Tutorial 2: Makefiles, basics of C, compiling source code and dealing with errors

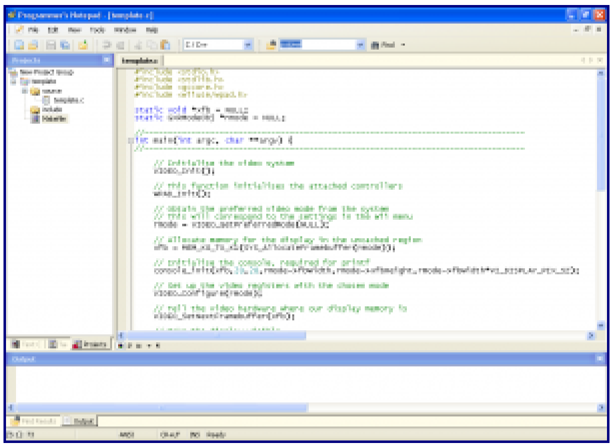
http://www.codemii.com/2008/08/16/tutorial-2-makefiles-basics-of-c-compiling-source-code-and-dealing-with-errors/

This second tutorial will cover makefiles, a very basic run down of the C programming language, compiling source code using the template examples found in DevKitPro and dealing with compiling errors. This tutorial came out way longer than I expected and I'm hoping to keep other tutorials shorter than this one.

When you're compiling source code, the program that does the compiling needs to know what to compile, which libraries to include, compiling options, optimisations, etc and this is what a makefile is used for.

Makefiles

A makefile is usually found that the root directory of the source code you have. Take the gamecube template source code example (C:\devkitPro\examples\gamecube\template). In that directory you will see a file name “Makefile”. If you open it up with notepad, you’ll be able to see the different parts of a makefile.


[illegible]

The important parts of a makefile for this tutorial are the following lines:

```
include $(DEVKITPPC)/gamecube_rules
```

By specifying “gamecube_rules” we are telling our compiler that the source code we want to compile will be run on the gamecube. You would use “wii_rules” to compile for the Wii.

By changing between these two, you are changing which libraries will be used when compiling the source code. Libraries are a bunch of files which we use in our source code to interact with the gamecube or Wii system.

```
SOURCES := source
DATA := data
INCLUDES :=
```

These lines tell the compiler which files should be compiled. Most of the time you can leave this alone as our source code will be in /source and our additional files like images, music, etc will be in /data.

```
LIBS := -logc –lm
```

This is a very important line as it tells the compiler which additional libraries we wish to use. Say we want to play an mp3 and have the relevant code in our source to do so. If we were to compile our source with the above line, the compiler would complain and say that it can’t find the functions we are using to play an mp3 file. For playing mp3 files, you need to include the “lmad” library.

The LIBS line when we add the lmad library looks like:

```
LIBS := -lmad -logc –lm
```

The order of how you include your libraries is also important as some libraries may reference other libraries and if you haven’t got them in the right order, the compiler will complain about it.

Basics of C

So we’ve quickly covered the more important parts of makefiles, we are now ready to learn some basics of the C programming language (which is kind of similar to most other programming languages). If you wish to learn much more about C programming just check the net for in depth tutorials which will explain things much clearer than what I’m doing.

Firstly we will cover variables. A variable as the name suggests is a symbol that can change its value. In the example below we are assigning 0 to the variable test.

```
int test = 0;
```

The variable test is of type int which means integer (number). So we are assigning the number 0 to the variable test.

There are different types of variables in the C programming language which include:

- boolean can either be true or false. E.g. bool test = false;
- int / long is integer that can range from -2,147,483,648 to 2,147,483,648. E.g int test =5;
- float is a precise decimal point number that can range from +/- 3.4e +/- 38 (~7 digits). E.g float test = 3.14159
- double is another precise decimal point number from +/- 1.7e +/- 308 (~15 digits). E.g double test = 3.1415926535897
- char is a single character or integer that represent a character. E.g. char test = ‘a’;

The next thing we’ll cover is control structures, some include if, else, while and for

```
int test = 5;
if (test == 5) {

   //do something

} else {

   //do something else

}
```

The above code is a demonstration of an “if” statement. If the variable “test” is equal (==) to 5 then “//do something” will be run. If we had set the “test” variable to 4, then “// do something else” would be run. The double slashes (//) means that all the text after the slashes is a comment in the code.

