CMPE-013/L

Introduction to “C” Programming

Max Dunne
Late days

Bottle boats

\[ \frac{1}{3} \text{ of one file} \quad 3 \]
Hardware Peripherals

- Digital pins
- Timers
- ADC
Hardware Peripherals

- Communications
- Pin change notification
- DMA
- Output compare
- Input capture
- Digital pins
- Timers
- ADC

UART SPI I²C

Direct Memory Access

PWM
Hardware Peripherals

Special function registers

- Peripherals are controlled by hardware registers
  - Referred to as Special Function Registers (SFRs)
- Memory-mapped unsigned 16-bit integers
- Accessible as global variables
  - Included from the `<xc.h>` header
Hardware Peripherals
Special function registers

- Declaration of Interrupt FlagS 0 register
- **volatile** qualifier indicates value can change outside of the code in this program
- **__attribute__** is a compiler directive to specify additional compiler parameters
  - **__sfr__** indicates that it's a memory-mapped SFR

**SFR: IFS0**

```c
extern volatile unsigned int IFS0 __attribute__((__sfr__));
```
Hardware Peripherals

Digital pins 0–5V

- Voltage
  - High ✓
  - Low ✓

- Direction
  - Input
  - Output

- Polling interface
  - ask
Hardware Peripherals

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

Synchronization
Hardware Peripherals

Digital pins

- **TRIS** – TRIState register. Sets pin direction.
  - Pin is an output when corresponding bit is 0, input when corresponding bit is 1
- **LAT** – LATch register. Sets pin value/gets pin's desired value
  - Desired output value of the pin
- **PORT** – PORT register. Sets pin value/gets pin's **actual** value
  - Actual value of the pin
Hardware Peripherals

Digital pins
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx

RD LATx

WR PORTx

RD PORTx

Synchronization

I/O Cell

I/O pin
Hardware Peripherals

Timers

• Multiple 16-bit timers
  – 5 total
• Interrupt-based
  – ISR is called every X seconds
• Configurable periodicity
  – Range from 20MHz to 305Hz
Hardware Peripherals

Timer SFRs

• TMRx – Timer counter
  – uint16
  – Ticks every instruction clock cycle (20MHz)

• PRx – Timer x prescalar
  – Limit for when to trigger the timer interrupt.
  – Valid values are [1, INT16_MAX]
  – 0 is a special value, disables peripheral.
Hardware Peripherals

Timers

• To modify timer interrupt period, set PRx register.

• To set a period of the timer interrupt:
  – 20MHz / PRx = periodicity

• PRx of 20000 -> 1kHz interrupts
Hardware Peripherals

Timers
Hardware Peripherals

Timers

65535

TMRx
Hardware Peripherals

Timers

PRx

TMRx

event
Hardware Peripherals

Timers
Hardware Peripherals

Timers

PRx → CPU

TMRx

event

Interrupt()
Hardware Peripherals

ADC

• Analog to Digital Converter
• Measures the voltage of a processor pin
• Used to read analog sensors
  – Temperature
  – Power
  – Battery levels
Hardware Peripherals

ADC SFRs

- ADCxBUFy: Buffer for holding samples
  - x is the ADC
  - y is the sample [0, 7]
  - 16-bit unsigned value
    - Only lowest 10-bits matter
Hardware Peripherals

ADC

- The input signal is continuously sampled
- Every 8th sample triggers an interrupt
Hardware Peripherals

ADC

- Voltage range from $V_{\text{ref}^-}$ to $V_{\text{ref}^+}$
  - 0V to 3.3V
- Values are unsigned 10-bits, from $[0, 1023]$
- Units are in $V_{\text{ref}} / 1023 = 0.0032V$
Hardware Peripherals

ADC
Hardware Peripherals

ADC

event

ADC1BUF0 = 2
ADC1BUF1 = 146
ADC1BUF2 = 288
ADC1BUF3 = 420
ADC1BUF4 = 563
ADC1BUF5 = 691
ADC1BUF6 = 829
ADC1BUF7 = 987
# Hardware Peripherals

