



Introduction to Embedded Systems

CMPE2200

AN INSTITUTE OF TECHNOLOGY COMMITTED TO STUDENT SUCCESS

NAIT CoursePack CP1503 Revision R10

All Rights Reserved

This publication © The Northern Alberta Institute of Technology (2017). 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
BCDTоЗex.....	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.....	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
The VT100/VT52 Terminal.....	100
Escape Sequences	100
SCI0_TxString.....	101
Interrupts.....	105
Interrupts in S12XCPU Assembly Language	105
Interrupts using ANSI C	109
Input-Driven Interrupt	111

Accurate Timing.....	112
Timer Initialization.....	114
Setting the Timer Compare Event Duration	116
Delays <i>vs.</i> Intervals.....	117
Delay Function for Misc_Lib	117
Interrupt-Driven Timer	118
Real-Time Loop.....	121
Input Capture and Pulse Accumulation	123
Input Capture.....	123
Pulse Accumulation.....	125
A To D Conversion.....	127
Setting up V_{RH}	128
Configuring ATD0	129
Using ATD0	132
Pulse-Width Modulation.....	133
Generating Waveforms.....	134
True Pulse-Width Modulation	139
I ² C Bus.....	141
Basic I ² C Communication Using the 9S12X.....	143
LTC2633HZ12 I ² C DAC – 16-bit Data Writes	148
MPL3115A2: Standard 8-bit Reads and Writes	151
Floating-Point Math in ANSI C.....	154
<stdio.h>.....	155
<math.h>.....	156
M41T81 Real-Time Clock – Standard 8-bit Reads and Writes	157
Position Information with the LSM303DLHC – Standard 8-bit Reads and Writes.....	160
3-Axis Accelerometer.....	160
3-Axis Magnetometer and Temperature Sensor	165
Device with 16-bit Internal Addresses (e.g. EEPROM) – Write and Read Functions.....	168
I ² C Reliability Measures.....	169
Parting Words.....	169

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, 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”.

