Introduction to PIC Microcontroller Programming in C & the Development Board for AER201

Slides by *Tyler* Gamvrelis

Who am I?

- AER201 survivor
- 3rd year EngSci ECE
- Former U of T employee under Prof. Emami
  - Designed the current generation of the development board
- UTRA Robosoccer embedded & electrical lead
- Future PEY intern at Intel

Goals for today

- Development board
  - Gain experience using the modules on the development board by running through various sample programs
  - Understand capabilities & limitations
- PIC Microcontrollers
  - Use the in-circuit debugger to trace code execution
  - Learn how to view disassembly
  - Learn how to use interrupts in C and the keywords `volatile and const`
  - Learn programming concepts useful for embedded software
- Other
  - Learn how to navigate datasheets and utilize them to develop low-level code

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

The Big Picture

The Big Picture

What sorts of things might we need to know to build a robot?

- Need a way of acquiring information about the environment → sensors
- Need a way of making decisions → processing
- Need a way of actualizing the decisions → actuators, communication

How do we unify these?

- Need a system-level design, and designs for each subsystem
- Need to understand how each device (sensor, processor, actuator) works
  - Datasheets
  - Online resources (videos, articles)

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Programming PIC Microcontrollers Using C

How will this programming experience be different than CSC190?

- We're much closer to the hardware
  - We are interested in the **status and control registers** for various peripherals and directly observe and manipulate their values
  - Our **memory is far more constrained** than what we're used to → need to be conscious of our program's space-efficiency
  - Our **clock speed** is on the order of 100 times slower than that of a modern desktop PC → need to be conscious of our program's time-efficiency
- Your code is the only thing running on the PIC. There's no operating system (OS) underneath (no `malloc`)
- In AER201, you inevitably have to learn how to **communicate with *devices***, which may be motors, sensors, etc. Through this, you will be exposed to a diverse array of computing systems and electronics

Loops (1/2)

- 3 types of loops in C: for, while, do-while

```
for(int i = 0; i < n; i++){
    //do something
}

while(n > 0){
    //do something
    n--;
}

do{
    //do something
}while(condition != true);
```

Loops (2/2)

- Do-while loops are particularly useful when you know you want to do something *at least once*, but possibly more times. For example, **polling** the status of a device.

```
do{
    status = getDeviceStatus();
}while(status != READY);
```

Prototypical statement for polling, that is, preventing the rest of your program from executing until a condition is met.

```
/* SD_SingleBlockRead */
do{
    response = spiReceive();
}while(response != START_BLOCK);
```

Code from the `SD_SingleBlockRead` function in SD_PIC.c

Control flow

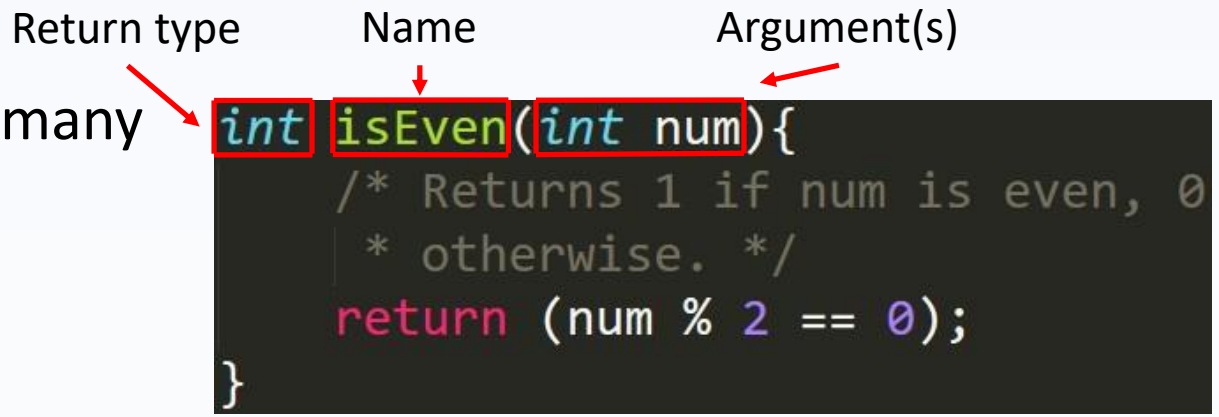
- Control flow statements provide a means of *decision-making* for programs
- if-else statements compile to conditional branches (inefficient)
- Switch statements can be compiled as look-up tables, accessed in constant time (i.e. $O(1)$)
- Switch statements can only be used for integer data types while if-else statements can be used for floating-point as well

```
switch(value){
    case 2: //do something if
            //value is 2

    case 3: //do something if
            //value is 3

    default: //do something if
             //value isn't 2 or 3
}
```

```
if(value == 2){
    //do something if
    //value is 2
}
else if(value == 3){
    //do something if
    //value is 3
}
else{
    //do something if
    //value isn't 2 or 3
}
```

Functions

- Functions give us a way of combining many instructions into a single line of code
- Functions are an **abstraction**



- **Abstraction** means that you're able to use something without understanding how it works internally
  - For example, given a linked list, we can call `printLinkedList` and get the desired results without having to know how the function is implemented

```
void printLinkedList(struct Node* head){
    /* Print a linked list. */
    Node* next = head;
    while(next != NULL){
        printf("%d\n", next -> data);
        next = next -> next;
    }
}
```

Data Types: Integers

- Integer-valued data types are:
  - `bit` (1 bit)
  - `char` (1 byte)
  - `short` (2 bytes)
  - `int` (2 bytes)
  - `short long` (3 bytes)
  - `long` (4 bytes)
  - `long long` (4 bytes)

TABLE 5-3: INTEGER DATA TYPES

Type	Size (bits)
`bit`	1
`signed char`	8
`unsigned char`	8
`signed short`	16
`unsigned short`	16
`signed int`	16
`unsigned int`	16
`signed short long`	24
`unsigned short long`	24
`signed long`	32
`unsigned long`	32
`signed long long`	32
`unsigned long long`	32

Figure: Table 5-3, MPLAB XC8 Compiler User's Guide

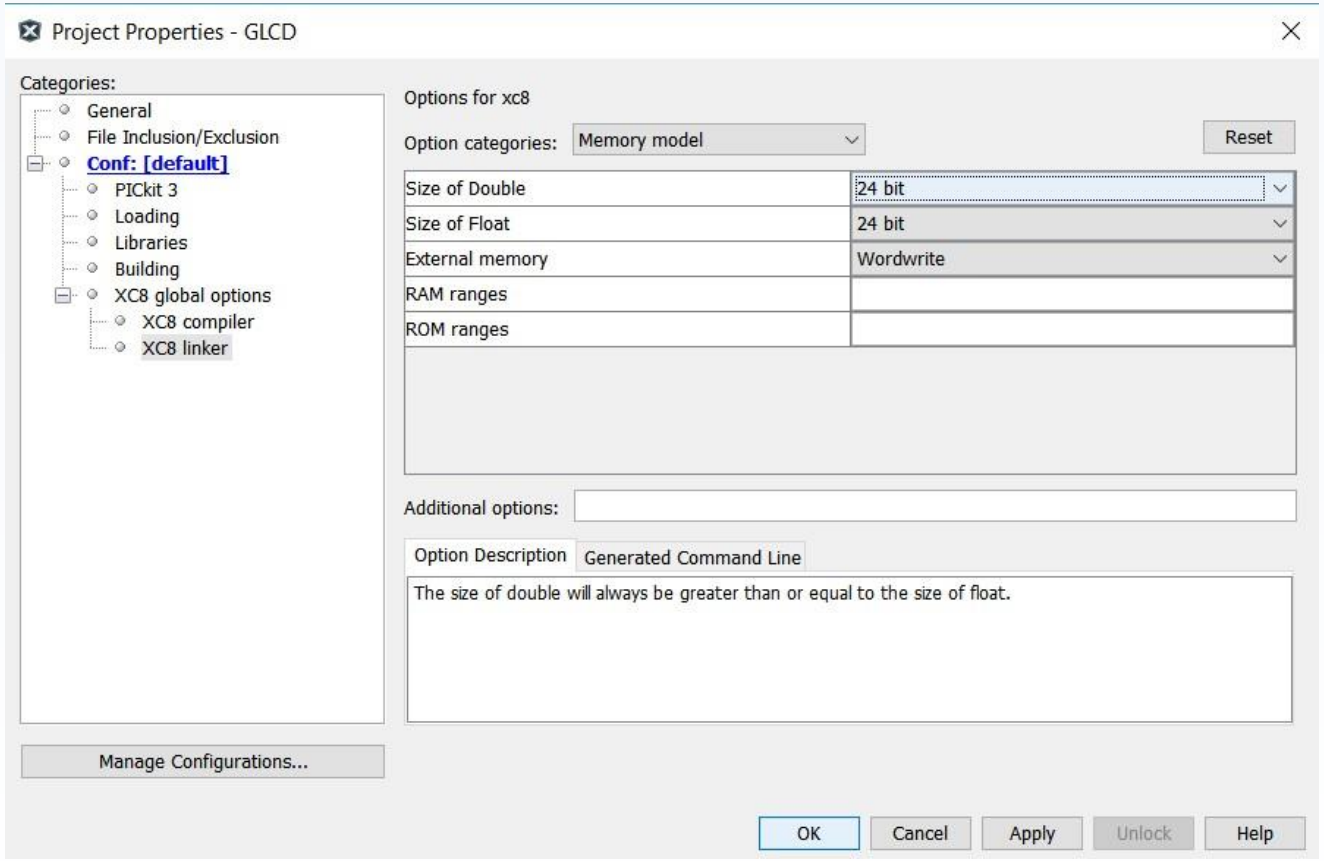
Data Types: “Real”-valued

- “Real”-valued data types are:
  - `float` (3 or 4 bytes)
  - `double` (3 or 4 bytes)
  - `long double` (3 or 4 bytes)

TABLE 5-5: FLOATING-POINT DATA TYPES

Type	Size (bits)
`float`	24 or 32
`double`	24 or 32
`long double`	same as `double`

Table 5-5, MPLAB XC8 Compiler User’s Guide



Specification of the type size for `float` and `double` in the project properties window in MPLAB X IDE

Data Types: Final Thoughts

- Question: are all variables of type `char` characters (e.g. ‘A’, ‘B’, ‘!’,…)?
- Integer data types are either signed or unsigned. Examples:
  - `unsigned char a = 192;` // takes on values from 0 to 255
  - `signed char b = -64;` // takes on values from -128 to 127
  - `a` and `b` in the above two examples are exactly the same in memory (0b11000000)
    - What differs is that for each usage of these variables in the code, the compiler generates different surrounding instructions that produce the signed or unsigned behaviour you’d expect

```
;main.c: 62: unsigned char a = 192;
   movlw 192
   movwf main@a,c

;main.c: 63: signed char b = -64;
   movlw 192
   movwf main@b,c
```

  - `char` variables are `unsigned` by default; all other integer data types are `signed` by default
- **Rule of thumb: Always explicate your variables’ signedness**

Enums (1/2)

- **Enumerated types** (“enums”) are a **collection of related named constants**. They are used to make code more readable (i.e. reduce the *semantic gap*)
- The names within an enum can be substituted for their corresponding numerical values throughout a project, and variables of type enum can be declared as well. For example `enum day today = friday;`
- In embedded systems, enums are often used to list the possible **states** that a part of the program might be in

```
enum day {sunday, monday, tuesday,
          wednesday, thursday, friday,
          saturday
         };
```

(Above) Here, `sunday` = 0, `monday` = 1, `tuesday` = 2,…,`saturday` = 6

```
enum day {sunday=1, monday, tuesday,
          wednesday=10, thursday,
          friday=10, saturday
         };
```

(Above) Here, `monday` = 2, `tuesday` = 3, `thursday` = 11, `saturday` =11. Each element is always 1 greater than the previous element unless explicitly assigned.