## ADC

<table>
<thead>
<tr>
<th>ADC1BUF0</th>
<th>ADC1BUF1</th>
<th>ADC1BUF2</th>
<th>ADC1BUF3</th>
<th>ADC1BUF4</th>
<th>ADC1BUF5</th>
<th>ADC1BUF6</th>
<th>ADC1BUF7</th>
</tr>
</thead>
<tbody>
<tr>
<td>950</td>
<td>600</td>
<td>100</td>
<td>65</td>
<td>81</td>
<td>93</td>
<td>107</td>
<td>122</td>
</tr>
</tbody>
</table>
Event-driven Programming

Events
Event loop
Event-driven Programming

- Real-time programming paradigm
- Build around the concept of events
- Events are then handled by specific event handlers
- Works well with systems with multiple inputs that need to be handled in a timely manner
  - Real-time system
- Integrates well with interrupts
Event-driven Programming

Events

- Any temporally-short sensor occurrence
- Usually the derivative of a signal
  - Button was pressed down
  - The mouse was clicked
  - This sensor value changed
  - This interrupt triggered
Event-driven Programming

The event loop

- A continual loop that checks for and processes events
- The core of an event-driven program
Event-driven Programming

The event loop

```c
{
    while (1) {
        // Check for events

        // Process events
    }
}
```
Event-driven Programming

The event loop

```c
{
    while (1) {
        // Check for event 1
        // Check for event 2
        ...
        // Check for event n

        // Process event 1
        // Process event 2
        ...
        // Process event n
    }
}
```
Event-driven Programming

Event priorities

```c
{
    while (1) {
        // Check for event 3
        // Process event 3

        // Check for event 1
        // Process event 1

        // Check for event 2
        // Process event 2
    }
}
```
Event-driven Programming

Real-world example

```c
{
    while (1) {
        if (buttonsEvent) {
            // Update fixed LED mask
        }
        if (adcEvent) {
            // Update OLED
        }
        if (timerEvent) {
            // Update bouncing LED mask
        }
        if (ledEvent) {
            // Update LEDs
        }
    }
}
```
Event-driven Programming

Real-world example

```c
static uint8_t buttonsEvent;

void main()
{
    while (1) {
        if (buttonsEvent) {
            // Event loop
        }
    }
}

void _ISR Timer1Int(void)
{
    buttonsEvent = ButtonsCheckEvents();
    IFS0 &= ~(1 << 3);
}
```
CMPE-013/L

RPN (stack)

Maxwell James Dunne
Stack Operation

Pop

Push

3
4
3
Stack
Struct Layout

struct Stack {
    float stackItems[STACK_SIZE];
    int currentItemIndex;
    uint8_t_t initialized;
};
Stack Functions

- `void StackInit(struct Stack *stack)`
  ```c
  struct stack my_stack, *stackptr;
  stackptr = &my_stack;
  SI(stackptr)
  SI(&my_stack)
  ```
- `int StackPop(struct Stack *stack, float *value);`

  `STANDARD ERROR SUCCESS`
RPN

- Reverse Polish Notation
  - Calculator that uses a stack as a scratchpad
  - We will read strings from the keyboard and parse

- Simple Examples
  - $4 \ 3 \ + \ = \ 4 \ + \ 3$
  - $1 \ 6 \ -3 \ -\ + \ =$
RPN
Error/End Conditions

• The RPN calculation is complete when the whole string has been processed.
• If only one item is left on the stack the string entered was valid.
• It can also fail in a few ways
  – Not one Item left on stack at end
  – Invalid RPN Token
  – Operator without at least 2 items on the stack
  – Stack is full (Specific to the class, not RPN in general)
1 + 4

RPN

\[ 8 2 / \ 8 \div 2 \]

Complex Example

\[
(1 + 4) \ast (6 - 4) / 8” \text{ as } “1 4 + 6 4 - \ast 8 /
\]

1 + 4 = 5

6 - 4 = 2

5 \ast 2 = 10

10 \div 8 = 1.25
Development Tips

test stack independently

my stack

my stack, initialized

mystack.csi = 19
"14 + 364 = \" \\
strtok(str, \" \") for
"1" "4" "+" "36" "4"
atoi("1") -> 1
14 3 64

// gets(str) ->
// str = "14 3 64"
fgets(
scanf("%\%s %\%s")
);