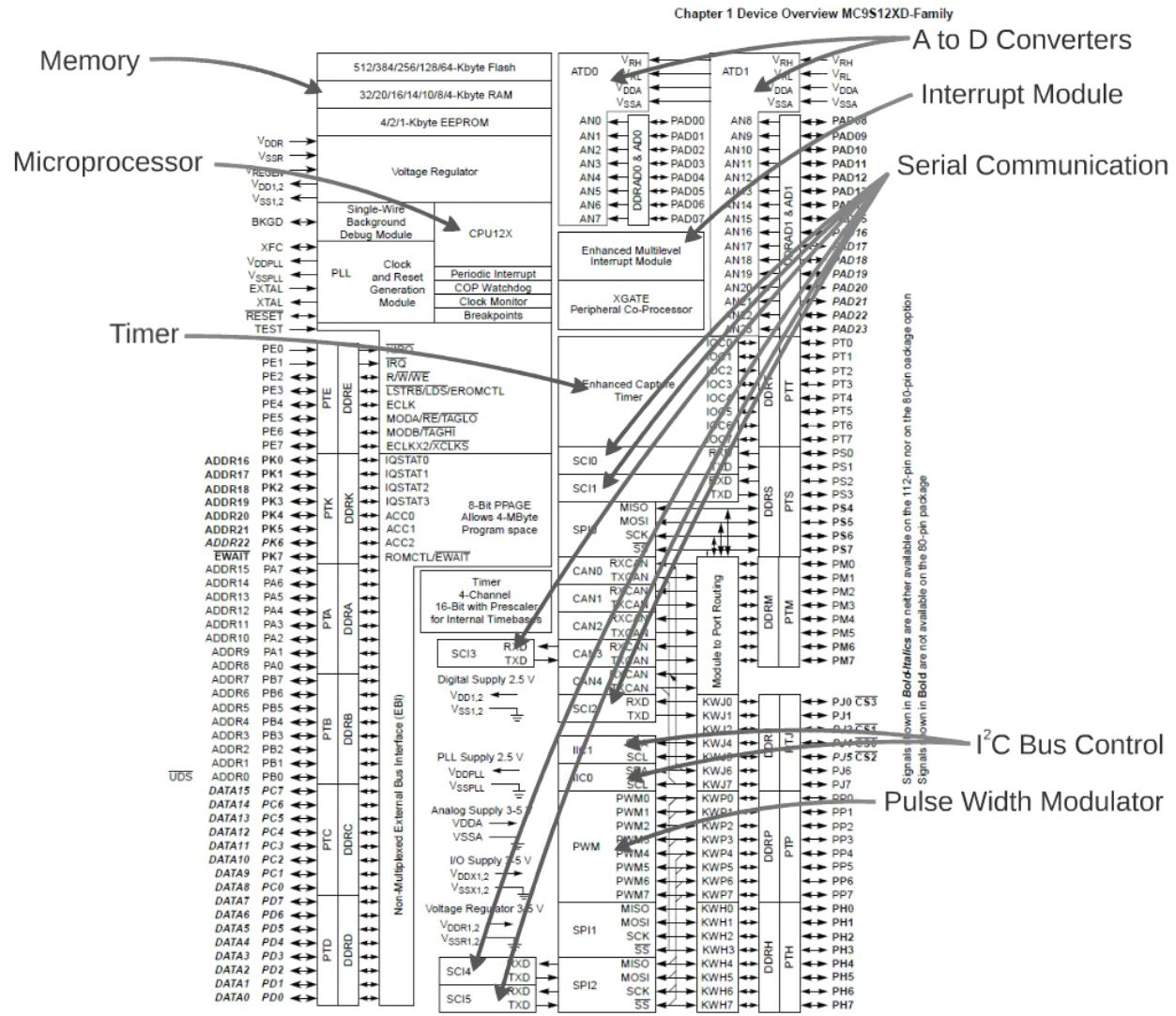


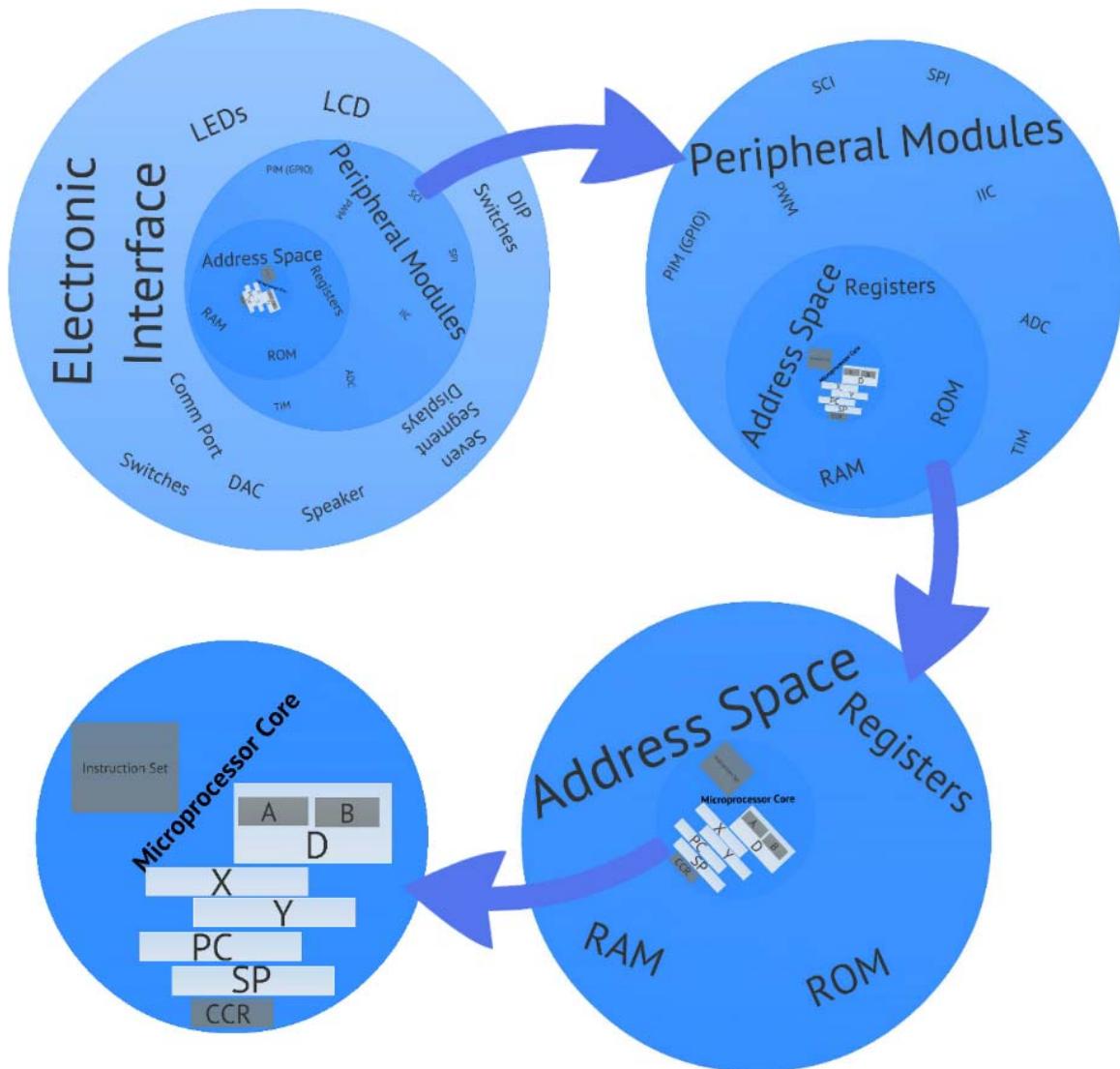
Figure 1-1. MC9S12XD-Family Block Diagram

MC9S12XDP512 Data Sheet, Rev. 2.21

Freescale Semiconductor

35

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 will get to know well in this 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. Notice that there’s something called “PIM” or “GPIO”. GPIO is “General Purpose Input/Output”, and refers to pins available on the IC that can be used 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. That’s where the “PIM” comes in: it’s the Port Integration Module. Using the PIM, we can choose to either connect the pins in a particular port to the internal peripheral associated with them, or we can choose to use those pins for 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 (I^2C) busses. Here's how our electronic interface is wired to the microcontroller in the kit designed for this course.

