Mix C TOOLS Table of Contents

Chapter 12 Compiler Controls

Chapter 1

The C Compiler

The C Compiler (CC.COM) is executed using a simple 1 line command of the following form. The command line parameters shown inside braces {} are optional.

CC source {-Listing} {[stacksize]}

where

source	is the name of the C source file
listing	is the name of the listing file or devices
stacksize	is the size of the stack

1.1 Source

The source name may contain an extension. However if an extension is not present the compiler appends the extension (.C) to the end of the name. There are several compiler options that may be placed in the source file or in a header file that is included by the source file. See Chapter 12 of the Reference Manual for a discussion of the compiler options.

1.2 Listing

The –L option is used to redirect the compiler listing. By default the listing is sent to the screen. The compiler listing can be turned off by simply using –L without a file or device name. The compiler listing can be sent to a file or device (eg. the line printer) by immediately following –L with a name. Any valid file name or any of the devices discussed in the Getting Started section of this manual may be used to redirect the listing.

If errors are detected by the compiler, error messages will appear in the listing. Error message lines have a string of asterisks "*****" at the beginning of the line. An up arrow symbol ^ points to the approximate location within the line where the error is detected. The ^ symbol is followed by one or more error codes. It is possible for a single error to generate more than one error code. The first error code identifies the original cause of the error.

If any errors are detected, a summary of the meanings of the error codes generated is printed at the end of the listing. Also, an error file named C.ERR is created that contains a list of the C source lines that contained errors. If an error occurs on a line containing reference to a macro definition (#define), the expansion of the macro is also displayed on the listing to aid in determining the cause of the error.

1.3 Stacksize

The compiler's stack size may be set by specifying a number inside square brackets []. The optional stack size parameter should be used only if the compiler terminates with an OUT OF STACK or OUT OF HEAP error message. If an OUT OF STACK error occurs, execute the compiler again with a larger stack size. If an OUT OF HEAP error occurs, execute the compiler again with a smaller stack size. The stack size may be specified as an integer number followed by the letter K. The letter K implies the number 1024. For example, [4K] is equivalent to [4096].

Large files that will not compile even after adjusting the stack size must be split into smaller files. The compiler creates a symbol for each identifier in the file it is compiling. An identifier is a function name, a variable name, or a constant defined using #define. Since a small file will generally contain fewer identifiers than a large file, splitting the file into several smaller files will normally solve any compiler memory problems. Excessive use of #define is a typical reason for the compiler not having enough memory. If you include a lot of header files containing #defines that are not used by the program, eliminate the unused #defines and you may have enough memory.

1.4 Object

The compiler creates an object file with the same name as the source file but with the extension (.MIX). The object file is written to the same drive that contains the source file. The object file must be linked to the necessary runtime routines before the program can be executed.

1.5 Examples

The following examples compile the source file named FIRST.C on drive B and create and object file named FIRST.MIX on drive B.

```
CC B:FIRST                  /* listing to screen              */
CC B:FIRST -L               /* listing turned off             */
CC B:FIRST -LLST            /* listing to printer             */
CC B:FIRST –LB:FIRST.LST    /* listing to B:FIRST.LST         */
CC B:FIRST –L [5K]          /* listing off / stack size 5124  */
```

Chapter 2

The Linker

The linker (LINKER.COM) links together separately compiled object files and builds an executable command file. The file named RUNTIME.OVY must be present when the linker is executed. This file contains the low level runtime routines that must be linked to a program. The linker first searches for RUNTIME.OVY on the selected drive and if not found then searches drive A. The READ.ME file explains how you can apply a patch to change this search path. You may want to change it if you use a hard disk, especially if the hard disk is not drive A. The file named CLIB.MIX should also be present when the linker is executed. The file contains the C function library. All programs use some of the functions contained in CLIB.MIX. The linker may be executed in either of two ways.

2.1 Single Line Command

The easiest way to build a command file is to use the single line command. The single command has the following form. The parameters inside braces {} are optional.

LINKER object {,object…} {[stacksize]}

where:

object	is the name of an object file
stacksize	is the size of a program's stack

Object must be the name of an object file created by the compiler. The linker appends the extension (.MIX) to the name if no extension is specified. Multiple object files can be linked by separating the object file names with a comma. There must be no spaces between the comma and the next file name.

