Chapter 4
Teaching KoolB to Write

Introduction:

Welcome back and I hope you enjoyed reading Chapter 3! That was a rough chapter to deal with both in length and in difficulty, so I will try to make this one shorter and more interesting. To give you a short preview of where we are heading, here are the tasks that we need to accomplish this chapter:

1. Teach KoolB how to create and write to files.
2. Set up our assembler, resource compiler, and linker to work with KoolB.
3. Add more modules and functions to perform specific tasks like generates the assembly language needed.
4. Teach KoolB how to produce actual programs.

Although this looks like a lot, it really isn't too hard (in fact, it will probably be easier than the reading module we created last chapter since we don't have to deal with as much logic). The last item, number 4, should excite you (or should I say 'excite you the most' since you should be excited with all of them) as it marks a real accomplishment when writing a compiler. When your compiler finally can produce working programs, you know that you are heading in the right direction. Before we can do this, we need to cover some theor…I mean some 'background info'!

Our first item, teaching KoolB to write to files, won't be too hard as we have already done a bit with files. We have also already seen how to add functions and modules to KoolB's ever growing program. However, item two and four might need a bit of explaining. First, we need to define what an assembler, resource compiler, and linker are and what they do:

1.	Assembler	Is a program that is very similar to a compiler. It takes assembly language (we will be studying a bit of that soon) and translates it into something called 'object code.' This object code is not human readable like BASIC and assembly language. Instead, it is only one step away from a true, working program. However, different Operating Systems have different object codes, so we need to be careful.
2.	Resource Compiler	Whereas an assembler takes code (actual instructions like do this, do that) and turns it into object code, a resource compiler takes all sorts of stuff like bitmaps, icons, or almost any file, and turns it into object code.
3.	Linker	So now that we have all this little pieces of the program floating around in object code (like the main program, its resources, and a library), what do we do

with them? We link them together to form a working program.

So what assembler, resource compiler, and linker will we use? Well, I originally thought about using FAsm, as it is small and fast, but some serious complaints were brought up:

1. It is not cross-platform, meaning a huge amount of work for us if we decided to try to make KoolB work on say, Linux.
2. It does not produce object code, meaning we cannot easily combine libraries together. The only other alternative is to re-compile large libraries with every compile, which would take a lot longer.

So what does that leave? Well, Pavel Minayev suggested the Netwide Assembler (or NASM) because it is cross-platform to nearly ever major OS based on Intel chips (unfortunately, it doesn't work with Macs on a PowerPC platform). It also has the features FAsm lacks like generating object files. If you don't like my choice of assembler, feel free to swap it out! Start your own compiler with your own choice of assembler if you feel like it. After all, you are going to have the source code to everything. As for a resource compiler, GoRC by Jeremy Gordon looked pretty good. Jeremy Gordon also has a really nice linker named GoLink which links object directly to the dynamic link libraries (DLL) were all the Windows API functions reside. If you look closely at GoRC or GoLink, you will see that these two programs are Windows only. However, since most platforms have different object codes, most other platforms have their own linkers. For now, we will be just programming Windows with a little bit of Linux as a bit of 'sugar' for KoolB. Well, enough planning. Let's get rolling.

Setting up NASM, GoRC, and GoLink:

Before we start doing the actual programming, let's get all the other utilities we need to produce fully working programs. First, we need to get the Netwide Assembler. There website is at http://nasm.2y.net/; secondly, we need GoRC and GoLink. They are available at http://www.godevtool.com. I am providing a download for the source with NASM, GoRC, and GoLink already set up, but for those curious ones out there, I will chronicle how to set it up using the downloads. The download that contains all source code to the compiler and NASM, GoRC, and GoLink are at my website (http://www.brianbecker.n2v.net) under the Windows Package for Chapter 4. If you are interested in how I set up these utilities, read on. First, navigate to the KoolB directory (C:\Compiler\KoolB) and create a folder named Asm. Asm will contain all our assembly related stuff such as the assembler and linker. First, download the latest version of NASM and extract it. The main program is nasmw.exe. Rename it to nasm.exe and move it into the Asm folder. Dowload the GoRC and GoLink packages and extract them. Move both GoRC.exe and GoLink.exe into the Asm directory. We now have all the programs we need to produce real Windows programs; however, we still need one more file: a file that contains some nice macros that reduce some of the work for us. You can this file in the Win32NASM package available at http://rs1.szif.hu/~tomcat/win32/: TomCat's Win32 Programming with NASM package. Extract it and look in the INC folder. After locating macros.asm, rename it to macros.inc and move it into the Asm folder.

That completes the setup for our Windows version of KoolB. Linux is a bit more difficult and I suggest you just download the Linux package at my website that has everything in it. In it you will find the Linux version of nasm, the GNU linker named ld, and the necessary files that ld needs. Now that you have it set up, how about we test it out.

Test It Out & Meet Grandfather Asm:

Before we can teach KoolB how to use the assembler, linker, and such, YOU need to know how it all works. Since we are teaching KoolB how generate working programs in assembly language, how about we create the simplest assembly language program we can think of: one that starts and immediately exits. Pretty straightforward, right? It is, but Grandfather Asm has many tricks up his sleeve and is worse than C++ in user-friendliness. Are you ready to meet him? No? Well, tough! May I introduce you to Grandfather Asm in his Windows costume:

```
;Library functions to import to the KoolB app
extern ExitProcess            ;     Add function ExitProcess to the program


;Initialize everything to prep the app to run
section .text                 ;     The section where all our code is held
%include "Asm\MACROS.INC"     ;     Include all the macros we need
global START ;    Make sure the linker can find where the program starts
START:                        ;     Where the program begins



;The main body of the app where the app actually runs



;Prepare the app to exit and then terminate the app
Exit: ;     Where everything winds down and the program prepares to exit
stdcall ExitProcess,0         ;     Exit normally from the program



;Data section of the KoolB app
section .data     ;     The section where all our data for variables are held
```

Hmm, is this Greek or Pakistani? Looks a bit more Greek, doesn't it? Seriously, this does look pretty intimidating, but it really isn't so. Let's analyze this line by line:

```
;Library functions to import to the KoolB app
```