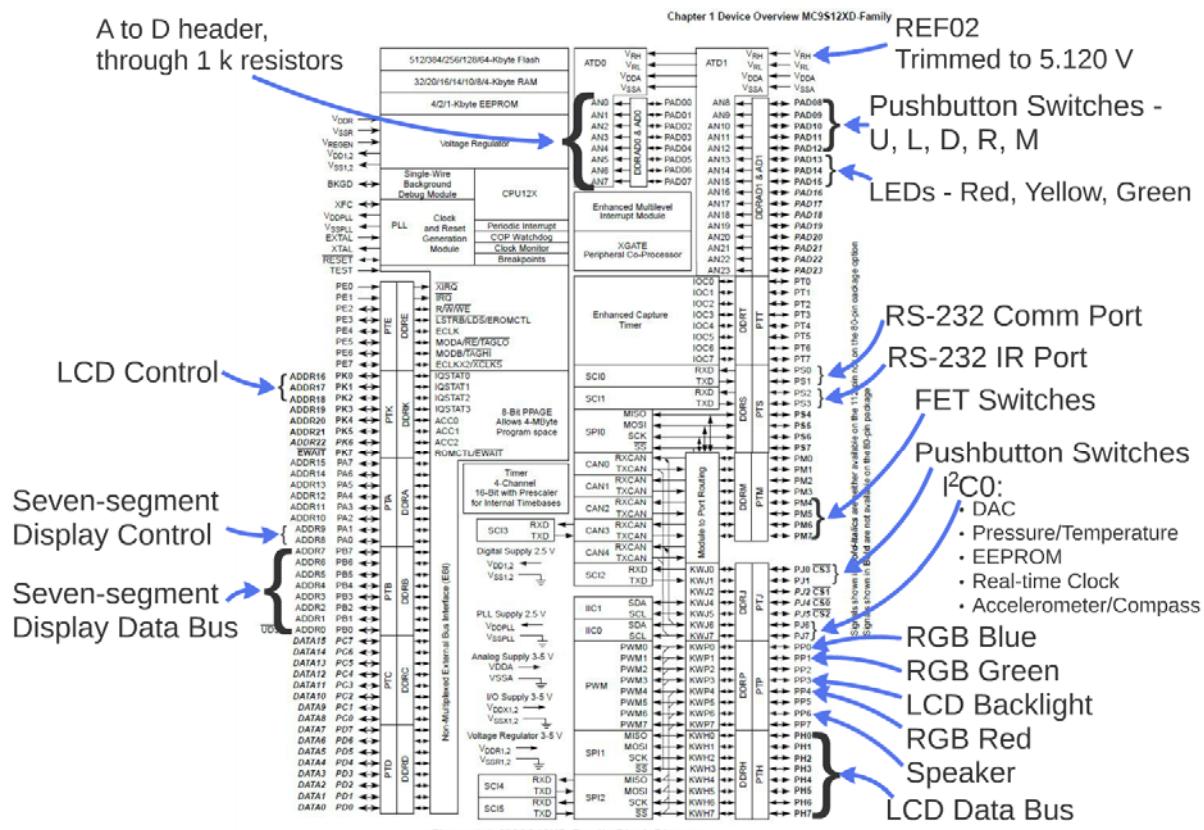
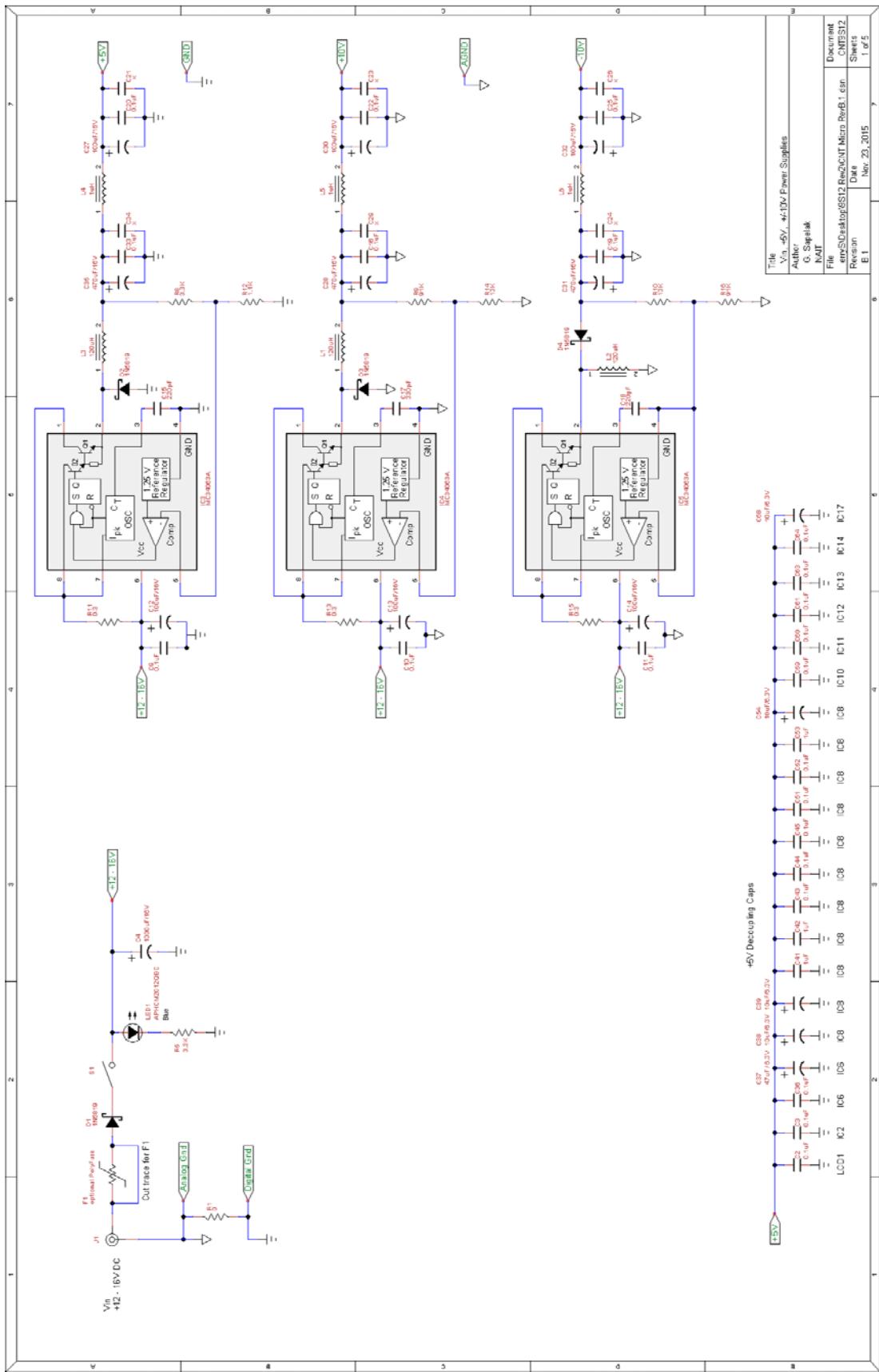
