Embedded C Basics

Cameron Patterson

ECE 2534

C versus C++

- C predates C++ and is mostly a subset of C++
  - No classes, templates, namespace, operator overloading, exceptions (C uses error codes), or run-time type identification
  - `for` loop index must be declared before the `for` loop
  - Uses stdio functions (e.g. `sprintf()`) rather than `cin>>` and `cout<<`
  - Uses `malloc()` and `free()` rather than `new` and `delete`
- C is a procedural language
  - C applications are decomposed into a set of functions
  - Some functions return values, some do not
- C++ supports object-oriented, procedural, or functional programming styles
  - C++ applications are normally decomposed into a set of classes

Why the switch from C++ to C?

- Most embedded applications are written in C
- Operating systems and drivers are usually written in C
- There are fewer run-time overheads in C
  - Important in embedded platforms requiring real-time performance and/or small memory footprints
- The Microchip tools and libraries support only C
- For new embedded applications, there is some adoption of C++
  - See http://www.caravan.net/ec2plus/

If performance is such a big deal, why not use assembly language?

- For embedded applications, there was a shift from assembly language to C in the 1980's
- Modern embedded applications are huge, and assembly does not offer enough productivity
- Especially for RISC architectures such as the PIC32 M4K, it's hard to write assembly code that is faster or smaller than code generated by a good C compiler
- Using assembly language throws portability out the window
- Even if the above hasn't convinced you, the 80-20 rule still applies to any code optimization
- Assembly language programming lives on in malware

C primitive types

Type / typedef	Implementation on our platform
`int / INT32`	32-bit signed (2's complement) integer
`short / INT16`	16-bit signed (2's complement) integer
`char / INT8`	8-bit signed (2's complement) integer Example `char` constants: `'a'`, `'\xFF'`, `0`
`unsigned int / UINT32`	32-bit unsigned integer
`unsigned short / UINT16`	16-bit unsigned integer
`unsigned char / UINT8`	8-bit unsigned integer
`<type>* /`	32-bit pointer (unsigned memory address)
` / bool`	Has constants `true` and `false` Must also `#include <stdbool.h>`

Note: there is no bit type

Pointer angst

- Pointers are just memory addresses
- A C pointer points to a specific type of variable

```
int * pointer_to_int;
```

- The unary & operator returns the address of a variable

```
int i = 10;
pointer_to_int = &i;     // memory address of i
```

- The unary * operator gets what is being pointed at

```
int k = *pointer_to_int;    // k = 10
```

- A C pointer tutorial is available at
  http://en.wikipedia.org/wiki/Pointer_%28computing%29
- Arrays will usually suffice in this course
  - `array[i]` is equivalent to `*(array+i)`
  - The array name is a pointer to the start of the array

Why is the `#include` needed for the `bool` type?

- The `bool` type was introduced to C around 1999
  - By that time, many programmers had rolled their own `bool` type with:

    ```
    typedef enum {FALSE, TRUE} BOOL;
    ```

    or

    ```
    typedef enum {false, true} bool;
    ```

  - The `#include` provides backwards compatibility
- How did C live without a `bool` type before 1999?
  - An `int` / `short` / `char` can be tested in an `if` statement
  - A value of 0 is interpreted as false
  - <u>Any</u> non-zero value (not just 1) is interpreted as true
    - » Hence, write "`if (int_variable)`" rather than "`if (int_variable == 1)`"

Homogeneous collection types (arrays)

- Declared as `<type> array_name[array_size];`
- Valid indices range from 0 to array_size – 1
  - No run-time index checking
- Inside a function, `array_size` can be a run-time expression
- C strings are implemented as `char` arrays

  e.g. `char string_name[string_size];`
  - Extra character needed at the end for the null delimiter (with value 0) to mark the end of the string
  - The null delimiter is <u>required</u>
  - Null delimiter is automatically inserted by the C compiler in declarations such as:

```
char word1[] = “Hello”;
char * word2 = “there”;
```

equivalent

Heterogeneous collection types (structs)

- Example definition:

```
struct person {
        char name[30];
        char address[50];
        int  zipcode;
};
```

- Examples:

```
struct person relative;
struct person * relative_p = & relative;
struct person friends[5];
```

- Example field access:

```
relative_p->address
friends[0].name
```

Bit manipulation (1)

- C does not provide builtin bit indexing
- To set / clear / toggle / test individual bits while leaving all other bits in a word unchanged, two steps are needed:
  1. Create a bit **mask** that has 1's in the bits of interest and 0's elsewhere (for bit setting, toggling and testing), or 0's in the bits of interest and 1's elsewhere (for bit clearing)
     - » The bit mask is generally created by left shifting a constant (often 1) by *n* bit positions
     - » Bit positions range from 0 (least significant bit on the right) to 31 (most significant bit on the left)
     - » Left shifting by *n* shuffles all bits *n* positions to the left, introducing 0's on the right and dropping the bits falling off the left

Bit manipulation (2)

2. Apply a logical function between the mask and the variable to be changed
   - Setting a bit requires a bitwise OR (|), not logical OR (||)
   - Clearing or testing a bit requires a bitwise AND (&), not logical AND (&&)
   - Toggling a bit requires an XOR operation (^)

- Examples:

```
int ioReg;
ioReg = ioReg | (1 << n);  // sets bit n
ioReg = ioReg & ~(1 << n); // clears bit n
ioReg = ioReg ^ (1 << n);  // toggles bit n
if (ioReg & (1 << n))      // tests bit n
```