The first line is a comment; just as ' starts the beginning of a line comment in Mrs. Rapid-Q, a semi-colon is the beginning of a line comment in Grandfather Asm. So we have a comment describing that the very first part of our assembly app contains library functions to import. "What are library functions?" you ask. Well, library functions are functions that are not created inside the actual program – in other words, library functions are functions provided by other people. For example, Windows APIs and Linux C libraries are both library functions. For Windows, these functions are often enclosed inside a DLL; for Linux, these functions are enclosed in a SO file. The great thing is we don't have to worry about how these functions work, we just know they are there and that we can use them. So do you want an example of a library function, do you? Well, here you go:

```
extern ExitProcess            ;     Add function ExitProcess to the program
```

That is how we make a library function available to use in our program. First, we tell Grandfather Asm 'extern' so that he knows that we just want to use a function provided by somebody else (namely, Microsoft). Then we tell Grandfather Asm what function we want to import into the program (or use in our program). In this case, the only Windows API function we want to use is ExitProcess. Bet you can't guess what that function does! Exit from the program, right? This whole process if very similar to Mrs. Rapid-Q's `DECLARE SUB ExitProcess LIB "Kernel32.DLL"` statement except we don't have to tell Grandfather Asm which DLL to use. We will have to tell the linker instead. After telling Grandfather Asm which functions we want to import, we then prepare to run the app:

```
;Initialize everything to prep the app to run
section .text                 ;     The section where all our code is held
%include "Asm\MACROS.INC"     ;     Include all the macros we need
```

First, we have a comment that describes what we are doing. Secondly, we tell Grandfather Asm what section we want. Right now, we want to prepare the app to actually run, so we tell Grandfather Asm that we want a text section (not a data section). As we do everything to make sure the program is ready to run, we also need to include our macros (you will see why later). So we include them just we would in Mrs. Rapid-Q, except we use the percent sign instead of the dollar sign. After including all the macros to simplify our lives, we tell were our program where to start running:

```
global START ;    Make sure the linker can find where the program starts
START:                        ;     Where the program begins
```

First, we need a place marker for where the program starts to run (With Mr. C++, this is the 'main' function). When we finally link everything together, the linker will want to know where the program starts. GoLink defaults to wanting the app to start at START. We can change that, but I don't know of anything better than START, do you?

But to make the GoLink see where START is, we need to make it global. So we do, no problem. Finally, we have a label named START. For those of you not too familiar with Grandfather Asm or old-time BASIC, a label is sort of like a bookmark or a landmark: it marks a certain place in the program and allows you to go there whenever you need to. A label is always followed by a colon and a new line. In addition to being a label, START is a special label because it marks where the OS will go to when you run the program. Think of START: as your driveway; that is where you start your car and start your voyage. We will be make use of labels quite a bit throughout our KoolB apps. Now that we have created everything necessary for our program to run, let's program the main part (this will be the hardest part):

```
;The main body of the app where the app actually runs
```

Hee, hee, fooled you! Remember, we are making the simplest program we can think of. To me, that is a program that really doesn't do anything! So our main body of the app is empty. Later, as we actually get into the compile process, this will probably be the largest section in our app, but for now, let's just move on to quitting the app:

```
;Prepare the app to exit and then terminate the app
Exit: ;     Where everything winds down and the program prepares to exit
```

Here we have comments describing what is happening: First, we have a label called Exit (we mark it in case we need to make a speedy exit in the middle of the program). The next line is a bit confusing.

```
stdcall ExitProcess,0          ;     Exit normally from the program
```

The first word on the line, stdcall, is a macro defined in MACROS.INC. Instead of using the regular call to call functions, we use a special call to call Windows APIs. All we need to know is that stdcall is the way Microsoft wants us to call Windows APIs, and since we want our programs to work, we use stdcall. The second word tells Grandfather Asm which Windows API (or library function) to call. In this case, we want to call Exitrocess to terminate our program. If you will recall, ExitProcess is the function we imported earlier in the library function sections. After the function we want to call, we add the parameters to the function. ExitProcess expects one of two parameters, either 0 or 1. If we pass ExitProcess 1, the functions exits the program and tells the OS that the program ended unnaturally with an error. If we pass 0, ExitProcess exits from the program and tells the OS that the program succeeded and ended naturally. Now what do you suppose we want? That's a dumb question because you can just look at the code snippet and see that we want zero to exit normally! The last section of this app is the data section:

```
;Data section of the KoolB app
section .data     ;     The section where all our data for variables are held
```

Here we tell Grandfather Asm that we want a section for data. Since we don't need any data, we will just leave this section blank. If you haven't already done it, copy & paste this into Notepad or the like and save it as "Test 1.asm" in the KoolB folder. Create a new text file and copy & paste this into it:

```
;Resource file - Generated by the KoolB compiler
1 ICON "Asm\Default.ico"            ;     Use the default icon for now
```

This is our resource file where we can include bitmaps, icons, and other resources into our final program. First, we have a comment telling everybody that this is a resource file made by our own KoolB compiler. The second line is the most interesting. For now, we just want to include an icon for our compiler. Later, we can add the ability for users to specify which icon to use, but until then, we will use deafult.ico (included in the package for Chapter 4 at my website). In order to add an icon to the program, we need to use the following format:

<ResourceNumber> <ResourceType> <ResourceInfo>

• <ResourceNumber>	An unique number to identify each resource.
• <ResourceType>	Tells the resource compiler what type of resource the resource number is (can be BITMAP, ICON, etc).
• <ResourceInfo>	Information that varies depending on the type of resource.

So with that in mind, let us examine second line of the resource file. First, we tell the GoRC that we want an ICON with a ID of 1. The ResourceInfo for an ICON is the filename (optionally with the path) to the icon. In our resource file, we make the filename "Asm\default.ico." Save that to "Test 1.rc" in your KoolB folder.

