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
• `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

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

Digital pins

- Voltage
  - High $T$ 3.3 - 5.5
  - Low $F$ 0 - .8
- Direction
  - Input
  - Output
- Polling interface
Hardware Peripherals

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

Synchronization
Hardware Peripherals

Digital pins

• TRIS – TRISState 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

RD LATx

WR PORTx

RD PORTx

Synchronization

I/O Cell

I/O pin

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

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

WR LAT x
WR PORT x
RD LAT x
RD PORT x

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

\[2^{16} = 32\]

\[1\text{ms} \div 2^{16} = 29\text{ms} \div 2^{29} = 30\text{days}\]
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 -> 1kHz interrupts

\[
\frac{20\text{MHz}}{2, 8, 16, 32, 64}
\]
Hardware Peripherals

Timers
Hardware Peripherals

Timers

\[
\frac{+1}{65535}
\]

TMRx

65535

0
Hardware Peripherals

Timers

$\neg k$

```
PRx
```

```
TMRx
```

event
Hardware Peripherals

Timers

PRx → CPU

TMRx

event
Hardware Peripherals

Timers

\[ SF T1 = 1 \]

PRx → CPU

Interrupt() → event
Hardware Peripherals

ADC

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

ADC SFRs

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

$0-1023$
Hardware Peripherals

ADC

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

average
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

event

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

Maxwell James Dunne – Spring 2016
Hardware Peripherals

ADC

**ADC1BUF0** = 950
**ADC1BUF1** = 600
**ADC1BUF2** = 100
**ADC1BUF3** = 65
**ADC1BUF4** = 81
**ADC1BUF5** = 93
**ADC1BUF6** = 107
**ADC1BUF7** = 122
// struct x = @0, False
x-initialized = False;

struct x;
struct *y = @x;
stack init(\((x)\));
Event-driven Programming

Events
Event loop
Stack.x
Stack.c
Stack.h

RPN.x
RPN.c

STACK_TESTING
fgets(str, a+0f()) 0

[+][3, 0]
[ ][3, 3, 2, 3]

(ok[0] == "+")
strcmp(strcspn strlen)
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

$\tau = 2$
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
    }
}
```
```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
        }
    }
}
```
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
Bit manipulation

Bit masks

ABAT

Example

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

Bit masks

Example

// Disable CAN message timestamping
C1CTRL1 = C1CTRL1 & 0xFFF7;
Bit manipulation

Bit masks

Example

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

<1 = \overline{0} 6(\overline{1} << 3);
<1 &= \overline{2}(\text{BIT}_3);
<1 &= \overline{10611}
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
    \[ C1CTRL1 \ |= \text{CxCTRL1\_MASK\_CANCAP}; \]

• Clearing a bit
  – ANDing with 0
    \[ C1CTRL1 \ &= \sim\text{CxCTRL1\_MASK\_CANCAP}; \]

• Toggling a bit
  – XORing with 1
    \[ C1CTRL1 \ ^= \text{CxCTRL1\_MASK\_CANCAP}; \]
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) {
    ...
}
```
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;
```
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
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

// 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__));
Bit Fields
How to Use a Bit Field

Example

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

    // Disable CAN message timestamping
    C1CTRL1 &= 0xFFF7;

    // If CAN message timestamping is enabled
    if (C1CTRL1 & 0x0008) {
        ...
    }
}
```
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;
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;
Metaprogramming: The C Preprocessor

- Directives
- Constants/Macros
- Conditionals
- Debugging
Preprocessor

Preprocessor stage

C Source File

C Compiler

Preprocessor

C Header Files

Compiler

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

```c
#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**

```c
#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

```c
#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

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

- Constants can be nested

Example

```c
#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

#undef

**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**

```
#define LABEL(arg_1, ..., arg_n) 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**

```
#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

```c
#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);
```
Preprocessor Directives

Argument Macros – Side Effects

Example

```c
#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(arg_1, ..., arg_n) {
    ... 
}
```

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

if (beginStartup)
  INIT();
else
  ...

Preprocessor Directives
Emulating functions
Preprocessor Directives

Emulating functions

- For encapsulation with expression-ness

Example

```c
#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
```
#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

#if 0

#if defined(_WIN32)

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

#if __STDC_VERSION__ > 199409L
```
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() (attr)
  do {
    printf("FATAL ERROR at %s:%s():%ld\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
```
Switch statements
switch Statement

Syntax

```
switch (expression)
{
    case const-expr_1: statements_1
    :
    case const-expr_n: statements_n
    default: statements_{n+1}
}
```

- `expression` is evaluated and tested for a match with the `const-expr` in each `case` clause
- The `statements` in the matching `case` clause is executed
**switch Statement**

Flow Diagram (default)

```
START

Const-expr_1 = expression?
  YES → statement_1
  NO

Const-expr_2 = expression?
  YES → statement_2
  NO

... (repeat for each expression)

Const-expr_n = expression?
  YES → statement_n
  NO

statement_{n+1}

END
```

Notice that each statement falls through to the next.

This is the default behavior of the `switch` statement.
switch Statement

Flow Diagram (modified)

START

Const-exp1 = expression?

YES

statement1
break;

NO

Const-exp2 = expression?

YES

statement2
break;

NO

Const-expn = expression?

YES

statementn
break;

NO

statementn+1

END

Adding a break statement to each statement block will eliminate fall through, allowing only one case clause's statement block to be executed.
switch Statement

Simple example

switch Example 1

```c
switch (channel) {
    case 2: puts("WBBM Chicago"); break;
    case 3: puts("DVD Player"); break;
    case 4: puts("WTMJ Milwaukee"); break;
    case 5: puts("WMAQ Chicago"); break;
    case 6: puts("WITI Milwaukee"); break;
    case 7: puts("WLS Chicago"); break;
    case 9: puts("WGN Chicago"); break;
    case 10: puts("WMVS Milwaukee"); break;
    case 11: puts("WTTW Chicago"); break;
    case 12: puts("WISN Milwaukee"); break;
    default: puts("No Signal Available");
}
```
switch (channel) {
    case 2:
        puts("WBBM Chicago");
        break;
    case 3:
        puts("DVD Player");
        break;
    case 4:
        puts("WTMJ Milwaukee");
        break;
    ...
}

Styling
switch Example 2

```c
switch (letter) {
    case 'a':
        puts("Letter 'a' found.");
        break;
    case 'b':
        puts("Letter 'b' found.");
        break;
    case 'c':
        puts("Letter 'c' found.");
        break;
    default:
        puts("Letter not in list.");
}
```
switch Statement

Fall-through

switch Example 3

```c
switch (channel) {
    case 4:
    case 5:
    case 6:
    case 7:
        puts("VHF Station");
        break;
    case 9:
    case 10:
    case 11:
    case 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
```
switch Statement

Range syntax

switch Example 3

```c
switch (channel) {
    case 4 ... 7:
        puts("VHF Station");
        break;
    case 9 ... 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
```
switch Statement

Real-world example

switch Example 2

```c
bool IsHex(char character)
{
    switch (character) {
    case 'a' ... 'f':
    case 'A' ... 'F':
    case '0' ... '9':
        return true;
    default:
        return false;
    }
}
```