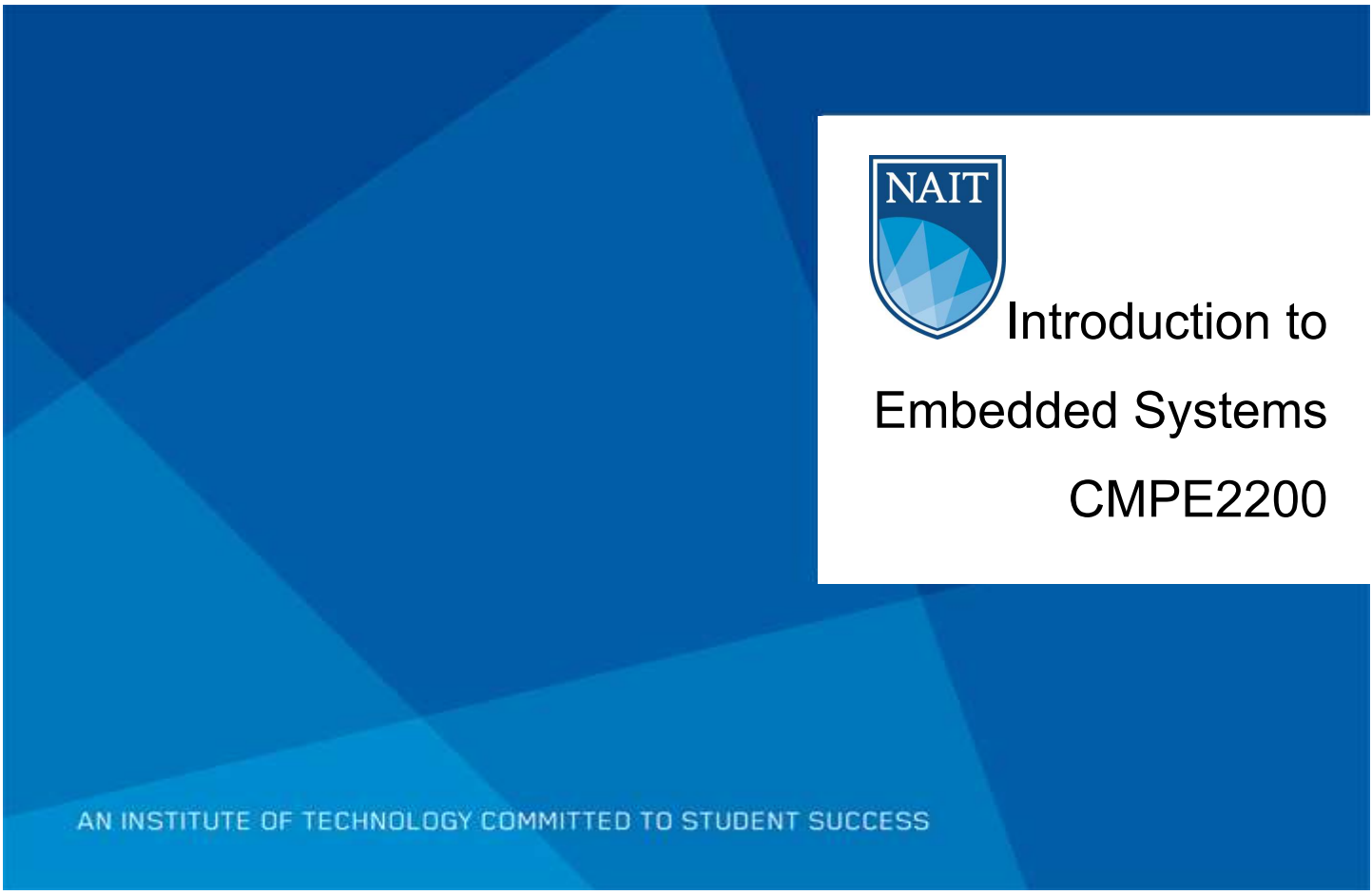



# NAIT CoursePack CP1503

Revision R12



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

## 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 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 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".