After creating the assembly language and the resources, how about we use NASM, GoRC, and GoLink to create a program? Before we can test it, download the latest source-code to KoolB at http://www.brianbecker.n2v.net and extract it to C:\Compiler\KoolB. That should give you the latest KoolB source-code and the Asm folder containing NASM, GoRC, and GoLink. Start a console follow along:

```
Microsoft(R) Windows 95
   (C)Copyright Microsoft Corp 1981-1996.

C:\WINDOWS>cd C:\Compiler\KoolB

C:\Compiler\KoolB>Asm\NASM -f win32 "Test 1.asm"

C:\Compiler\KoolB>Asm\GoRC /r "Test 1.rc"

GoRC.Exe Version 0.58 - Copyright Jeremy Gordon 1998/2002 - JG@JGnet.co.uk
Output file: Test 1.res
```

```
C:\Compiler\KoolB>Asm\GoLink /console "Test 1.obj" "Test 1.res" kernel32.dll

GoLink.Exe Version 0.11 (beta) - Copyright Jeremy Gordon 2002 -
JG@JGnet.co.uk
Output file: Test 1.exe

C:\Compiler\KoolB>
```

This is the output on my console (I have bolded what you need to type in). Once you have a console up, `cd` (change directory) to `C:\Compiler\KoolB`. Then we are set up to build our program. First, we execute `Asm\NASM -f win32 "Test 1.asm"`. This tells NASM to assemble "Test 1.asm" (we use the quotes because we want long-file names) into the win32 object file format (that is what the -f win32 does). The strange thing is that NASM doesn't say anything if it succeeds, it only complains if it encounters errors. If you switch back to the KoolB directory, you will see that NASM produced an object file named "Test 1.obj". Secondly, we need to compile our resource file. So we execute `Asm\GoRC /r "Test 1.rc"`, which tells GoRC to compile "Test 1.rc" into a RES file (that is what the /r switch does). GoRC does give a success message telling you that it produced "Test 1.res". Now that we have assembled the assembly language and the resource file, we need to link them together to form a true program. So we use GoLink: `Asm\GoLink /console "Test 1.obj" "Test 1.res" kernel32.dll`. This tells GoLink to link "Test 1.obj" and "Test 1.res" into a console app (that is what the /console switch does). The kernel32.dll at the end of line tells GoLink to search kernel32.dll for any external library functions. Why do we need to do this? Remember the "extern ExitProcess" statement in our assembly program? That is a Windows API function from Kernel32.DLL. Earlier I said that just like Mrs. Rapid-Q always wants to know what DLL each function is from, GoLink wants to know in which DLL it can find all Windows APIs used in the program. If all goes well, GoLink tells you that it produced a WORKING WINDOWS PROGRM! Hurrah! Swap back to the KoolB directory and you should see "Test 1.exe" with a nice shiny KB icon (under Win95, it might not show up correctly). Try running it; it does doesn't anything, but it does run without crashing, doesn't it?

The Linux version is slightly different:

```
;Library functions to import to the KoolB app
extern exit       ;     Add function exit to the program



;Initialize everything to prep the app to run
section .text                 ;     The section where all our code is held
%include "Asm\MACROS.INC"     ;     Include all the macros we need
global _start ;   Make sure the linker can find where the program starts
_start:                       ;     Where the program begins
```

```
;The main body of the app where the app actually runs



;Prepare the app to exit and then terminate the app
Exit: ;     Where everything winds down and the program prepares to exit
ccall exit,0      ;     Exit normally from the program



;Data section of the KoolB app
section .data     ;     The section where all our data for variables are held
```

A couple things worth noting with the Linux version:

- No resource files
- We use `_start` instead of `START`
- We use `exit` instead of `ExitProcess`
- We use `ccall` to call exit because we are calling the C library function exit, not a Windows API function.

To assembly that that under Linux, cd to your KoolB directory and type in: `./Asm/nasm –f elf "Test 1.asm"`. That will produce "Test 1.o" elf object code. To link that to a Linux program, type in: `./Asm/ld –o "Test 1" –dynamic-linker /usr/lib/ld-linux.so.2 "Test 1.o" –lc`. That links "Test 1.o" to the program "Test 1" using the linker ld-linux. The last part "-lc" tells ld to search the C library for any C functions (like printf). In this case, we do this so we can use the C library function `exit`.

Now that you know how to do all this, it is time to teach KoolB how to do it. Let's get rolling.

Creating the Writing Module:

First, open up your KoolB project with Dev-C++. Now add a new file to it, hit the save button (or File->Save Unit), and name it something like Write. Don't forget to change the extension from .cpp to .h. Now we are ready to add a Writing class:

```
class Writing{
 private:

 public:


};
```

Here we are creating a class (or an object) named Writing with two sections: private (for all the Writing variables) and public (for all the Writing functions). The beginning curly bracket tells Mr. C++ where the class starts and the last curly bracket (along with the semi-colon) tells him where the Writing class ends. Now that we have a skeleton for the Writing object, how about we fill it in with variables and functions? Alright! Let's do it. First, copy & paste these variables to the private section of the Writing object:

```
    string AppData;
    string FireUpApp;
    string MainApp;
    string FinishUpApp;
    string Resources;
    string Library;
```

If you think that Mr. C++ is bad, just wait until you start really using old Grandfather Asm – everything *has* to go his way…or else. Grandfather Asm divides everything into sections (sort of like those people who feel that no one type of food must touch another on their plate – everything must be separate piles. Even though you tell them it will all end up in their stomach all mixed together, they don't listen to you). So all the variables go in one section while all the resources go in anther while all the code goes in another. So here we have six strings to hold different parts of the assembly program. Hopefully they are pretty self-explanatory, but let's run through them quickly.

• AppData	Contains all the variables and data that the program will need. For example, we will store strings and numbers here.
• FireUpApp	Contains all the initialization routines for the program like object constructors and internal code that gets the program ready to run the user's code.
• MainApp	Bet you can't guess what this is! Yes, this contains all the user's code; it will probably be the largest section (unless the user is making a "Hello World" program!).
• FinishAppUp	Contains all the routines to clean up after the user and prepare the program to exit. For example, it will free all memory associated with GUI objects, variables, and the program in general.
• Resources	This will be a separate file from the rest of the sections. It will contain instructions for the resource compiler to tell it what files and resources to include and how to include them.
• Library	Contains all the external functions that the program will need. By external, I mean functions from outside the app like the Windows API or Linux C-Library calls.

Those are the strings that we will be using. You might be asking why we are using strings and not files to store all this in, and that's good as it shows you are thinking. The reason is rather simple: strings are faster. To write less than about 20 MB, strings are about twice as fast when compared to writing to a file. Also, many of these sections will be written to the same file. For example, AppData, FireUpApp, MainApp, FinishAppUp, and Library will all be stored in the same file, just in a different order.