The optional stacksize may be used to specify a default stack size that the linker builds into the command file. If the stacksize parameter is omitted, the linker allocates half of the available memoryto the stack and the other half to the heap. As with the compiler, the letter K may be used when specifying the stack size.

After reading the object file(s), the linker automatically searches for the file named CLIB.MIX to extract the C library functions that are referenced. The linker searches for the CLIB.MIX on the selected drive and if not found searches drive A. As with the RUNTIME.OVY file, the search path may be altered by applying a patch to the linker.

After linking the C library functions in CLIB.MIX to the object file(s), the linker builds a command file with the same name as the first object file but with the extension (.COM). The command file is written to the drive containing the first object file. When multiple

object files are linked, only one should contain the main function. The file containing the main function is typically the first object file specified.

The command file is executed by simply typing its name. The default stack size built into the command file can be overridden by specifying the stack size in brackets following the command file name. For example if the command is named TEST.COM, then typing TEST [6000] will execute the program with a stack size of 6000 bytes.

The single line command builds a command file that does not contain any of the low level runtime routines in RUNTIME.OVY. As a result the command files are very small. All command files built this way share the RUNTIME.OVY file which must be present when the command files are executed. The command file searches for the RUNTIME.OVY file on the selected drive and if not found searches drive A. If the linker is patched to change the search path to RUNTIME.OVY, the command files built using the single line command will inherit the same search path.

If unresolved references remain after loading the object file(s) and searching CLIB.MIX, the linker displays a menu of commands and lets you continue the link process. The Symbols may be used to determine the name of the function or external variable that is referenced but not linked to the program. The following section discusses all of the linker commands.

Examples:

```
LINKER TEST                    /* builds TEST.COM on selected drive     */
LINKER TEST [3000]             /* builds TEST.COM with stack = 3000     */
LINKER A:TEST1,B:TEST2 [8K]    /* builds A:TEST1.COM with stack = 8K */
```

2.2 Command Menu

The linker can also be executed by simply typing LINKER. The linker then displays a menu of commands and waits for a command to be selected.

```
H=Help,      L=Load,      F=Find,      FA=Find All,
S=Symbols    R=Run,       B=Build,     C =Change Runtime
I=Init,      E=Exit,      O=Overlay,   OA=Overlay Area
>>
```

All commands require only the single letter, although longer names will also be accepted. To execute a command, simply type its first letter and press the return key. If more information is required, additional prompts will be supplied. You may type H and press the return key to get help on each command.

2.2.1 Load Command

The load command is used to load object files into memory. To load an object file, type L and press the return key. The load command will prompt with "File)s) =". Type the name of one or more files in standard notation. If the extension is omitted, then the extension (.MIX) is appended to the file name. Multiple file names must be separated by commas. You may optionally follow the Load command with the file names to avoid the prompt.

The Load command opens the object file(s) and loads the object code into memory. As each function is loaded, its name is displayed on the screen. This allows you to monitor the load process as the object file(s) are loaded into memory.

The Load command should be used to load the object file containing the main function and the object files containing any other separately compiled functions. These functions will cause references to some of the library functions in CLIB.MIX. The Find command should then be used to link the C library functions to the program.

2.2.2 Find Command

After loading the program with the Load command, the Find command can be used to extract any commands needed from one or more library files.

The Find command is identical to the Load command except that the object files are searched. Only referenced (called) functions in the object file are loaded. The Find command makes one pass through the library file. As each function id encountered, the Find command checks to see if it has been referenced (called). The function is extracted only if it has been referenced.

In some cases, the order in which library files are searched is important. For example, if library X contains a function that calls another function in library Y, then library X should be searched first. Searching library X first causes a reference to the function in library Y. Then when library Y is searched, the function is extracted. If library Y is searched first, there is no reference to the function and it therefore is not extracted. Although library functions can be searched multiple times to remove all references, time can be saved by searching them in the proper order.

The order in which functions appear in a single library can also be important. For example, if function X calls function Y then function X should precede function Y in the library file. Then if function X is extracted, it causes a reference to function Y which will also be extracted because it appears later in the file. If the functions in a library file are not in the proper order, the library will have to be searched more than once to remove all references.