Enums (2/2)

- When compiled, statements involving enums simply undergo text replacement

```
enum day {sunday, monday, tuesday,
          wednesday, thursday, friday,
          saturday
         };
enum day today = friday;
```

Here, `sunday` = 0, `monday` = 1, `tuesday` = 2,…, `friday` = 5, `saturday` = 6

```
;main.c: 63: enum day {sunday, monday, tuesday,
;main.c: 64: wednesday, thursday, friday,
;main.c: 65: saturday
;main.c: 66: };
;main.c: 68: enum day today = friday;
   movlw 5
   movwf main@today,c
```

The C line "`enum day today = friday`" assigns the value 5 to the variable today

- Here, the text "friday" is replaced with "5" upon compilation. Text replacement is one of the initial stages of compilation

#define

- `#define` statements are called **preprocessor macros**
- We can use these to define constants and functions as follows:

```
#define _XTAL_FREQ 40000000
#define __lcd_newline() lcdInst(0xC0);
```

- **Caution:** Notice how these statements do not specify a data type anywhere? This is because #define statements are not **type safe**, meaning that if you're not careful with how you use them, unexpected behaviour may arise.

Constants

- Constants are type-safe, and are used when we want *something* to be **read-only**

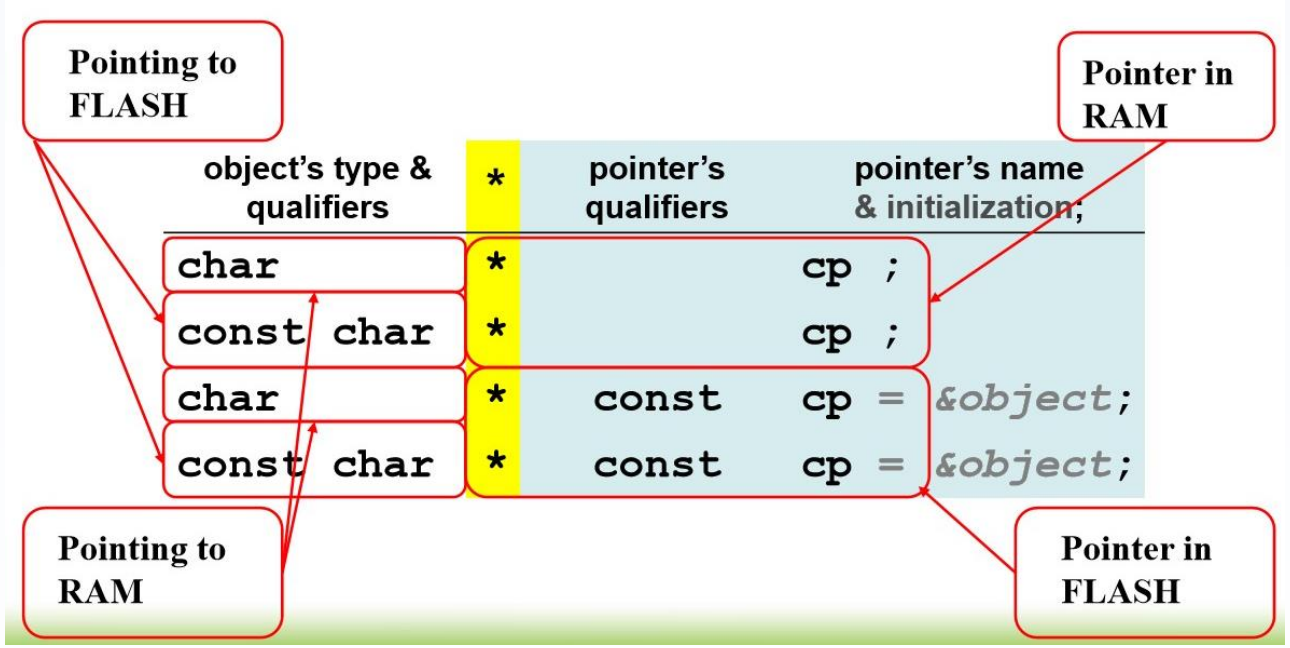
```
const unsigned long WHITE = 0xFFFFFF;
```

- Usefulness:
  - If we try to change the value of a constant the compiler will throw an error
  - Constants usually signify some fundamental immutable quantity
    - My AER201 project had to sort 12 cans in 3 minutes or less. The maximum allowed time exemplifies an immutable quantity (i.e. beyond my control to change)
  - Improve code readability

- **Also important:** constants are placed in the **program memory** (i.e. flash) while variables are placed in RAM

Pointers (1/2)

- Pointers are variables that hold memory addresses



(From Microchip MASTERs slides) Pointer syntax

Pointers (2/2)

- The “address of” operator, `&`, retrieves the address of the variable/data following it

```
unsigned int myAge = 20;
unsigned int* ptr_myAge = &myAge;
```

- The indirection operator, `*`, accesses the value stored at the address in the pointer

```
if(myBirthday){
    *ptr_myAge = *ptr_myAge + 1;
}
```

Structs

- Structs are collections of several variables which may be of different types
- They are useful for grouping related quantities

```
struct Animal{
    int color;
    char name[20];
};

struct Animal cat;
struct Animal dog;

cat.color = BLACK;
strcpy(cat.name, "Tom");
```

Illustration of syntax for structs

typedef

- The `typedef` keyword is used to “create new types” (not really, but you can think of it this way)
- This improves code readability and often improves development efficiency by creating abstraction
- We can place `typedef` before structs, enums, ints, etc.

```
typedef struct{
    int color;
    char name[20];
}Animal;

Animal cat;
Animal dog;

cat.color = BLACK;
strcpy(cat.name, "Tom");
```

```
typedef unsigned char uint8_t;
uint8_t myByte = 0x01;
```

Unions

- Unions provide several different ways of interacting with the same space in memory

pic18f4620.h

Source | History

```
883 typedef union {
884     struct {
885         unsigned LATA0          :1;
886         unsigned LATA1          :1;
887         unsigned LATA2          :1;
888         unsigned LATA3          :1;
889         unsigned LATA4          :1;
890         unsigned LATA5          :1;
891         unsigned LATA6          :1;
892         unsigned LATA7          :1;
893     };
894     struct {
895         unsigned LA0            :1;
896         unsigned LA1            :1;
897         unsigned LA2            :1;
898         unsigned LA3            :1;
899         unsigned LA4            :1;
900         unsigned LA5            :1;
901         unsigned LA6            :1;
902         unsigned LA7            :1;
903     };
904 } LATAbits_t;
905 extern volatile LATAbits_t LATAbits @ 0xF89;
```

These are the least-significant bits

Writing the following have the same result:

```
LATAbits.LATA0 = 1;

LATAbits.LA0 = 1;

LATA = 1; // LATA @ 0xF89 as well
```

Interrupts (1/2)

- All we need to do for interrupts is define 1 function (named whatever you want) as follows:

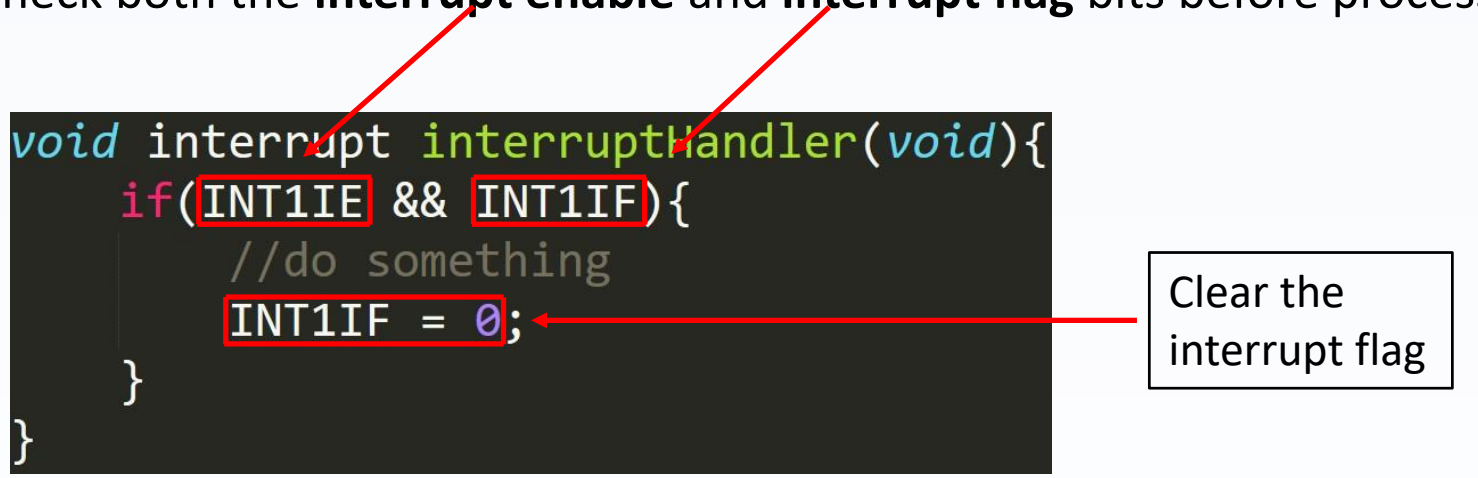
```
void interrupt interruptHandler(void){
    //handle interrupts in here
}
```

- **IMPORTANT:**
  - You only define **ONE** interrupt handler function
  - The interrupt keyword tells the compiler to map this function's address to the interrupt exception vector
  - The function must **return void**, and have a **void argument list**.

```
void interrupt interruptHandler(void){
    if(INT1IE && INT1IF){
        //do something
        INT1IF = 0;
    }
}
```

Interrupts (2/2)

- When we enter the ISR, we need to first determine which interrupt(s) require servicing
- The best practise is to check both the **interrupt enable** and **interrupt flag** bits before processing



- **Note:**
  - Interrupt priorities are determined by the order of the if statements (highest priority first)
  - If an interrupt is enabled and it doesn't have a handler to clear the interrupt flag, unexpected behaviour could occur, **possibly resetting the PIC** due to nesting

Volatile

- Wikipedia: the `volatile` keyword indicates (to the compiler) that a value may change between different accesses, even if it doesn't appear to be modified
  - What this means is that every time a statement involving this variable is encountered, the PIC loads it from its location in memory instead of a CPU register
  - This ensures the most up-to-date value is used
  - Compilers don't have to do this by default, and often they won't because it's better for optimizations

- **Rule of thumb:** global variables modified in an ISR should be declared `volatile`

```
volatile int myGlobalInt = 0;
```

- **Further reading:** atomic instructions and ISRs

PIC: SFRs (1/2)

- **Special function registers** (SFRs) control & monitor certain functionality
  - Interrupts
  - Timers
  - I/O peripherals

- In C, SFRs are defined in <pic18f4620.h>
  - They are **accessed like normal variables**
  - Their names in code *usually* coincide with their names in the datasheet (e.g. LATA from the slide on unions)

- Terminology:
  - "**set**" → bit is 1
  - "**cleared**" → bit is 0

PIC: SFRs (2/2)

Example: INTCON

- INTCON is the interrupt control register (pg. 111)