Now that we have all the variables to help KoolB store the assembly files, let's create some functions so KoolB can actually store stuff in these strings. Here is the public section of the Writing object:

```
Writing(){
  Resources   = ";Resource file - Generated by the KoolB compiler\n";
  Library     = ";Library functions to import to the KoolB app\n";
  FireUpApp   = "\n\n;Initialize everything to prep the app to run\n";
  MainApp     = "\n\n;The main body of the app where it actually runs\n";
  FinishUpApp = "\n\n;Prepare the app to exit and then terminate the app\n";
  AppData     = "\n\n;Data section of the KoolB app\n";
}
void Line(int Section, string Line);
void Comment(int Section, string Comment);
void File(string FileName);
void BuildApp(string FileName);

enum Sections{ToData, ToFireUp, ToMain, ToFinishUp, ToResource, ToLibrary};
```

Here we have five functions and an enumeration for the sections. Let's analyze these functions:

• Writing	The constructor for the object (or the function that runs when the object is actually created). It just assigns each of the sections a line of text describing what each part of the program is. The first two (Resources and Library) are the first line of each file. All the strings end with a newline (\n) and last four strings start off with \n\n, which creates two new lines before that section. You will also notice that each string begins with ';'. In assembly, the character ';' starts a line comment just like in BASIC, ' or REM starts a line comment. So the beginning of each section has a comment describing what the section does. Neat, huh?
• Line	This function formats the string Asm (containing the assembly language code) into a line and adds it to the desired section.
• File	This function writes all the sections of the programs to files based on the FileName.
• BuildApp	Runs the assembler, resource compiler, and linker to produce a working program. Wahoo!

Finally, we have:

```
enum Sections{ToData, ToFireUp, ToMain, ToFinishUp, ToResource, ToLibrary};
```

This creates text labels that we can then pass to Line to specify which section to write to.

Now let us actually create the code for these functions. First, we will create Line:

```
void Writing::Line(int Section, string Asm){
  string Line = Asm + "\n";
  switch(Section){
    case ToData:
      AppData += Line;
      break;
    case ToFireUp:
      FireUpApp += Line;
      break;
    case ToMain:
      MainApp += Line;
      break;
    case ToFinishUp:
      FinishUpApp += Line;
      break;
    case ToResource:
      Resources += Line;
      break;
    case ToLibrary:
      Library += Line;
      break;
    default:
      cout << "Error: Attempt to write to an invalid section" << endl;
      break;
  }
  return;
}
```

This is a rather longish function, but it isn't too complex. We have a function that is part of the Writing object and doesn't return anything. It takes Asm, which is a line of assembly language, and adds it to the desired Section. Before doing this, it needs to format the line:

```
  string Line = Asm + "\n";
```

To format the Assembly we create a string named Line and add the parameter Asm to it followed by a newline. That is pretty simple formatting, but it works. If you want to make it better or add your own formatting routines, feel free to do so. Now that we have it all formatted it, let us add it to a section of the program:

```
  switch(Section){
    case ToData:
      AppData += Line;
      break;
    case ToFireUp:
      FireUpApp += Line;
      break;
    case ToMain:
      MainApp += Line;
      break;
    case ToFinishUp:
      FinishUpApp += Line;
      break;
    case ToResource:
      Resources += Line;
      break;
    case ToLibrary:
      Library += Line;
      break;
    default:
      cout << "Error: Attempt to write to an invalid section" << endl;
      break;
  }
  return;
}
```

That's a rather long code, but it really isn't too bad. First, we take Section and then we compare it to a list of values. When we find a match, we add the formatted line to the correct string. The += just means add to the end of the string. For example, MainApp += Line is just shorthand for MainApp = MainApp + Line. If for some strange reason the Section doesn't match up to any of the choices, we default to printing out an error. After adding the line to a section, we return from the function. That covers the assembly language, but what if we want to add comments? To do that, let's create a Comment function:

```
void Writing::Comment(int Section, string Comment){
 Line(Section, ";\t" + Comment);
}
```

Basically all we to do is add the line comment ';" and a tab right before the comment and then call Line actually do the hard work. During the actual compile process (which we will probably be starting next chapter), we will be using Line to write all our assembly language stuff to the appropriate sections. However, once we have finished compiling, what do we do? We certainly cannot leave all the assembly language code in those strings! So we fill out our function File to write all these sections containing all the Asm to files:

```
void Writing::File(string FileName){
  string  AsmFileName      = FileName + ".asm";
  string  ResourceFileName = FileName + ".rc";
  ofstream AsmFile;
  ofstream ResourceFile;
  AsmFile.open(AsmFileName.c_str(), ios::out);
  ResourceFile.open(ResourceFileName.c_str(), ios::out);
  AsmFile      << Library     << endl;
  AsmFile      << FireUpApp   << endl;
  AsmFile      << MainApp     << endl;
  AsmFile      << FinishUpApp << endl;
  AsmFile      << AppData     << endl;
  ResourceFile << Resources   << endl;
  AsmFile.close();
  ResourceFile.close();
  return;
}
```

Here we have a rather large function named File that is part of the Writing object. It takes only one parameter, and that is the FileName of the BASIC source code. Let us examine the first part of this function:

```
  string  AsmFileName      = FileName + ".asm";
  string  ResourceFileName = FileName + ".rc";
  ofstream AsmFile;
  ofstream ResourceFile;
  AsmFile.open(AsmFileName.c_str(), ios::out);
  ResourceFile.open(ResourceFileName.c_str(), ios::out);
```

First, we create two strings named AsmFileName and ResourceFileName. We assign each the BASIC filename (like Test.bas) plus the correct extensions: .asm for the assembly language file and .rc for the resource file. Then we create two file components named AsmFile and ResourceFile. Finally, we open each with the respective filenames. We use .c_str() function for each string to convert it to a string that Mr. C++ can understand; we use the ios::out option when opening to specify that we want to only write to it. Now we are ready to write each section to one of the two files!

```
  AsmFile      << Library     << endl;
  AsmFile      << FireUpApp   << endl;
  AsmFile      << MainApp     << endl;
  AsmFile      << FinishUpApp << endl;
  AsmFile      << AppData     << endl;
  ResourceFile << Resources   << endl;
```

OK, this segment of code isn't two bad. Remember how we print something to a console? Like: cout << "Sometext" << endl;? Well, this is the same concept except instead of cout, we use a file. First, we put the whole library section into the AsmFile, followed by a new line. Then we add the section FireUp, mainApp, FinishUpApp, and finally the AppData section. That's all for the AsmFile, so now let us write the Resource section to the separate ResourceFile. Once we have finished writing all the assembly language to files, we can now close them:

```
  AsmFile.close();
  ResourceFile.close();
  return;
}
```