2.2.3 Find All Command

The Find All command is identical to the Find command except that it searches an object file multiple times if necessary to extract all of the functions that are referenced. If the functions in an object file are not in the proper order, the file will be searched more than once.

The Find All command can be used to avoid having to manually execute the Find command until all references are resolved. When multiple files are specified with this command, the files are searched from left to right. If all references are not resolved, searching starts over again with the first file. The search is terminated when either all references are resolved or a pass through all of the files does not resolve any more references.

2.2.4 Symbols Command

The linker stores the name and addresses of each function as an object file is loaded. Also stored are the names of functions that have been referenced (called) by another function, but have not yet been loaded into memory. This symbol can be displayed with the Symbols command.

The Symbols command all currently defined or referenced symbols. The display is paused if the screen fills and the prompt "* MORE *" appears at the bottom of the screen. Press the return key to view the remaining symbols.

One function name is displayed per line. After the function name is a character that describes the use of that function. A "D" indicates that the name is defined. This means that the function has been loaded into memory. An "R" indicates that the function has been referenced but not yet defined. This means that a function that has already been loaded makes a call to this function. All functions that are called must be loaded before the program can be executed.

The last item on the line is the address of the symbol. If the symbol is defined "D", then this is the address in memory where the function is located. If the symbol has not been defined "R", then this is the address of the last place it was used (called).

2.2.5 Run Command

After all of the object files of a program have been linked to the necessary library functions, the program can be executed with the Run command. The linker prompts for the amount of stack space required by the program. The default is to allocate half of the unused memory to the stack and the other half to the heap. If these space allocations are sufficient then simply press the return key. Otherwise enter a value. The letter K may be used as a multiplier of 1024 when specifying the stack.

2.2.6 Build Command

Once a program has been linked, the Build command can be used to store the program to disk as an executable command file. Like the Run command, the Build command prompts for the stack size. Simply pressing the return key allocate equal amounts of memory to the stack and heap.

The next prompt asks weather or not to include runtime support in the command file. The runtime support is contained in the file named RUNTIME.OVY. The prompt may be answered by typing Y for yes of N for no. Simply pressing the return key is equivalent to typing N. Very small command files are created if the runtime is not included. If a command file is built without the runtime support, the RUNTIME.OVY must be present when the command file is executed. If the runtime support is included in the command file, the command file can be executed without any other files being present.

The last prompt asks for the name of the command file. This is the name of the file that will contain the program. You must type in a file name with a COM extension. For example, PAYROLL.COM might be the name of the command file. The Build command causes the program to be saved to disk in command file format. The linker terminates after building the command file.

The program may then be executed by simply typing the name of the command file. Parameters may be specified following the command file name. The < and > symbols may be used to redirect stdin and stdout respectively. The >> symbol indicates that the output sent to stdout will be appended to the end of its associated file.

Example:

```
PAYROLL <PAYROLL.DAT >>REPORT.DAT
```

2.2.7 Change Command

The Change command allows the linker to use a runtime file other than the default RUNTIME.OVY file. The SMALLCOM.OVY file contains a stripped down version of RUNTIME.OVY. It has everything except the routines for performing long integer and floating point arithmetic. If your program does not use the types long, float, or double then the SMALLCOM.OVY file can be used to build smaller command files. This only makes a difference when you use the Build command and specify that the runtime be included in the command file.

2.2.8 Init Command

The Init command clears the symbol table and redisplays the command menu. This command may be used if the wrong file is load by mistake. It is equivalent to exiting to the operating system and then executing the linker again.

2.2.9 Exit Command

The Exit command causes the linker to terminate and return to the operating system.

2.2.10 Overlay Command

When a program becomes too large to fit in the available memory it becomes necessary to overlay portions of it. The Overlay command allows a group of functions to share the same area in memory with one or more other groups of functions. Each group may contain one or more functions. When groups of functions are overlayed with one another, only one of the groups will reside in memory at any given time. If a function is called that is not currently in memory, the group in which it resides will automatically be loaded into memory. The size of the shared area of memory is equal to the size of the largest group of functions.

