Late days

Bottles boats

$\frac{1}{3}$ of one file 3
Hardware Peripherals

Digital pins
Timers
ADC
Hardware Peripherals

- Communications
- Pin change notification
- DMA
- Output compare
- Input capture
- Digital pins
- Timers
- ADC
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
- \texttt{volatile} qualifier indicates value can change outside of the code in this program
- \texttt{__attribute__} is a compiler directive to specify additional compiler parameters
  - \texttt{__sfr__} indicates that it's a memory-mapped SFR

\begin{verbatim}
extern volatile unsigned int IFS0 __attribute__((__sfr__));
\end{verbatim}
Hardware Peripherals

Digital pins 0-5V

- Voltage
  - High ✓
  - Low ✓

- Direction
  - Input
  - Output

- Polling interface

3.3 * 15 = 49.5
3.3 * 8 = 26.4
Car company

$P_{\text{ ISIL }} = 100,000$
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx

WR PORTx

RD LATx

RD PORTx

Synchronization

I/O pin

I/O Cell

ODCx

TRISx

LATx

Gabriel Hugh Elkaim – Winter 2015

CMPE-013/L: “C” Programming
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
LAT = 0.1
Port reads 0
Hardware Peripherals

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

Synchronization
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

Digital pins

Dedicated Port Module

RD TRIx

WR TRIx

WR PORTx

RD PORTx

Synchronization

I/O pin

I/O Cell

CMPE-013/L: “C” Programming

CMPE-013/L: “C” Programming
Hardware Peripherals

Timers

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

FRC

Quartz Oscillator

PLL

8.0000000000MHz

20MHz

laser trimmed resistors
Hardware Peripherals

Timer SFRs

- **TMRx** – Timer counter
  - `uint16`
  - Ticks every instruction clock cycle (20MHz) \( \div 2 \)

- **PRx** – Timer x prescaler
  - Limit for when to trigger the timer interrupt.
  - Valid values are `[1, INT16_MAX]`
  - 0 is a special value, disables peripheral.
$\frac{20\text{MHz}}{2} = \frac{2}{2}$
Hardware Peripherals

Timers

• To modify timer interrupt period, set PRx register.

• To set a period of the timer interrupt:
  – \( 20\text{MHz} / \text{PRx} = \text{periodicity} \)

• PRx of 20000 -> 1kHz interrupts
Hardware Peripherals

Timers
Hardware Peripherals

Timers

3.25 milliseconds
Hardware Peripherals

Timers

\[ \frac{1}{20 \text{MHz}} = 50 \text{ns} \]

\[ NR = 20k \]
Hardware Peripherals

Timers

```
PRx
```

```
TMRx
```

event

```
CPU
```

Gabriel Hugh Elkaim – Winter 2015
Hardware Peripherals

Timers

- PRx
- TMRx
- CPU
  - Interrupt()

Event:
- 12
- 34
- 5
- Different
- 4/5
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

guess and check

input match

Gabriel Hugh Elkaim – Winter 2015
Hardware Peripherals

ADC

- 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

while (1)
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
    }
}
```
{ 
    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

```
static uint8_t buttonsEvent;

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

void _ISR Timer1Int(void)
{
    buttonsEvent = ButtonsCheckEvents();
    IFS0 &= ~(1 << 3);
}
```
while (1)

if (event1) // process event1
  continue;

.
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Bit manipulation

- Bit masking
- Bit flags
- Bit fields
Bit manipulation

Bit packing

- Data is commonly packed into larger unsigned integers on embedded systems
- Generally a tie in to hardware or when space is critical
  - Hardware
  - Storage
  - Binary formats

32 bits
1 bit
Bit manipulation

Bit packing

C1CTRL1 - dsPIC33EP256MC502
Bit manipulation

Bit masks

Example

// Abort the current CAN message transmission
CICTRL1 = CICTRL1 | 0x1000;
Bit manipulation

Bit masks

Example

// Disable CAN message timestamping
ClCTRL1 = ClCTRL1 & 0xFFF7;
Bit manipulation

Bit masks

\[ 1 << 3 \quad 1000 \]

Example

// Disable CAN message timestamping
ClCTRL1 &\(=\) \(\sim(1 << 3)\);

\[ \begin{array}{cc}
123 & 1000 \\
\end{array} \]
Bit manipulation

Bit masks

• A constant that indicates which bits are relevant for a given variable
• One bits indicate significant bits
• Zero bits indicate ignore bits
#define CxCTRL1_MASK_CANCAP (1 << 3)

// Disable CAN message timestamping
C1CTRL1 &= ~CxCTRL1_MASK_CANCAP;

C1CTRL1 &= Cx

BIT_0
Bit manipulation

Bit masking

- Setting a bit
  - ORing with 1
    - `C1CTRL1 |= CxCTRL1_MASK_CANCAP;`

- Clearing a bit
  - ANDing with 0
    - `C1CTRL1 &= ~CxCTRL1_MASK_CANCAP;`

- Toggling a bit
  - XORing with 1
    - `C1CTRL1 ^= CxCTRL1_MASK_CANCAP;`

LED
Bit manipulation

Bit masking

- Setting a bit can OR multiple masks together