There we go! Close the AsmFile and the ResourceFile and then exit from the function. We have now succeeded in filling out the Asm sections and writing them to files. What does that leave us with? We need to have NASM and the linker generate our programs! That's exactly what the last function, BuildApp does:

```
void Writing::BuildApp(string FileName){
  File(FileName);
  if (OS == Windows){
    Run("Asm\\NASM -E results.txt -f win32 \"" + FileName + ".asm\"");
    Run("Asm\\GoRC /r /ni /nw \"" + FileName + ".rc\" > results.txt");
    if (AppType == GUI){
      Run("Asm\\GoLink /ni /nw \"" + FileName + ".obj\" \"" + FileName +
          ".res\" kernel32.dll user32.dll gdi32.dl msvcrt.dll > results.txt");
    }
    if (AppType == Console){
      Run("Asm\\GoLink /ni /nw /console \"" + FileName + ".obj\" \"" + FileName
          + ".res\" kernel32.dll user32.dll gdi32.dl msvcrt.dll > results.txt");
    }
    return;
  }
  if (OS == Linux){
    Run("Asm\\NASM -E results.txt -f elf \"" + FileName + ".asm\"");
    return;
  }
}
```

OK, this looks like a mess of obfuscated code, but let's break it down into simple sections to (guess what?) analyze. We have a function named BuildApp that is part of the Writing module. The first thing we do is call File and pass it the BASIC source code filename to write all the sections to files. Then we have something new! We are comparing OS to Windows and Linux. So what exactly is OS, Windows, and Linux? Good question because at this point we don't know because we haven't created them! So let us create them. Swap back to Dev-C++ and then to our main program (KoolBmain.cpp). Right before you include all the KoolB modules like Read.h and such, add this:

```
enum OS{Windows, Linux};
enum AppType{GUI, Console, CGI};
int  OS      = Windows;
int  AppType = Console;
```

Before we can analyze the BuildApp function, we need to know what all this junk means. On the first line we enumerate (or list) all our OS types. For now, I am going to stick to Windows and Linux (you can add more if you want). On the second line, we enumerate the different types of programs. These include GUI (with forms and buttons and such), Console (just plain text), and CGI (Internet programs like PERL). On the last two lines, we create integers OS and AppType to hold what OS we are targeting and what type of program we want (for the Windows version, we will default to Windows and Console). Why do we need these specifications? Well, as you have seen, the assembly language between OS's is slightly different, especially when using functions provided by the OS. So when we come to these differences, we can check what OS the user wants to program for and spit out the correct assembly language for the specified OS. We will do the same with the types of programs; we will need to perform different actions depending on what type of app the user wants. Lets examine what will happen if the target OS is Windows:

```
  if (OS == Windows){
```

```
Run("Asm\\NASM -E results.txt -f win32 \"" + FileName + ".asm\"");
Run("Asm\\GoRC /r /ni /nw \"" + FileName + ".rc\" > results.txt");
```

If the target OS is Windows, we run NASM first. You will see that this is very similar to how we ran NASM to assemble (or compile) "Test 1.asm". However, several things are different. First, you notice that instead of a single backslash, there are two! Why two? Well, that is simple to answer: because Mr. C++ wants two! A backslash is the beginning of an escape character in Mrs. Rapid-Q, right? So if you want a backslash embedded in a string, you have to use two. The next part is –E results.txt; if you look in the NASM docs, you will see that this redirects the output into a file named results.txt. Why do this? Because if any errors occur, wouldn't it be nice to alert the user that the compile failed? Otherwise, we could have some very unhappy users who couldn't find their programs! The rest of the part is pretty simple, we tell NASM we want the format to be Windows (or Win32). Finally, we add ".asm" to the end of the BASIC source code and put quotes around it (in case the filename was something like "This would not compile without quotes.bas.asm"). Then we run GoRC (which is also in the Asm folder) with the switches /r, /ni, and /nw. We used the /r switch earlier, it tells GoRC to produce a standard RES file. The /ni and/nw are new, though. They tell GoRC to report no information (/ni) and no warnings (/nw). I am sure some of you are wondering why we don't want info or warnings. That becomes evident towards the end of the command where you see "> results.txt". That writes all the output from GoRC to results.txt. Here is the idea: we tell each program to write its output to a file named results.txt. After the program is done, we open up the file and see what is in it. If you will remember, NASM doesn't ouput *anything* if it succeeds, it only reports errors, and if we tell GoRC to report *only* errors (not info or warnings), KoolB will know that if results.txt is empty, everything went well. However, if results.txt has some text in it, that means that something went wrong and KoolB can read all of results.txt and print that out the user so the user can send a bug report. Pretty nifty, huh? I can almost hear some of you more advanced C++ users saying back there, "Wait a minute…where does the Run function come from, and how does it check results.txt?" Good question – Run doesn't come from anywhere, we have to make it! Yes, you heard me correctly, we have to program it. But before we do, let's finish up the rest of the BuildApp function:

```
if (AppType == GUI){
  Run("Asm\\GoLink /ni /nw \"" + FileName + ".obj\" \"" + FileName +
    ".res\" kernel32.dll user32.dll gdi32.dl msvcrt.dll > results.txt");
}
if (AppType == Console){
  Run("Asm\\GoLink /ni /nw /console \"" + FileName + ".obj\" \"" + FileName
    + ".res\" kernel32.dll user32.dll gdi32.dl msvcrt.dll > results.txt");
}
```

So here we have two actions: one if the AppType is GUI and another one if the user wants a Console app. These aren't too hard if you understand the previous ones. If we want a GUI app, we run GoLink (which is in the Asm folder) with the same switches (/ni /nw) to tell GoLink not to report any info or errors. We also tell GoLink to link the object file (the BASIC source code filename + ".obj") and the RES file (the

BASIC source code filename + ".res") together, using Kernel32.DLL, User32.DLL, GDI32.DLL, and Msvcrt.DLL. Why these DLLs? Because these are some of the most common DLLs; later we will learn how to add DLLs to this list so we can handle other Windows API calls. We also redirect GoLink's output to the same file: results.txt If the user wants a Console app, we do the exact same thing, except we add one extra switch when running GoLink, the /console switch. That tells GoLink to make what: a) GUI app, b) Linux app, c) Console app, or d) CGI app. For all those who answered option c, correct! The last part of the BuildApp function reads:

```
  if (OS == Linux){
    Run("Asm\\NASM -E results.txt -f elf \"" + FileName + ".asm\"");
  }
  return;
}
```

This is pretty simple. If the target OS is Linux, we run NASM with the ELF (the Linux object format) instead of the Win32 format. Unfortunately, we cannot link a Linux app in Windows, so that is as far as we can go. The user will have then move over to Linux and link the object file (perhaps somebody wants to make a small utility to do this easily?).