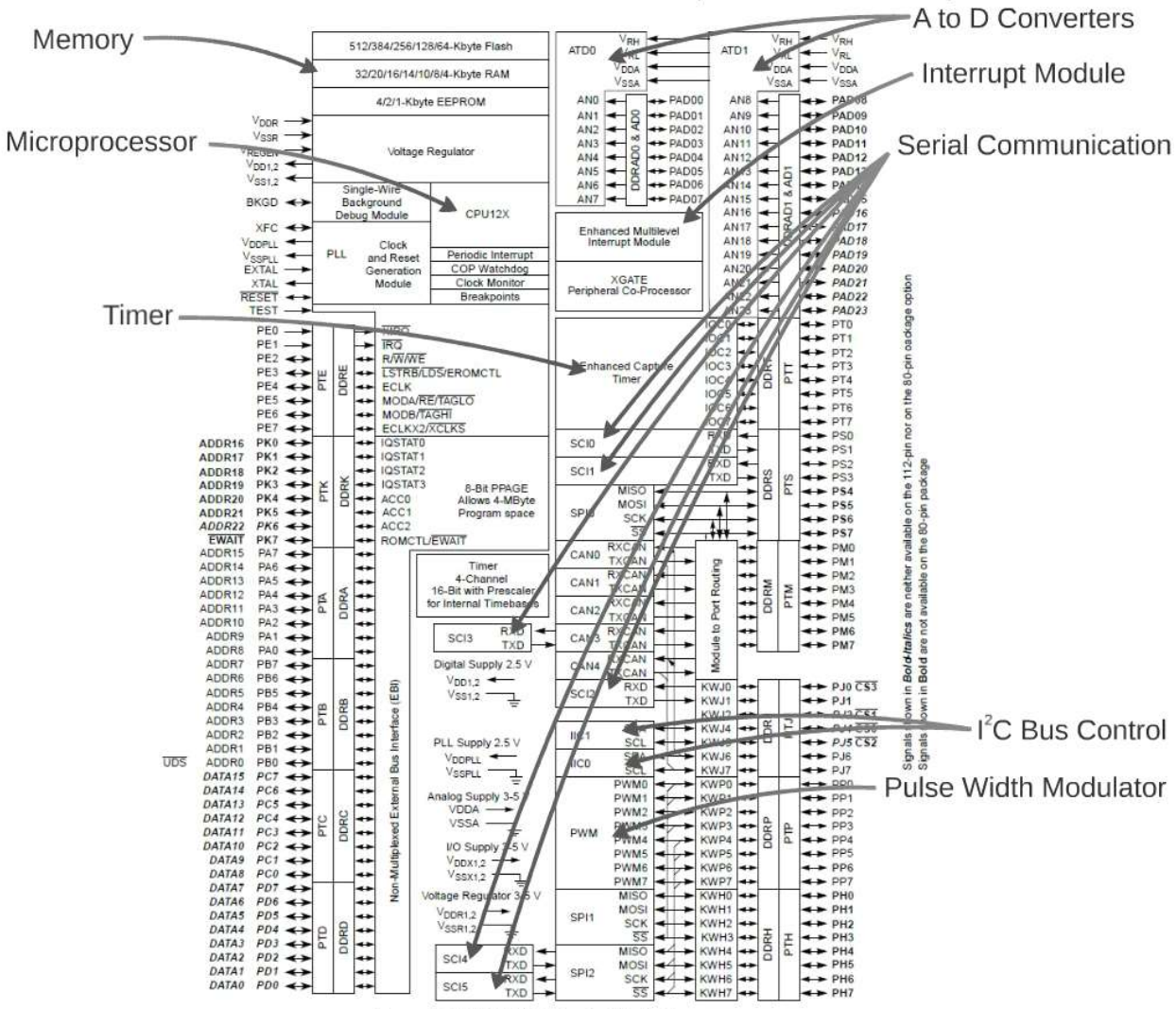
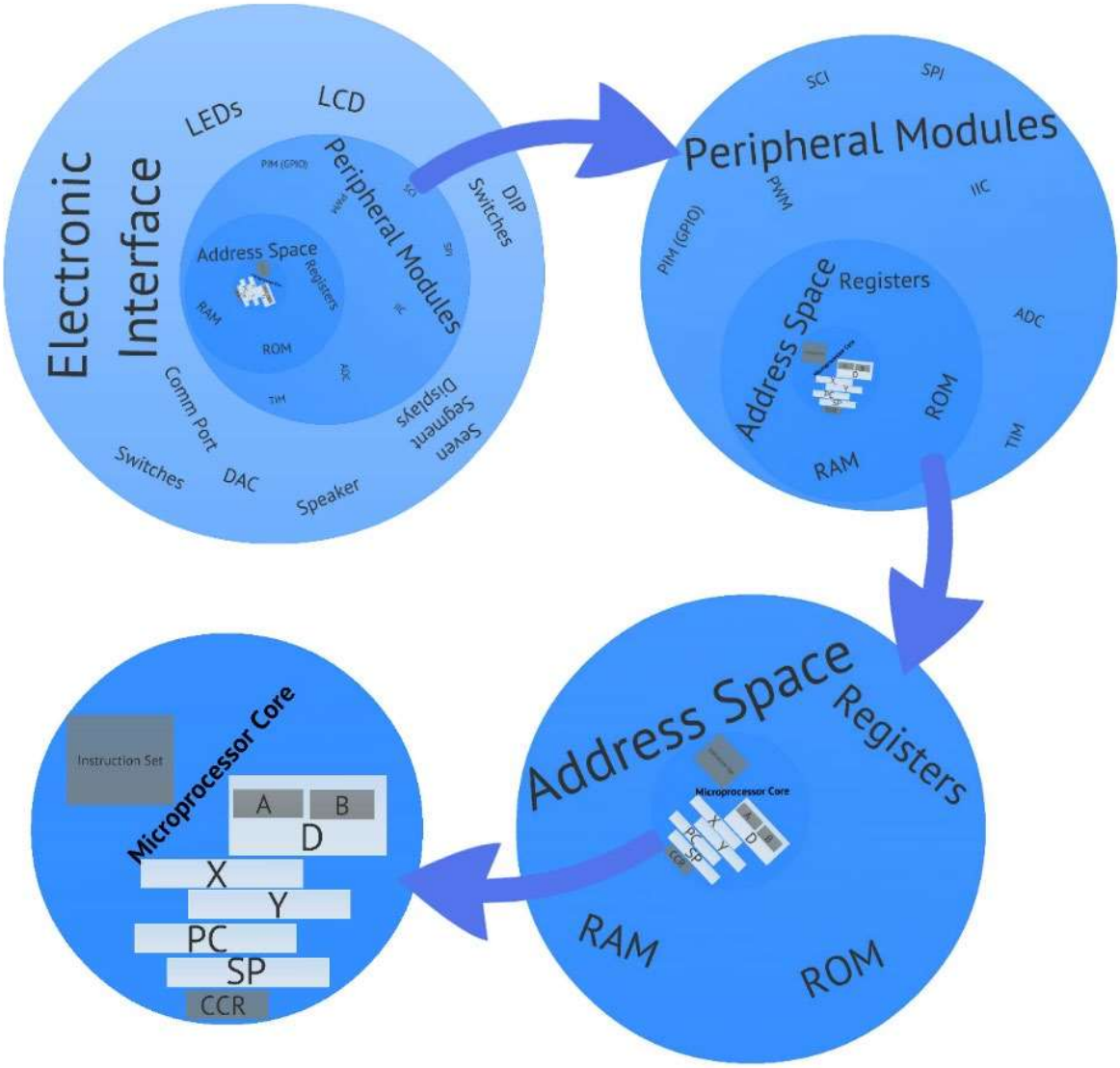



