CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Dynamic Memory

malloc()
free()
Dynamic Memory

Memory leaks

- If pointers returned by `malloc()` are lost, that memory is then "lost"
- Easy to do because this may not crash your program, possibly only causing errors over long periods of time

Example

```c
void MyFunc(void)
{
    Complex *x = malloc(sizeof(Complex));
    ...
}
```
Dynamic Memory

Memory leaks

• So for every pointer obtained from `malloc()`, there should be an equivalent `free()` at some point

```
void MyFunc(void)
{
    Complex *x = malloc(sizeof(Complex));
    ...
    free(x);
}
```

```
valgrind
```
Dynamic Memory

When to use the Heap

• For unknown amounts of data
  – Arrays are always fixed-length at compile time

• When data needs to be accessible outside of the scope it was created in
  – Pointers need to be passed around
Pointers

Pointers to pointers
Pointers
Pointers to pointers

- Since pointers can point to any valid datatype, they can also point to other pointers

- No limit on levels of indirection
Pointers

Pointers to pointers

**Example**

```c
{  
    int x = 6;
    int *y = &x;
    int **z = &y;
    printf("%d\n", **z);
}
```

**Output**

6

**32-bit Data Memory (RAM)**

```
Address   0x3F50  0x3F54  0x3F58  0x3F5C
```

```
X  0000 0006  
Y  0000 3F54  
Z  0000 3F58  
```

Maxwell James Dunne
Pointers
Passing by reference, again

• Passing by reference only allows persistently changing the value 1 level of indirection from the pointer and further
  – If a pointer is passed to a function, the data it points to can be altered
  – If a pointer-to-a-pointer is passed, the pointer it points to and the data that pointer points to can be altered
Pointers
Passing by reference, again

Example interrupt

```c
void MyFunc(int *x)
{
    *x = 6;
}

int main(void)
{
    int myInt;
    int *myIntPtr = &myInt;
    MyFunc(&myIntPtr);
}
```
Pointers
Passing by reference, again

Example interrupt

```c
void MyFunc(int **x)
{
    *x = malloc(sizeof(int));
    if (*x) {
        **x = 6;
    }
}

int main(void)
{
    int *myInt;
    MyFunc(&myInt);
}
```
void pointer

void *p = NULL;

foo = (complex)p

p++
 Enums
Enumerations

**Definition**

Enumerations are integer data types that you can create with a limited range of values. Each value is represented by a symbolic constant that may be used in conjunction with variables of the same enumerated type.

- **Enumerations:**
  - Are unique integer data types
  - May only contain a specified list of values
  - Values are specified as symbolic constants
Enumerations
How to Create an Enumeration Type

- Creates an ordered list of constants
- If unspecified, each label’s value is one greater than the previous label

**Syntax**

```c
enum typeName {label_0, label_1,...,label_n}
```

Where compiler sets `label_0 = 0`, `label_1 = 1`, `label_n = n`

**Example**

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
```

Label Values:

- SUN = 0
- MON = 1
- TUE = 2
- WED = 3
- THR = 4
- FRI = 5
- SAT = 6
Enumerations
How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

Syntax

```c
enum typeName {label_0 = const_0,...,label_n}
```

Where compiler sets `label_0 = const_0, label_1 = (const_0 + 1), ...`

Example

```c
enum people {Rob, Steve, Paul = 7, Bill, Gary};
```

Label Values:

```
Rob = 0, Steve = 1, Paul = 7, Bill = 8, Gary = 9
```
Enumerations
How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

Syntax

```c
enum typeName { label_0 = const_0, ..., label_n }
```

Where compiler sets `label_0 = const_0`, `label_1 = (const_0 + 1)`, ...

Example

```c
enum people { Rob = 'a', Steve, Paul, Bill, Gary };
```

Label Values:

Rob = 'a', Steve = 'b', Paul = 'c', Bill = 'd', Gary = 'e'
Enumerations
How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

Syntax
```
enum typeName { label_0 = const_0, ..., label_n }
```
Where compiler sets `label_0 = const_0, label_1 = (const_0 + 1), ...`

Example
```
enum people { Rob = -4, Steve, Paul, Bill, Gary };
```
Label Values:
```
Rob = -4, Steve = -3, Paul = -2, Bill = -1, Gary = 0
```
Enumerations

How to Declare an Enumeration Type Variable

- Declared along with type:

```c
enum typeName {const-list} varname_1,...;
```

- Declared independently:

```c
enum typeName varName_1,...,varName_n;
```

Example

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT} today;

enum weekday day; // day is a variable of type weekday
```
Enumerations
How to Declare a ‘Tagless’ Enumeration Variable

• No type name specified:

Syntax

```c
enum {const-list} varName1, ..., varName_n;
```

• Only variables specified as part of the `enum` declaration may be of that type

• No type name is available to declare additional variables of the `enum` type later in code

Example

```c
enum {SUN, MON, TUE, WED, THR, FRI, SAT} Today;
```
Enumerations
How to Declare an Enumeration Type with `typedef`

- Variables may be declared as type `typeName` without needing the `enum` keyword

Syntax
```
typedef enum {const-list} typeName;
```

- The enumeration may now be used as an ordinary data type (compatible with `int`)

Example
```
typedef enum {SUN, MON, TUE, WED, THR, FRI, SAT} Weekday;

Weekday day; // Variable of type weekday
```
Enumerations

How to Use an Enumeration Type Variable

If enumeration and variable have already been defined:

Syntax

\[ \text{varName} = \text{label}_n; \]

- The labels may be used as any other symbolic constant
- Variables defined as enumeration types must be used in conjunction with the type’s labels or equivalent integer