Well, well. I have good news and bad news, which would you like to hear first? The good news? OK, the good news is that we are finished with the Writing module. The bad news is we still have other modules to program. Earlier, I told you that we needed to create the Run function we used in the Writing function BuildApp. How about we start with that? To do so, we need a file that will hold miscellaneous functions and stuff. Swap back to Dev-C++ and create an new unit. Name it Misc.h or something like that.

Now that you have your file for miscellaneous functions, let's create Run:

```
void Run(string Command){
  ifstream File;
  int FileStart;
  int FileEnd;
  char * Spoon;
  double StartTime = clock() / CLK_TCK;
  system(Command.c_str());
  File.open(".\\results.txt", ios::in);
  while (File.is_open() == false){
    File.open(".\\results.txt", ios::in);
    if ((clock() / CLK_TCK) - StartTime > 30000){
       cout << "Time out running: " + Command << endl;
       exit(1);
    }
  }
  FileStart = File.tellg();
  File.seekg(0, ios::end);
  FileEnd   = File.tellg();
  File.seekg(0, ios::beg);
    if (FileEnd - FileStart > 0){
    cout << "Error: NASM, GoRC, or GoLink failed. Please attach error 
messages to bug repots:" << endl;
```

```
    cout << "Ran " + Command + " and got:" << endl;
    Spoon = new char[FileEnd-FileStart];
    File.read(Spoon, FileEnd-FileStart);
    Spoon[FileEnd-FileStart-1] = '\0';
    cout << Spoon << endl;
    delete[] Spoon;
    exit(1);
  }
  File.close();
}
```

Before you freak out, let me tell you that this function really isn't that important to your understanding of compiler construction, so I am not going to describe what each piece of code does (e-mail me if you want to know how each piece of this function works). First, let me explain what Run does. Run basically tells the OS (either Linux or Windows) to run the string Run got in its parameter. The parameter contains instructions for either the app (like NASM) or the OS to put all the results in a text file named results.text. After telling the OS to do this, Run attempts to open up results.txt. Since the program being run might take a while (a second or two) and might not be done, Run will continue to try to open results.txt for 30 seconds. If 30 seconds passes and Run cannot open results.txt, Run gives a timeout error. If Run is able to open results.txt, it opens it and checks the size. If the size is zero, that means the program being run did not encounter any errors. In that case, Run returns. If Run finds that the size of results.txt is greater than zero, it reads the contents, reports the errors, and exits from the program.

Creating the Assembly Module:

That's it for the Writing module! Before we test it out, we need create yet another module: the module that actually generates the assembly language for us! That is a rather important module, probably only second to the module we will be looking at in the next chapter: the compile module. But back to what we were talking about. Swap back to Dev-C++ and add another module and save it as "Assembly" or something similar. (remember to change it to a header file (.h)). As always, let's create an assembly object:

```
class Assembly{
 private:

 public:


};
```

That is our Assembly object. First, let's fill in the private section of our object Assembly with some data:

```
   int LabelNumber;
```

Here we have a number named LabelNumber. As assembly doesn't really have control statements like if, else, for, while, etc, so we sort of have to figure out how to do these by using goto (or jumping) back and forth between labels. When we just need a temporary label (as we will often need), we just use the format Label<*number*> where *number* is a unique number. LabelNumber is the variable that keeps track of this number (basically, we start out with LabelNumber being zero and then add one to it each time we use it). We will see how this actually works when we analyze the function that uses LabelNumber. Speaking of functions, let's add some of them to the public section of our object:

```
   Assembly(){LabelNumber = 0;}
   void AddLibrary(string FunctionName);
   string GetLabel();
   void PostLabel(int Section, string Label);
   void BuildSkeleton();
   void FinishUp();
```

Not a whole lot of functions, but here is a quick overview of what each does.

• Assembly	The constructor for our Assembly object. It initializes LabelNumber to zero.
• AddLibrary	Adds a function (either a Windows API call or a Linux C-Library call) to the library section of our assembly so we can use that function in our program.
• GetLabel	We talked about this earlier; this is the function that generates and returns a unique label for whenever we need a temporary label.
• PostLabel	Posts a label to the desired section of the assembly language.
• BuildSkeleton	This function basically adds all the assembly that we saw earlier in the chapter, like including the macros, telling the linker where the program starts, adding a default icon to the program, and other such stuff.
• FinishUp	This is the opposite of the function BuildSkeleton; it does a few last minute things so that the KoolB app can close down and exit properly. Think of it as the cleanup function.

Now that we have an overview of the functions, let us analyze them in a bit more depth. The first, AddLibrary, takes only one parameter and that is the name of the function to import from the Windows API or the Linux C-library:

```
void Assembly::AddLibrary(string FunctionName){
  Write.Line(Write.ToLibrary, "extern " + FunctionName);
  return;
}
```

Hurrah! A short function! We won't complain about that will we? AddLibrary, part of the Assembly object, takes its parameter (the name of a library function) and writes a line in the following format to the library section of the KoolB app:

```
extern <FunctionName>
```

And then it returns. The Write object that we use here will be created momentarily. That wasn't so difficult, now was it? Now lets examine the GetLabel function:

```
string Assembly::GetLabel(){
  string Result;
  LabelNumber++;
  Result = "Label" + ToStr(LabelNumber);
  return Result;
}
```

Here is the GetLabel function that we have been hearing so much about. First, we create a string name Result; that will hold the temporary label we generate. Then we increment (or add one) to the LabelNumber to get the next unique number. Finally we store the string "Label" + LabelNumber in Result and return Result. However, you will notice that since we cannot add a string and a number, we need to convert LabelNumber to a string. You will notice that this is exactly what we do with ToStr. Unfortunately, ToStr is just like Run: we have to create it. So swap to Dev-C++ and to the Misc.h file (or whatever you called it) and add this function below the Run function:

```
      string ToStr(int Number){
        string Result;
        char * Spoon;
        Spoon = new char[1024];
        sprintf(Spoon, "%i", Number);
        Result = Spoon;
        delete[] Spoon;
        return Result;
      }
```