REGISTER 10-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF(1)
bit 7							bit 0

```
6376  // bitfield definitions
6377  typedef union {
6378      struct {
6379          unsigned RBIF                   :1;
6380          unsigned INT0IF                 :1;
6381          unsigned TMR0IF                 :1;
6382          unsigned RBIE                   :1;
6383          unsigned INT0IE                 :1;
6384          unsigned TMR0IE                 :1;
6385          unsigned PEIE_GIEL              :1;
6386          unsigned GIE_GIEH               :1;
6387      };
6388      struct {
6389          unsigned                        :1;
6390          unsigned INT0F                  :1;
6391          unsigned T0IF                   :1;
6392          unsigned                        :1;
6393          unsigned INT0E                  :1;
6394          unsigned T0IE                   :1;
6395          unsigned PEIE                   :1;
6396          unsigned GIE                    :1;
6397      };
6398      struct {
6399          unsigned                        :6;
6400          unsigned GIEL                   :1;
6401          unsigned GIEH                   :1;
6402      };
6403  } INTCONbits_t;
6404  extern volatile INTCONbits_t INTCONbits @ 0xFF2;
```

- If we want to set the "**global interrupt enable**" (GIE) bit, we can write:

```
INTCONbits.GIE_GIEH = 1;
```

- Alternatively, we notice that this the GIE_GIEH bit in the top struct overlaps the same memory as the GIE bit in the middle struct, so we can write:

```
INTCONbits.GIE = 1;
```

PIC: I/O (1/2)

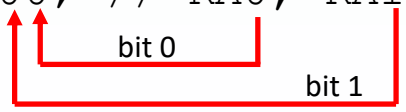
3 REALLY important SFRs are:

- TRISx (data direction latch)
- LATx (data output latch)
- PORTx (data input latch)

where x $\in \{A, B, C, D, E\}$

- Examples: TRISA, LATB, PORTC
- Read about them on **pg. 91 of the PIC18F4620 datasheet** (ch. 9)

- **TRISx**
  - Determines data direction (i.e. write or read)
  - E.g. `TRISA = 0b11111100; // RA0, RA1 outputs; RA2-RA7 inputs`



Rule of thumb: Write data to LATx, read data from PORTx

PIC: I/O (2/2)

Rule of thumb: Write data to LATx, read data from PORTx

Example: Writing 5 [V] onto RA0

```
LATAbits.LATA0 = 1; // load 1 into the output latch for RA0
TRISAbits.RA0 = 0; // set data direction for RA0 to output (i.e. 0)
```

Example: Reading digital input from RA0

```
TRISAbits.RA0 = 1; // set data direction for RA0 to input (i.e. 1)
unsigned char input = PORTAbits.RA0; // read input latch (either 1
                                     // or 0)
```

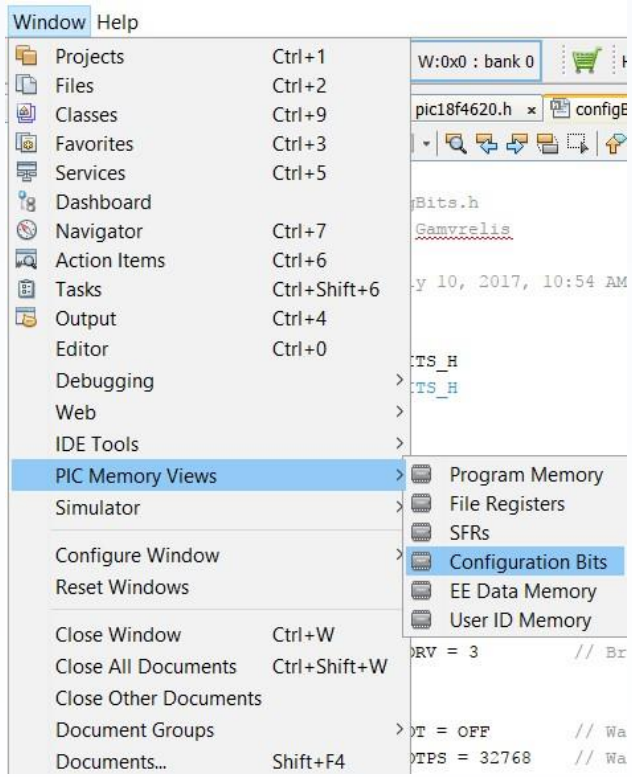
PIC: Configuration Bits

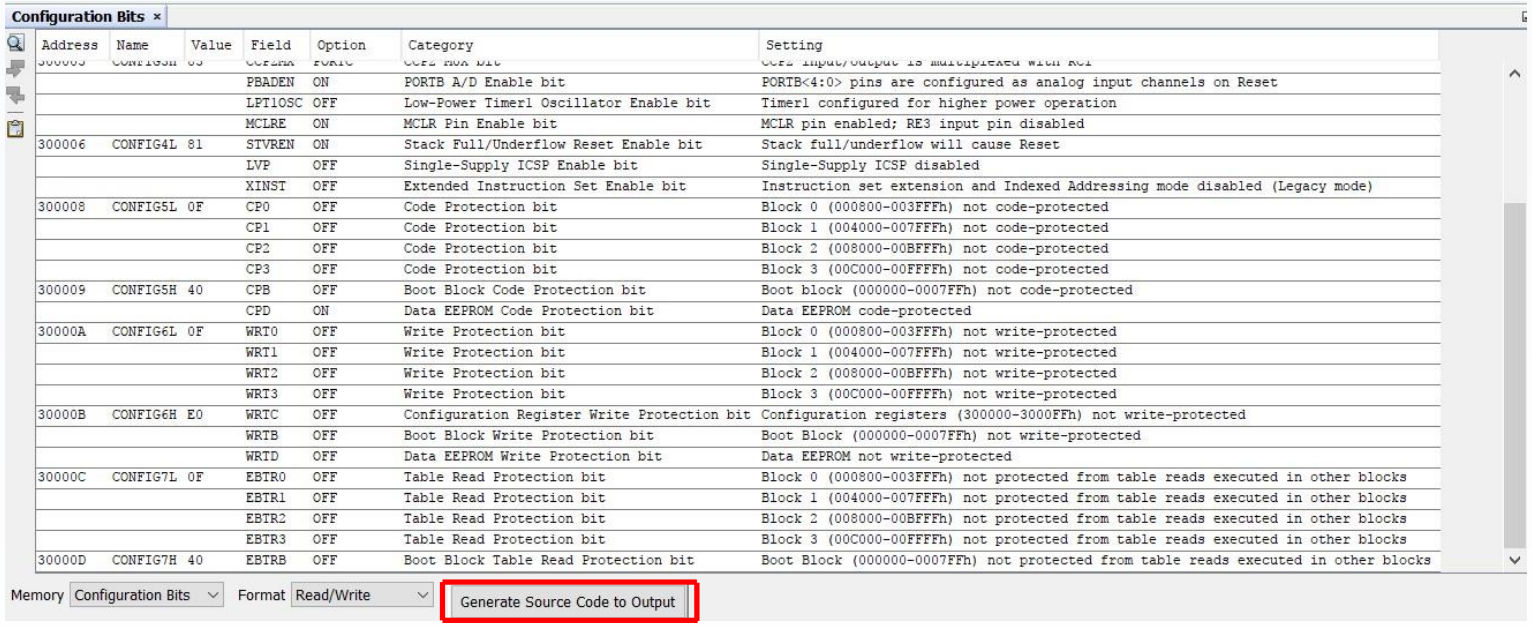
- **Question**: How do the various hardware modules on the PIC know how to behave by default (i.e. before any code starts executing)?
- **Answer**: configuration bits ("**config bits**")
  - These are programmed into the PIC and are not changed during runtime

- Examples of configuration bits include:
  - Oscillator selection bits
  - MCLR pin enable bit
  - Watchdog timer (WDT) enable bit (**don't enable this**)

PIC: Configuration Bits

- **MPLAB X IDE has a GUI for the config bits that can generate the code for you**





Address	Name	Value	Field	Option	Category	Setting
			PBADEN	ON	PORTB A/D Enable bit	PORTB<4:0> pins are configured as analog input channels on Reset
			LPT1OSC	OFF	Low-Power Timer1 Oscillator Enable bit	Timer1 configured for higher power operation
			MCLRE	ON	MCLR Pin Enable bit	MCLR pin enabled; RE3 input pin disabled
300006	CONFIG4L	81	STVREN	ON	Stack Full/Underflow Reset Enable bit	Stack full/underflow will cause Reset
			LVP	OFF	Single-Supply ICSP Enable bit	Single-Supply ICSP disabled
			XINST	OFF	Extended Instruction Set Enable bit	Instruction set extension and Indexed Addressing mode disabled (Legacy mode)
300008	CONFIG5L	0F	CP0	OFF	Code Protection bit	Block 0 (000800-003FFFh) not code-protected
			CP1	OFF	Code Protection bit	Block 1 (004000-007FFFh) not code-protected
			CP2	OFF	Code Protection bit	Block 2 (008000-00BFFFh) not code-protected
			CP3	OFF	Code Protection bit	Block 3 (00C000-00FFFFh) not code-protected
300009	CONFIG5H	40	CPB	OFF	Boot Block Code Protection bit	Boot block (000000-0007FFh) not code-protected
			CPD	ON	Data EEPROM Code Protection bit	Data EEPROM code-protected
30000A	CONFIG6L	0F	WRT0	OFF	Write Protection bit	Block 0 (000800-003FFFh) not write-protected
			WRT1	OFF	Write Protection bit	Block 1 (004000-007FFFh) not write-protected
			WRT2	OFF	Write Protection bit	Block 2 (008000-00BFFFh) not write-protected
			WRT3	OFF	Write Protection bit	Block 3 (00C000-00FFFFh) not write-protected
30000B	CONFIG6H	E0	WRTC	OFF	Configuration Register Write Protection bit	Configuration registers (300000-3000FFh) not write-protected
			WRTB	OFF	Boot Block Write Protection bit	Boot Block (000000-0007FFh) not write-protected
			WRTD	OFF	Data EEPROM Write Protection bit	Data EEPROM not write-protected
30000C	CONFIG7L	0F	EBTR0	OFF	Table Read Protection bit	Block 0 (000800-003FFFh) not protected from table reads executed in other blocks
			EBTR1	OFF	Table Read Protection bit	Block 1 (004000-007FFFh) not protected from table reads executed in other blocks
			EBTR2	OFF	Table Read Protection bit	Block 2 (008000-00BFFFh) not protected from table reads executed in other blocks
			EBTR3	OFF	Table Read Protection bit	Block 3 (00C000-00FFFFh) not protected from table reads executed in other blocks
30000D	CONFIG7H	40	EBTRB	OFF	Boot Block Table Read Protection bit	Boot Block (000000-0007FFh) not protected from table reads executed in other blocks

In case you forget…

ALL this information is contained in the XC8 user manual and the PIC18F4620 datasheet

(Both are included with the sample code in the “Datasheets and Other References” folder)

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

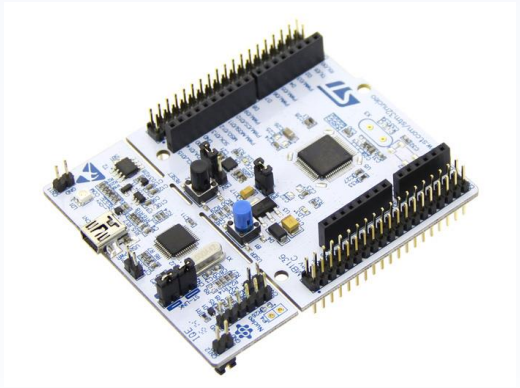
Development Board Orientation

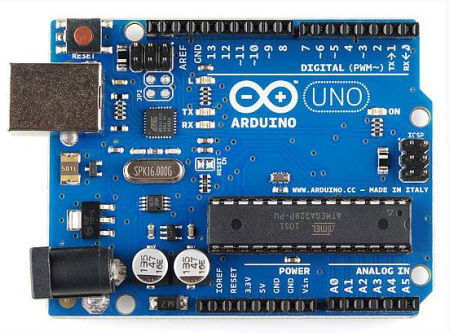
What is the purpose of the board?

