Hardware Peripherals

Digital pins
Timers
ADC
Hardware Peripherals

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

Serial SPI I²C
Direct Memory Access
PWM
Timing 1ns 1.25ns

Gabriel Hugh Elkaim – Winter 2015
CMPE-013/L: “C” Programming
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

pic24

m.154k
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

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

Digital pins

- **Voltage**
  - High: 3.3V
  - Low: 0
- **Direction**
  - Input: 0 - 3.3V
  - Output: 0, 1
- **Polling interface**
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
programm
PC
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

I/O Cell

I/O pin

WR LATx
WR PORTx
RD LATx
RD PORTx

Synchronization

LATx

TRISx

WR TRISx

RD TRISx

ODCx

CK

EN

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q

Q
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx

RD LATx

WR PORTx

RD PORTx

I/O Cell

I/O pin

Synchronization
Hardware Peripherals

Timers

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

16 16
32-bit
Hardware Peripherals

Timer SFRs

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

- **PRx** – Timer x prescaler
  - Limit for when to trigger the timer interrupt.
  - Valid values are `[1, \text{UINT16\_MAX}]`
  - 0 is a special value, disables peripheral.

\[
\frac{20 \text{MHz}}{65536} = \frac{20 \text{MHz}}{65536} = 0.00030520871796875 \text{ MHz}
\]
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

1000 times per second
millisecond
16616
32616
$\approx 3$ minutes
$\approx 40$ days
Hardware Peripherals
Timers
Hardware Peripherals

Timers

65535

0

TMRx
Hardware Peripherals

Timers

1200

PRx

TMRx

event
Hardware Peripherals

Timers

PRx \rightarrow \text{event} \quad \text{event} \quad \text{CPU}
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

1024
0-1023

1,000,000

36-bit
Hardware Peripherals

ADC

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

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

ADC

delta sigma
Hardware Peripherals

ADC

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

event

Gabriel Hugh Elkaim – Winter 2015
Hardware Peripherals

ADC

event

event

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

```c
{
    while (1) {
        // Check for events
        flags you set
        // Process events
        from interrupts
    }
}
```
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
    }
}
```
```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);
}
```
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
Bit manipulation

Bit packing

C1CTRL1 – dsPIC33EP256MC502

CAN
Cor
Bit manipulation

Bit masks

Example

// Abort the current CAN message transmission
C1CTRL1 = C1CTRL1 | 0x1000;
Bit manipulation

Bit masks

Example

// Disable CAN message timestamping
C1CTRL1 = C1CTRL1 & 0xFFFF7;

```
   (1111111 /\ / \ / \ / \ ) 10111
```
Bit manipulation

Bit masks

Example

```c
// Disable CAN message timestamping
ClCTRL1 &= ~(1 << 3);

BIT_0 - 31
```
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
Bit manipulation

Bit masks

Example

#define CxCTRL1_MASK_CANCAP (1 << 3)

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

Bit masking

• Setting a bit
  – ORing with 1
  
• Clearing a bit
  – ANDing with 0

• Toggling a bit
  – XORing with 1

\[
\begin{align*}
C1CTRL1 \ &= \ CxCTRL1\_MASK\_CANCAP; \\
C1CTRL1 \ &= \ \sim CxCTRL1\_MASK\_CANCAP; \\
C1CTRL1 \ &= \ CxCTRL1\_MASK\_CANCAP;
\end{align*}
\]
Bit manipulation

Bit masking

- Setting a bit can OR multiple masks together

Example

```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) {
  ...
}
```

1 right bit
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

// Retrieve the operating mode of the CAN hardware
int opmode = (ClCTRL1 & 0xE0) >> 5;
Bit Fields

Definition:

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

32 → 64
Bit Fields
How to Create a Bit Field

Syntax

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

Byte in Data Memory (RAM)

```
   7 6 5 4 3 2 1 0
  1 1 1 0 1 0 0 1
```

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

```c
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 CANCAP : 1;
    unsigned OPMODE : 3;
    unsigned REQQP : 3;
    unsigned CANCKS : 1;
    unsigned ABAT : 1;
    unsigned CSIDL : 1;
} C1CTRL1BITS;
extern volatile C1CTRL1BITS C1CTRL1bits __attribute__((__sfr__));
```
```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;

0   -1
Bit Fields

Maximum bitness

Example

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;

Union

Signed int

Unsigned
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Metaprogramming: The C Preprocessor

Directives
Constants/Macros
Conditionals
Debugging
Preprocessor
Operation of

- Preprocessor operates on all sources files before they're pass to the compiler
- Processes special preprocessor directives specified in the code
- Final text of the source file after all preprocessor directives are processed is then compiler
Preprocessor Directives are parts of the code that give special instructions to the compiler. They always begin with a # at the beginning of the line, and are used to direct the compiler with a number of specific commands.

