the same thing as what you entered manually.

11. In the “Memory” window, locate $4000 and check what appears there against what you entered previously. The Assembler has converted your assembly language code into machine code for the microcontroller to run.

12. Why is your assembled code at $4000, rather than at $2000, as it was when you entered it manually?

13. Turn off your board and disconnect the BDM pod. Wait a few seconds, then turn the board back on again. If nothing happens, hit the Reset button (under the LCD display). What do you see?

14. Power down, reconnect the BDM pod, and power back up. Check locations $2000 and $4000. What do you discover, and why?

It has probably occurred to you that you need to know more about Assembly Language in order to program your 9S12. It may not be as obvious to you that you also need to know more about the structure of the microprocessor itself. For example, you now know that $7A is represented by “STAA”. This is a short form for “Store Accumulator A”, which means only a little bit more to you now than it did a moment ago: What’s Accumulator A? Most of the pertinent information you’ll need for programming is in the “CPU12 Reference Guide”, which is for the CPU12 microprocessor that’s at the core of our MC9S12XDP512 microcontroller. You should download and print a paper copy of this document. Here’s the link to access it: http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU12RG.pdf

Look at the front page of the CPU12 Reference Guide: it shows you the arrangement of registers in the CPU12 core. The CPU12 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 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 much 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.

Learning Exercise 9. Do the following at a computer loaded with CodeWarrior.

1. If you're not still in the project you set up earlier, you can "Load Previous Project" and select it from the list. "Run/Debug" the project to access the Debugger.
2. Type "ABCD" into the D accumulator, and hit Enter or click somewhere else in the Register window. What appears in the A and B accumulators?
3. Type "10" in the A accumulator. What happens to the other accumulators?
4. Type "34" in the B accumulator. What happens to the other accumulators?

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've already been introduced to the ***Program Counter Register***. During the running of a program, this register is constantly updated to keep the microprocessor moving through the program.

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 CCR:

- 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). However, if the result is more than one bit too big (like trying to display $32C in an 8-bit register), the Overflow flag will be set to indicate that there's no way to determine what the correct result is.
- 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, in the code you typed into your micro earlier, we had a "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 command, as the micro restores the original flags before it completes this (and other) compound (two-part) instructions.

When we start looking at Assembler coding, we’ll notice that each possible operation uniquely controls the bits in the CCR, and we will need to pay attention to the results.

We’re almost able to start programming in CPU12 Assembly Language now. One more thing we need to know is the arrangement of memory 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 9S12 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 directly 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” (a 1348-page document!) provides the necessary 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 ports that won’t be connected to any hardware and internal modules that don’t need to be addressed.

There’s a huge document called the “Data Sheet” for the MC9S12XDP512 device 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. There’s a link in Moodle, and here it is, as well:
http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

Locate the Memory Map for the S12X CPU (page 39 in the current version).

The “Global Memory Map” 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.

Unless you have good reason to do so, you're probably wise to avoid the windows and work with the unpaged regions in the "Local Memory Map", which can be accessed without any concern using the 16-bit address bus.

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, you will put your code in the fixed FLASH block at $4000, and will place working variables in RAM ($2000). You may choose to put constant data in FLASH at $C000, although this isn't necessary. 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.

Finally, we're at the point where we should know enough to begin programming in CPU12 Assembly Language.

When programming in CPU12 Assembly Language there are two fundamental types of commands: Assembler Directives and Instructions.

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. ***Instructions***, on the other hand, are translated into machine language for the microprocessor to carry out.

Learning Exercise 10. Do the following at a computer loaded with CodeWarrior.

1. If you're not still in the project you set up earlier, you can "Load Previous Project" and select it from the list.
2. In the "main.asm" code window, you will see mostly Assembler Directives at this point. Locate each of the following:
   a. INCLUDE
   b. EQU
   c. ORG
   d. DS.W
   e. DC.W
3. There are also a couple of Instructions. Locate each of the following:
   a. LDS
   b. CLI