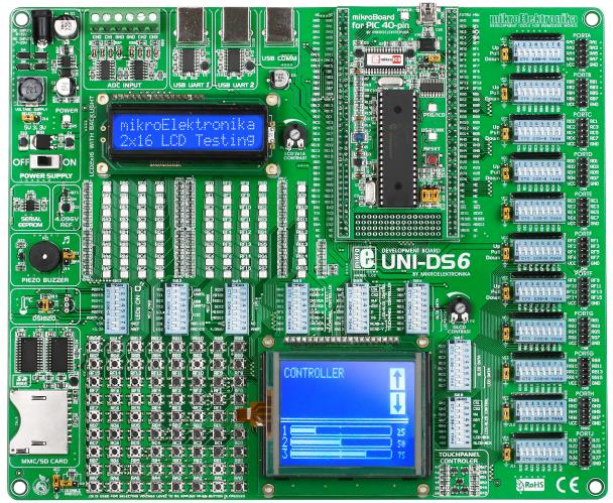
- Platform for developing **robotic control solutions**
  - Process inputs signals (**sensors**)
  - Generate output (control) signals for **actuators**

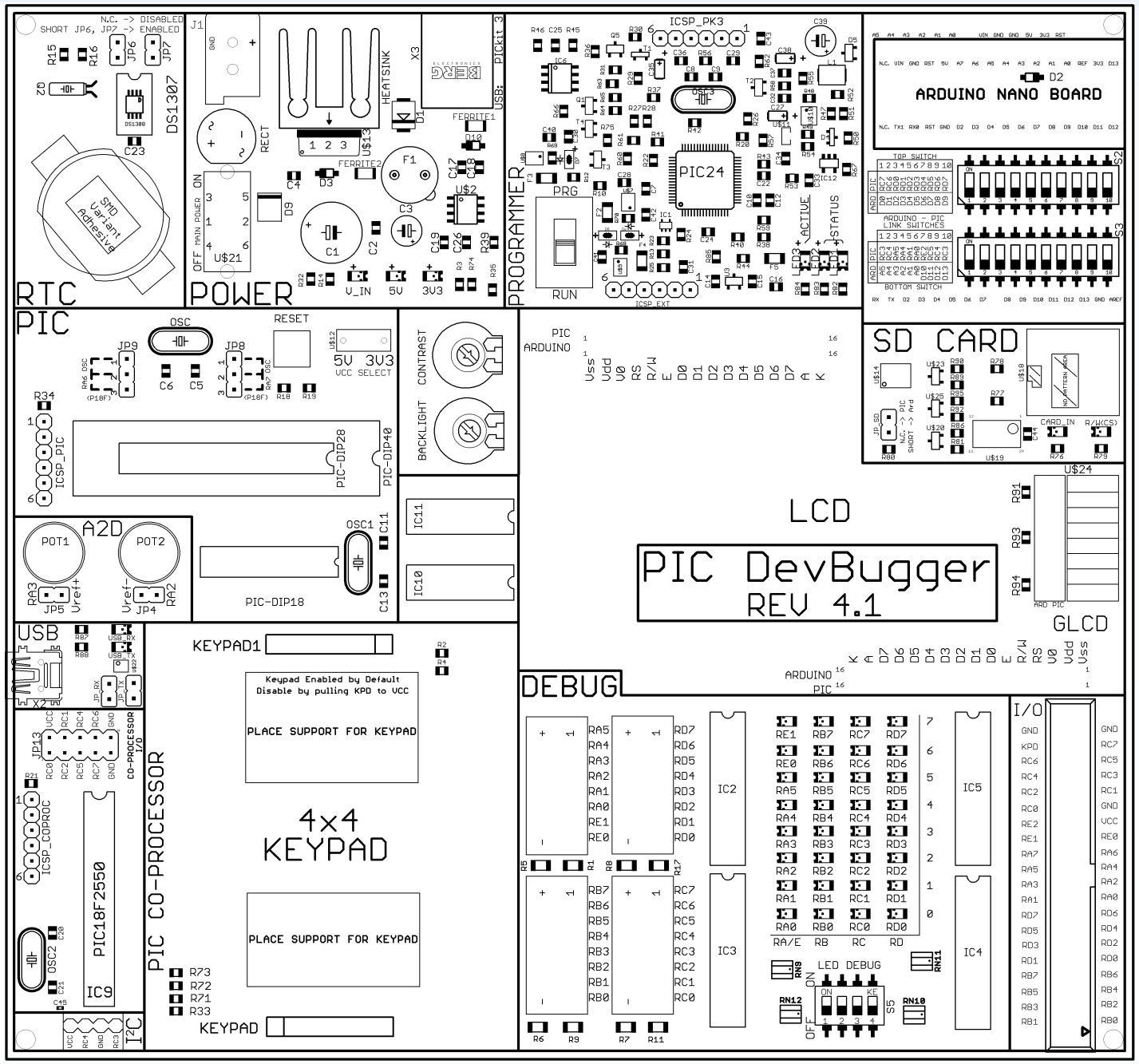
- There are many such platforms (Arduino is a common one, ST Nucleo boards are another)which each have their own trade-offs

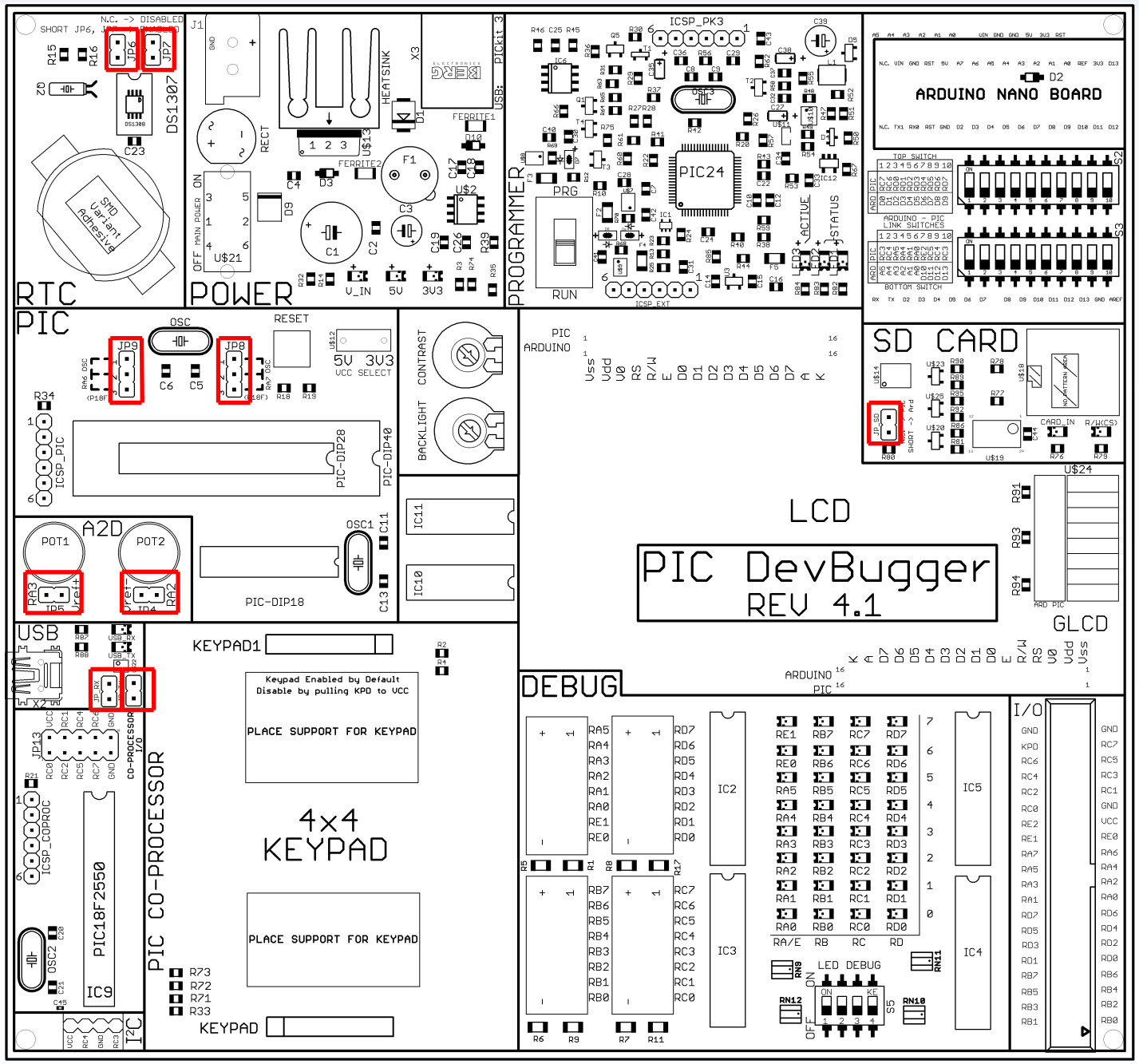
- The DevBugger **has integrated modules** that touch on broad & relevant applications. They are approachable for people learning to develop programs for **embedded systems**
  - Time-keeping
  - User input via key presses
  - Displaying text
  - Displaying graphics
  - Storage
  - Inter-device communication








RTC
POWER
PROGRAMMER
PIC
SD CARD
LCD
PIC DevBugger
REV 4.1
GLCD
DEBUG
I/O
PIC CO-PROCESSOR
4x4
KEYPAD
ARDUINO NANO BOARD
PIC18F2550
PIC24
DS1307
USB
A2D
BACKLIGHT
CONTRAST
RUN
ICSP_PK3
ICSP_EXT
ICSP_PIC
ICSP_COPROC
PIC-DIP28
PIC-DIP40
PIC-DIP18
PLACE SUPPORT FOR KEYPAD
Keypad Enabled by Default
Disable by pulling KPD to VCC
LED DEBUG
HEATSINK
RECT
5V 3V3
VCC SELECT


RTC
POWER
PROGRAMMER
PIC
SD CARD
LCD
PIC DevBugger
REV 4.1
GLCD
DEBUG
I/O
PIC CO-PROCESSOR
4x4
KEYPAD
ARDUINO NANO BOARD
PIC18F2550
PIC24
DS1307
USB
A2D
I2C

5 important things to know

1. The **user manual** contains information & tips for each module. If something doesn't seem to be working, read up on it here
2. **Wall power is superior to USB power**. Also, the programmer module takes approximately 2 seconds to start up (to get current from PC), so if powered via USB only, the PIC might reset 2 seconds after power-up
3. Pins have an indeterminate state (i.e. "floating") unless you specify otherwise. **Rule of thumb:** don't leave pins floating
4. PRG/RUN switch should always be in the **RUN position when you're running code, and the PRG position when you're programming/debugging**. In the PRG position, RB6 and RB7 are unavailable for I/O since they're being used to connect the PIC18F4620 to the programmer module
5. The co-processor can boot into different programs. If you want to use the USB feature, you need to supply your own 20 MHz oscillator

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Exercise: I/O

Exercise: I/O

Goal: Blink the RC0 LED in the debug module at a frequency of 1 Hz

Given:

- Walkthrough on how to create a new project
  - main.c
  - Config bits setup (OSC = HSPLL, WDT = OFF, PBADEN = OFF, LVP = OFF)
  - Outside main: `#define _XTAL_FREQ 40000000`
- Walkthrough on how to get code onto the PIC18F4620
- **TRISx** controls the data direction for port x
- **LATx** controls the output to the pins on port x
- **PORTx** reads from the pins on port x
- __delay_ms() exists in <xc.h>

If finished early: Blink all LEDs for port C at a frequency of 20 Hz

Exercise: I/O

Solution:

Posted later

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Exercise: Interrupts

Exercise: Interrupts

Goal: Make the RC0 LED in the debug module toggle whenever a key on the keypad is pressed, using RB1 external interrupt

Given:

- Interrupt control (**INTCONx**) register information begins on **pg. 110** of the PIC18F4620 datasheet
- Information about RB1 is on **pg. 14** of the PIC18F4620 datasheet
- Interrupt handler function prototype: `void interrupt interruptHandler(void)`

Steps:

1. Look at the information about **RB1**. What is the pin name for the interrupt we want to use?
2. Look at the **INTCONx** registers. Which bits in which **INTCONx** registers do we need to set to enable the RB1 interrupt? (hint: there are 2 registers, and 1 bit in each must be set)
3. Write the interrupt handler. What bits do we need to check for to handle the **RB1** interrupt?
4. What goes inside the interrupt service routine for the **RB1** interrupt? Use the previous exercise

If done early:

- Implement a routine to toggle the RC0 LED whenever a key on the keypad is pressed WITHOUT using interrupts
- Place a breakpoint inside your ISR and explore some of the debugger windows

Exercise: Interrupts

Solution:

Posted later

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Libraries and APIs

What are libraries? (1/2)

Library → Collection of related functionality

- **Header files (.h)** → Tell you what to expect from the library
  - #include, #define, type definitions (typedef), **function prototypes**, inline function definitions, etc.
- **Source files (.c)** → Implementation of functionality
  - Functions, private variables, etc.

stdio is a C library you should all recognize (“`include <stdio.h>`”)

What are libraries? (2/2)

Example: Graphical LCD (GLCD)

Suppose we want to command a graphical LCD to display a circle. We might end up with something like this:

```
void drawCircle(unsigned int x, unsigned int y, unsigned int radius)
    void glcdDrawPixel(unsigned char XS,
    unsigned char YS, unsigned long color)
        void glcdDrawRectangle(unsigned char XS, unsigned char XE,
        unsigned char YS, unsigned char YE, unsigned long color)
            void glcdTransfer(unsigned char byte, unsigned char cmd)
                void spiSend(unsigned char val)
                    unsigned char spiTransfer(unsigned char byteToTransfer)
```

Yellow = Shapes library
blue = GLCD library
red = SPI library

What are APIs? (1/6)

- An "**application programming interface**" (API) typically refers to the functions available in a library
- The goal of an API is to allow programmers to focus on ***creating applications*** by providing them with simple ways to use low-level implementations

What are APIs? (2/6)

Taking a step back

What is an interface?

- **"a point where two (or more) systems meet and interact"**

Examples:

- Pressing keys on your keyboard to type text on your screen
- Navigating your touchscreen phone with your finger

This "meeting" and "interacting" from the definition consists of an **agreed-upon I/O relationship**

- When you press the A key on your keyboard, you expect an A to appear where you're typing. You don't have to know how your computer does it, you just need to know what **input** you need to provide to get the desired **output**

With programming, functions are the mechanism implementing this I/O relationship

- To draw a circle on a display, you don't need to know how the display works. Instead, you just need to know how to tell *other code* to draw a circle

What are APIs? (3/6)

What's the I/O relationship here?

```
void drawCircle(unsigned int x, unsigned int y, unsigned int radius)
```

Input(s): `x`, `y`, `radius`

Output(s): circle of radius `radius` centred at the coordinates (`x`, `y`) on the display

What are APIs? (4/6)

Example: The `glcdDrawPixel` function provides a ***straightforward interface*** for programmers to draw on a GLCD.

```
void glcdDrawPixel(unsigned char XS, unsigned char YS, unsigned long color){
    /* Draws the color specified at the coordinates specified relative to the
     * origin (top-left).
     *
     * Arguments: XS, the x-position of the pixel (min: 0, max: 127)
     *            YS, the y-position of the pixel (min: 0, max 127)
     *            color, the color specified
     *
     * Returns: none
     */
```

The ***implementation*** of this functionality, however, requires an understanding of the **SPI hardware** on the PIC18F4620 (as detailed in the datasheet), and the architecture of the **ST7735R GLCD controller**.

What are APIs? (5/6)

High abstraction

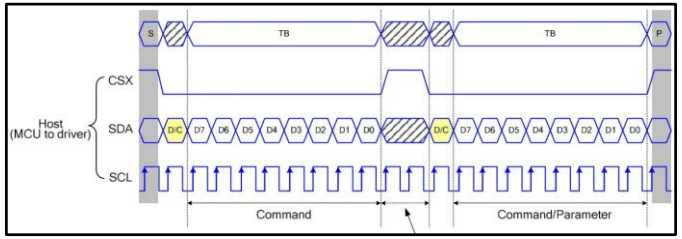
Low abstraction

```
184  /* Fundamental communication interface. */
185  void glcdTransfer(unsigned char byte, unsigned char cmd);
```

Expression of intent (the ***interface*** programmers would use to build GLCD functions like rectangle drawing)

```
18  void glcdTransfer(unsigned char byte, unsigned char cmd){
19     /* Driver to interface with the SPI module to send data to the GLCD.
20      *
21      * Arguments: byte, the byte corresponding to the desired command/data value
22      *            cmd, 1 --> command for display controller, 0 --> data for RAM
23      *
24      * Returns: none
25      */
26
27     RS_GLCD = (cmd == 1) ? 0 : 1; // RS low for command, and high for data
28
29     /* Enable serial interface and indicate the start of data transmission by
30      * selecting the display (slave) for use with SPI. */
31     CS_GLCD = 0;
32
33     spiSend(byte); // Transmit data
34
35     CS_GLCD = 1; // Deselect display
36  }
```

Implementation



Datasheet

What are APIs? (6/6)

Q: Why do we do this?

A: To write graphics functions, we don't want to worry about how to transfer bytes to the GLCD. Instead, we solve the problem once, then invoke the solution every time we require it.

This is **modular design**, and is more organized, readable, and easy to maintain.

Separation of concerns (SoC) design principle: Separate your program into *distinct sections* that each address a separate concern. One direct implication of this is that **for each concern, we should have a distinct pair of .c and .h files**

More C: Extern & Static Keywords

- There are two keywords in C that modify the visibility of functions and variables: `extern`, and `static`
  - These have slightly different implications for functions than for variables

- Using these helps us build better libraries/APIs since we can choose which information to hide and which to expose

We don't have time to go into the details right now, but here're some examples:

- Sometimes, we want a function to only be visible within one .c file. In this case, we qualify it with the keyword `static`. This is similar to "private" in OOP.
- `static` variables stick around forever, even if they only have local scope. This means that `static` variables local to the scope of a function retain their value upon exiting and re-entering the function

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Interfacing with Devices

You may have heard of a driver…

Device drivers

- Getting a device to do something on your behalf
- Requires deep knowledge of the low-level interface

Polled v.s. Interrupt-driven I/O (1/2)

Polled

- Program execution is blocked while waiting for the device to respond
- Implemented using a loop, often a do-while ("are you ready? How about now? How about now? …")

Interrupt-driven

- Program execution continues until device notifies you that it is ready
- Implementation requires an interrupt service routine (ISR). Also, you need some variable to store the data in that both the ISR and the rest of your program can access
  - Since this variable is accessed from multiple *execution points*, code that uses this variable needs to always make sure it reads the value from memory (i.e. variable is `volatile`) and not from a CPU register since the register won't be kept up-to-date
- Further reading: *mutexes, locks, threads*

Polled v.s. Interrupt-driven I/O (2/2)

Polled I/O is sometimes the most suitable option for a project

On slide 11 (loops), we saw the following code exemplifying polled I/O.

```
/* SD_SingleBlockRead */
do{
    response = spiReceive();
}while(response != START_BLOCK);
```

Code from the `SD_SingleBlockRead` function in SD_PIC.c

- This code snippet is part of the function SD_`SingleBlockRead`, which reads 512 bytes from a SD card
- After we command the SD card to give us the 512 bytes, it needs some time to find the bytes and read them into a buffer
- We are forced to poll the SD card because it has no way to tell us when it is ready

Remark: We use SD cards in the **SPI bus mode**, but there is a more efficient mode called SDIO that supports interrupts. Unfortunately, SD cards are complicated and the SDIO standard is very expensive.

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Exercise: RTC library

Exercise: RTC Library

Goal: Given some working code, rewrite it in a modular fashion using the techniques discussed for libraries and APIs

Given:

- Code (url: **https://tinyurl.com/aer201-pic-2018**)
- Quick run through of how the code works

Steps:

1. Look at the code. What are the 2 ways we're interacting with the RTC? Which high-level functions should be available to users? What should be the input/output relationship of these?
2. Considering separation of concerns, how many .c and .h pairs do we need? Which are these?
3. Move the functions, macros, and any other data to the appropriate files
4. Are there any new functions, macros, etc. you can create to improve readability and overall organization? Why or why not?

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Exercise: Debugging

Exercise: Debugging

Goal: To become familiar with the in-circuit debugger

Given:

- RTC library code from the last exercise
- In MPLAB X, use the “Debug” menu as well as “Window -> Debugging” and “Window -> PIC Memory Views”

Steps: (must be completed using debugging tools)

1. What is the value of `TRISE` upon reset?
2. At what address does the function `I2C_Master_Init` begin executing?
3. What is that ASCII character corresponding to the YEAR data?
4. How many breakpoints can we set?
5. What do each of these buttons do? 
6. At what address is the `I2C_Master_Read` function?
7. How do we step through instructions at an assembly level? At a C level?
8. How can we watch a global variable? How can we monitor a local variable?

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Sample: 00_PortTest

Sample: 00_PortTest (1/6)