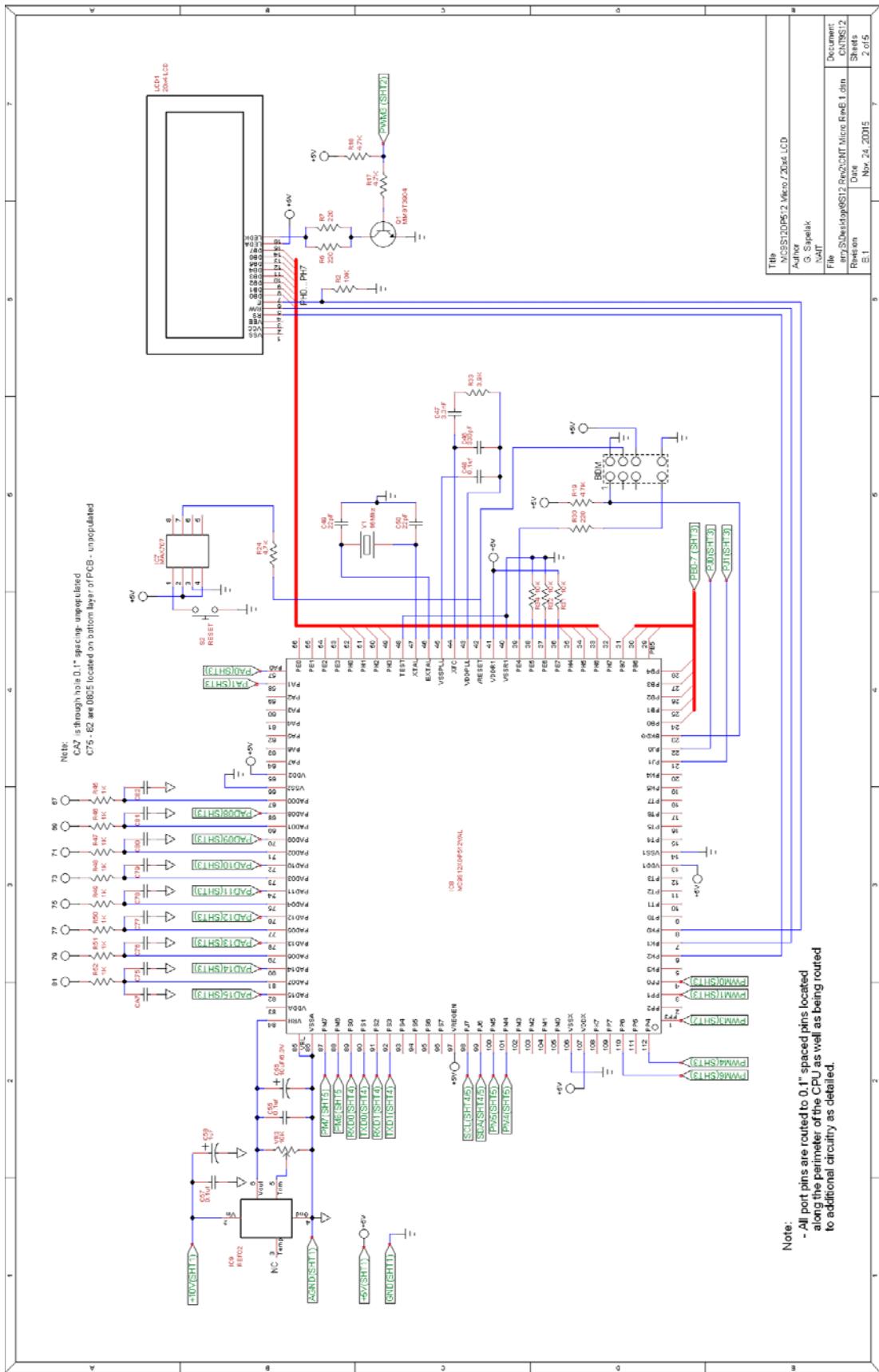