- **Groups:**
  - #defines: constants, macros
  - Conditionals

- **Usage:**
  - Code organization
  - Debugging
# Preprocessor Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define</td>
<td>Define a preprocessor macro.</td>
</tr>
<tr>
<td>#elif</td>
<td>Alternatively include some text based on the value of another expression, if the previous #if, #ifdef, #ifndef, or #elif test failed.</td>
</tr>
<tr>
<td>#else</td>
<td>Alternatively include some text, if the previous #if, #ifdef, #ifndef, or #elif test failed.</td>
</tr>
<tr>
<td>#endif</td>
<td>Terminate conditional text.</td>
</tr>
<tr>
<td>#error</td>
<td>Produce a compile-time error with a designated message.</td>
</tr>
<tr>
<td>#if</td>
<td>Conditionally include text, based on the value of an expression.</td>
</tr>
<tr>
<td>#ifdef</td>
<td>Conditionally include text, based on whether a macro name is defined.</td>
</tr>
<tr>
<td>#ifndef</td>
<td>Conditionally include text, based on if a name is not a defined macro.</td>
</tr>
<tr>
<td>#include</td>
<td>Insert text from another source file.</td>
</tr>
<tr>
<td>#line</td>
<td>Reset the line number for compiler output</td>
</tr>
<tr>
<td>#pragma</td>
<td>Allows for extending preprocessor directives beyond what's in the standard</td>
</tr>
<tr>
<td>#</td>
<td>Null directive</td>
</tr>
<tr>
<td>#warning</td>
<td>Emits a warning described by the rest of the line</td>
</tr>
</tbody>
</table>
Preprocessor Directives

Text substitution using `#define`

- Defines a text substitution label

**Syntax**

```
#define label text
```

- Each instance of `label` will be replaced with `text` by the preprocessor unless `label` is inside a string
- `text` is optional
- Uses no memory

**Example**

```
#define PI 3.14159
#define MOL 6.02E23
#define MCU "PIC32MX320F128H"
#define PI_2 2 * PI
#define _STDIO_H_
```
Preprocessor Directives

Text substitution using \#define

• Labels must be valid identifiers

Example

\#define 0 1
\#define _WRONG
\#define __WRONG
\#define RIGHT
Preprocessor Directives

Text substitution using `#define`

- Text goes until the end of the line
  - Unless newline is escaped with a '\'

**Example**

```
#define true false
#define true \false
```

- Constants can be nested

**Example**

```
#define OLED_NUM_LINES (OLED_DRIVER_PIXEL_ROWS \ / ASCII_FONT_HEIGHT)
```
## Preprocessor Directives

### Predefined constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>Full path of current file</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>The current line in the file</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The current date as a string, like &quot;Jan 27 2014&quot;</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>The current time as a string, like &quot;17:20:50&quot;</td>
</tr>
<tr>
<td><strong>func</strong></td>
<td>The current function as a string, like &quot;main&quot;</td>
</tr>
<tr>
<td><strong>DEBUG</strong></td>
<td>When debugging is specified in MPLAB X, not part of the standard!</td>
</tr>
</tbody>
</table>
Preprocessor Directives

#define

Syntax

```c
#undef LABEL
```

- Deletes a macro definition
- Allows you to change a macro
  - Error when macros are redefined otherwise

Example

```c
#define M_PI 3.14
#undef M_PI
#define M_PI 3.141592653589793238462643383279502884197
```
Preprocessor Directives
Argument Macros

• Create a function-like macro

Syntax

```c
#define LABEL(arg₁, ..., argₙ) code
```

- The `code` must fit on a single line or use `\` to split lines
- Text substitution used to insert arguments into `code`
- Each instance of `LABEL()` will be expanded into `code`
- This is not the same as a C function! No stack allocation.