An executable program that uses overlays will consist of just two files, the command file and a single overlay file. However in order to utilize the Overlay command, your program must be designed in a modular fashion. Each group of functions that reside in an overlay must be compiled in a separate file from the other groups. For example if the two functions X and Y are to overlay each other, then function X and function Y must reside in two separate object files. Similarly if a group of functions A, B, and C are to be overlayed with a second group of functions D and E, then functions A, B, and C should be together in one file and functions D and E should be together in a separate file. If you design your program such that only related functions reside in the same source file, then it will be quite easy to overlay the program if it becomes too large.

The overlay loader in the file named OVLOADER.MIX provides automatic overlay loading. If a function that resides in an overlay is called, the overlay loader will automatically load the appropriate overlay if necessary. The overlay loader even allows a function in one overlay to call a function in another overlay. There are no restrictions with regard to function calling when using overlays. The only restriction is that static variables cannot be used inside overlays. The only concern that you have when overlaying a program is efficiency. You want to overlay the program in a manner that does not dramatically effect performance. For example, if you have a function that repeatedly calls a second function, then you would not want these two functions to reside in two different overlays. You would want to put them in the same overlay.

The first step in overlaying a program is to split the program up into groups of functions. Try to keep related functions together in the same group. Each group should be in a separate file. The second step is to identify which groups could be overlayed with other groups without severely impacting performance. Most large programs contain some relatively independent sections that can be overlayed with one another without impacting performance. Once these two steps are completed, you are ready to link the program. The third step is to execute the linker and use the Load command to load in the main function and all other functions that are not part of the overlay. The fourth step is to use the Find command to search the CLIB.MIX and extract any library functions that are referenced. Now you are ready to execute the Overlay command.

Type O and press the return key. The first time that the O command is executed it prompts for two file names. The first prompt allows you to specify a file name for your program's overlay file. The linker will use this name when it creates the overlay file. The second prompt is for the name of the file containing the overlay loader. If you simply press the return key, the file named OVLOADER.MIX is loaded into memory. The overlay loader is the last thing loaded before the start of the overlays. Everything up to this point constitutes the root of the program. Everything in the root is always memory resident. All subsequent object files loaded become part of an overlay.

After answering the two prompts, you are ready to load in the object files containing the functions to be overlayed. First load in the object file(s) which contain the functions for the first overlay. Then execute the O command again and load in the object file(s) which contain the functions for the second overlay. Each time the O command is executed the linker sets the load address back to the beginning of the overlay area and a new overlay is started. This process should be repeated until all of the overlays have been loaded.

Now type OA to execute the Overlay Area command and use the Find command to search the CLIB.MIX file once again. This insures that all library functions referenced by functions in the overlays are linked to the program. Only the library functions not already present in the root are extracted.

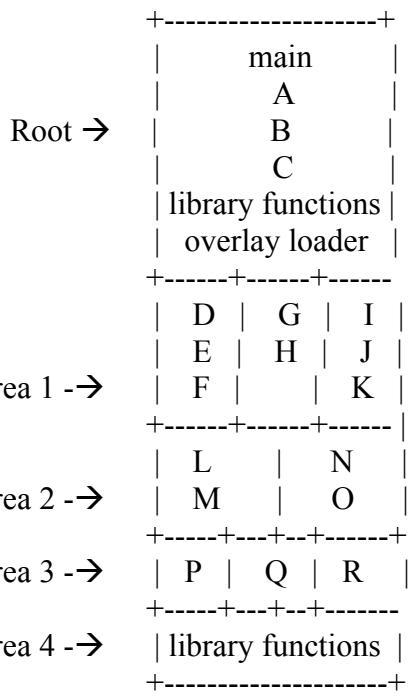
The last step is to execute the Build command. When the Build command is executed the linker will create two files. All the functions in the root are written to the command file. All other functions are written to the overlay file that you named.

2.2.11 Overlay Area Command

The OA command is used to start a new overlay area. The address of the new overlay area is equal to the address of the previous overlay area plus the size of its largest overlay. With multiple overlay areas, the overlay is memory resident in each overlay area at any given time. Having multiple overlay areas gives you a great deal of flexability. If the functions are well arranged you can save a lot of memory without sacrificing performance.

