How to Write Awesome C

(A quick intro to C for the Happyboard)

ross@glashan.net

January 9, 2008

Introduction

C is a very unforgiving programming language, especially on a small embedded platform like the Happyboard. This document provides a brief introduction to the language and covers things you should watch out for while writing code for your robot.

The C Language

C is an imperative, low level programming language initially written for systems development (OS and other low level work). This means that C provides a bare minimum level of functionality over the underlying hardware. This gives the programmer a lot of power, but at the same time doesn't provide much in the way of support and debugging capabilities, nor does it protect programmers from their mistakes.

Types and Variables

As with most programming languages C allows the programmer to declare *variables*, locations in memory in which values (sensor readings, results of calculations, etc) may be stored. Each variable in C must be defined as a specific *type*, which determines what type and range of value may be stored in the variable. For example, below we define two variables, one called `my_velocity` of type `int8_t`, and one called `light_sensor` of type `uint16_t`.

```
int8_t my_velocity;
uin16_t light_sensor;
```

The C standard defines a number of basic types:

- Integers
  - `uintN_t`, where N is one of 8, 16, or 32. This is an unsigned integer of size N bits. It can hold an integer value in the range $0...2^N - 1$. For example the uint16_t above can hold a value between 0 and 65535 ($2^{16} - 1$).

  - `intN_t`, where XX is one of 8, 16, or 32. This is a signed integer of size N bits. It can hold an integer value in the range $-2^{N-1}...2^{N-1}-1$. The int8_t above can hold a value between -128 and 127.

- Real Numbers
  - `float`. This is a single-precision floating point number of size 32 bits.
  - `double`. This is a double-precision floating point number of size 64 bits.

- Other
  - `char`. This is a single ASCII text character and is equivalent to `int8_t`.

Using a type that is too small for the value you are trying to store can lead to strange results (for example, 240+17 when stored in an `uint8_t` equals 1), so it's best to use a larger integer type (uint16_t, uint32_t, or float if you need really big numbers or fractional quantities).

The one exception to this rule is where you are using a large number of variables, such as an array (which we'll look at next). In this case the difference between 1024 x `uint8_t` vs 1024 x `uint32_t` is significant (One is 1KB and will fit on the happyboard, while the other is 4KB and won't!)

In many situations, you will want to store a collection of values, grouped together (such as a series of sensor readings). Arrays are perfect for this type of situation. Arrays are defined and accessed using square brackets:

```
uint16_t readings[32]; // array defined as uint16_t to fit the value 6270

readings[0] = 42;
readings[1] = 6270;
readings[2] = readings[0] + readings[1];
```

Here we're defining a new array of type `uint16_t` with space for 32 values. C arrays are zero-indexed, meaning our 32 element array has indices 0 to 31. We then assign 42 to the first element and 6270 to the second element. Finally we assign the sum of the first 2 element to the third element.

You can define arrays of any size and any type, but the type must be known at compile time (that is, you can't define an array of size N, where N is some variable you've just calculated). Text strings, for example, can be defined as arrays of characters:

```
char message[16]; // must be at least 1 character longer than text to be stored

message = "Awesome!";
```

Math, Logic, and Assignments

Variables are rather useless unless you have something to put into them. You've already seen the assignment operator `=`, which assigns the value on the right to the variable on the left. You've also seen how the value on the right can be a calculation. C provides a variety of simple mathematic operators which work with most of the types:

- Basic math: (+, -, *, /). These operators work pretty much as expected, though you must be careful about the types used in the operations (see below).
- Modulo (%). This operator is the modulo (remainder) of 2 values. For example $8\%3 = 2$.
- Bitwise operators (&, |, $\hat{}$). These operators work at the bit level and perform bitwise AND, OR and XOR (not exponent!) respectively.
- Logic (>, >=, <, <=, ==, !=). These operators (greater-than, greater-than-or-equal, less-than, less-than-or-equal, equal, not-equal), operate on most types and return 1 if the comparison is true, and 0 otherwise. It's important to remember the difference between = (assignment) and == (equality comparison), as the compiler may not warn if you use the wrong one.