Figure 1-1. MC9S12XD-Family Block Diagram

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.



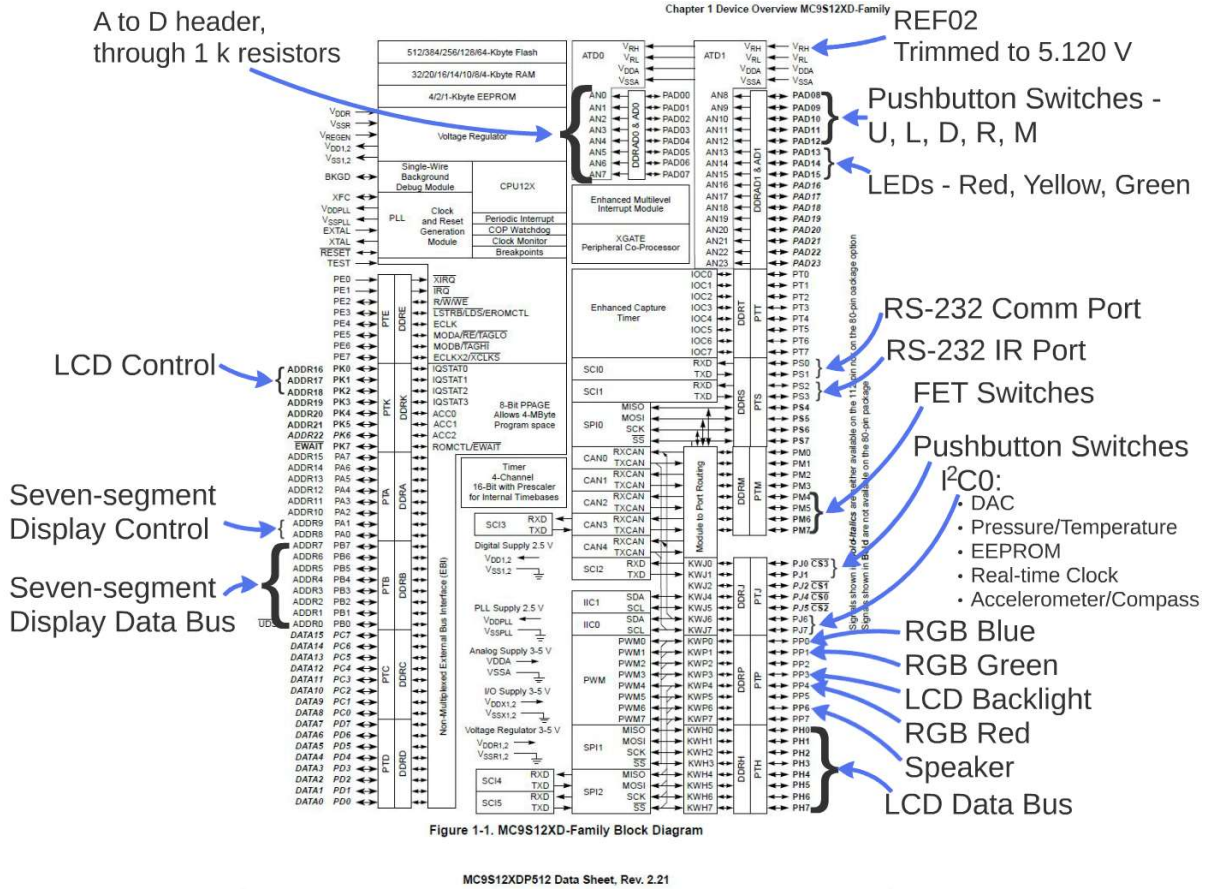

Let's work backwards through the diagram above.

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.