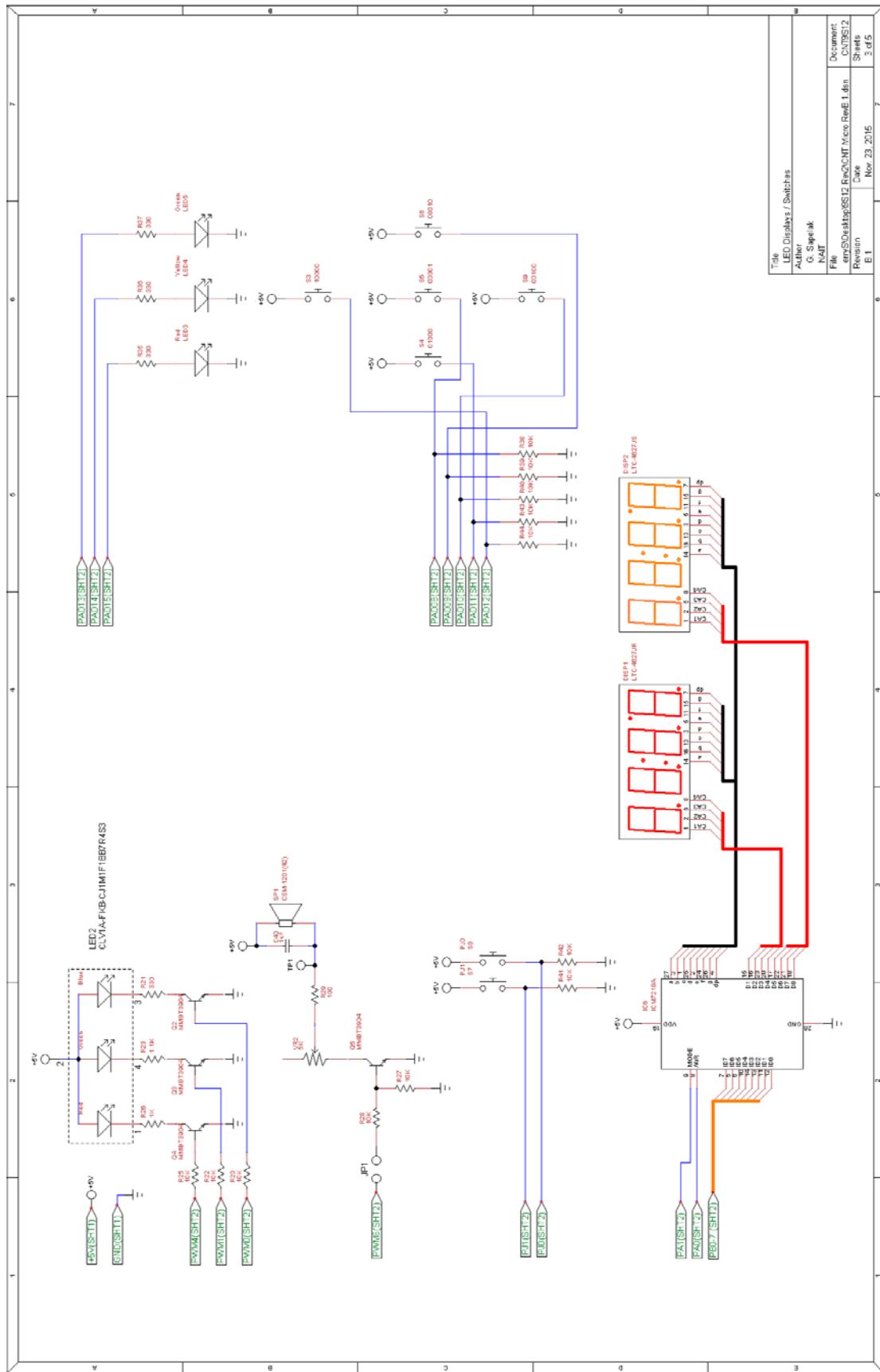
Figure 1-1. MC9S12XD-Family Block Diagram