Functions

C code is divided up into collections of statements and expressions called *functions*. Below is a function which calculates the average of 2 integers, *a* and *b*.

```
int16_t average(int16_t a, int16_t b) {
  int16_t sum;

  sum = a+b;
  return sum/2;
}
```

When a function is called, the code it contains is executed and a value is returned. For example if we wanted to calculate the average of -10 and 14, we would do the following:

```
int16_t result = average(-10, 14); // result now contains 2
```

A function must always have a "signature" describing what inputs it takes and what it returns. In our example above the average function returns a value of type `int16_t` (the first type definition before the function name), and accepts 2 arguments - *a* and *b* - of type `int16_t`.

Parameters to functions (like *a* and *b* in this example) act like normal variables, but are only accesible from within the function. Also, you can see from the above example that variables may be declared within functions (also known as local variables), and like function parameters they are only accessible from within the function. This is an example of "scoping"–the range in which a variable is accessible in code. In general, in C, a variable is accessible within the range of the curly braces ({ }) in which the variable was declared.

In this example you also see the use of the *return* statement. This statement will immediately exit the function and return the value to its right. In the average function, `sum/2` is returned to the calling code. In the case that you don't need a function to return anything, you can use the *void* type in the function definition.

While it's possible to write code without using functions, function-less code is not awesome code. Functions allow you to re-use common functionality (`drive_forward()`, `turn_degrees(x)`, etc) and allow you to build up powerful abstractions. Functions form the basis of writing good, modular code (something we'll discuss later).

Main Function

On the Happyboard the first piece of user code to be run is a function called *usetup*. This function is called at the beginning of the 60 second calibration period and is meant for robots to perform any calibration routines.

Once the calibration period is complete, the Happyboard enters the *umain* function. This function is the main function for your robot and once this function is complete the robot will stop.

Conditions and Loops

At some point your robot will need to make a choice (robots that don't make choices are not awesome robots). The standard C conditional is the if-then-else statement:

```
int8_t simple_compare(int16_t a, int16_t b) {
  if (a > b) {
    return 1;   // if a is greater than b, return 1
  } else if (a < b) {
    return -1;  // else if b is greater than a, return -1
  } else {
    return 0;   // otherwise return 0
  }
}
```

In this example we see there are three separate blocks which could be executed depending on the relation between a and b. An if statement can be composed of as many "else if" sections as needed, and may have either 0 or 1 "else" section.

Like conditionals, looping is very important in most programs (do something X times, or do something while X is true). To do something while an expression is true, a while loop can be used:

```
while ( analog_read(16) > 100 ) { // light sensor on port 16
  drive_forward_a_bit();
}
```

This example will call the drive_forward_a_bit function as long as analog_read(16) returns a value greater than 100. Code similar to this is likely what would be used to drive until some sensor has been triggered. Always rememeber to have timeouts though, in case that sensor never triggers!

To repeat some code a number of times the for loop is useful:

```
for (uint8_t i = 0; i < 10; i++) {
  motor_set_vel(0, i*25);
}
```

This code is quite compact, so lets look at the first line. The "for" statement is divided into 3 sections, separated by semi-colons. The first section, `uint8_t i = 0`, defines a new variable

and initializes for counting (it can be initialized to any value). The next section, `i < 10`, is checked every time the body of the loop is about to be run. If it evaluates to true, the body is run, otherwise the for loop exits. Finally, the last section, `i++`, is executed at the end of every iteration of the loop, and in this case serves to increment the loop counter.

Thus, in our example the process is as follows:

1. define new variable i and initialize it to 0.
2. check if i is less than 10, which it is, so execute the body
3. execute motor_set_vel, with i as 0.
4. increment i
5. jump back to step 2 and repeat.

So, combining everything we've seen thus far, we can create a *find_maximum* function that will return the maximum value in an array:

```
uint16_t find_maximum( uint16_t values[], uint16_t length ) {
  uint16_t max = 0;                       // initialize maximum to 0

  for (uint16_t i = 0; i < length; i++) { // loop through values
    if ( values[i] > max ) {              // if value is greater than max
      max = values[i];                    // update max
    }
  }
  return max;                             // finally, return max
}
```

Displaying information

A very import function while debugging your robot is `printf`. This function allows you to print formatted text to the LCD.

```
uint16_t a = 18, b = 24;
printf("\nThe sum of %d and %d is %d", a, b, a+b);
```

This code will clear and print "The sum of 18 and 24 is 42" to the LCD. Usually the n code is used to move to the new line, but since the Happyboard LCD is treated as a single line, the newline character is used as a clear screen.

