Writing Programs in C

Prof. John McLeod

ECE3375, Winter 2021

This lesson note introduces how to use C to write programs for the ARM®Cortex-A9 instead of ARMv7 Assembly. I know everyone loves Assembly, but the *load-modify-store* paradigm that turns something as simple as $z = y + x$ into five (or more) lines of code does get tedious.

General Introduction

Assembly language is convenient for simple programs to demonstrate various functionalities of the ARM®Cortex-A9, as I have done in my notes.

- This is mostly because it is very easy to directly interact with hardware peripherals: just load the memory address of the peripheral into a register and start reading/writing to it.
- The downside is that functional programming, loops, and case-based conditionals are all rather complicated in Assembly, and writing useful subroutines involves a lot of "busy work".

For programs written to *solve problems* rather than *play with peripherals*, a higher-level programming language is probably a better choice than Assembly. In this case, the most common language is C.

- The online simulator allows you to write code in C. Compiling this code will automatically convert it to Assembly and then allow you to execute the result continuously or line-by-line.

In this sense, looking at the Assembly language generated form compiling a program written in C is also a good way of understanding Assembly better, and the shear amount of Assembly generated from a simple C program may make you appreciate the value of higher-level programming language.

Declarations and Types

Like Java, C requires variables to be declared with explicit types: integers, floats, characters, etc. [1] All of these are clearly defined by recognized programming standards, but the names aren't as meaningful today as they once were. Often you may see alternative (and more informative) variable types used.

- A single-byte variable can be defined as a `char` or a `uint8_t`. [2]
- A half-word can be defined as an `unsigned short` or a `uint16_t`. If you want a signed half-word, use `short` or `int16_t`.
- A word can be defined as an `unsigned int` or a `uint32_t`. If you want a signed word, use `int` or `int32_t`.
- A 64-bit integer can be defined as an `unsigned long` or a `uint64_t`. If you want a signed 64-bit integer, use `long` or `int64_t`.
- A 32-bit decimal number can be defined as an `float`. These are always signed.
- A 64-bit decimal number can be defined as an `double`. These are always signed.

I imagine you will rarely need a signed variable in this course, [3] but it's easy enough to use them if you need them.

Note: The textbook (and our notes derived from the textbook) often use `uint32_t` (and similar). This notation is shorter to write

[1] C is very similar to Java in many respects — or perhaps it is fairer to say that "Java is similar to C" — including using semicolons at the end of each line of code.

[2] `char` referring to a "character", as the now-outdated ASCII character set contains just 256 characters.

[3] And almost certainly won't need any decimal values.

but still very clear on whether the value is signed or unsigned, and on how many bits the variable is. However, the simulator doesn't recognize `uint32_t` (and similar), so either define them yourself or use `unsigned int` (and similar) instead.

In Assembly we work directly with registers, so no declarations are possible. No matter how much you may want to, you cannot define a new piece of hardware using only software! In Assembly we also used some registers to hold memory addresses instead of data values. In C this is done with **pointers**

- To define a pointer, use a "*" after the variable type in the declaration.
- "*" is also the **dereferencing operator**, that implies the given operation is to be done to the memory address the pointer *references*, rather than the pointer itself.
- "&" is the **addressing operator**, that looks up the *memory address* of a variable rather than the value of that variable.

Some examples:

```
unsigned int n = 2;
unsigned int* n_ptr = &n;
*n_ptr = 4;
n_ptr = 2;
```

Here `n_ptr` is defined as a pointer to an unsigned integer, and initialized with the address in memory used by the previously-defined unsigned integer `n`.

- When the code `*n_ptr=4` is executed, the value of `n` will change to 4.
- When the final line of code, `n_ptr=4` is executed, the pointer will now point to memory address 4 (which, in all probability, is *not* the address of `n` anymore).

Using "*" can get confusing, especially when writing microcontroller in C. For example, if you wanted to store a constant (such as `0xA5`, say) to a fixed memory address (such as `0x01FC`, say) you could use the C code:

```
*((unsigned int*)(0x01FC)) = 0xA5;
```

Here the first "*" used in the above line of code is the **dereferencing operator**, indicating that it is the contents of the following memory address that should be modified rather than the memory address itself. The second "*" is part of the **variable type**, namely that the following number (`0x01FC`) is the address to an unsigned integer variable.
As another example, if you want store the contents of a known memory address (say `0xFF2001FF`) to a variable, you could write:

```
unsigned char x = *(unsigned char *)(0xFF2001FF);
```

Here we are only reading a single byte of data, as the variable type is `unsigned char` and the pointer is similarly a `unsigned char *`. Note that the actual address is *not* a byte: it is a word.