When you execute the OA command a new overlay area is started. Load in the object files that go into the first overlay. Then execute the O command and load in the object files that go into the second overlay. Repeat the O command until all of the overlays for this overlay area have been loaded. You can then execute the OA command to start the next overlay area, etc..

The following diagram provides an example of how memory might be partitioned using overlays and multiple overlay areas. Each letter represents a function. The root contains the main function, the 3 functions A, B, and C, library functions extracted from CLIB.MIX, and the overlay loader. Overlay area 1 is shared by 3 overlays, 2 of which contain 3 functions and one that contains 2 functions. Overlay area 2 is shared by two overlays, each of which contains 2 functions. Overlay area 3 is shared by 3 overlays, each of which contains a single function. Overlay area 4 contains any library functions referenced by an overlay but not present in the root. The only limit to the number of overlays is the size of the table in the overlay loader. The size of this table can be increased using the eXtra command discussed next.



2.2.12 eXtra Command

The X command sets the number of eXtra table entries in the overlay loader. Each function in an overlay that is called from either the root or from another overlay needs one table entry. Functions that are only called from inside the overlay in which they reside do not require a table entry. When the first overlay is started, all functions that are called from the root but not yet loaded are added to the table in the overlay loader. Extra empty slots are reserved for calls from one overlay to another. The number of extra entries is set by the X command (the default is 32). If you wish to use the X command to change the number of extra entries, it must be executed before the first O command.

2.2.13 Map Command

The M command causes the linker to send link information to a file. When executed it prompts for the name of the file to use. For each subsequent function that is loaded, the linker prints the name of the function, its load address, and its size (in both hex and decimal) in the file. The origin of each overlay is also printed.

2.3 Error Messages

The linker displays the following messages when an error is detected.

*** CANNOT OPEN FILE

This message is generated when the linker cannot find the file specified with the Load, Find, or Find All commands. This may be caused by a misspelling or specifying the wrong disk drive.

*** UNRESOLVED REFERENCES

When the Run command is used to execute a program or the Build command to generate a command file, the linker checks to insure that all of the functions that are called within the program have been loaded. If there are functions that have been called but have not been loaded then this message is generated. At this point, you can load the required files and repeat the command. The Symbols command may be used to list names of the functions that are not yet defined. These will be followed by an R tag.

*** INVALID OBJECT TAG

This message is displayed when a load is attempted on a file that is not a valid object file. The most frequent cause of this error is an attempt to load the source program instead of the object.

*** ILLEGAL REFERENCE

This message signifies an inconsistent structure is an object file. It is an indication that the file has been damaged. The typical solution is to recompile the offending program.

2.4 Examples

The following examples illustrate use of the command menu to build command files. The examples assume that the two files MAIN.MIX and FILL.MIX (see Getting Started section) are on drive B the selected drive is A. The second example assumes that a copy of the SMALLCOM.OVY file is on drive A.

Include runtime from RUNTIME.OVY

```
LINKER
>>L B:MAIN,B:FIX
>>F CLIB
>>B
Stack size: 15K
Include runtime support in command file? Y

FILE     = B:MAIN.COM
```

Include runtime from SMALLCOM.OVY

```
LINKER
>>C SMALLCOM.OVY
>>L B:MAIN,B:FIX
>>F CLIB
>>B
Stack size 15K
Include runtime support in command file? Y

FILE     = B:MAIN.COM
```

Chapter 3

Optional Development Tools

The compiler and linker are tools that must be used in the development of a program. This chapter describes other tools that may optionally be used.

3.1 The Shrink Utility

The shrink utility (SHRINK.COM) optimizes an object file for space. Its purpose is to reduce the amount of space used by the object code when it is loaded into memory.

SHRINK object

where:

object is the name of the object file (.MIX extension)

The shrink utility appends the extension (.MIX) to the object file name if no extension is specified. It then loads the object file into memory and optimizes it function by function. The optimized function code is stored in a temporary file. When the last function has been optimized the original object file is deleted and the temporary file is renamed as the original.

The shrink utility should generally be used only when developing large programs. The amount of memory saved from optimization is usually in the range of 10% to 25%.

3.2 The Speedup Utility

The speedup utility (SPEEDUP.COM) optimizes an object file for speed. Its purpose is to increase the execution speed of the object code.