Example

```c
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define SQUARE(x) ((x) * (x))
#define SWAP(x, y) {((x) ^= (y);(y) ^= (x);(x) ^= (y));}
```
Preprocessor Directives

Argument Macros – Side Effects

Example

```
#define SQUARE(x) x * x

Extreme care must be exercised when using macros. Consider the following use of the above macro:
i = 5;
a = SQUARE(i + 3);
```
Example

#define SQUARE(x) ((x)*(x))

Extreme care must be exercised when using macros. Consider the following use of the above macro:

\[
i = 5; \\
a = SQUARE(i++);
\]
Macros with `#define`

Argument Macros – Side Effects

```
#define ABS(x) (((x) > 0) ? (x) : (-x))
#define NORM1(x, y) (ABS((x)) + ABS((y)))

int x = NORM1(5, 6.6);

int x = (((5) > 0)?(5):(-5)) + (((6.6) > 0)?(6.6):(-6.6));
```
Macros with `#define`

Emulating functions

- Functions provide useful features:
  - Encapsulation
  - Evaluate as an expression
  - Return values
Preprocessor Directives
Emulating functions

• For encapsulation

Example

```c
#define LABEL(arg1, ..., argn) {
    ... 
    
}
```

- Code blocks forces all code in the macro to execute in the same context
- Also allows for temporary variables within the macros
Preprocessor Directives

Emulating functions

Example

```c
#define INIT() TRISA = 5; LATA = 5;

if (beginStartup)
    INIT();
```
Preprocessor Directives

Emulating functions

Example

```c
#define INIT() {TRISA = 5; LATA = 5;};

if (beginStartup)
    INIT();
else
    ...
```
Preprocessor Directives

Emulating functions

- For encapsulation with expression-ness

**Example**

```
#define LABEL(arg1, ..., argn) do {
    ...
} while (0)
```

- Code blocks forces all code in the macro to execute in the same context
  - Also allows for temporary variables within the macros

- `while`-statement allows for semi-colon termination
  - Generates a single statement
Preprocessor Directives

Emulating functions

- To "return" values, just have the statement evaluate to a value

Example

```
#define LABEL(arg₁, ..., argₙ) VALUE
```
Preprocessor Directives
Stringification of macro values

Example

```c
#define VERSION 6.3
#define TEXTIFY(x) #x

printf("\%s", TEXTIFY(VERSION));
```

6.3
Preprocessor Directives
Stringification of macro values

- You need another layer of indirection

Example

```c
#define TEXTIFY(x) TEXTIFY_HELPER(x)
#define TEXTIFY_HELPER(x) #x
#define MAJOR_VER 1
#define MINOR_VER 3
#define VERSION_STRING TEXTIFY(MAJOR_VER) \ 
    "." \ 
    TEXTIFY(MINOR_VER)

printf("%s", TEXTIFY(VERSION));
```

1.3
Preprocessor Directives

Token concatenation

- To combine argument with existing token to generate identifiers

Example

```c
#define DEBUGIFY(x) x ## _DEBUG

printf("%s", DEBUGIFY(asdf));
```
Preprocessor Directives
Conditional compilation

- Control what code actually gets compiled
  - Already seen this with header guards

Example

```c
#ifndef BUTTONS_H
#define BUTTONS_H

...
#endif
```
Preprocessor Directives

Conditional compilation

- Family of if-statements
  - `#if`
  - `#ifdef`
  - `#ifndef`
- Ended with `#endif`
- `#if` is the general case
  - `#ifdef/#ifndef` only check if a macro has been defined
Preprocessor Directives

Emulating functions

Example

```c
#if INIT
#endif

#if 0
#endif

#if defined(_WIN32)
#endif

#if defined(__unix__) && !defined(__APPLE__)
#endif

#if __STDC_VERSION__ > 199409L
#endif
```
Preprocessor Directives

Conditional compilation

- `#ifdef text`
  - Same as `if defined(...)`
- `#ifndef text`
  - Same as `if !defined(...)`
- `#elif text`
  - Else-if, follows same rules as `if`
- `#else`
- `#endif`
Preprocessor Directives

Unit testing

- Conditionally compile in test code

Example

```c
int main(void)
{
    // Initialization code

    #if 0
        // Test code
    #endif

    // Main program
}
```
Preprocessor Directives

Fatal errors

• Output location of failure and stop running

Example

```c
#define FATAL_ERROR() \
    do { \
        printf("FATAL ERROR at %s:%s():%zd\n", \
                __FILE__, __func__, __LINE__); \
        TRISE = 0; \
        LATE = 0xFF; \
    } while (1);
```
Preprocessor Directives
Forcing compilation errors/warnings

- `#warning text`
  - Outputs compilation warning

- `#error text`
  - Outputs compilation error

Example

```c
#if __STDC_VERSION__ < 199901
#error "Must be compiled with C99 or greater"
#endif
```