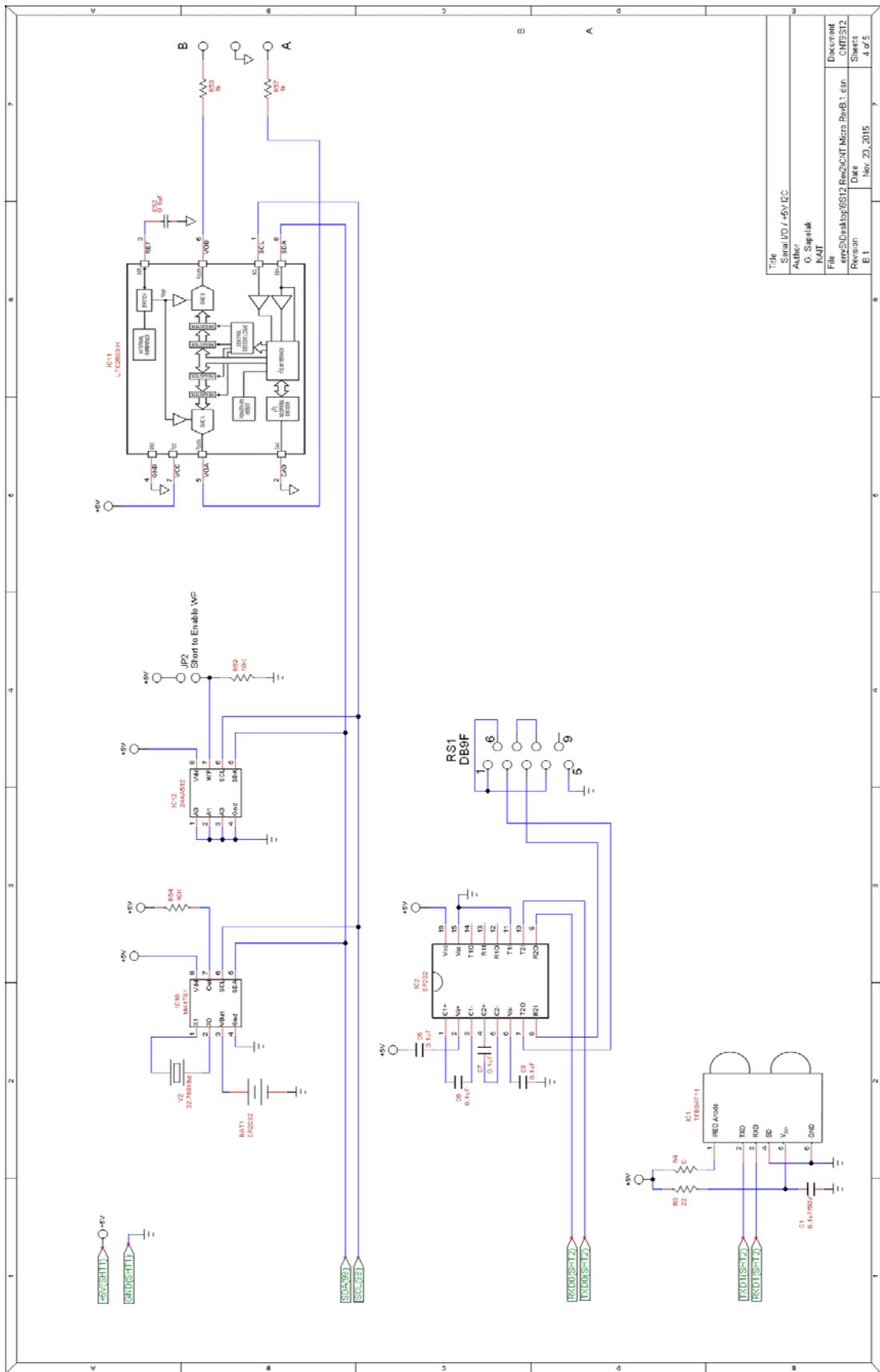
The CNT MC9S12XDP512 I/O Board

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.

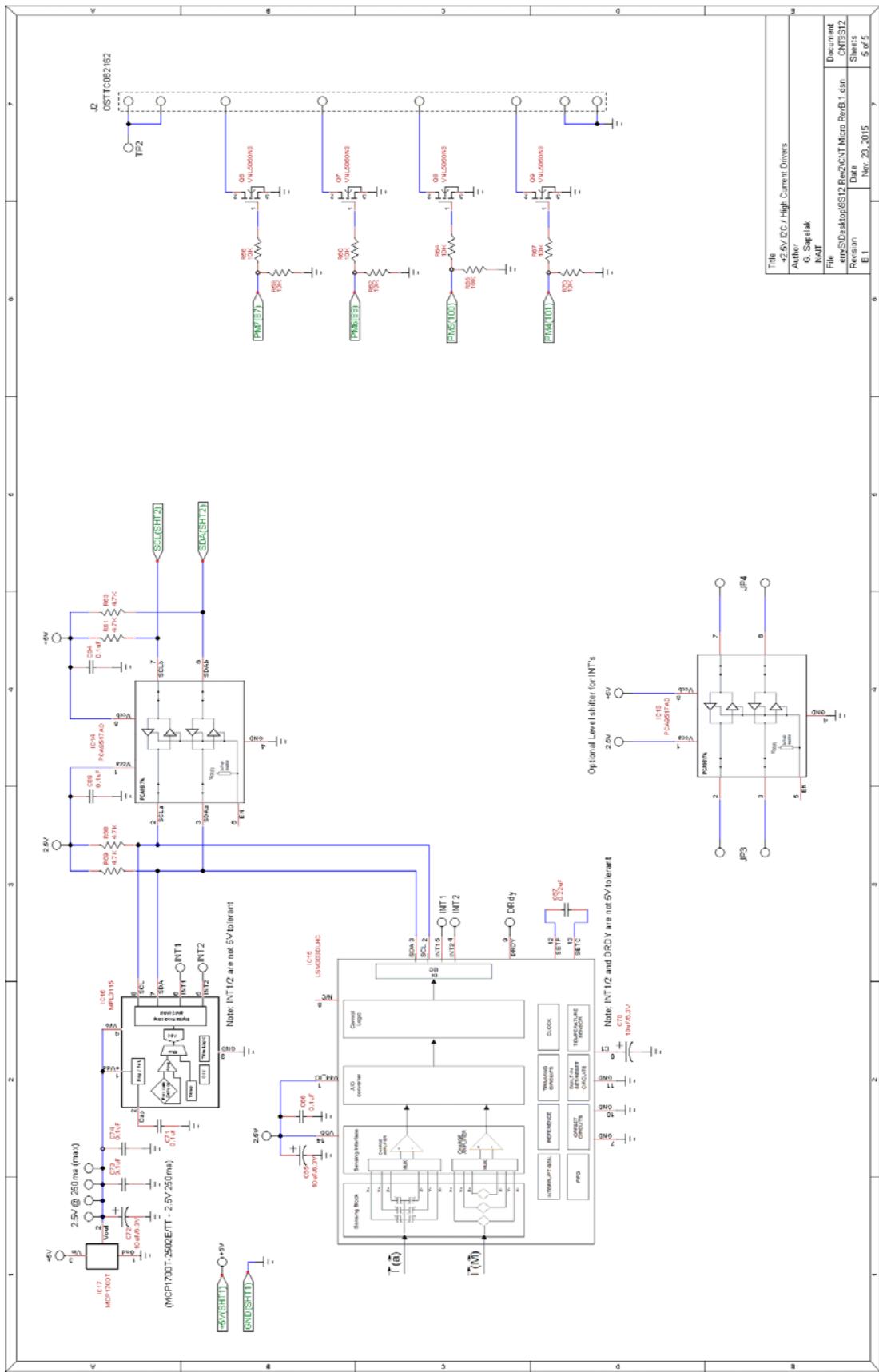








Title		Author		Date		Document	
Serial I/O / 8x16 I ² C		G. Sapkota		Nov 23, 2015		CMPS12	
File		Rev: Schematic of PIC16F877A Microcontroller		Rev: 1		Sheets	
Page		1		4 of 5		4	



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.