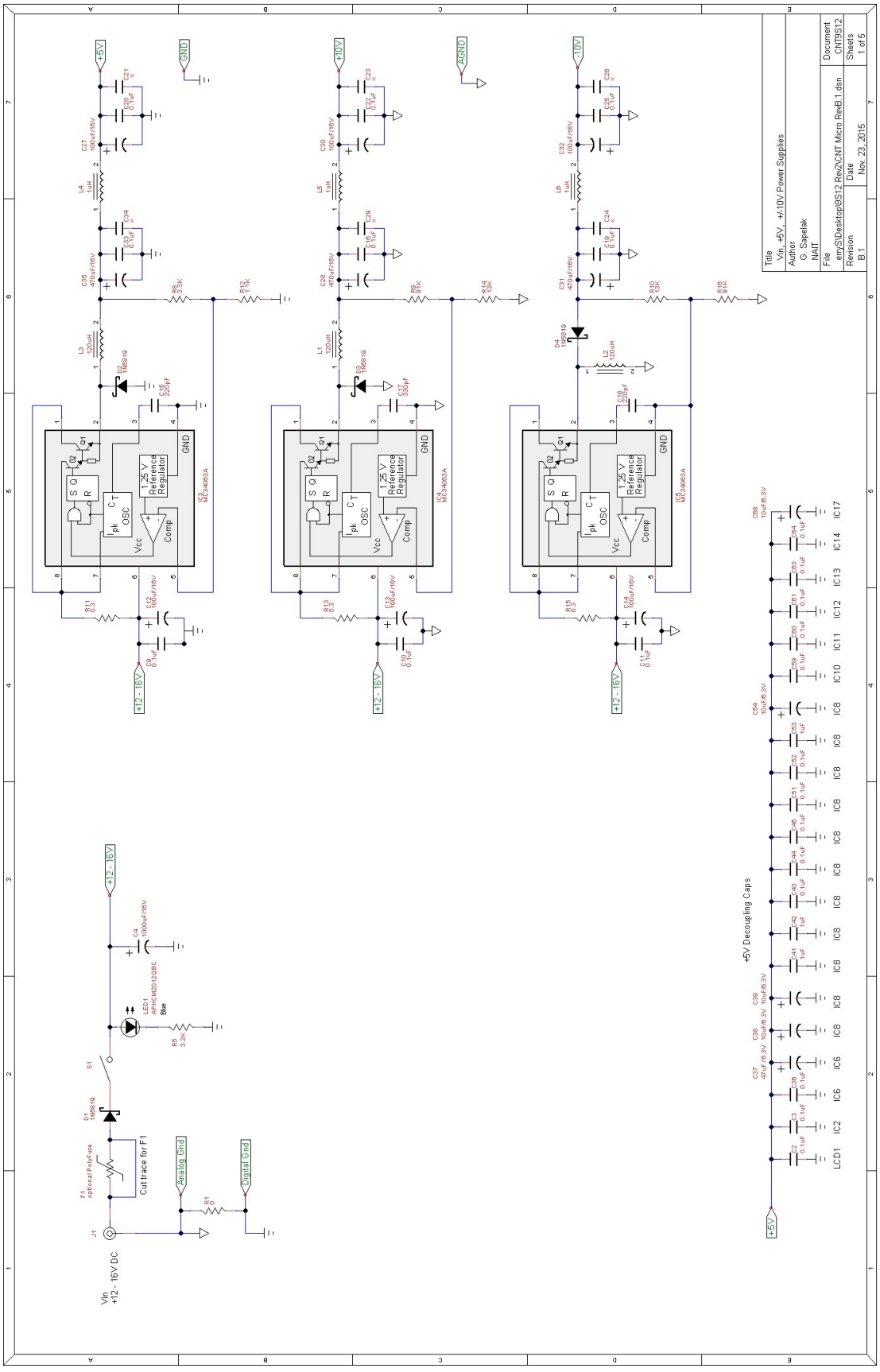
The ***peripheral modules*** are the interface with the "outside world". We've identified a number of these previously. Between the modules and the pins in the following diagram is the "PIM" – Port Integration Module – which allows us either to connect to the peripheral or to use the pins as "GPIO". 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.

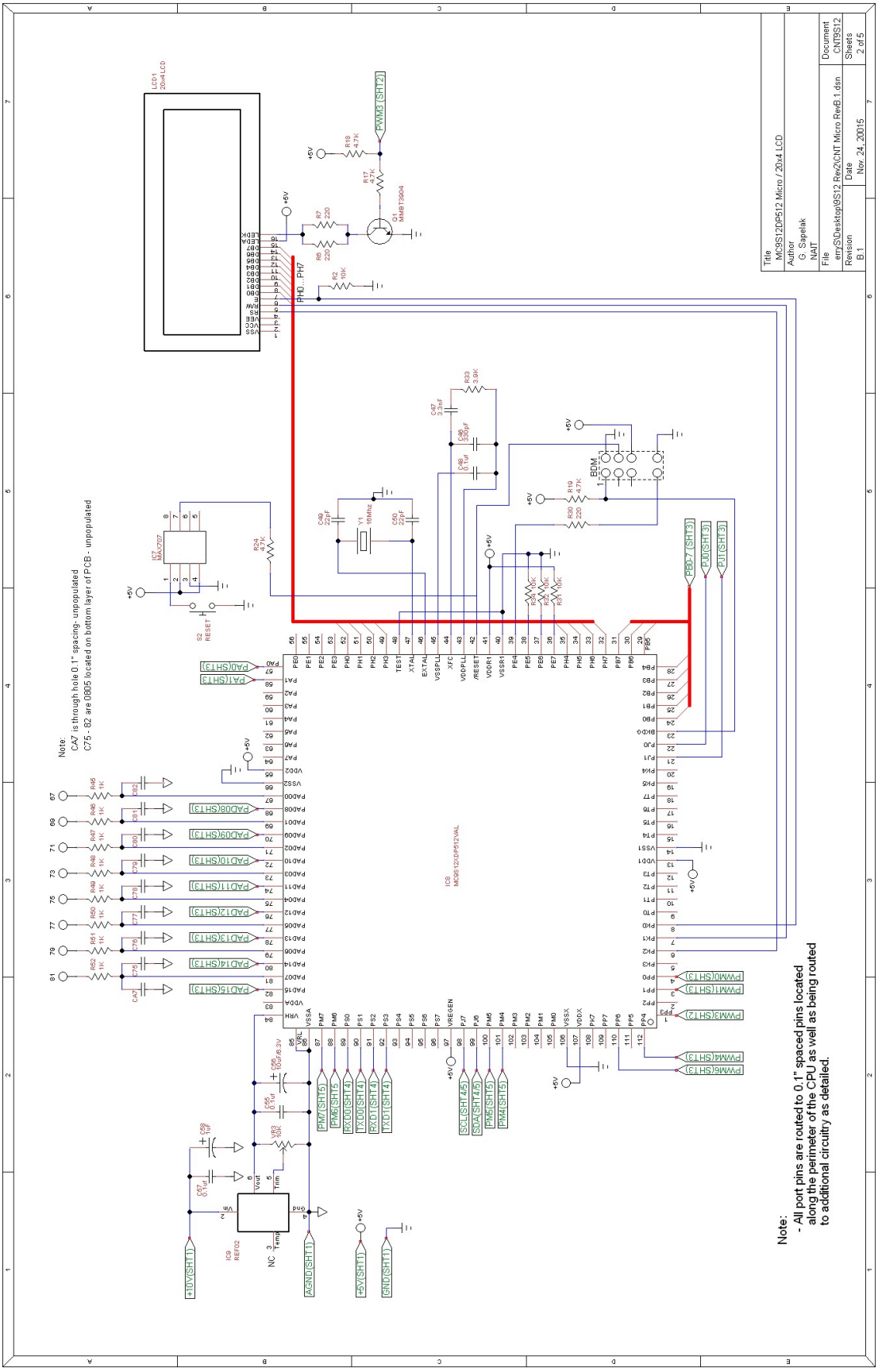
The biggest circle represents the ***electronic interface***, which contains all the components on the printed circuit board. Most of the external peripherals on the printed circuit board are accessed using GPIO, although the speaker is intended for operation using the pulse-width modulator module (PWM), 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 ($I^2C$) busses. Here's how our electronic interface is wired to the microcontroller in the kit designed for this course.

