Compiling with Let's C

This section describes how to compile C programs with **Let's C**.

In brief, a C compiler transforms files of C source code into machine code. Compilation involves several steps; however, **Let's C** simplifies it with the **cc** command, which controls all the actions of the compiler.

The phases of compilation

Let's C is not just one program, but a number of different programs that work together. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs. It translates the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates it into machine code. The code generation is table driven, with entries for each operator and addressing mode.

cc2 The optimizer/object generator. This phase optimizes the generated code and writes the object module.

cc3 **Let's C** also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. This phase is optional, and allows you to examine the code generated by the compiler. If you want **Let's C** to generate assembly language, use the **-VASM** option on the **cc** command line.

Unless you specify the **-VASM** option, **Let's C** creates an *object module* that is named after the source file being compiled. This module has the suffix **.obj**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

Edit errors automatically

The first option, and one that you'll use most often, is the MicroEMACS option **-A**. Often when you're writing a new program, you try to compile it, only to have the compiler tell you that you've made a mistake. You must then invoke your editor, change the program, exit from the editor, and start compiling the program again.

To make this process easier, **cc** command has the *automatic* (or MicroEMACS) option, **-A**. If **Let's C** detects any errors in your program, it will automatically invoke the MicroEMACS screen editor. MicroEMACS will display all error messages in one window and your source code in another, with the cursor set at the number of the line where the first error occurred.

Try the following example. Use MicroEMACS to create a program called **error.c**. To invoke MicroEMACS, type the command

```
	me error.c
```

at the MS-DOS prompt, or use the display interface to MWS, the Mark Williams shell, as described in section 1 of this manual. Then type the following code:

```
main()
{
        printf("Hello, world")
}
```

Note that the semicolon was left off of the **printf** statement. Type **<ctrl-X><ctrl-S>** to save the file to disk, and **<ctrl-X><ctrl-C>** to exit from MicroEMACS. Now, try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

or use MWS's display interface, as described in section 1. You will see no messages from the compiler because they are all being diverted into a file to be used by MicroEMACS. Then, MicroEMACS will appear automatically. In the upper window you will see the message:

```
4: missing ';'
```

and in the lower window you will see your source code for **error.c**, with the cursor set on line 4. If you had more than one error, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error.

With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on; it will point to a line that is near the source of the error.

Now, use **<ctrl-E>** to move the cursor to the end of line 3, and type a semicolon to correct the error. Type **<ctrl-X><ctrl-S>** to save the file to disk, and then type **<ctrl-X><ctrl-C>** to exit from MicroEMACS. **cc** will recompile the program automatically, to produce a normal working executable file.

cc will continue to invoke the MicroEMACS editor either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

Renaming executable files

When **Let's C** compiles a source file, by default it names the executable program after the source file. For example, when you compiled **error.c**, **Let's C** automatically named the executable file **error.exe**.

If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name. For example, should you wish the executable file to have the name **example.exe**, use the command:

```
cc -o example.exe error.c
```

This command will compile the source file **error.c** and generate an executable file called **example.exe**. The suffix **.exe** tells MS-DOS that the file is executable.

Floating-point numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask **Let's C** to include these routines with your program by using the **-f** option with the **cc** command.

For example, if the program **example.c** used floating-point numbers, you would compile it with the following command line:

```
cc -f example.c
```

If your program prints floating-point numbers or reads them from an input device, and it is *not* compiled with the **-f** option, it will print the following error message when it is run:

```
You must compile with the -f option
to include printf() floating point!
```

Compiling multiple source files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with **Let's C**, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** type the following:

```
cc -f factor.c atod.c -lm
```

This command compiles both C source files to create the program **factor**.

When the **cc** command line includes several file name arguments, by default it uses the *first* to name the executable file. In the above example, **cc** produces the non-executable object modules **factor.obj** and **atod.obj**, and then links them together to produce the executable file **factor.exe**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Wildcards

A *wildcard* character is one that represents a variety of characters. MS-DOS recognizes the asterisk '*' and the question mark '?'. The asterisk can represent any string of characters of any length (including no character at all), whereas the question mark can represent any one character.

For example, if the current directory held the following files:

```
a.c
ab.c
abc.c
abcd.c
```

typing **dir a?.c** would print:

```
ab.c
```

whereas typing **dir a*.c** would print all four files.

The **cc** command lets you use wildcards in your command line to save you time and effort. For example, you can compile all of the C source files in the current directory simply by typing:

```
cc *.c
```

This command compiles all of the files with the suffix **.c** and links the resulting object modules.