The rest of the printf functionality is relatively straightforward. For each %d code printf finds in the "formatting string" (the first paramter to printf), it will look for an integer argument after the formatting string. In this case there are 3 %d's, so there are 3 parameters after the formatting string.

%d isn't the only formatting code; others exist for most types:

- %d - integer
- %c - character
- %s - string
- %f - float

Defines

During compilation a C file will be passed through a program known as the C PreProcessor (cpp) which performs a collection of transformations on the code in the file. Any line starting with a hash symbol (#) is a preproccessor definition.

The first preprocessor directive we will look at is the define (#define).

```
#define SENSOR_LEFT_FRONT  16

uint16_t sensor_lf = analog_read(SENSOR_LEFT_FRONT);
```

In this example we define a new preprocessor definition `SENSOR_LEFT_FRONT` with the value 16. When the preprocessor runs over this file, it will now replace every instance of `SENSOR_LEFT_FRONT` with 16. Unlike a variable, a preprocessor definition takes up no space in RAM, and cannot be changed. From the Happyboard's perspective this code is exactly the same as

```
uint16_t sensor_lf = analog_read(16);
```

This provides a quick and convenient way to define port number, motor velocities, sensor limits and a variety of other constants by name, rather than littering your code with various "magic" numbers.

Includes

The next important preprocessor directive is the include (#include). The include allows a file to include code from another file - when the preprocessor encounters an include it will actually copy all of the code in the file specified by the include into the current file. You'll see include definitions at the top of most C files, for example, happytest.c:

```
#include "board.h"
```

This includes the board.h "header file" from JoyOS, which includes definitions for all of the JoyOS functions. A header (or H file for short) file is special file that is usually paired to an equivalent C file (for example, board.h matches a board.c file in JoyOS). The H file contains "prototype definitions" of the functions in the corresponding C file–function definitions without the actual function body. For example, in button.h we see the following.

```
#ifndef _BUTTONS_H_
#define _BUTTONS_H_

uint8_t go_press();
uint8_t stop_press();
void go_click();
void stop_click();
uint16_t frob_read();
uint16_t frob_read_range(uint16_t min, uint16_t max);
uint16_t read_battery();
void beep(uint16_t freq, uint16_t duration);

#endif
```

The preprocessor directives you see above (#ifndef, #define, and #endif) are a preprocessor trick to ensure that the function definitions in the header are only included once. The main body of the H file is the function definitions, and were you to look at button.c, there would be a matching function implementation for each function declared in button.h.

Modules

Using the preprocessor functionality shown above C code can relatively easily be made modular. For example a robot may have a driving.c file (with matching .h) which defines functions like `drive_forward`, `turn_degrees`, etc. Then there might be a sensing.c with `check_front_bumpers`, `get_table_color`, etc.

While not strictly neccessary, modularising your code in such a fashion makes it easier to debug, test and reason about your robot program.

Tips

Don't Optimize (yet)

99% of the code written for a 6.270 robot will not need to run quickly. So don't worry about optimising it for speed. Instead focus on making it readable and correct.

Watch that RAM

Keep in mind that the happyboard has very limited RAM and every variable, array, and string takes up space. This doesn't mean you should ignore the tip above and try optimize RAM usage from the start, but it's helpful to keep RAM usage in mind while coding. Especially bad culprits for RAM usage are debugging strings, which can easily consume hundreds of bytes of RAM. If you find this becoming a problem, look into the PROGMEM command in AVR-LibC for defining program-space strings.

Modularize

Separate code into separate well defined modules. Have one file dedicated to driving code, one for sensing, etc. This helps separate functionality and allows for easier development and debugging.

Comment, Comment, Comment

Comment your code! While the competition is barely more than 3 weeks away, that's more than enough time for code to become completely indeciperable.

Know Your Debug Tools

The LCD, while space-limited, is an invaluable debugging tool. When you need more space, consider using the USB serial port. With a program like hyperterminal or minicom you can print text from a happyboard to your computer. If even more in-depth debugging is required, talk to an organizer about hooking a JTAG interface to your board to run GDB.

Assume the Worst

Assume that your robot will get stuck half-way through a turn, and that the bump sensor to detect the wall won't trigger, and that the other robot will accidentally knock you off course. In short, assume Murphy's Law will hold, and code to compensate for it. Where possible do multiple checks to ensure reliable readings, have alternatives to every choice, and timeouts while waiting for expected events.