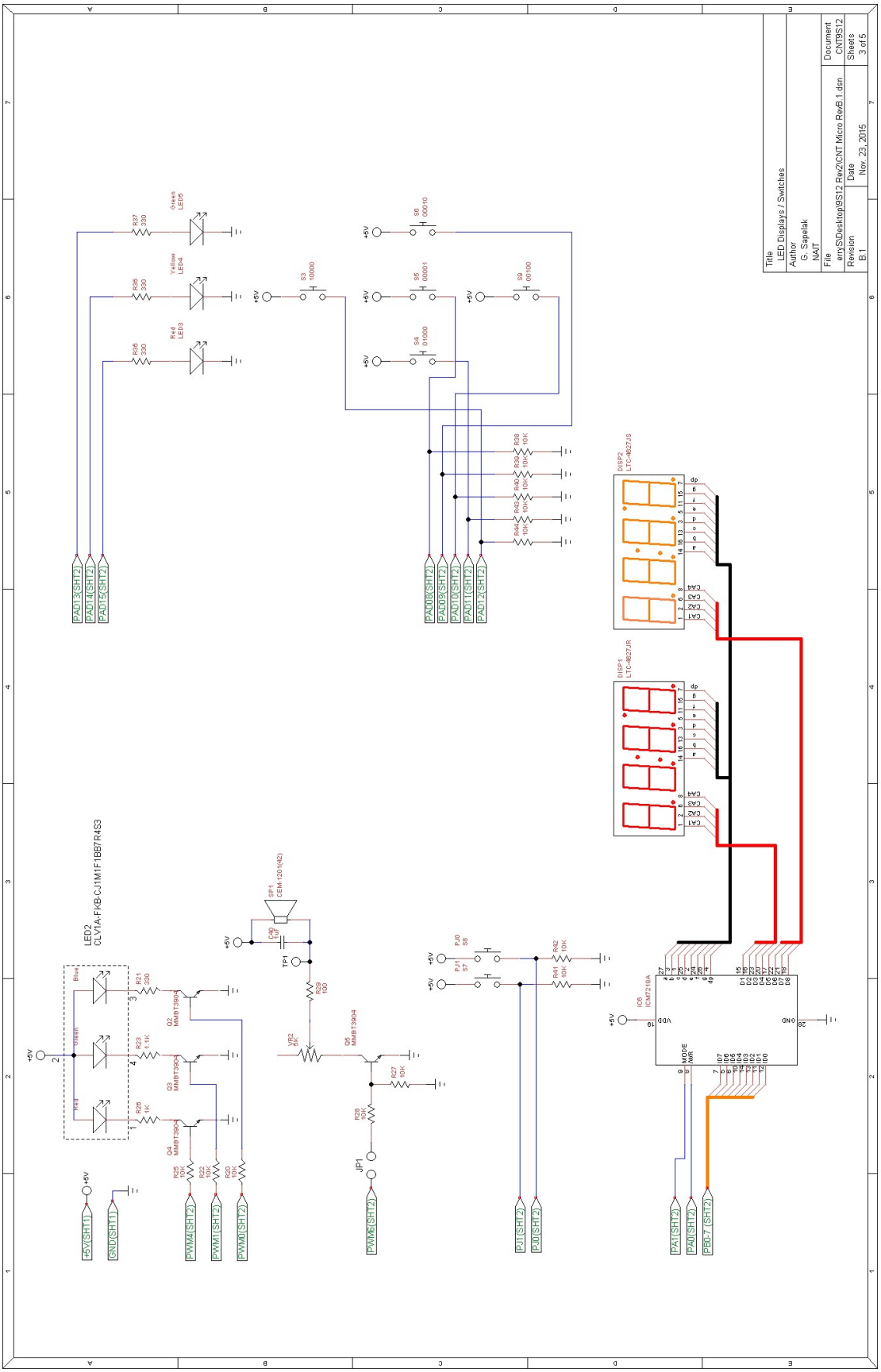


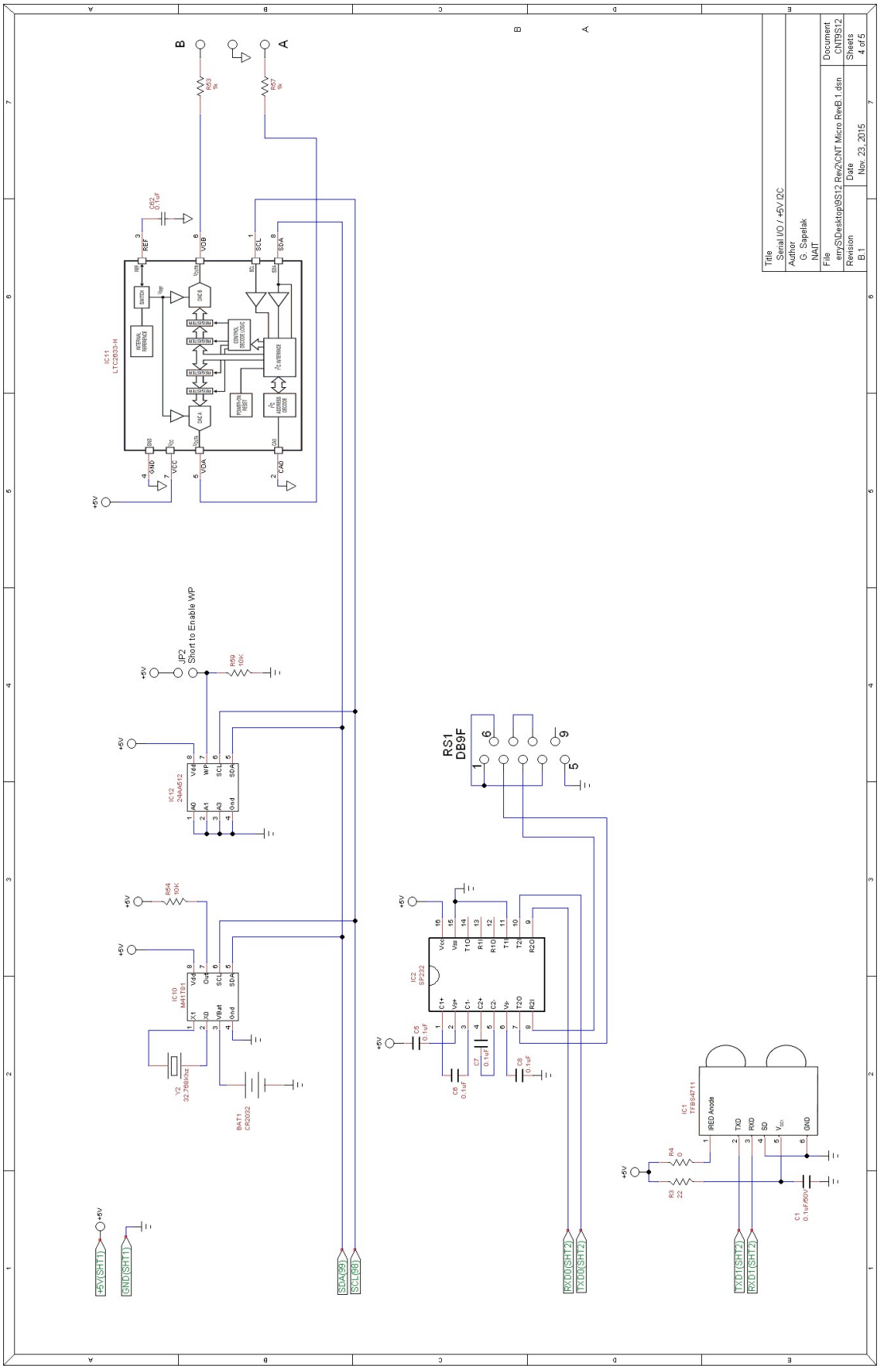

## The CNT MC9S12XDP512 I/O Board

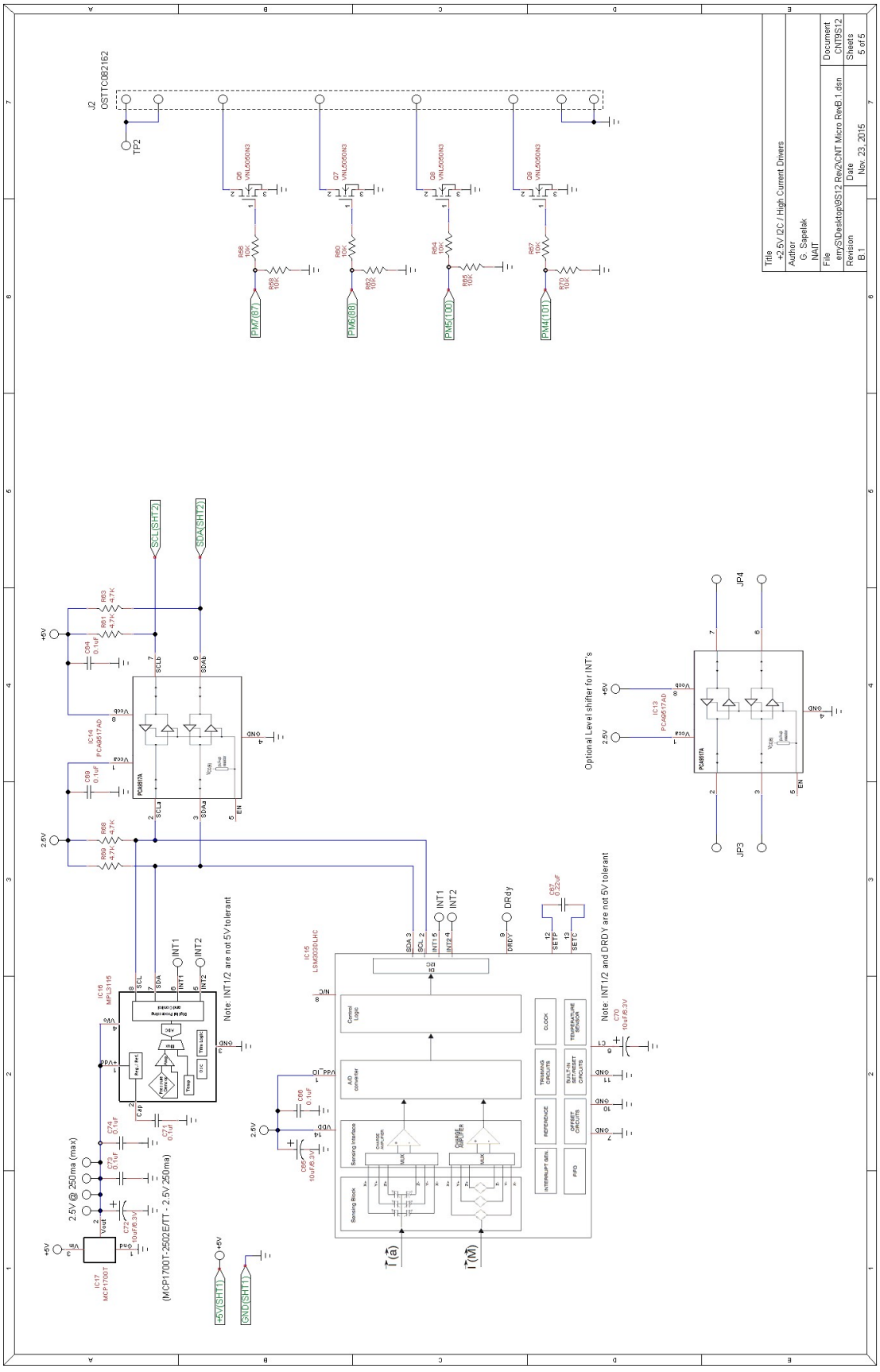
On the pages following, you'll find the schematics for the microcontroller kit, showing how these connections are actually wired up to provide us with the electronic interface shown in the previous bubble diagrams and block diagrams. Higher-resolution versions should be available in Moodle or from your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


Title
Vin, +5V, +/-10V Power Supplies
Author
G. Sapelak
NAIT
File
erry\StDesktop\9S12 Rev2\CNT Micro Rev\B.1.dsn
Revision
B.1
Date
Nov. 23, 2015
Document