Example

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
enum weekday day;

day = WED;
day = 6; // May only use values from 0 to 6
if (day == WED) {
    ...
```
Enumerations

Proper formatting

Example

typedef enum {
    SUN, // comments
    MON, // class
    TUE,
    WED, // class
    THR,
    FRI,
    SAT
} Weekday;

Weekday day = WED;
Enumerations

Proper formatting

Example

typedef enum {
    SUN,
    MON,
    TUE,
    WED,
    THR,
    FRI,
    SAT
} Weekday;

Weekday day = 3; // No compilation warning/error
Enumerations
Datatype usage

Example

typedef enum {
    SUN = 0,
    MON,
    TUE,
    WED,
    THR,
    FRI,
    SAT
} Weekday;

void PrintDayName(Weekday d)
{
    if (d == SUN) {
        printf("Sun\n");
    }
    ...
}

PrintDayName(WED); // No compilation warning/error
Enumerations

Why enumerations?

- Enumerations are a proper datatype as well as the possible values for them
- Some compile-time checking
- Doesn't do text replacement, done during the compiler stage
- Use for a group of related values
Interrupts
Interrupts

- High-priority alerts that an event requires immediate attention
- Generally interrupts can be assigned priorities
- Event is handled by an Interrupt Service Routine (ISR)

short fast
Interrupts

- ISR is a special function that is written by the developer, but called directly by the processor.
- ISRs have no inputs or outputs.
  - All processing through global/module-level variables.
- ISRs are written a specific way and the processor is told they have been implemented by the compiler/developer.
Interrupts

Traps

• Software interrupts are generally referred to as exceptions or traps

• Examples:
  – Division by zero
  – Invalid address dereference
  – Debugging breakpoint
  – Stack overflow

NULL Pointer


Example interrupt

```c
void _ISR IsrName(void)
{
    // Process data from the interrupt
    // Store results in global/module variable
    // Clear interrupt flag
}
```
Example interrupt

```c
void _ISR Uart1TxInterrupt(void)
{
    // Stall until transmission finishes
    while (!U1STAbits.TRMT);

    // Continue transmitting next batch of data
    Uart1StartTransmission();

    // Clear interrupt flag
    IFS0bits.U1TXIF = 0;
}
```
Interrupts

Calling

Example program

```c
int main(void)
{
    int x = 20;
    int y;
    y = x / 2;
}
```

Interrupt: UART1 Post-transmission

```c
void _ISR_U1TXInt(void)
{
    IFS0bits.U1TXIF = 0;
}
```
Interrupts

Calling

Example program:

```c
int main(void)
{
    int x = 20;
    int y;
    _U1TXInt();
    y = x / 2;
}
```

Interrupt: UART1 Post-transmission:

```c
void _ISR _U1TXInt(void)
{
    IFS0bits.U1TXIF = 0;
}
```
Interrupts

- Interrupts are important events that happen in real-time
- ISRs are the functions that handle these events
- ISRs are called outside of regular program execution order
Interrupt Vector Table

UART TX
CMPE-013/L

Introduction to “C” Programming

Max Dunne
Hardware Peripherals

- Digital pins
- Timers
- ADC
Hardware Peripherals

- Communications: UART, SPI, I^2^C
- Pin change notification
- DMA: Direct Memory Access
- 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
• `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

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

- Voltage
  - High: 1 (2.7 - 3.3 V)
  - Low: 0 (0 - 0.5 V)
- Direction
  - Input
  - Output
- Polling interface
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

Dedicated Port Module

- RD TRISx
- WR TRISx
- WR LATx
- WR PORTx
- RD LATx
- RD PORTx

Synchronization

I/O Cell

I/O pin
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx

RD LATx

RD PORTx

Synchronization

I/O Cell

I/O pin

D

Q

Q

Q

Q

Q

Q

Q

Q

ODCx

TRISx

LATx

CMPE-013/L: “C” Programming

CMPE-013/L: “C” Programming

Gabriel Hugh Elkaim – Winter 2015
Hardware Peripherals

Digital pins
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:
  - $20\text{MHz} / \text{PRx} = \text{periodicity}$
- PRx of 20000 $\rightarrow$ 1kHz interrupts

\[
\frac{20\text{MHz}}{1, 2, 4, 8, \ldots, 256}
\]
Hardware Peripherals

Timers
Hardware Peripherals

Timers

$65535$

TMRx
Hardware Peripherals

Timers

PRx

TMRx

event
Hardware Peripherals

Timers

PRx → CPU

TMRx

event
Hardware Peripherals

Timers

PRx ———> TMRx ———> event

———> CPU ———> 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

\[
\frac{3.3V}{1024} = 0.00322V
\]

0 - 1023

0 - 3.3V
Hardware Peripherals

ADC

- The input signal is continuously sampled
- Every 8\textsuperscript{th} 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

Diagram of ADC components and connections.
\[
\begin{array}{c}
-2.5V \\
-2.5V
\end{array}
\]

\[
\begin{array}{c}
1.65V \\
2.475
\end{array}
\]
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>2</td>
<td>146</td>
<td>288</td>
<td>420</td>
<td>563</td>
<td>691</td>
<td>829</td>
<td>987</td>
</tr>
</tbody>
</table>
Hardware Peripherals

ADC

ADC1BUF0 = 950
ADC1BUF1 = 600
ADC1BUF2 = 100
ADC1BUF3 = 65
ADC1BUF4 = 81
ADC1BUF5 = 93
ADC1BUF6 = 107
ADC1BUF7 = 122
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

{
  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

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

        // Check for event 1
        // Process event 1

        // Check for event 2
        // Process event 2
    }
}
```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);
}
```