In another example, if the program **example** were built from the source files **example1.c**, **example2.c**, and **example3.c**, you could compile them with the following command:

```
cc example?.c
```

Tailoring the command line interface

With **Let's C**, you can tailor the command-line interface that your compiled programs use. Some programs do not use command-line arguments; others take a few; whereas others may need to read the environment and expand wildcard characters. The following options allow you to select the interface you want for your program.

The option **-na** (for "no arguments") tells **Let's C** that a program does not use command line arguments. The **-na** option may be used with or without the **-ns** option, which suppresses STDIO.

The option **-w** (for "wildcard") tells **Let's C** to include code that expands the wildcards '?' and '*' used in command-line arguments. For example, if the program **example.exe** is compiled with the **-w** option, it will expand the command:

```
example *.c
```

The wildcard argument ***.c** will expand into all file names in the current directory that end in **.c**.

If your program defines a global array **char _cmdname[]** that gives the name of the command, then compiling the program with the **-w** option will include code that fills in **argv[0]** with the command name and looks for environmental variables of the form *name***HEAD** and *name***TAIL**. If found, these are added to the **argv[]** array, respectively, before and after the command-line arguments.

For example, the word-count command **wc** is built with the **-w** option. If you set the environmental variable **WCHEAD** to **-l**, then the command

```
wc foo.c
```

has the same effect as the command

```
wc -l foo.c
```

The arguments to the function **main** are usually defined as

```
main(argc, argv)
int argc; char *argv[];
```

On some systems, a third argument is available:

```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
```

The argument **envp** is a NULL-terminated array of pointers to environmental variables, each of the form *var***=***value*. If a program is compiled without the **-w** option, **Let's C** passes an empty list as **envp**. If a program is compiled with this option, **Let's C** passes an **envp** that points to all of the MS-DOS environmental variables. Note that your program does not have to use **envp**; like **argc** and **argv**, it is available should you want it.

Linking without compiling

When you are writing a program that consists of several source files, you will need to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** *and* relink the entire program with the following command:

```
        cc -f factor.c atod.obj -lm
```

The first two arguments are the C source file **factor.c** and the *object module* **atod.obj**. **cc** recognizes that **atod.obj** is an object module and simply passes it to the linker **ld** without compiling it. You will find this particularly useful when your programs consist of many source files and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** command that is included with **Let's C**. For more information on **make**, see the entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling without linking

At times, you will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use the **-c** option to tell **cc** not to link the compiled program. This option is used most often to create relocatable object modules that can be archived into a library for later use.

For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
        cc -c factor.c
```

To link the resulting object module with the object module **atod.obj** and with the appropriate libraries, type the following command:

```
        cc -f factor.obj atod.obj -lm
```

Mini-make option

When you write a program that consists of several files of source code, you may find that, at one time or another, you need to alter the code in just one or two files, to update the program or to fix a bug. You must then recompile and relink the program to create an executable file; however, it is wasteful to recompile every file of source code when did not modify all of them. What you need is an easy way to recompile only the files that you edited, and then relink all of the object modules into an executable file.

The **-m** (mini-**make**) option allows you to create an up-to-date version of your program without recompiling all of your source files. When you use the **-m** option, the compiler compares the **date** the source file was last modified with the date its object module was last created. If the object module has a later date than the source file, then the source file has not been modified since it was last compiled, and **Let's C** will not recompile it. It will, however, re-link the previously compiled object module to build a new executable file.

This option is quite useful when recompiling programs that are built out of many different modules because unchanged source files are not recompiled unnecessarily. Note, however, that the **-m** option does not recognize header file dependencies, so you should use it with some caution.

Note, too, that this option will not work properly if you do not reset your system's time whenever you reboot. If you do not, files will be date-stamped to the default time, and **cc** will not be able organize them properly.

Assembly-language files

C makes most assembly language programming unnecessary. However, you may wish to write small parts of your programs in assembly language for greater speed or to access processor features that C cannot use directly. **Let's C** includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **example.c** and the assembly-language source file **example.s**, simply use the **cc** command as usual:

```
        cc example1.c example2.s
```

cc recognizes that the suffix **.s** indicates an assembly-language source file, and assembles it with **as**; then it links both object modules to produce an executable file.

If you wish, you can also write programs that combine assembly language with C preprocessor instructions. These files should have the suffix **.m**. When you name a **.m** file in a **cc** command, **cc** will pass it first to the C preprocessor **cpp**, and then pass what **cpp** produces to the assembler **as**. These allow you to write assembly-language programs that are independent of i8086 memory model. For more information on how to use the **.m** format, see the Lexicon entries for **larges.h** and for **as**.

Changing the size of the stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. **Let's C** by default sets the size of the stack to two kilobytes (2,048 bytes). This is enough stack space for most programs; however, some programs, such as the example program on page 26 of the first edition of *The C Programming Language*, ed. 2, require more than two kilobytes of stack. A program that uses more than its allotted amount of stack will cause a *stack overflow*; this may force you to reboot your computer.

The size of the stack cannot be altered while a program is running. Should your program need more than two kilobytes of stack, use the **-ys** option to the **cc** command. For example, to increase the stack size to 8,000 bytes, use the following command to the **cc** command:

```
        cc -ys8000 hello.c
```

Note that this option indicates the number of bytes to which you wish to set the stack, not the number of kilobytes. This must be a decimal number.

i8086 memory models

The i8086/88 microprocessor uses a *segmented architecture*. This means the i8086/88 divides memory into segments of 64 kilobytes each. No program or data element can exceed that limit.

Intel Corporation has devised a number of *models* for organizing the segments of memory into a program that is larger than any single segment. **Let's C** implements the two most useful of these: SMALL model and LARGE model.

SMALL model C programs use 16-bit pointers and **near** calls. Because a 16-bit pointer can address 65,536 bytes (64 kilobytes) of memory, SMALL model programs are limited to 64 kilobytes (one segment) of code and 64 kilobytes of data and stack.

LARGE model C programs use 32-bit pointers and **far** calls. In the LARGE model, the 32-bit pointers are converted by the processor to 20-bit addresses, so LARGE model programs can access up to a total of 1,048,576 bytes (one megabyte) of code and data. The IBM PC and and its imitators have a physical limit of 640 kilobytes.

In terms of execution, LARGE-model programs run more slowly than SMALL-model programs, but for many purposes the advantages of the expanded address space of the LARGE model outweigh the decreased efficiency.

When **Let's C** compiles a program with the **-VSMALL** option, the resulting object module follows the rules of the SMALL model. This is the default setting for the compiler. When the **-VLARGE** option is used with the **cc** command, the object program follows the rules of the LARGE model.

When you compile a program with the **-VLARGE** option, **cc** defines the manifest constant **LARGE** to the C preprocessor. This allows you to use the **#ifdef LARGE** conditional to flag model-dependent code.

Note that you cannot mix SMALL-model object modules with those compiled into LARGE model.

Debugging information

One powerful feature of **Let's C** is its ability to generate programs that you can debug with **csd**, the revolutionary Mark Williams C source debugger. **csd** lets you debug C *source code*: you can use it even if you do not know i8086 assembly language.

csd uses debugging information that **Let's C** writes into the object module as it compiles a C program. Because this information slightly enlarges the file that contains the object modules, **Let's C** does not produce it unless you request it. To include debugging information in an object module, use the **-VCSD** option before the file name argument on the **cc** command line:

```
        cc -VCSD hello.c
```

The manual for **csd** describes the C source debugger in full.

A module compiled with the **-VCSD** option will run exactly the same as one compiled without it, but the size of the object module will increase by a few bytes. The size of the executable file will increase, due to the special symbol table that the **-VCSD** option builds.

With some programs that already approach the limits of the SMALL model, compiling with the **-VCSD** option may make them too large to be executed as SMALL model programs. In that case, recompile the program with the **-VCSD** and **-VLARGE** options; the latter option will create a LARGE model output.

To remove the debug symbol table from the programs that you compile with the **-VCSD** option, use the **strip** command. **strip** is described in the Lexicon.

i8087 programs

The Intel i8087 chip is a numeric data processor that is designed to execute mathematics routines. It increases the speed with which programs can compute floating-point numbers. Because of its expense, however, many personal computers do not include this chip.

Let's C by default uses a special set of libraries that *sense* if an i8087 is present. When you compile a program with these libraries and then run it, the library routines automatically check to see if an i8087 is present on your computer. If an i8087 is present, then floating-point arithmetic is automatically computed it; otherwise, it is computed in software. Thus, a program compiled with **Let's C** can be run to best advantage on machines that have an i8087 as well as on machines that do not, without needing to recompile the program.

If you know that the program you are compiling will always be run on a machine with an i8087, you may wish to use the libraries that use the i8087 exclusively. You can do this by specifying the **-VNDP** option to the **cc** command. For example, to compile the program **factor** to run exclusively with an i8087, use the following command:

```
        cc -VNDP factor.c atod.c -lm
```

This program will *not* run on a machine that does not have an i8087; however, the executable file will be somewhat smaller than one that uses the sensing libraries, and will run slightly faster.

Options passed to MS-LINK

The compiler controller **cc** passes a number of its options directly to MS-LINK. The following summarizes them.

-y/*switch*
: This option sends *switch* directly to MS-LINK. *switch* can be any MS-LINK command or option.

-ym Tell MS-LINK to create a *map file* that can be used with the MS-DOS utility **DEBUG**. For more information on **DEBUG** and its uses, see your MS-DOS manual.

-yn **Increase the number of segments allowed in a program to 1,024 using the MS-LINK** *segments* switch. Note that the *segments* switch is used only version of MS-LINK later 3.0. Earlier versions use the *x* switch to increase the number of segments.

-ys*number*
Set the stack size to *number* where *number* is a decimal integer that gives the number of bytes you desire. The stack is set by default to two kilobytes; to set the stack, for example, to 16,000 bytes type:

```
cc -ys16000 foo.c
```

-yf Tell MS-LINK to write a linker command file. This option is useful, should you ever have trouble linking a program and wish to see just what MS-LINK is doing, or if you wish to fine-tune how your program is linked.

-yu*name*
Undefine the variable *name* for MS-LINK. This tells MS-LINK to link in the library module called *name* even though it is not named explicitly in your program. For example, the command line

```
cc -yuprintf example.c
```

tells MS-LINK to link the library module **printf** into your program, even if your program does not explicitly call **printf**. This tactic is sometimes quite useful.

Compiling programs without STDIO

STDIO is an abbreviation for *standard input and output*. Library routines use STDIO to write to the screen or read the keyboard. Most of the runtime startup routines included with **Let's C** call STDIO, whether your program uses any STDIO functions or not.

If you have a small program that does not use any of the STDIO functions, you can stop STDIO from being linked into your program by using the **-ns** option. This will make your program noticeably smaller and more efficient. Note that the **-ns** option gives your program a different version of the **exit** command, one that does not call **fclose** or **fflush**

Using default options

To make using **Let's C** even simpler, **cc** helps you specify default options with the environment variables **CCHEAD** and **CCTAIL** These variables give options that **cc** adds to the command line you give it: it adds **CCHEAD** to the start of the command line (after the "cc"), and it appends **CCTAIL** to the end of the command line.

How you can build a *name***HEAD** and *name***TAIL** feature into your program is described above, in the sub-section *Tailoring the command line interface.*

When you installed **Let's C**, the **install** utility instructed you to set **CCHEAD** so that **Let's C** would read the file **CCARGS**. If you wish, though, you can attach additional variables to **CCHEAD**, or add them to the file **ccargs**.

For example, suppose you always wish to use the options **-V** and **-f** (for "verbose" compilation and floating-point routines), and always link in the mathematics library with the **-lm** option (which, as you recall, must be mentioned *after* the source and object modules). Rather than retype these options every time you type a **cc** command line, you can set **CCHEAD** and **CCTAIL** as follows:

```
set CCHEAD=@a:\lib\ccargs\ -V -f
set CCTAIL=-lm
```

Note that if your computer has a hard disk, **CCHEAD** should indicate that **ccargs** is on drive C, rather than drive A, as shown above. Thereafter, when you type

```
        cc factor.c atod.c
```

it will be as if you had typed

```
        cc -V -f factor.c atod.c -lm
```

in addition to the arguments contained in **ccargs**. These environmental variables allow you to pass variables to **Let's C** with ease. To ensure that these variables are set every time you boot your system, be sure to enter the **set** commands described above into the file **autoexec.bat** on your MS-DOS boot disk.

Where to go from here

For more information on compiling, see the Lexicon entry for **cc**. This entry summarizes all of **cc**'s options, and presents many that are not discussed here. For more information on the assembler **as**, see its entry in the Lexicon as well.

The following section introduces the MicroEMACS screen editor. If you have worked the exercises in this part of the book, you have already used MicroEMACS a little; this tutorial, however, will show you how to use all of its advanced features to input text quickly and easily.

Then comes an introduction to **make**, the Mark Williams programming discipline. If you are building programs that use multiple files of source code, you will find **make** to be an invaluable tool.

Section 6, *Questions and Answers*, answers frequently asked questions about **Let's C** and its utilities. If you have a question about **Let's C**, look here first. You may well find the information you need.