- For more information, see http://en.wikipedia.org/wiki/Bit_manipulation#Bit_manipulation_in_the_C_programming_language

Bit shifting

```
int word = -31;
```

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1
31																															0

```
word = word << 1;
```

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0

```
word = word >> 2;
```

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

```
unsigned int uword = 0xFF00FF00;
```

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

```
uword = uword << 7;
```

1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
uword = uword >> 7;
```

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

String manipulation

- String surgery can be accomplished using arrays operations and/or the string library
  - Must first #include <string.h>
  - e.g. strlen(string)
  - For a list of string library functions, see
    http://www.cplusplus.com/reference/clibrary/cstring/
- String formatting can be accomplished with sprintf()
  - Must first #include <stdio.h>
  - e.g.
    ```
    char timeString[20];
        sprintf(timeString, "Time: %u ms", elapsedTime);
    ```
  - A useful tutorial is available at
    http://www.cplusplus.com/reference/clibrary/cstdio/sprintf/

Compiling a collection of .c files

- By default, functions have global scope
  - Functions defined in one .c file can be used in another .c file
  - Functions must be declared before use (just like variables), which is the purpose of .h (header) files
    - e.g. `int getButtonState(unsigned int button);`
- Functions should have global scope <u>only</u> when needed in other packages or `main()`
- Functions not needed in other packages should have local scope
  - Accomplished by adding the `static` attribute
- Global variables are a bad idea
  - Any variables declared outside functions should have the `static` attribute

What belongs in an include (header / .h) file?

- From http://www.eetimes.com/discussion/barr-code/4215934/What-belongs-in-a-header-file
  - Create a .h file for each package (e.g. peripheral, bus, application) in the system
  - Include prototypes for the package's public-scope functions
  - Include only declarations, not executable code
  - Include only public-scope `#defines`
  - Don't include functions local to the package
    - Declare these as `static` in the package's .c file
  - Don't include variables
- Header files should not reveal implementation details
  - Makes it easier to change implementations (provided in the .c file) without affecting other packages
- See Lab 1 for examples

The C preprocessor

- Runs before compilation, and not when the program executes
- Application- and platform-specific constant values should given symbolic names as follows:

  ```
  #define SYMBOLIC_NAME value
  ```

  – As in a text editor's find-and-replace command, every occurrence of `SYMBOLIC_NAME` is replaced with `value`
  – Unlike C assignment statements, there is no equals sign or semicolon
  – By convention the SYMBOLIC_NAME uses uppercase words separated by "_"
- Makes the program easier to read and maintain
- See Lab 1 for examples
- Microchip library macros begin with "m"

How to include a header file?

- Use the preprocessor's `#include` directive to copy one file's lines into another file
- Two variations:

  `#include “file.h”`

  » Includes files contained in the project directory

  `#include <file.h>`

  » Includes library files such as `plib.h`
- Include file order can matter
  - Library files should probably be included before anything else

The many meanings of the `static` attribute (1)

- The normal connotation of `static` is unchanging or constant
- This is <u>not</u> the C meaning!
  - `const` is a separate attribute
- Functions can be declared as `static`
  - This means the function's scope (visibility) is limited to the .c file in which the function was defined
  - A good thing
- Variables declared outside any function can be declared as `static`
  - This means the variable's scope (visibility) is limited to the .c file in which the variable was defined
  - A very good thing

The many meanings of the `static` attribute (2)

- Variables declared inside a function can be declared as `static`
  - This means that the variables retains its value from one function call to the next
  - Rarely needed

When to declare a variable as `volatile`

- Means that a memory-mapped variable may spontaneously change value
- Instructs the C compiler not to cache the variable in a CPU register
- Arises in the following situations:
  - A (hopefully `static`) variable updated in an interrupt service routine and used in another function (see Lab 1 for an example)
  - An I/O peripheral control / status / data register that spontaneously changes value in response to some asynchronous event
- Mind bender: a variable can be given both the `volatile` and `const` attributes
  - These are not antonyms in the C language!

Source code comments (1)

- The most important comments are the ones that describe:
  - **what** a function does
  - the assumptions its makes
  - the parameters used
  - what is returned (if anything)
- There should be little need to describe **how** a function works (by adding comments attached to statements) if the function:
  - performs a specific operation
  - is well structured
  - is not too long (because of helper functions)
  - uses meaningful variable names

Source code comments (2)

- Documenting function interfaces (rather than implementations) is especially important because most software and even hardware bugs arise from misunderstandings and incorrect assumptions about function and module interfaces
  - This can occur when you are using a library function (e.g. Microchip's `PORTReadBits()`) or a function written by someone else (e.g. `lcdInstruction()`)
  - Sometimes you may not have have the source code for library function implementations (e.g. proprietary code)

Additional references

- A useful web site with lots of mini-tutorials:
  http://www.cprogramming.com
- A comprehensive online C reference:
  http://publications.gbdirect.co.uk/c_book/
- A challenging embedded C quiz:
  http://www.netrino.com/Embedded-Systems/Embedded-C-Quiz
- Embedded C quiz worldwide results:
  http://www.eetimes.com/design/embedded/4008870/Embedded-systems-programmers-worldwide-earn-failing-grades-in-C
- Common embedded system interview questions:
  www.sanjayahuja.com/Interview%20questions.pdf
  - Spoiler alert: answers some of the embedded C quiz questions