```c
enum {
    BUTTON_EVENT_1UP = 0x01,
    BUTTON_EVENT_2UP = 0x04
};

{ 
    uint8_t event = (BUTTON_EVENT_1UP | BUTTON_EVENT_2UP); 
}
```
Bit manipulation

Bit masking

• Getting a bit
  – ANDing with 1

Example

```c
#define CxCTRL1_MASK_CANCAP (1 << 3)

// If CAN message timestamping is enabled
if (C1CTRL1 & CxCTRL1_MASK_CANCAP == CxCTRL1_MASK_CANCAP) {
    ...
}
```
Bit manipulation

Bit masking

- Getting a bit
  - ANDing with 1

Example

```c
#define CxCTRL1_MASK_CANCAP (1 << 3)

// If CAN message timestamping is enabled
if (C1CTRL1 & CxCTRL1_MASK_CANCAP) {
    ...
}
```
Bit manipulation

Bit masking

Example

```c
// Retrieve the operating mode of the CAN hardware
int opmode = (C1CTRL1 & 0xE0) >> 5;
```

```c
\|
\|
\|
```
Bit Fields are (unsigned) int members of structures that occupy a specified number of adjacent bits from one to sizeof(int). They may be used as an ordinary int variable in arithmetic and logical operations.

- Bit Fields:
  - Are ordinary members of a structure
  - Have a specified bit width
  - Provide bit access to a variable without masking operations
Bit Fields

- Bit Fields:
  - May only be integers (short, long, __, long long)
    - No larger than the base type
  - Unsigned by default, but may be signed
  - Non-portable across architectures/compilers!
    - Just like regular structs
Bit Fields
How to Create a Bit Field

Syntax

```c
struct StructName {
    ((un)signed) int memberName1: bitWidth;
    ...
    ((un)signed) int memberName n: bitWidth;
}
```

Example

```c
struct ByteBits {
    unsigned int a: 1;
    long b: 1;
    short c: 2;
    unsigned d: 1;
    long long e: 3;
};
```
Bit Fields
How to Use a Bit Field

Example

typedef struct {
    unsigned int a: 1;
    long b: 1;
    short c: 2;
    unsigned d: 1;
    long long e: 3;
} ByteBits;

ByteBits x;

bitfield struct may be declared normally or as a typedef
Bit Fields
How to Use a Bit Field

Example

```c
struct ByteBits {
    unsigned a: 1;
    unsigned b: 1;
    unsigned c: 2;
    unsigned d: 1;
    unsigned e: 3;
} x;

int main(void)
{
    x.a = 1;    // x.a may contain values from 0 to 1
    x.b = 0;    // x.b may contain values from 0 to 1
    x.c = 0b10; // x.c may contain values from 0 to 3
    x.d = 0x0;  // x.d may contain values from 0 to 1
    x.e = 7;    // x.e may contain values from 0 to 7
}
```
Bit Fields
Microchip's SFRs

Example

```c
// SFR register declaration
extern volatile unsigned int C1CTRL1 __attribute__((__sfr__));

// SFR bitfield declaration
typedef struct {
    unsigned WIN : 1;
    unsigned : 2;
    unsigned CANCAP : 1;
    unsigned : 1;
    unsigned OPMODE : 3;
    unsigned REQOP : 3;
    unsigned CANCKS : 1;
    unsigned ABAT : 1;
    unsigned CSIDL : 1;
} C1CTRL1BITS;
extern volatile C1CTRL1BITS C1CTRL1bits __attribute__((__sfr__));
```
win: 1
Bit Fields
How to Use a Bit Field

Example

```c
int main(void)
{
    // Abort the current CAN message transmission
    C1CTRL1 |= 0x1000;
    C1CTRL1bits.ABAT = 1;

    // Disable CAN message timestamping
    C1CTRL1 &= 0xFFF7;
    C1CTRL1bits.CANCAP = 0;

    // If CAN message timestamping is enabled
    if (C1CTRL1 & 0x0008) {
        if (C1CTRL1bits.CANCAP) {
            ...
        }
    }
}
```
Bit Fields

Signed values

Example

typedef struct {
    signed int a: 3;
    short b: 2;
    signed short c: 2;
    long long d: 3;
} ByteBits;

ByteBits x;
Bit Fields

Signed values

Example

typedef struct {
    signed int     a: 3;
    short          b: 2;
    signed short   c: 1;
    long long      d: 3;
} ByteBits;

ByteBits x;
typedef struct {
    signed int    a: 3;
    short         b: 2;
    signed short  c: 1;
    long long     d: 3;
} ByteBits;

ByteBits x;
Bit Fields

Maximum bitness

Example

typedef struct {
    signed short  a: 3;
    short         b: 2;
    signed short  c: 1;
    short         d: 3;
} ByteBits;

ByteBits x;