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

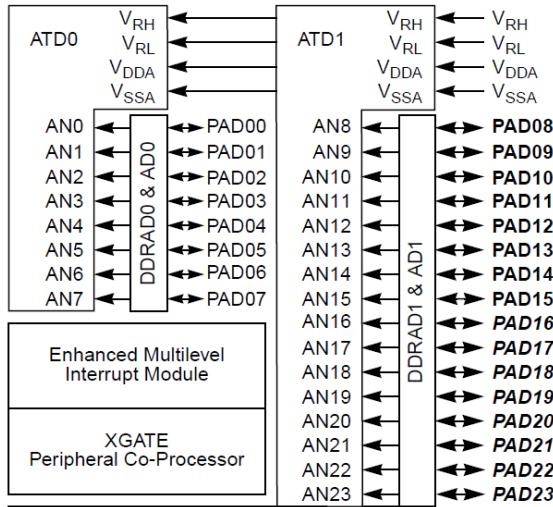
Bussed (Parallel) Interface – A parallel interface involves the simultaneous transfer of multiple bits of information on separate (parallel) copper traces. The microprocessor in a personal computer operates in bussed mode. This requires an address bus capable of locating each unique address in the address 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

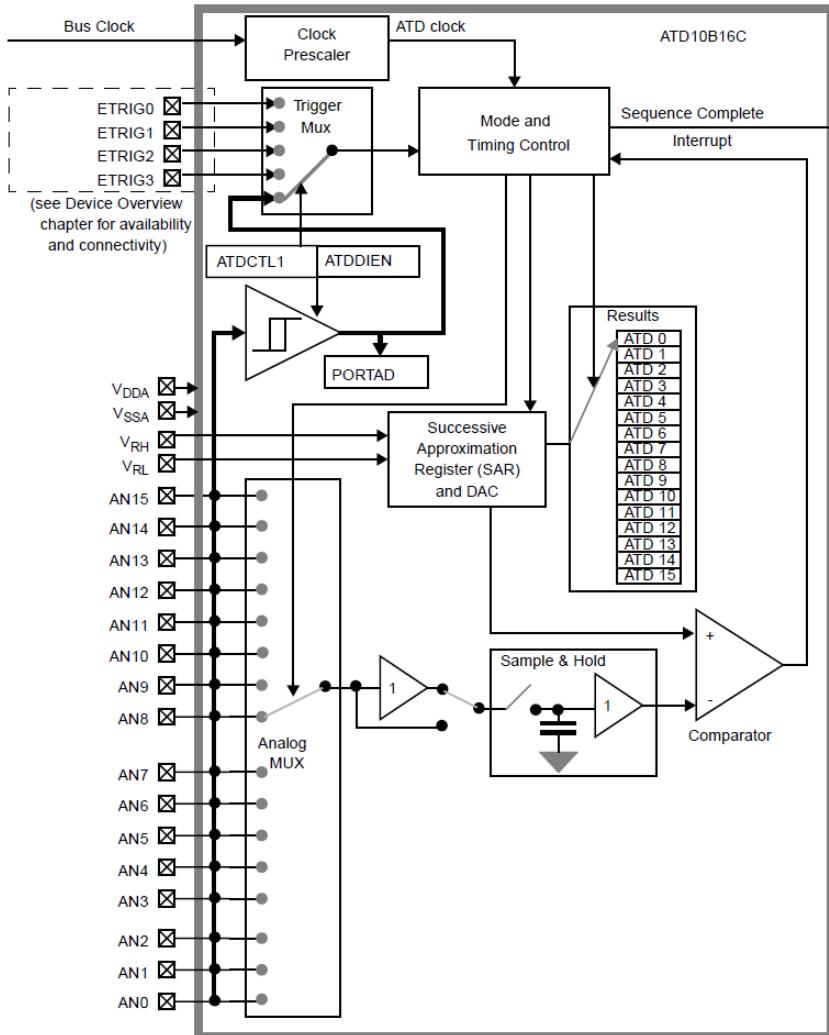


Figure 4-1. ATD10B16C Block Diagram

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 (<i>mc9s12xdp512.inc</i>)	Addresses	Function
AD0	PT1AD0	\$0271	Lowest 8 bits
AD1	PT01AD1	\$0278	Upper 16 bits
	<i>PT0AD1</i>	\$0278	<i>Highest 8 bits (n/a)</i>
	<i>PT1AD1</i>	\$0279	<i>Middle 8 bits</i>
DDRAD0	DDR1AD0	\$0273	Lowest 8 Data Direction
DDRAD1	DDR01AD1	\$027A	Upper 16 Data Direction
	<i>DDROAD1</i>	\$027A	<i>Highest 8 DDR (n/a)</i>
	<i>DDR1AD1</i>	\$027B	<i>Middle 8 DDR</i>
ATDDIEN	ATD0DIEN	\$02CD	Lowest 8 Input Enable
ATDDIEN	ATD1DIEN	\$008C	Upper 16 Input Enable
	<i>ATD1DIENO</i>	\$008C	<i>Highest 8 IE (n/a)</i>
	<i>ATD1DIEN1</i>	\$008D	<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.