Assembler Directives

- The "INCLUDE" directive tells the assembler/compiler/linker to include files you've added to the project when it creates machine code for the microprocessor.
- "EQU" assigns a "label" to a particular address location. (You can think of this as a variable name, although labels are more general than that.) For example, any reference to "ROMStart" in the code that follows will be interpreted as address $4000, which is the beginning of the FLASH block in the MC9S12XDP512's memory.
- "ORG" stands for "origin", and tells the assembler where to create the code that follows. In this case, the assembled code will start at $4000 (think that through). Variables will be assigned starting at some place called "RAMStart", which happens to be defined in *mc9s12xdp512.inc*, which in turn is included indirectly through *derivative.inc*.
- "DS.W 1" means "define storage for one 16-bit word". This is typically combined with assigning a label so this location can be used as a variable. It makes sense to use DS in RAM only, so that values can be written into these storage spaces.
- "DC.W" means "define a constant 16-bit word". It makes sense to make constants in ROM only; otherwise, whatever is supposed to be "constant" will disappear if the power is lost.

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

A summary of the CPU12 Assembly Language Instruction Set can be found in the CPU12RG/D Reference Guide, 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 proceed 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.

<u>Learning Exercise 11.</u> Refer to the CPU12RG/D Reference Guide for the following.

1. Locate the "LDAA" instruction. This stands for "Load Accumulator A".
2. How many different ways can this instruction can be used? (Think of these as "overloads" from your C# terminology.)
3. The first and simplest of these uses the "IMM" addressing mode. This means that the A 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. From this little blurb, explain what "#opr8i" tells you, and why the Machine Language version says "86 ii".
4. One other version of this command that you will use extensively is the "EXT" mode. Explain what "opr16a" means, and what "B6 hh ll" tells you.
5. The "Access Detail" column tells you how many bus clock cycles this command takes, and what happens for each clock cycle. 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. How long would it take this micro to carry out the "IDX" version of the LDAA instruction?
6. The last two columns tell us what to expect in the Condition Code Register. Which flags do you expect to see some action on for the LDAA instruction?

Learning Exercise 12. Do the following at a computer loaded with CodeWarrior.

1. If you're not still in the project you set up earlier, you can "Load Previous Project" and select it from the list. Locate the "main.asm" window.
2. Enter the following code after the "CLI" line. (There are lots of editor settings for the IDE in CodeWarrior – you might want to play with the tab settings; I usually turn off the part about "Tab inserts spaces", for example.) The stuff after the semicolons below is not, hopefully, something you are a stranger to!

```
            BSET  DDR1AD1,%11100000       ;make LED 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
```

3. Run/Debug the program, and make any corrections required to make it work.
4. Using the CPU12RG/D Reference Guide and what you know by now, go through this code line by line to gain an understanding of what each instruction is doing. Take your time with this! The sooner you start to think like a machine in this course, the better!
5. Now, explain each of the following:
    a. The commands "BSET" and "BCLR"
    b. The use of "%" and "$"
    c. The use of a mask (e.g. %11100000 or %10000000)
    d. The use of "#" in the "LDY" instruction
    e. "*"
    f. How many times will the "DBNE" line be executed before the program continues?
    g. Approximately how long will it take for the program to get past the "DBNE" line?
    h. What does "BRA" do?

6. We’re going to try some rudimentary debugging skills, which will definitely come in handy when you start writing your own code.
    a. Stop/Reset the program using the black arrow in a red circle (hover over the buttons with the mouse to see what they do).
    b. Beside the green arrow, you’ll find a “Single Step” button. Which Function Key could you use instead?
    c. While looking at the “Source” and “Assembly” windows, Single-Step through the first few instructions in your program. (You might want to set the Assembly window to hexadecimal to help you understand what you’re seeing.)
    d. When you get to the “DBNE” line, as predicted, you seem to be trapped. Watch what happens to “Y” in the “Register” window as you step through.
    e. Change “Y” to 2, and watch what happens with the next couple of steps.
7. When you get to the next “DBNE” line, let’s try a different method of breaking out. It’s called “running to a breakpoint”. In the “Source” window, right click on the line following the one that’s holding you up, and select “Set Breakpoint”. Now, hit F5 or the green arrow to run from the current program counter location. Describe what happens.

8. You have now programmed your microcontroller to run as a stand-alone device. To prove this, turn off the board, disconnect the BDM Pod, and turn the board back on. You’ve burned your program into EEPROM on board, and, until you reprogram it, it will continue to run the same instructions faithfully.