The double equals signs (==) is what we call an operator. There are many operators, some include:

- <= means less than or equal to
- >= means more than or equal to
- != means not equal to

```
int test = 0;
while (test < 5) {
   printf(“ok”);
   test++
}
```

The above code is a while loop, which continually checks to make sure that if “test” is less than 5 then it will print the text “ok” to the screen and then increment the “test” variable, which means it will add 1 to “test”. In an example run, “test” would start out at 0, it would print “ok” to the screen and then “test” would be “test” + 1. It would then go back to the while condition and test if “test” is less than 5 which in this case “test” would be equal to 1 and so it would print “ok”, etc.

The for loop is similar to the while loop except that we will know before we enter the loop, how many times we will run the loop.

```
int test;
for (test = 0; test < 5; test++) {
   printf(“ok”);
}
```

Notice how the “test” variable hasn’t been declared and we are assign the integer in the for loop. The for loop has three arguments, the starting count (test = 0), the condition (test < 5) and what to do after each loop which is usually increment the starting variable (test++). This example for loop would print “ok”5 times.

There is a data structure which is often used which is called an array. An array is a set of elements contained in a structure.

```
int myarray[50];
myarray[0] = 1234;
```

In the above code we are setting up an array of type int and setting 50 elements to be in that array. The next line shows us assigning the first element (yes, 0 is always the start of an array) of the array the value of 1234. In almost any source code that you will see, it will most likely contain an array of some sort, so it’s best that you learn about arrays.

Functions are useful if need to do something repetitive, for example, if your program were to print a list of numbers in an array on the screen you wouldn’t want to individually print each element of the array by writing the code, but rather use a function which could print all the elements of that array for you.

```
void myfunction () {
   printf(“hi”);
}
```

In the above code, we have the function name “myfunction” which prints out “hi” to the screen. All functions need to return something once they have finished except if they have void before the function name. Void just means that this function will not return anything.

We can call our function “myfunction” by using: myfunction();

If we want to have a function return a certain value, be it an int, float, bool, etc, we can stick these types just before the function name as shown below.

```
int myfunction (int number) {
   printf(“%i”, number);
   return number;
}
```

What we are doing in the above function is we will be printing and returning the integer we receive in our function argument. So if we called int test = myfunction(5); it would print 5 to the screen and return 5 to the variable “test”.

Now the %i part of printf means that we will be replacing %i with an integer, in this case the integer is the taken from the “number” variable. We can use %f for printing out a float, %l for a long, etc.

Reading the source code

So that’s the basics of the C programming language which should get us through this tutorial. We are now ready to compile our first source code. We will be using the Wii template file instead of the Gamecube one as it’s actually got comments in the source which will help us to learn more.

Navigate to C:\devkitPro\examples\Wii\template and open up the template.pnproj file which should open up with Programmer’s Notepad. When loaded, on the left hand side you will see the directory structure. Open up the template.c file by double clicking on it. You’ll now see the source code being displayed.

Below is the break down of source code for the template.c file.

```
#include <stdio.h>
#include <stdlib.h>
#include <gccore.h>
#include <wiiuse/wpad.h>
```

These are our include files, which specify the libraries we are including in our source code. “stdio” represents the Standard Input/Output library which lets us print text to the screen, read and write files, etc. “stdlib” represents the Standard Utilities Library, and so on. You’ll need to use at least these 4 libraries when compiling code for the Wii.

```
static void *xfb = NULL;
static GXRModeObj *rmode = NULL;
```

We can skip these lines for now, they are just variables which are used to show video on the screen.

```
int main(int argc, char **argv) {
```

This line is the start of our “main” function, which is the first piece of code that is run after our libraries and variable definitions.

Most of the lines below this line have comments so it’s pretty easy to know what the each line does. For every application you develop that you will need all the lines from **VIDEO_Init();** to **if(rmode->viTVMode&VI_NON_INTERLACE) VIDEO_WaitVSync();**. These lines aren’t really too important to focus on so we can skip them for now.

```
printf(”Hello World!”);
```

This line is saying that we would want to write “Hello World!” on to the screen.

```
while(1) {
```

We are starting an infinite loop which will run the below code.

```
WPAD_ScanPads();
```

This tells the system that we want to read the controller state. This will let us get information such as which buttons were pressed, which ones were let go, etc. You will always need to call this function when inside a loop, otherwise when you run your application on the Wii, you won't be able do anything with the controller.

```
u32 pressed = WPAD_ButtonsDown(0);
```