Preprocessor Macros

C is a compiled language. Before the code is compiled into an executable, [4] a preprocessor is used to prepare the code. There are several preprocessor macros that can be included into C code to make programming easier. There are a whole lot of these macros, but in this course you probably only need two of them:

- An additional file of code can be inserted into the current file using the `#include` macro. Typically these are **header** files.
- You can specify string to be replaced using the `#define` macro. Typically this is used to make hard-coded constant numbers more readable.

The DE10-Standard development board has a lot of peripherals, all of which are memory mapped. For your convenience, a file called `address_map_arm.h` is provided with the laboratories; this file contains all the peripheral memory addresses identified by a (usually) self-explanatory label. You can include this these definitions in your code using the following at the start of your main file:

```
#include "address_map_arm.h"
```

Of course this requires the file `address_map_arm.h` to be at a known location, typically in the same directory as the main code file.

- This won't work on the simulator because the simulator runs in a browser and is hosted on someone else's webserver. It doesn't know where to find the header file.

[4] Or a long list of syntax errors!

Instead, you can define the necessary addresses yourself (or copy+paste the necessary lines from `address_map_arm.h`) at the start of your code.

```
#define LED_BASE     0xFF200000
#define TIMER_BASE   0xFF202000
```

Now it is clearer what the value `0xFF200000` is for, and your code is easier to read:

```
*((unsigned int*)(LED_BASE)) = 0xFF;
```

It should be pretty obvious now that the above code writes `0xFF` to the LEDs (i.e. lights up 8 of the 10).
Note that the preprocessor acts more like a *word processor* than a *compiler* — in these examples, it is simply inserting text or performing a search+replace within the code prior to running it through the compiler.

- For example, if you wrote a program declaring variables as `uint32_t` or `uint16_t`, only to find that the simulator doesn't recognize those types, you can use a preprocessor macro to define them. [5]

```
#define uint32_t unsigned int
#define uint16_t int
```

There is a bunch of other stuff that can be done with the preprocessor, but it probably isn't needed for this course.

[5] Alternatively, you can use C code: `typedef unsigned int uint32_t;` does the same thing. Which is better? Ask a CS major...

Accessing Hardware

Writing programs in C makes nested functions and complicated data manipulation much easier to code, but the trade-off is that the interaction with hardware is a bit clumsy.

- Basically you need to define a pointer to the memory-mapped peripheral, then you can manipulate the data at that address.
- Because the peripheral is at a fixed memory address rather than a memory address that is determined by the compiler, the keyword `const` should be used after the data type to clarify that the address is fixed and will not change.
- Because the peripheral is external to the CPU the keyword `volatile` should be used to let the compiler know that the contents at this address may change without explicit instructions from the CPU. [6]

For example, to turn on a LED:

```
#define LED_BASE 0xFF200000

void main()
{
 volatile unsigned int* const LED_ptr
                    = (unsigned int *) LED_BASE;
 // turn on first LED
 *LED_ptr = 1;
}
```

[6] Basically we use the keyword `volatile` to prevent the compiler from trying to optimize any of the code that involves that peripheral.

Defining a pointer to control a peripheral basically works the same as dereferencing a constant address, as the following code does the same thing such as the code above.

```
#define LED_BASE 0xFF200000

void main()
{
 // turn on first LED
 *((unsigned int* )(LED_BASE)) = 1;
}
```

Dereferencing a constant is simpler, but sometimes it is handy to have an explicit variable to control the peripheral — especially if you want to pass control to a function, or if that peripheral has structure.

Structures

A structure is a variable with multiple parts, and possibly a different type for each part. Structures are useful to define complex data sets, but in this course they are primarily useful for defining complex peripherals.

```
// extremely important structure
typedef struct _TwoInt
{
  int x1;
  int x2;
} TwoInt;

TwoInt b;
b.x1 = 3;
b.x2 = 1;
```

In the above code, an extremely important and exceedingly useful structure is defined that contains two integers. Not one, but two! The future is now.