Location

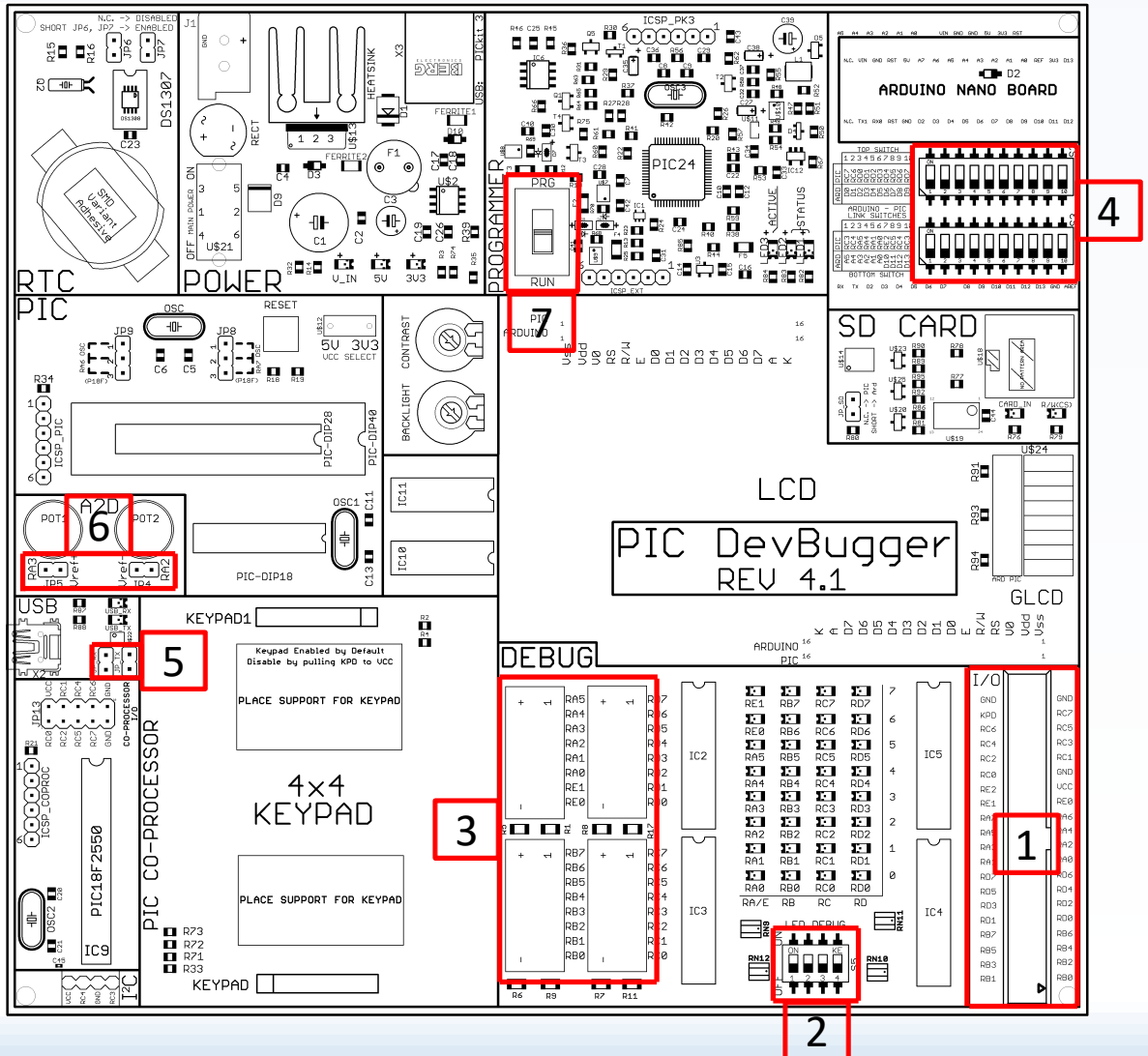
Code\00_PortTest\PIC\C\PortTest.X

What this program does

Iterates through the pins on the pic, toggling them sequentially as seen on the LED array in the debug module of the development board

Preconditions

1. No external circuitry driving the data lines
2. LED debug switches are all enabled
3. Debug DIP switches are all in the middle position
4. Arduino-PIC link switches are all OFF
5. JP_TX and JP_RX in the USB module are open (i.e. not shorted)
6. The A2D jumpers are open (i.e. not connected to the PIC)
7. The PRG/RUN switch must be in the PRG position (this is assumed for ALL sample code)



Sample: 00_PortTest (2/6)

```
* Description: Iterates through the pins on the PIC, making the debug LEDs
*              toggle sequentially.
*
* Preconditions:
*   1. No external circuitry driving the data lines
*   2. LED debug switches are all enabled
*   3. Debug DIP switches are all in the middle position
*   4. Arduino-PIC link switches are all OFF
*   5. JP_TX and JP_RX in the USB module are open (i.e. not shorted)
*   6. The A2D jumpers are open (i.e. not connected to the PIC)
*   7. The PRG/RUN switch must be in the PRG position (this is assumed for ALL
*      sample code)
*
* Note: RB1 is held low by the co-processor when it runs the default keypad
*       encoder program. To see it toggle, simply boot the co-processor into
*       the UART keypad encoder by holding key 1 on the keypad while turning
*       on power, or use a jumper to short KPD on the I/O bus to VCC.
```

Sample: 00_PortTest (3/6)

```
/***** Includes *****/
#include <xc.h>
#include "configBits.h" // For _XTAL_FREQ, required in delay macros

/***** Defines *****/
#define DELAY_TIME  250
```

Sample: 00_PortTest (4/6)

```
void main(void){

    // <editor-fold defaultstate="collapsed" desc="Machine Configuration">
    /********************************* PIN I/O *********************************/
    /* Write outputs to LATx, read inputs from PORTx. Here, all latches (LATx)
     * are being cleared (set low) to ensure a controlled start-up state. */
    LATA = 0x00;
    LATB = 0x00;
    LATC = 0x00;
    LATD = 0x00;
    LATE = 0x00;

    /* After the states of LATx are known, the data direction registers, TRISx
     * are configured. 0 --> output; 1 --> input. Default is  1. */
    TRISA = 0x00;
    TRISB = 0x00;
    TRISC = 0x00;
    TRISD = 0x00;
    TRISE = 0x00;

    /************************** A/D Converter Module ***************************/
    ADCON0 = 0x00;  // Disable ADC
    ADCON1 = 0b00001111; // Set all A/D ports to digital (pg. 222)
    // </editor-fold>
```

Sample: 00_PortTest (5/6)

```
/* Main loop. This demonstrates writing to the latches for each I/O pin on
 * the IC18F4620. Note that RE2, RA6, and RA7 do not have LEDs in the debug
 * module. To view the latches for these pins being set and cleared, the
 * I/O bus can be used to connect external LEDs with series resistors to
 * ground. */
while(1){
    /* Initialize local variables. */
    unsigned char i = 0;
    unsigned char temp = 0x00;

    temp = 1 << 7; // Set temporary byte to 0b10000000.
    /* Write temp to the latch, then bit shift to right. Iterate over the port
     * size. */
    for (i = 0; i < 8; i++){
        LATA = temp;
        temp >>= 1;
        __delay_ms(DELAY_TIME);
    }
    LATA = 0x00; // Clear latch outputs after demonstrating the port.

    /* Same process for all other ports. */
    temp = 1 << 7;
    for (i = 0; i < 8; i++){
        LATB = temp;
        temp >>= 1;
        __delay_ms(DELAY_TIME);
    }
    LATB = 0x00;
```

Sample: 00_PortTest (6/6)

Q: Who can tell me why we only iterate through port E three times while all others are iterated through 8 times?

```
        temp = 1 << 7;
        for (i = 0; i < 8; i++){
            LATC = temp;
            temp >>= 1;
            __delay_ms(DELAY_TIME);
        }
        LATC = 0x00;

        temp = 1 << 7;
        for (i = 0; i < 8; i++){
            LATD = temp;
            temp >>= 1;
            __delay_ms(DELAY_TIME);
        }
        LATD = 0x00;

        /* Note: Look in the SD card module to see the LED for RE2. */
        temp = 1 << 2;
        for (i = 0; i < 3; i++){
            LATE = temp;
            temp >>= 1;
            __delay_ms(DELAY_TIME);
        }
        LATE = 0x00;
    }
}
```

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Sample: 03_KeypadLCD

Sample: 03_KeypadLCD (1/10)

Location

Code\03_KeypadLCD\KeypadLCD_IO\Polling\C\

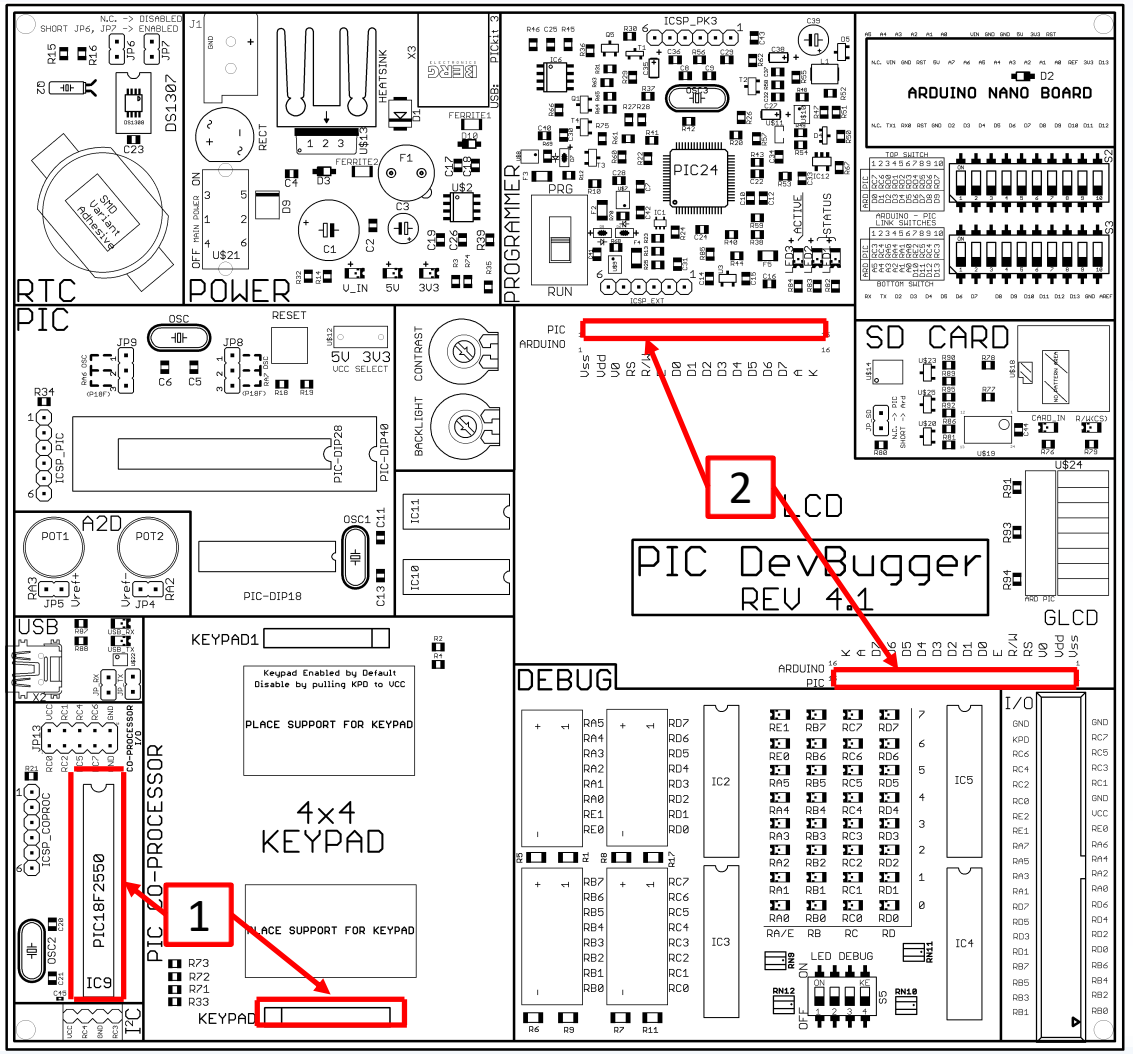
PortTest.X

What this program does

Writes keypress data to the character LCD

Preconditions

1. Co-processor running default program
2. Character LCD in PIC socket



Sample: 03_KeypadLCD (2/10)

```
* Description: Receive keypress data from co-processor and display the
*              corresponding character on the character LCD.
*
* Preconditions:
*   1. Co-processor running default program
*   2. Character LCD in PIC socket
```

Sample: 03_KeypadLCD (3/10)

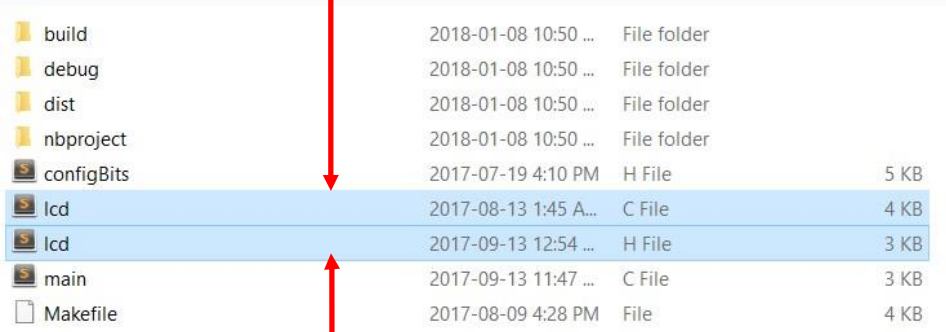
```
/***** Includes *****/
#include <xc.h>
#include "configBits.h"
#include "lcd.h"

/***** Constants *****/
const char keys[] = "123A456B789C*0#D";
```