As the comments for this line read, the "pressed" variable will tell us if anything was pressed on the controller. If we pressed

```
if ( pressed & WPAD_BUTTON_HOME ) exit(0);
```

Here we are just checking firstly, if a button was pressed ("pressed") and secondly if the Home button was the button that was pressed ("& WPAD_BUTTON_HOME"). If both these conditions are true, then the application will be terminated ("exit(0);").

```
VIDEO_WaitVSync();
```

WaitVSync just waits for the Vertical Sync from the TV screen. It's exactly the same as the option that you have in some computer games. If you didn't have this line of code and your application was doing a lot of graphics displaying and clearing of the screen then the screen would begin to flicker (although that wouldn't happen in computer games, the game fps would just speed up).

```
}
```

This bracket ends the indefinite loop.

```
return 0;
```

All C programs must return 0 at the end of the program. If you don't return 0 then the compiler will complain because it won't know where the end of the program is.

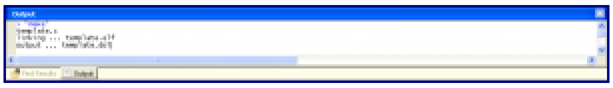
Compiling the source code

So we know understand the source code. It will firstly initialise the video and controller, print out "Hello World!" to the screen and wait for us to press the Home button on the Wiimote to exit. Go ahead and press Alt + 1 to compile the project.

You will see the following text in the Output window:

```
> "make"
template.c
linking … template.elf
output … template.dol

> Process Exit Code: 0
> Time Taken: 00:02
```



```
> "make"
template.c
linking ... template.elf
output ... template.dol
```

As you can see, we have run the "make" command and it has compiled the template.c file which contained our source. It then outputs the machine executable format, an elf and dol, either can be used on the Wii.

Another command which is useful is the "clean" command which can be run by pressing Alt + 2. The clean command removes all traces of the compiled project, so that you can re-compile the project from scratch. It's always a good idea to run the clean command and then recompile from scratch when you make major changes to any files.

Dealing with Compiling errors

Now we'll talk about dealing with compiling errors. Most of the time when the compiler complains about something it's usually right and gives us the line number where the problem occurs.

Say we forget to put the ; after assigning something to a variable, the compiler responds with:

```
c:/devkitpro/examples/wii/template/source/template.c: In function ‘main’:
c:/devkitpro/examples/wii/template/source/template.c:21: error: expected ‘;’ before ‘rmode’
```

So the compiler tells us which function this error occurred in and leads us close to the line in question. The second line is highlighted in light purple and if you click on it, it jumps you straight to a line that is close to the error.

The compiler can also give you warning about things which still will work but they are something you should look at. For example, if I have int test; and didn't use it anywhere in the project, the compiler would say:

```
c:/devkitpro/examples/wii/template/source/template.c: In function ‘main’:
c:/devkitpro/examples/wii/template/source/template.c:52: warning: unused variable ‘test’
```

If you were to declare something like: int test = "ok"; which is incorrect as an integer can't be a string then the compiler would say:

```
c:/devkitpro/examples/wii/template/source/template.c: In function ‘main’:
c:/devkitpro/examples/wii/template/source/template.c:52: warning: initialization makes integer from pointer
without a cast
```

Most of the times reading what the compiler says does make sense. If it doesn't you can always try to google the error message you are getting. Another thing to do if you've writing a whole heap of code all at once is to begin to comment out chunks of the code by using /* and */.

In the example below, the compiler will not compile the code that is inside the */ and */.

```
int test = 0;

/* if (test > 0) {
   printf("more than 0");
} */
```

We've now got some fundamentals out of the way and now we can test out example project on the Wii. I'm assuming that you have the Homebrew Channel installed on your Wii. If not, you really should think about installing it, it makes things a lot easier.

In the C:\devkitPro\examples\Wii\template directory you will see a template.dol and template.elf file. Rename the template.elf file to boot.elf. Create a new directory on your SD card in the /apps/ directory called "template". Copy your boot.elf file to the directory you created (/apps/template/).

Put the SD card back in your Wii and launch the Homebrew Channel. You should now see the directory you created in the listing of applications on the Homebrew Channel. Click on the application and click Load.

You will then see it show on the screen "Hello World!" and there you have it, you've compiled your first project for the Nintendo Wii.