- Anyway, this structure is defined as the new variable type `TwoInt`, allowing us to declare as many instances of the structure as we want.
- In this case, we declare `b` to be that kind of structure, and then initialize each integer in `b` to whatever value.

Students who are keen to master C programming may wish to

Google "typedef struct vs struct" to read a page of explanation why it is a good idea to use typedef here even though it is not strictly necessary. Most students probably won't care. [7]
Once the structure is defined, we can also define pointers to the structure. This allows us to use the visually-pleasing -> operator for dereferencing inside a structure:

```
TwoInt* c = &b;
c->x1 = 8;      //sets b.x1 = 3
(*c).x2 = 15; //sets b.x2 = 15
c.x1 = 3;       //oops, error!
b.x1 = 3;       //this still works
```

As indicated, `ptr->part` is just another, and arguably more intuitive, way of writing `(*ptr).part`.

Ok, so structures seem cool, but why are they useful? As we will soon learn, most peripherals are more complicated than a single memory address.

- In fact, the memory address give is just the *base* address. [8] Many peripherals extend up from this base address over several words-worth of memory.
- Often each of these words above the base address has a different purpose: for setting output, for reading input, for controlling the operation, for checking the status, etc...

As an example, the interval timers on the DE10-Standard extend across six words above the base address. Starting from the base address, there is:

[7] Short answer: there are some rare instances where typedef struct works better than just struct, and basically zero instances where it is worse, so just do it.

[8] Hence the word "base" in the addresses defined in address_arm_map.h.

- A status register, to check whether the timer is currently running.
- A control register to start or stop the timer.
- Two period registers, to set the interval (number of clock cycles) that the timer should count for.
- Two counter registers, to read the current count from the timer.

This timer is shown visually in Figure 1. Structures make working with these kind of peripherals somewhat simpler. The nice thing about structures is that the variables defined sequentially within the structure are assumed to exist in the same sequence in memory. So a good structure for this timer might be as follows:

```
// data structure for interval timer
typedef struct _interval_timer
{
        int status;
        int control;
        int low_period;
        int high_period;
        int low_counter;
        int high_counter;
} interval_timer;
```

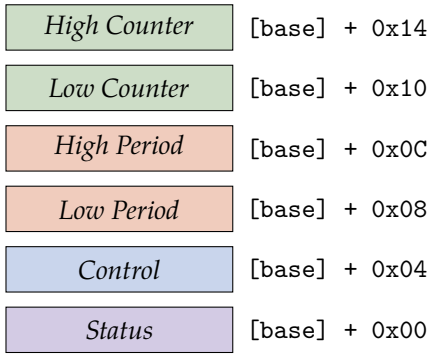

Figure 1: Visual representation of DE10-Standard interval timer.

The DE10-Standard has two of these timers, at addresses 0xFF202000 and 0xFF202020. So we might create a pointer to the first one:

```
// define pointer to timer
volatile interval_timer* const timer
            = (interval_timer*)(0xFF202000);
```

We can then read and write to the various parts of the timer fairly intuitively.

```
// set count period
timer->low_period = 312;
// set timer control
timer->control = ctrl_bits;
// check timer status
if (timer->status == stuff)
    hard_reboot();
```

This is probably clearer than constantly dereferencing fixed addresses.

```
// set count period
*((unsigned int*)(0xFF202008)) = 312;
// set timer control
*((unsigned int*)(0xFF202004)) = ctrl_bits;
// check timer status
if (*((unsigned int*)(0xFF202000)) == stuff)
    hard_reboot();
```

Functions

Like variables, functions (or subroutines) must also have an explicit type, which defines their return variable.

- Functions that do not return anything can be defined as `void`.

If you've been trying to run the code listed above and failing, it is because it is not a complete program.

- The bulk of your program must be contained within the function `void main()`.

In general C textbooks, "main" is often of type `int`. However that is because the main program is supposed to return some value upon completion to indicate whether or not an error has occurred. Since microcontroller programs do not run on any underlying operating system, there is nothing to receive any error codes. [9] Consequently, for our purposes, the "main" function will be `void`.