Sample: 03_KeypadLCD (4/10)

```
void main(void){

    // <editor-fold defaultstate="collapsed" desc="Machine Configuration">
    /********************************* PIN I/O ********************************/
    /* Write outputs to LATx, read inputs from PORTx. Here, all latches (LATx)
     * are being cleared (set low) to ensure a controlled start-up state. */
    LATA = 0x00;
    LATB = 0x00;
    LATC = 0x00;
    LATD = 0x00;
    LATE = 0x00;

    /* After the states of LATx are known, the data direction registers, TRISx
     * are configured. 0 --> output; 1 --> input. Default is  1. */
    TRISA = 0xFF; // All inputs (this is the default, but is explicated here for learning purposes)
    TRISB = 0xFF;
    TRISC = 0x00;
    TRISD = 0x00; // All output mode on port D for the LCD
    TRISE = 0x00;

    /************************** A/D Converter Module ***************************/
    ADCON0 = 0x00;  // Disable ADC
    ADCON1 = 0b00001111; // Set all A/D ports to digital (pg. 222)
    // </editor-fold>
```

Sample: 03_KeypadLCD (5/10)

```
    /* Initialize LCD. */
    initLCD();

    /* Main loop. */
    while(1){
        /* RB1 is the interrupt pin, so if there is no key pressed, RB1 will be
         * 0. The PIC will wait and do nothing until a key press is signaled. */
        while(PORTBbits.RB1 == 0){  continue;   }

        /* Read the 4-bit character code. */
        unsigned char keypress = (PORTB & 0xF0) >> 4;

        /* Wait until the key has been released. */
        while(PORTBbits.RB1 == 1){  continue;   }

        Nop();  // Apply breakpoint here to prevent compiler optimizations

        unsigned char temp = keys[keypress];
        putch(temp);    // Push the character to be displayed on the LCD
    }
}
```



These functions reside in the lcd.c and lcd.h files

Sample: 03_KeypadLCD (6/10)

Next Steps

- I strongly recommend looking at the sample project 01_CharacterLCD_1 on your own time. It uses `printf` from the **standard library** `stdio.h` to write to the character LCD
- We'll look briefly at how this works now

```
17  #include <stdio.h> // This gives us access to the standard print formatting library
```

A standard library is a collection of functionality that's made available across all implementations of a programming language

```
92          printf("Hello World! :)");
```

Using the standard library function `printf` to print the string "Hello World! :)"

Sample: 03_KeypadLCD (7/10)

`printf`

- Formats text
- Passes formatted output to a function called `putch`, which writes bytes to a specified output stream

What does printf() output to in an XC8 program?

The `printf()` function performs two main tasks: formatting of text, and printing this formatted text to stdout. The exact location of stdout is determined by a second function called `putch()`, which is called by `printf()` to output each character.

The `printf()` function performs the formatting and then calls a helper function, called `putch`, to send each byte of the formatted text. By default the `putch()` function is empty. It should be customised to suit the project at hand. By customizing the `putch` function you can have `printf` send data to any peripheral or location.

Explanation of `printf` in XC8, available online: http://microchipdeveloper.com/faq:29

If we write an implementation for `putch`, we can use `printf` on the PIC the exact same way we do when we write programs on our personal computers. Here, we direct the bytes from `printf` to the `lcdNibble` function

```
49  void putch(char data){
50      /* Sends a character to the display RAM for printing. */
51
52      RS = 1;
53      lcdNibble(data);
54      __delay_us(100);
55  }
```

Implementation of `putch` in the code, which explicitly passes data to `lcdNibble`

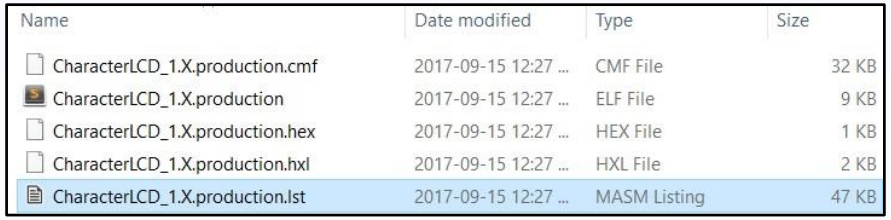
`printf` → `putch` → `lcdNibble`

Sample: 03_KeypadLCD (8/10)

Everyone navigate here: Code\01_CharacterLCD_1\CharacterLCD_1.X\dist\default\production

Open: CharacterLCD_1.X.production.lst

- **File type:** MASM Listing
- **Size:** 47 kB



Name	Date modified	Type	Size
CharacterLCD_1.X.production.cmf	2017-09-15 12:27 ...	CMF File	32 KB
CharacterLCD_1.X.production	2017-09-15 12:27 ...	ELF File	9 KB
CharacterLCD_1.X.production.hex	2017-09-15 12:27 ...	HEX File	1 KB
CharacterLCD_1.X.production.hxl	2017-09-15 12:27 ...	HXL File	2 KB
CharacterLCD_1.X.production.lst	2017-09-15 12:27 ...	MASM Listing	47 KB

Navigate to:

- **Line 146** of the assembly listing
- Here, main is defined

```
146 ;; *************** function _main ****************
147 ;; Defined at:
148 ;;   line 87 in file "main.c"
149 ;; Parameters:    Size  Location     Type
150 ;;   None
151 ;; Auto vars:     Size  Location     Type
152 ;;   None
153 ;; Return value:  Size  Location     Type
154 ;;                  1    wreg      void
155 ;; Registers used:
156 ;;   wreg, status,2, status,0, tblptrl, tblptrh, tblptru, cstack
157 ;; Tracked objects:
158 ;;   On entry : 0/0
159 ;;   On exit  : 0/0
160 ;;   Unchanged: 0/0
161 ;; Data sizes:     COMRAM   BANK0   BANK1   BANK2   BANK3   BANK4   
BANK10  BANK1
   +1  BANK12  BANK13  BANK14  BANK15
162 ;;      Params:         0       0       0       0       0       0
0       0
   +0       0       0       0       0
163 ;;      Locals:         0       0       0       0       0       0
0       0
   +0       0       0       0       0
164 ;;      Temps:          0       0       0       0       0       0
0       0
```

Sample: 03_KeypadLCD (9/10)

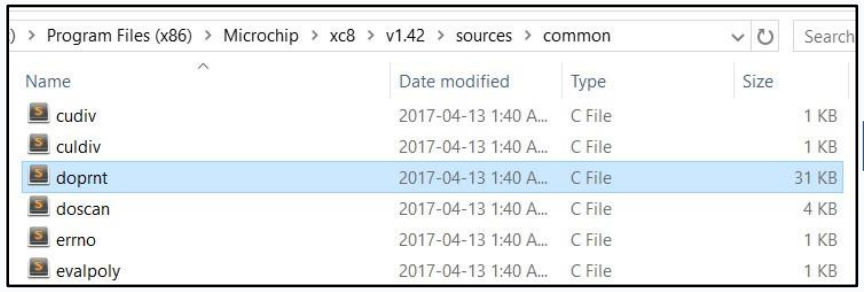
On **line 196**, the call to `printf` begins

```
195                                 ;main.c: 92: printf("Hello World! :)");
196  0010E4  0E01                      movlw low STR_1
197  0010E6  6E04                      movwf printf@f,c
198  0010E8  0E10                      movlw high STR_1
199  0010EA  6E05                      movwf printf@f+1,c
200  0010EC  EC57  F008                call  _printf ;wreg free
```

On **line 208**, we see the definition of `printf`, as well as where its definition can be found on disk

```
208 ;; *************** function _printf ****************
209 ;; Defined at:
210 ;;   line 464 in file "D:\Program Files (x86)\Microchip\xc8\v1.42\sources\common\doprnt.c"
211 ;; Parameters:    Size  Location     Type
212 ;;  f               2    3[COMRAM] PTR const unsigned char
213 ;;    -> STR_1(16),
214 ;; Auto vars:     Size  Location     Type
215 ;;  tmpval          4    0        struct .
216 ;;  cp              3    0        PTR const unsigned char
217 ;;  ap              2    5[COMRAM] PTR void [1]
218 ;;    -> ?_printf(2),
219 ;;  len             2    0        unsigned int
220 ;;  val             2    0        unsigned int
221 ;;  c               1    7[COMRAM] char
222 ;;  flag            1    0        unsigned char
223 ;;  prec            1    0        char
224 ;; Return value:  Size  Location     Type
```

Sample: 03_KeypadLCD (10/10)




```
463  #define pputc(c)    (putch(c) INCR_CNT)
464  printf(const char * f, ...)
465  {
```

Definition of `printf` begins on line 464, as specified in the assembly listing

```
536      while(c = *f++) {
537  #ifdef  ANYFORMAT
538          if(c != '%')
539  #endif  //ANYFORMAT
540          {
541              pputc(c);
542              continue;
543          }
544  #ifdef  ANYFORMAT
```

`printf` makes calls to `pputc` on line 541, as specified in the assembly listing. But `pputc` is simply `putch`

```
276                                 ;doprnt.c: 540: {
277                                 ;doprnt.c: 541: (putch(c) );
278  0010B8  5008                     movf  printf@c,w,c
279  0010BA  EC79  F008               call  _putch
```

Assembly call to `putch`, which we defined

This is just an illustration of what sorts of things you can find by tracing code!

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Sample: 07_GLCD

Sample: 07_GLCD (1/9)