CNT9S12
Sheets
1 of 5
+5V
GND
+10V
AGND
-10V
+12 - 16V
Vin
+12 - 16V DC
Cut trace for F1
Analog Gnd
Digital Gnd
+5V Decoupling Caps


Title
MC9S12DP512 Micro / 20x4 LCD
Author
G. Sapelak
NAIT
File
...ery\Desktop\9S12 Rev2\CNT Micro Rev6.1.dsn
Document
CNT9S12
Revision
B.1
Date
Nov 24, 2015
Sheets
2 of 5
Note:
CA7 is through hole 0.1" spacing- unpopulated
C75 - 82 are 0805 located on bottom layer of PCB - unpopulated
Note:
- All port pins are routed to 0.1" spaced pins located along the perimeter of the CPU as well as being routed to additional circuitry as detailed.


Title
LED Displays / Switches
Author
G Sapelak
NAIT
File
etry3\Desktop\9S12 Rev2\CNT Micro Rev8.1.dsn
Document
CNT9S12
Date
Nov. 23, 2015
Sheets
3 of 5
Revision
B.1
PA013(SHT2)
PA014(SHT2)
PA015(SHT2)
PA08(SHT2)
PA09(SHT2)
PA010(SHT2)
PA011(SHT2)
PA012(SHT2)
DISP2
LTC-4627JS
DISP1
LTC-4627JR
LED2
CLV1A-FKB-CJ1M1F1BB7R4S3
SP1
CEM-1201(42)
PWM6(SHT2)
PWM4(SHT2)
PWM1(SHT2)
PWM0(SHT2)
PJ1(SHT2)
PJ0(SHT2)
PA1(SHT2)
PA0(SHT2)
PB0-7 (SHT2)
+5V(SHT1)
GND(SHT1)
JP1
MMBT3904
ICM7218A


Title
Serial I/O / +5V I2C
Author
G. Sapelak
NAIT
File
erry\StDesktop\9S12 Rev2\CNT Micro Rev\B1.dsn
Document
CNT9S12
Date
Nov. 23, 2015
Sheets
4 of 5
Revision
B.1
JP2
Short to Enable WP
RS1
DB9F
+5V(SHT11)
GND(SHT11)
SDA(98)
SCL(96)
RXD0(SHT2)
TXD0(SHT2)
TXD1(SHT2)
RXD1(SHT2)