[9] As mentioned previously, the main program also should run endlessly.

- Functions have **scope**: variables defined within the function are only accessible within that function.
- **Global variables** can be defined outside the "main" function.

Global variables are often treated dismissively in programming courses, but they are rather useful in microcontroller programming, as we will see.

The following example is a complete program that uses a global variable to assist in a very valuable task:

```
unsigned int global_n = 5;

int add_five(unsigned int x)
 {
 return x + global_n;
 }

void main()
 {
 int y = 16;
 int z = add_five(y);
 }
```

Because of function **scope**, arguments are passed by **value**.

```
void dump_that_var(unsigned int byebye)
 {
 byebye = 0;
 }

void main()
 {
 int x = 17;
 dump_that_var(x);
 }
```

After calling `dump_that_var(x)` and returning to the main program,

the value of `x` remains 17.

- As the **value** of a pointer is the address to a place in memory, passing a pointer as an argument to a function *does* allow that function to change the value of the variable the pointer is pointing at.
- Arrays are technically just a construction of pointers, so arrays are passed to functions by **reference** rather than **value**. But I doubt you will need to use arrays in this course.

Functions should either be **defined** before they are first called, or **declared** before they are first called, and then defined at an arbitrary point in the code.

- A function is **defined** by providing the complete code.
- A function is **declared** by just providing the type, name, and arguments.
- Functions must always be **defined** somewhere in the code; they do not need to be **declared**.

In a conventional C programming class, you might be advised to put function declarations in a header file (like `my_functions.h`) that is included at the start of your main code (as `#include "my_functions.h"`), and then have the definitions in a separate code file (like `my_functions.c`) that is combined with your main code using the linker at compile time.
You can do the same things here, of course, but if you want to test code in the simulator that code has to all be in one file.

As an example of functions with only definitions:

```
int gimme_five()
{
  return 5;
}


void main()
{
  int x, y;
  x = gimme_five();       //this works
  y = gimme_fake_five(); //error!
  // compiler doesn't know what
  // 'gimme_fake_five()' is yet!
}

int gimme_fake_five()
{
  return 17;
}
```

This code will fail to compile because the compiler is not aware of what `gimme_fake_five()` is when it reaches that line in the main program. To fix it, we would have to put the definition of `gimme_fake_five()` above the main program (before or after the definition of `fake_five()` — these two functions don't call each other, so they can be defined in any order).

It is probably better programming practice to declare the functions first, then define the functions at the end.

```
// function declarations
int gimme_five();
int gimme_fake_five();

void main()
{
  int x, y;
  x = gimme_five();
  y = gimme_fake_five();
}

// function definitions
int gimme_fake_five()
{
  return gimme_five() + 12; //nested call!
}

int gimme_five()
{
  return 5;
}
```

Here the order of declarations and definitions does not matter, as long as the declarations are made first. That way the compiler knows right away what sort of thing `gimme_five()` is, even if the compiler doesn't know what it does yet.

Arithmetic, Logical & Bitwise Operations

C has all the usual arithmetic, logical, and bitwise operations. One nice thing is that many operators can be used effectively as **unary operators** that apply to, and eventually replace, the original variable. Some of these are summarized in Table 1. The only tricky thing is remembering the difference between **bitwise logic** and **variable logic**.

- `&&` is the *logical* `AND`. `x && y` evaluates as *true* (1) or *false* (0), regardless of the size of `x,y`. If both `x` and `y` are non-zero, this expression will evaluate as *true*.
- `&` is the *bitwise* `AND`. `x & y` returns a sequence of bits that is the same size as `x,y`. Bit n in this sequence is 1 if both bits n from `x` and `y` are 1, otherwise it is zero.
- `||` is the *logical* `OR`. `x || y` evaluates as *true* (1) or *false* (0), regardless of the size of `x,y`. If either of `x` and `y` are non-zero, this expression will evaluate as *true*.
- `|` is the *bitwise* `OR`. `x | y` returns a sequence of bits that is the same size as `x,y`. Bit n in this sequence is 1 if either bit n from `x` or `y` is 1, otherwise it is zero.

As noted in the table, there is a **bitwise exclusive or** `^`. However there is no equivalent **logical exclusive or** that applies to an entire variable.
There are also convenient logical and bitwise **complement** operators.