We don't really have to analyze this as converting numbers to strings is not really crucial to your understanding of compilers. Besides, if we were using Mrs. Rapid-Q, we would have the Str$ function. All this function does is take an integer, convert it to a string, and return the string. Nothing too difficult. So now that we have a label, how do we use it? By posting it to a section with PostLabel:

```
void Assembly::PostLabel(int Section, string Label){
  Write.Line(Section, Label + ":");
  return;
}
```

PostLabel is another easy function. All it does is pass the section and the label (after adding a colon to the end) to the function Line from the Writing module. Then it returns. Now let's look at a function that has a bit more meat to it:

```
void Assembly::BuildSkeleton(){
  if (OS == Windows){
    AddLibrary("ExitProcess");
    Write.Line(Write.ToResource, "1 ICON \"Asm\\Default.ico\"");
    Write.Line(Write.ToData,      "section .data");
    Write.Line(Write.ToFireUp,    "section .text");
    Write.Line(Write.ToFireUp,    "%include \"Asm\\MACROS.INC\"");
    Write.Line(Write.ToFireUp,    "global START");
    PostLabel(Write.ToFireUp,     "START");
    PostLabel(Write.ToFinishUp,  "Exit");
  }
  if (OS == Linux){
    AddLibrary("exit");
    Write.Line(Write.ToData,     "section .data");
    Write.Line(Write.ToFireUp,  "section .text");
    Write.Line(Write.ToFireUp,  "%include \"Asm\\MACROS.INC\"");
    Write.Line(Write.ToFireUp,  "global _start");
    PostLabel(Write.ToFireUp,   "_start");
    PostLabel(Write.ToFinishUp, "Exit");
  }
}
```

In the BuildSkeleton function, we see our dear old friends OS. Previously, we had two different versions of the assembly app: one for Windows and another for Linux. Well, KoolB does the same thing. First it sees what OS the user wants to compile for. If the OS is Windows, it just spits out what we have earlier developed. Before anything else, it adds the Windows API function ExitProcess to the library section of the app. You might ask why we don't actually call ExitProcess, and that is easily answered. This is the BuildSkeleton function that initializes everything. ExitProcess immediately terminates the program, and we certainly don't want to do that when we run the program! We want to do that in the function FinishUp, which closes the app down. After importing the function ExitProcess, it adds the default icon to the resource section of the app. Then it defines where the data and code sections of the app start. It also includes the macros that will prove to be very useful in the coming chapters. Finally, it creates the starting an ending labels for the program. If the target OS is Linux, some slight changes must be made. First, Linux doesn't support resources, so we skip adding the default icon. Secondly, we change the starting point of the program from START to _start. You might also notice that instead of importing ExitProcess to the library section of the app, we import the function exit. Why do we do this? Because Linux doesn't have the Windows API's (yet, WINE and Lindows might make that change), so we have to use another function. Here is where we cheat; instead of making our own function, we use functions provided for C programs! So exit is part of the C library, and although our program is an assembly program, we can use the C library anyhow. Some of you might be wondering why we don't use the C library for Windows. Well, we can, and that is more of a matter of opinion than anything else. First, I think that the Windows API's are more powerful and more diverse. They also are built into Windows, where as the C library isn't on some versions of Windows 95. If you feel differently, feel free to change it (it's not like I have the source under lock and key!). Now for our clean up program:

```
void Assembly::FinishUp(){
  if (OS == Windows){
    Write.Line(Write.ToFinishUp, "stdcall ExitProcess,0");
  }
  if (OS == Linux){
    Write.Line(Write.ToFinishUp, "ccall exit,0");
  }
}
```

So, here is where we use the functions ExitProcess and exit! FinishUp does exactly what its name implies: it adds instructions to exit to the very end of the FinishUp section of the assembly code. So if the OS is Windows, we call the Windows API function ExitProcess with one parameter: zero. Why zero? Well, ExitProcess exits the program and tells the OS whether the app ended normally or abnormally. If we pass 0 to ExitProcess, that tells the OS that we exited normally. Later, when we get into error checking, we will learn how to exit abnormally by passing 1 instead of 0. But what is all this stdcall stuff? Why not just use call? Although most compilers hide it from you, each function has a different way of handling parameters. Instead of getting into the technical details of all the ways functions can do it, how about we leave that to later when we have to handle compiling functions? Agreed? OK, then let's just get an overview of what we need to know to get by. First, Microsoft has their own way to call functions and they have named it the standard calling convention. In MACROS.INC, we have a bunch of macros that keep all the tedious details of calling functions hidden from us. So to call a function that uses the standard calling convention, we use the macro stdcall. Simple, huh? All we have to know is when to use stdcall – and that is pretty simple as well: almost all Windows API calls are called using stdcall. However, if we use Linux, we have to use the C library function exit. It works almost identically to the Windows API function ExitProcess. However, since it is part of the C library, we have to call it differently. C has its own way of calling functions, and that is the C calling convention. To use that, MACROS.INC has provided ccall. So for Linux, we will be primarily using ccall and for Windows, we will be using primarily stdcall.

Testing It Out:

Whew! You have nearly finished Chapter 4. If you can hang on in there for a bit longer, you will be rewarded with a compiler that can actually produce working programs. So let's get started. First, we need to add all our modules to the main KoolB app. Swap back to Dev-C++ and click on the KoolBmain.cpp file. Right after you have:

```
#include "Read.h"
  Reading Read;
```

Add the rest of the objects:

```
#include "Misc.h"
#include "Write.h"
  Writing Write;
#include "Assembly.h"
  Assembly Asm;
```