Title
+2.5V I2C / High Current Drivers
Author
G. Sapelak
NAIT
File
erry\SUDesktop\9S12 Rev2\CNT Micro Rev\B1.ddsn
Date
Nov. 23, 2015
Document
CNT9S12
Revision
B.1
Sheets
5 of 5
J2
OSTTC082162
TP2
PM7(87)
PM6(88)
PM5(100)
PM4(101)
SCL(SHT 2)
SDA(SHT2)
Optional Level shifter for INT's
JP4
JP3
Note: INT1/2 are not 5V tolerant
Note: INT1/2 and DRDY are not 5V tolerant
2.5V @ 250ma (max)
(MCP1700T-2502E/TT - 2.5V 250ma)
+5V(SHT1)
GND(SHT1)

## 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.

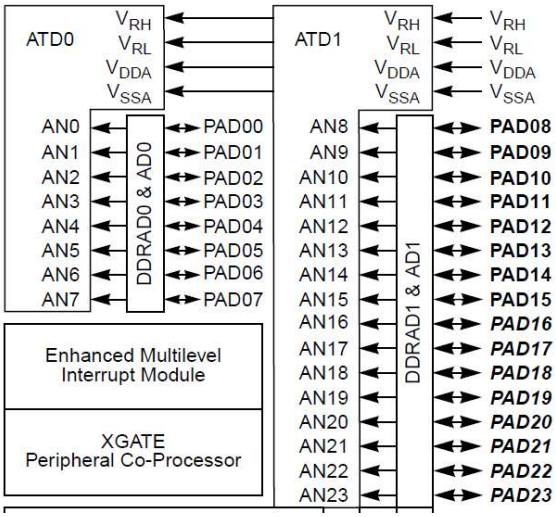
***Bussed (Parallel) Interface*** – A parallel interface involves the simultaneous transfer of multiple bits of information on separate (parallel) copper traces. The microprocessor in a personal computer operates in bussed mode. This requires an address bus capable of locating each unique address in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

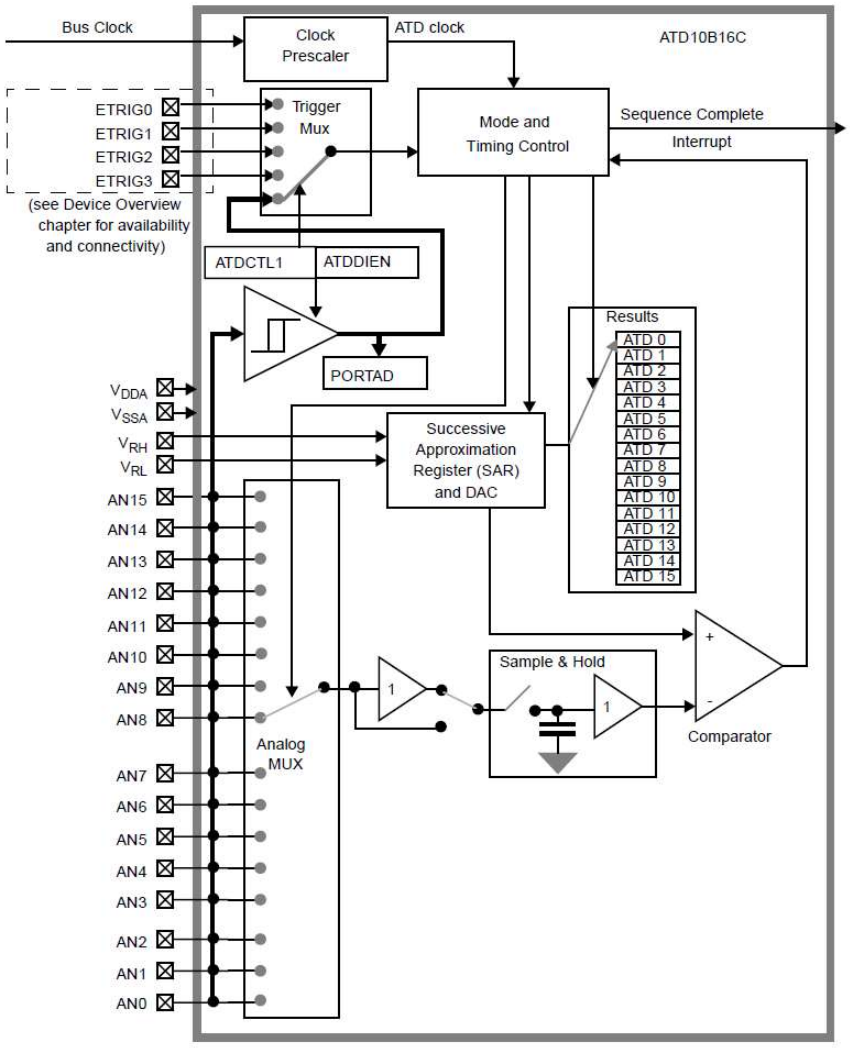



Figure 4-1. ATD10B16C Block Diagram



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 |
| | *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* |
| ATDDIEN | ATD0DIEN | $02CD | Lowest 8 Input Enable |
| ATDDIEN | ATD1DIEN | $008C | Upper 16 Input Enable |
| | *ATD1DIEN0* | *$008C* | *Highest 8 IE (n/a)* |
| | *ATD1DIEN1* | *$008D* | *Middle 8 IE* |

## 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.