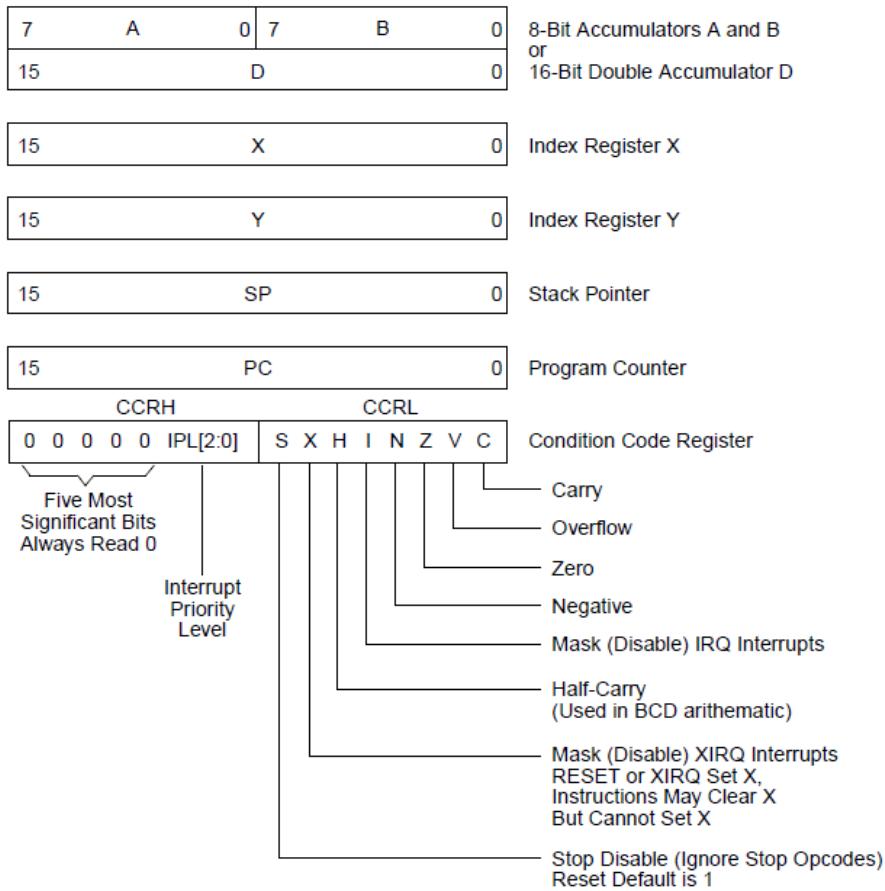


Figure A-1. Programming Model

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) \Rightarrow A Add Accumulators A and B	INH	18 06	00	00	--	-	Δ	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.

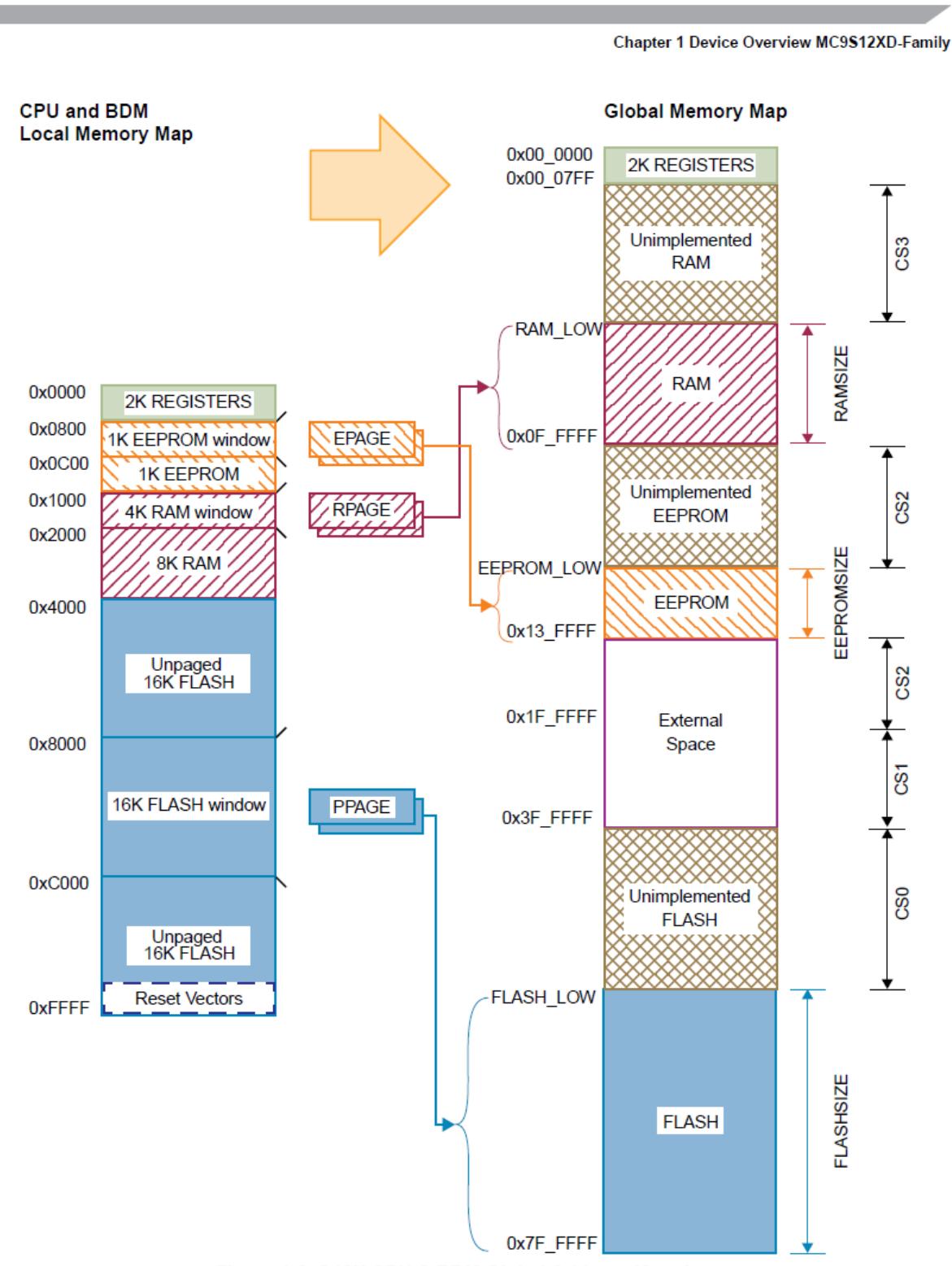


Figure 1-3. S12X CPU & BDM Global Address Mapping

Unless you have good reason to do so, you're probably wise to avoid the windowed areas of memory and work with the unpaged 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.