There we go. Now we have a Writing object named Write and an Assembly object named Asm. Now that we have that, erase everything in the Compile function and replace it with this:

```
  string FileName;
  string TargetOS;
  if (argc < 2 || argc > 4){
    cout << "Usage: KoolB [OS] <filename>" << endl << endl;
    cout << "\tOptions:" << endl;
    cout << "\t-Windows\tCompile for Windows" << endl;
    cout << "\t-Linux\t\tCompile for Linux" << endl;
    exit(1);
  }
  if (argc == 3){
    TargetOS = argv[1];
    FileName = argv[2];
    if (TargetOS == "-Windows" || TargetOS == "-Linux"){
      if (TargetOS == "-Linux"){
        OS = Linux;
      }
    }
    else{
      cout << "Usage: KoolB [OS] <filename>" << endl << endl;
      cout << "\tOptions:" << endl;
      cout << "\t-Windows\tCompile for Windows" << endl;
      cout << "\t-Linux\t\tCompile for Linux" << endl;
      exit(1);
    }
  }
  else{
    FileName = argv[1];
  }
  Read.OpenBook(FileName);
  Asm.BuildSkeleton();
  Asm.FinishUp();
  Write.BuildApp(FileName);
  return;
```

That is quite a bundle of code there. There is a lot of logic here, so let's proceed with caution:

```
  string FileName;
  string TargetOS;
```

First, we create two strings. One to hold the FileName, and one to hold the TargetOS. Next, we check to see how many parameters the command line contains. But what is the correct number of parameters? Somehow we need to allow the user to specify which OS to compile for. So I propose this usage for KoolB:

```
Usage: KoolB [OS] <filename>

        Options:
        -Windows        Compile for Windows
        -Linux          Compile for Linux
```

Here, we KoolB requires the filename of the BASIC source code, but the target OS is optional. If you count the number of parameters, you will see that there are two valid possibilities. Either there are only two (KoolB <filename>) or three (KoolB <OS> <filename>). This is how we handle this:

```
  if (argc < 2 || argc > 4){
    cout << "Usage: KoolB [OS] <filename>" << endl << endl;
    cout << "\tOptions:" << endl;
    cout << "\t-Windows\tCompile for Windows" << endl;
    cout << "\t-Linux\t\tCompile for Linux" << endl;
    exit(1);
  }
```

We check to see if argc (which holds the number of parameters) is less than two or greater than four. If it is, we print out the usage and exit. Otherwise, we continue:

```
  if (argc == 3){
    TargetOS = argv[1];
    FileName = argv[2];
    if (TargetOS == "-Windows" || TargetOS == "-Linux"){
      if (TargetOS == "-Linux"){
        OS = Linux;
      }
    }
    else{
      cout << "Usage: KoolB [OS] <filename>" << endl << endl;
      cout << "\tOptions:" << endl;
      cout << "\t-Windows\tCompile for Windows" << endl;
      cout << "\t-Linux\t\tCompile for Linux" << endl;
      exit(1);
    }
  }
```

So if we have the correct number of command line parameters, we check to see if we have three parameters. If we do, we assign TargetOS the second parameter (remember that the first parameter is in argv[0]) and FileName the third parameter. Then we check to see if TargetOS is equal to "-Windows" or if it is equal to "-Linux. If it is, we check again to see if it is "-Linux". If it is, we assign OS to Linux. If TargetOS is "-

Windows," we don't do anything because by default TargetOS is Windows. If TargetOS is neither "-Windows" nor "-Linux" we print out the usage and exit.

```
  else{
    FileName = argv[1];
  }
```

If the number of parameters on the command line is not three, we know that the user didn't specify what OS to use, so since earlier we defaulted to Windows, we just assign FileName the second parameter and continue. Finally, we have the engines of Read, Write, and Asm do the hard work we created them to do:

```
  Read.OpenBook(FileName);
  Asm.BuildSkeleton();
  Asm.FinishUp();
  Write.BuildApp(FileName);
  return;
```

First, we tell Read to open the file that contains the BASIC source code. Then we tell Asm to build the skeleton of the assembly app. Normally, we would then call the actual compiler module that would read the BASIC and output the Asm, but we haven't got that far, so we just tell Asm to finish up generating the assembly language. Finally we tell Write to make the program and return. Compile that and create a file named "test.bas" in your KoolB folder. Now lets compile "test.bas" into a real Windows (or Linux) program. Grab a console and follow along:

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\>cd C:\Compiler\KoolB

C:\Compiler\KoolB>KoolB test.bas
Welcome to the KoolB Compiler by Brian C. Becker.
Compile Time:   0.16 seconds.
```

If all goes well, you should see that KoolB generated several files listed in your KoolB folder:

• results.txt	Holds all the reported errors.
• test.bas.asm	The assembly language generated by KoolB
• test.bas.obj	The object code for the assembly language generated by NASM
• test.bas.rc	The resource file generated by KoolB
• test.bas.res	The compiled resource file generated by GoRC
• test.bas.exe	The Windows program generated by GoLink

Although the other files are interesting, by far the most exciting is test.bas.exe. Before you run it (if you haven't already), admire the nice shiny KB icon that “kutjemuk” so kindly made for us. Then run the test.bas.exe: it should display a console very quickly and then immediately exit.

CONGRATULATIONS! You have just written a completely working compiler! That means you now classify as a compiler writer! Quick, put it on your resume! Just kidding. Our compiler doesn’t do much, but hey, you can make it more functional later. And that is exactly what we will be doing.

Conclusion:

Well, that’s all for Chapter 4. You have now survived Part I of my compiler tutorial. The first 4 chapters of this tutorial have been introductory chapters to the general parts of a compiler work together to form working programs. Chapters 1 through 4 basically build the skeleton for our KoolB compiler with the main, reading, writing, assembly, and miscellaneous modules. So what have we been preparing for? Well, everything has been building up for the actual compiler module. Until now, we have operated each module on its own – we read from the BASIC source code file and we wrote to files and we generated assembly language, but it was sort of disjointed. Here is where the compiler module comes in: it is sort of like a music director. Just as a music directory organizes all the players and keeps them on task and on time to produce a harmonious (or if you don’t like classical, a dissident) melody, the compiler module organizes all the modules and keeps them on task to produce a worthwhile result: a program. So as we enter a new a new part of the compiler tutorial, we will be digging deeper into how to analyze the BASIC source code (not just printing each word out and its type) and how to generate the correct assembly language code.

With that in mind for the future part of the compiler tutorial, I am sure you want to know what we will be doing in the immediate future, namely what we will be doing in Chapter 5. In Chapter 5, we will be creating a new module named the Compile module and we will be also expanding the Assembly module quite a bit. As for what we will be doing with the Compile module, I think it is high time we teach KoolB how to create simple data types. So in the next tutorial we will be showing KoolB how to create several different types of data, including Integers, Doubles, and Strings probably. I think I have whetted your appetite enough, so I will see you then!