Operation	Short Form	Long Form
Addition	`x += y;`	`x = x + y;`
Subtraction	`x -= y;`	`x = x - y;`
Multiplication	`x *= y;`	`x = x * y;`
Division	`x /= y;`	`x = x / y;`
Bitwise And	`x &= y;`	`x = x & y;`
Bitwise Or	`x |= y;`	`x = x | y;`
Bitwise Exclusive Or	`x ^= y;`	`x = x ^ y;`
Bitwise Logical Shift Left	`x <<= y;`	`x = x << y;`
Bitwise Logical Shift Right	`x >>= y;`	`x = x >> y;`

Table 1: Some useful operators in C.

- `!` is the **logical not**. `!x` evaluates as *true* (1) or *false* (0), regardless of the size of `x`. If `x` is zero, this expression will evaluate as *true*.
- `~` takes the **complement**, or **bitwise not**. `~x` returns a sequence of bits that is the same size as `x`. Bit n in this sequence is 1 if bit n from `x` is 0, otherwise it is zero.

Note that the *same* operator in C is used for arithmetic regardless of whether the data is signed or unsigned (i.e., `x/y` is used for both signed and unsigned division), as this is explicitly accounted for by the declared type of the variable.

Other Useful Stuff

Most of the other useful stuff for programming in C is the same as Java. C supports conditional statements with the same *if ... else if... else* syntax as Java:

```
if (x == y)
{
  z += 2;
}
else if (x < y)
{
  apples_for_whom( everyone );
}
else
{
  destroy_earth();
}
```

It is important to remember that == is the **equality operator** while = is the **assignment operator**.

```
x = 2;
y = 3;

// oops, wrong command
if (x = y)
{
  // now x = 3
  // the assignment 'x=y' always returns true
```

```
  // so this will happen
  destroy_earth();
}
else
{
  // this won't happen
  distribute_cookies();
}
```

This can be tricky to debug, because something like the above example will not return an error.

- In this case, the `if` statement is not determining "does `x` have the same value as `y`?"
- Rather, the `if` statement is not determining is determining "can `y` be set to the same value as `y` successfully?", which basically always evaluates as true.

C supports *for...* loops with similar syntax to Java:

```
int i;
int x = 0;
for (i = 0; i < 10; i++)
{
  x += i;
}
```

The following computes the running total of `x`.

- Note that technically, all local variables should be defined at the start of the function in C. (Unlike Java or C++.)

- Consequently, handy immediate declaration of counters is not allowed:

```
int x = 0;
// strictly speaking, this should generate
// an error because the variable 'i' should
// not be defined in-place
for (int i = 0; i < 10; i++)
{
  x += i;
}
```

- However this practice is so popular that many C compilers quietly support it. [10]

C also supports *while...* and *do...while* loops. The only difference between these types of loops is that a *do...while* loop is guaranteed to cycle once; while if the loop-exit condition is immediately met a *while...* loop will not cycle.

```
a = b;
while(a != b)
{
  //good luck getting out of this loop!
  a = b - 1;
  // but fortunately, this loop never runs
}
```

[10] The simulator supports it, I am too lazy to check the hardware compiler. As long as your code works, we don't care how it is written. But if you get weird compiler-specific errors then maybe check your code for this sort of thing.

```
x = 5;
y = 15;
do
{
  x = y;
} while(x != y);
```

As you are probably already aware, you can use ++ or -- to increment or decrement a variable in place.

```
// these all do the same thing
x = x + 1;
x += 1;
x++;
++x;
// and these all do the same thing
x = x - 1;
x -= 1;
x--;
--x;
```

The difference between x++ and ++x is similar to the difference between **pre-indexing** and **post-indexing**. With x++, the variable is not incremented until after the statement has executed, while ++x increments the variable first, then executes the rest of the statement.

- As stand-alone operations (like the ones show above), x++ and ++x do the same thing.
- However in conditionals or *for...* loops, they provide different behaviour.

```
for (i=0; i<10; i++)
  // this code runs 10 times,
  // for i = 0, 1, 2, ..., 9

for (i=0; i<10; ++i)
  // this code runs 9 times,
  // for i = 1, 2, 3, ..., 9
```