Location

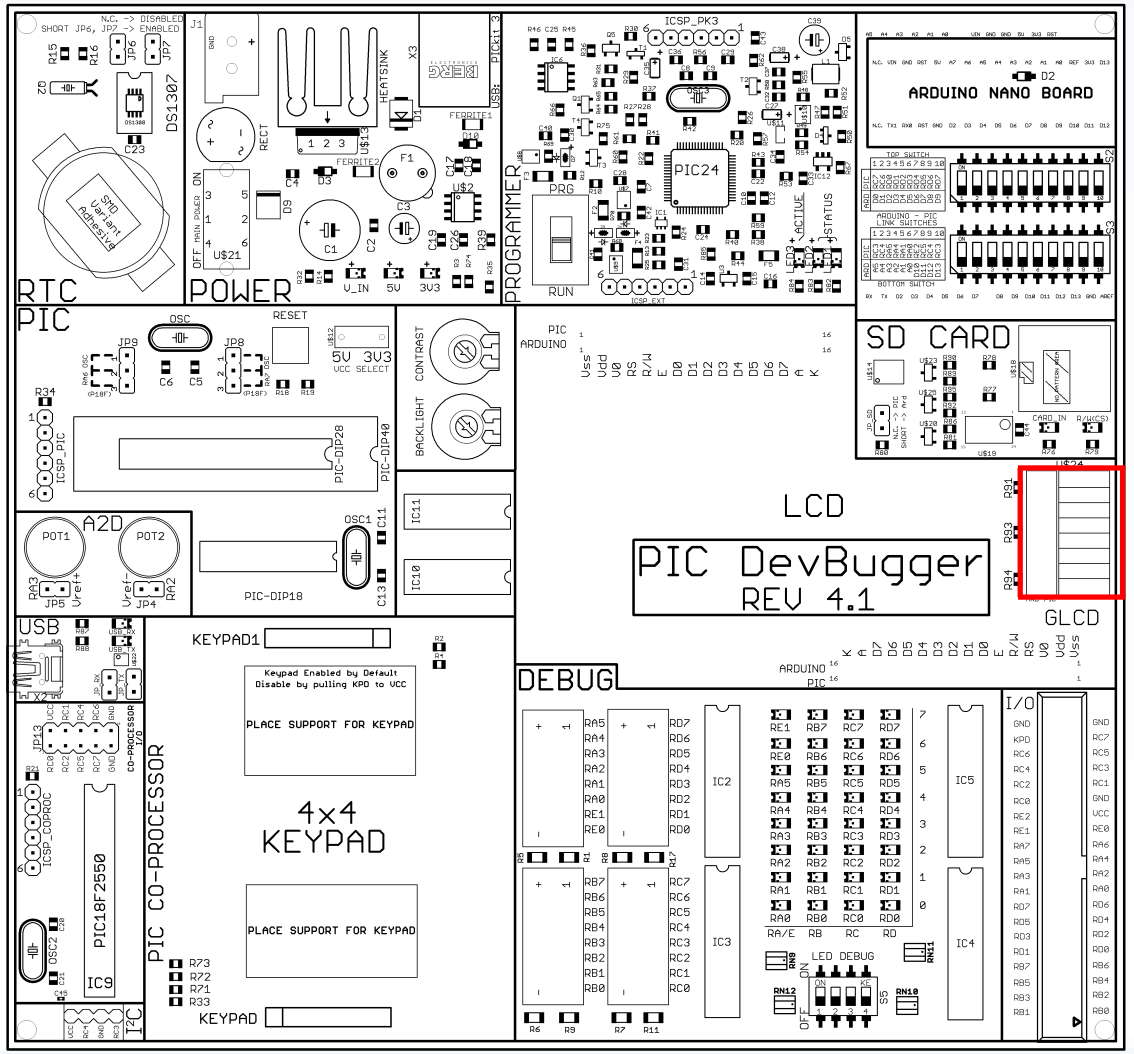
Code\07_GLCD\GLCD.X

What this program does

Demonstrates the fundamental graphics function of drawing rectangles, of which drawing pixels are a special case. This technique can be used to draw arbitrary bitmaps.

Preconditions

1. GLCD in PIC socket (bottom)

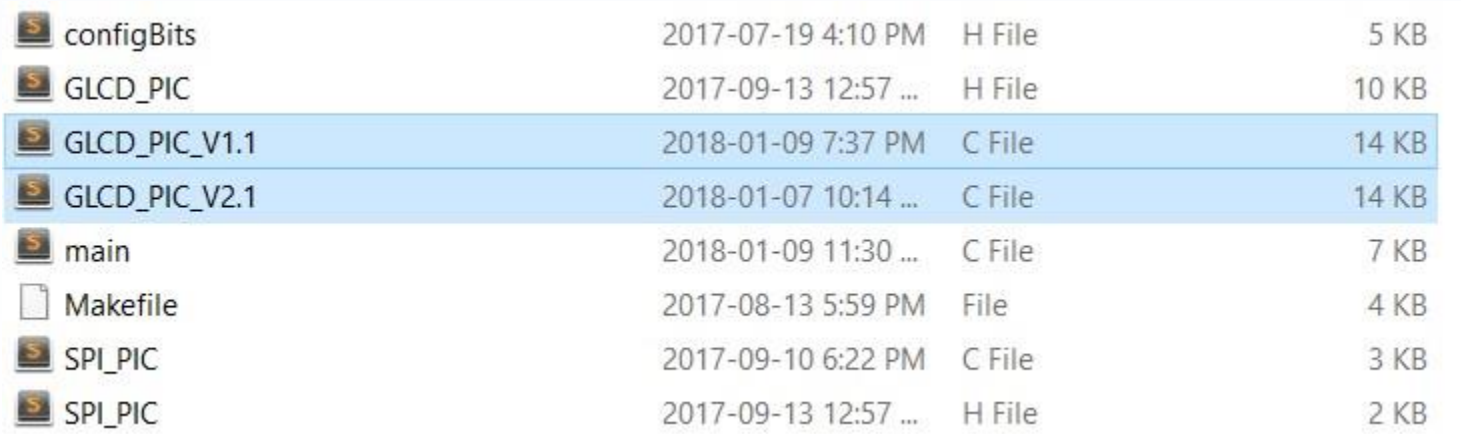


1

Sample: 07_GLCD (2/9)

What are the two GLCD_PIC_Vx.1 files?

- Look on the bottom of the GLCD. On the red PCB, you'll notice the version name "V1.1"
- If for some reason you replace the GLCD, it may have a different version name. The ones from Creatron are "V2.1"
- Different version numbers may **map different data RAM addresses to the display panel**. We'll see what this means in a minute, but at a high level it just means there are some small adjustments that need to be made in software



Sample: 07_GLCD (3/9)

```
* Description: Demonstration of GLCD initialization, and the fundamental
*              graphics "function", drawing rectangles. Drawing pixels are also
*              demonstrated as a special case of drawing rectangles.
*
* Preconditions:
*   1. GLCD is in a PIC socket
*/
```

Sample: 07_GLCD (4/9)

```
15  /***** Includes *****/
16  #include <xc.h>
17  #include "configBits.h"
18  #include "GLCD_PIC.h"
```

Sample: 07_GLCD (5/9)

```
20  void main(void) {
21
22      // <editor-fold defaultstate="collapsed" desc="Machine Configuration">
23      /********************************* PIN I/O *********************************/
24      /* Write outputs to LATx, read inputs from PORTx. Here, all latches (LATx)
25       * are being cleared (set low) to ensure a controlled start-up state. */
26      LATA = 0x00;
27      LATB = 0x00;
28      LATC = 0x00;
29      LATD = 0x00;
30      LATE = 0x00;
31
32      /* After the states of LATx are known, the data direction registers, TRISx
33       * are configured. 0 --> output; 1 --> input. Default is  1. */
34      TRISA = 0xFF; // All inputs (this is the default, but is explicated here for learning purposes)
35      TRISB = 0xFF;
36      TRISC = 0b10000000; /* RC3 is SCK/SCL (SPI/I2C),
37                           * RC4 is SDA (I2C),
38                           * RC5 is SDA (SPI),
39                           * RC6 and RC7 are UART TX and RX, respectively. */
```

Sample: 07_GLCD (6/9)

```
40      TRISD = 0b00000001; /* RD0 is the GLCD chip select (tri-stated so that it's
41                           * pulled up by default),
42                           * RD1 is the GLCD register select,
43                           * RD2 is the character LCD RS,
44                           * RD3 is the character LCD enable (E),
45                           * RD4-RD7 are character LCD data lines. */
46      TRISE = 0b00000100; /* RE2 is the SD card chip select (tri-stated so that
47                           * it's pulled up by default).
48                           * Note that the upper 4 bits of TRISE are special
49                           * control registers. Do not write to them unless you
50                           * read §9.6 in the datasheet */
51
52      /************************** A/D Converter Module ***************************/
53      ADCON0 = 0x00;  // Disable ADC
54      ADCON1 = 0b00001111; // Set all A/D ports to digital (pg. 222)
55      // </editor-fold>
```

Sample: 07_GLCD (7/9)

```
57      /* Initialize GLCD. */
58      initGLCD();
59
60      /* Declare local variables. */
61      unsigned char rotation = 0, x = 0, y = 0;
62
63      /* Main loop. */
64      while(1){
65          /* Fill screen with red. */
66          glcdDrawRectangle(0, GLCD_SIZE_HORZ, 0, GLCD_SIZE_VERT, RED);
67          __delay_ms(1000);
68
69          /* Fill screen with rainbow colors. */
70          glcdDrawRectangle(0, 18, 0, GLCD_SIZE_VERT, RED);
71          glcdDrawRectangle(18, 36, 0, GLCD_SIZE_VERT, ORANGE);
72          glcdDrawRectangle(36, 54, 0, GLCD_SIZE_VERT, YELLOW);
73          glcdDrawRectangle(54, 72, 0, GLCD_SIZE_VERT, GREEN);
74          glcdDrawRectangle(72, 90, 0, GLCD_SIZE_VERT, BLUE);
75          glcdDrawRectangle(90, 108, 0, GLCD_SIZE_VERT, INDIGO);
76          glcdDrawRectangle(108, 128, 0, GLCD_SIZE_VERT, VIOLET);
77
78          /* Demonstrate inversion. */
79          __delay_ms(500);
80          __INVON(); // Turn on color inversion
81          __delay_ms(500);
82          __INVOFF(); // Turn off color inversion
83          __delay_ms(500);
```

Sample: 07_GLCD (8/9)

```
85          /* Draw white in the four corners pixels. */
86          glcdDrawPixel(0, 0, WHITE);
87          __delay_ms(500);
88          glcdDrawPixel(GLCD_SIZE_HORZ, 0, WHITE);
89          __delay_ms(500);
90          glcdDrawPixel(GLCD_SIZE_HORZ, GLCD_SIZE_VERT, WHITE);
91          __delay_ms(500);
92          glcdDrawPixel(0, GLCD_SIZE_VERT, WHITE);
93          __delay_ms(500);
94
95          /* Place 128 pixels in accordance with some arbitrarily-chosen math. */
96          for(x = 0; x < GLCD_SIZE_HORZ; x++){
97              if(x % (GLCD_SIZE_VERT/16) == 0){
98                  glcdDrawPixel(GLCD_SIZE_VERT - (x * 8), (x % 3) * 33, x * 2048);
99              }
100             else{
101                 glcdDrawPixel(x, GLCD_SIZE_VERT - x, x * 2048);
102             }
103             __delay_ms(10);
104         }
105
106         /* Fill screen with blue/pink pattern, pixel-by-pixel. */
107         for(y = 0; y < GLCD_SIZE_VERT; y++){
108             for(x = 0; x < GLCD_SIZE_HORZ; x++){
109                 glcdDrawPixel(x, y, x*y*16);
110             }
111         }
```

These patterns really were chosen randomly and they just so happened to look neat!

Sample: 07_GLCD (9/9)

```
145         /* Rotate display at the end of each iteration. */
146         switch(rotation % 4){
147             case 0:
148                 glcdSetOrigin(ORIGIN_TOP_RIGHT);
149                 break;
150             case 1:
151                 glcdSetOrigin(ORIGIN_BOTTOM_RIGHT);
152                 break;
153             case 2:
154                 glcdSetOrigin(ORIGIN_BOTTOM_LEFT);
155                 break;
156             case 3:
157                 glcdSetOrigin(ORIGIN_TOP_LEFT);
158                 break;
159         }
160         rotation++;
```

You can rotate the GLCD via software

Agenda

2:10 – 3:30

- The big picture: microcontroller development resources & high-level example
- Programming PIC microcontrollers using C
- Development board orientation
- Exercise: I/O
- Exercise: Interrupts

3:30 – 5:00

- Libraries and APIs
- Interfacing with devices
- Exercise: RTC library
- Exercise: debugging

5:00 – 6:00

- Sample: 00_PortTest
- Sample: 03_KeypadLCD
- Sample: 07_GLCD

Also: 10 minute break at the turn of each hour

Recap

Goals for today

- Development board
  - Gain experience using the modules on the development board by running through various sample programs
  - Understand capabilities & limitations
- PIC Microcontrollers
  - Use the in-circuit debugger to trace code execution
  - Learn how to view disassembly
  - Learn how to use interrupts in C and the keywords `volatile and const`
  - Learn programming concepts useful for embedded software
- Other
  - Learn how to navigate datasheets and utilize them to develop low-level code

Wrap-up

Thoughts for the future…

Traditionally, AER201 students have been very involved in growing the course. Be on the lookout for any opportunities where you might be able to contribute to the course.

Acknowledgements

Thank you Prof. Emami for providing the opportunity to work on improving the learning experience in AER201.

Questions? Contact me.

Please do not hesitate to contact me at anytime.

- Facebook
- LinkedIn
- ty.gamvrelis@mail.utoronto.ca
- ty.gamvrelis@gmail.com

Thank you!