SPEEDUP object

where:

object is the name of the object file (.MIX extension)

The speedup utility appends the extension (.MIX) to the object file name if no extension is specified. It then loads the object file into memory and optimizes it function by function. The optimized object code is stored in a temporary file. When the last function has been optimized the original object file is deleted and the temporary file is renamed as the original.

The speedup utility would generally be used only as a last step in the development process. Once a program is working you can use the speedup utility to increase the execution speed. The size of the object code increases after being processed by the speedup utility. Therefore with large programs, it is best to optimize only the functions that are frequently executed. In typical large programs, about 90% of the time is spent executing only about 10% of the functions. If you can identify the functions in this critical 10%, the speed of the program can be increased without substantially increasing its size.

3.3 The Convert Utility

The convert utility (CONVERT.COM) converts an object file from binary to ASCII or vice versa.

CONVERT object

where:

object is the name of the object file (.MIX extension)

The convert utility appends the extension (.MIX) to the object file name if no extension is specified. If the object file contains binary object code it is converted to ASCII. If the object file contains ASCII object code it is converted converted to binary. The convert utility always produces the opposite of what the object file currently contains. A temporary file is used to store the object code as it is converted. When the conversion process is complete, the original object file is deleted and the temporary file is renamed as the original.

ASCII object files are quite useful. They can be edited or processed by a program just like any normal ASCII text file. If you decide to change or add additional functions to the STDLIB.C, PRINTF.C, or SCANF.C files you will need to use the convert utility. First make the changes and compile all three files. Next use the convert utility to convert these three files and the NOSOURCE.MIX object file to ASCII. The NOSOURCE.MIX contains the C functions for which source code is not supplied. Now append the object files together to form a single file named CLIB.MIX. The object files should be appended together in the order PRINTF.MIX, SCANF.MIX, STDLIB.MIX, and then NOSOURCE.MIX. You can do this using an editor or a simple C program. Now use the convert utility to convert the CLIB.MIX file to binary. You now have a new C library.

The functions in an ASCII object file are very easy to identify. Each function module starts at the beginning of a line with the letter J followed by the first 8 characters of the function name. Sometimes it is very convenient to use the convert utility to convert an object file to ASCII, use an editor to replace one of the object modules, and then use the convert utility to convert the file back to binary. The linker will load ASCII format object files but binary format object files are smaller and therefore faster.

Chapter 12

Compiler Controls

12.1 Preprocessor Statements

Statements beginning with the # sign are known as preprocessor statements. The term preprocessor is used for historical reasons. Most C compilers make a separate pass through a C program to process just the statements beginning with the # sign, creating an intermediate file that is then compiled. The # sign is usually required to begin in the first column. This compiler however makes only a single pass through source program and the # sign may begin in any column. The following sections explain the preprocessor statements.

12.1.1 #include

Format:

#include "file_name"

where file_name is the name of a disk file

Description:

The #include statement is used to include another C source file during a compile. When the compiler encounters a #include statement, it compiles the C source in the named file before continuing. It has the same effect as inserting the source in the file at the location of the #include statement. The file name used with #include must be enclosed in either double quotes "" or angle brackets <>. On some systems, the angle brackets may cause the file to be searched for on a specific disk drive.

Example:

```
#include "stdio"
#include "globals"
main()
{
        #include "locals"
        #include "body:
}
```

12.1.2 #define

Format:

```
#define MACRO_NAME macro_defination

where MACRO_NAME is an identifier
and     macro_definition  is a text string
```

Description:

The #define statement is used to define a macro. A macro is an identifier and the text that the compiler substitutes when the identifier is later encountered in the program. The identifier corresponds to the name of the macro while the text corresponds to the actual definition of the macro.

The main use of macros is to define program constants. However, a macro can also have arguments. This allows macros to be used as shorthand notations for complex expressions that are used in many different places in a program. An argument list may follow the identifier to provide a means of passing information to the macro definition. The macro definition text would then contain the arguments as part of the text string. The macro is used much like a function call. The identifier is followed by an argument list and the compiler makes the appropriate substitution. The arguments passed to a macro are passed as text.

Example:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

main()
{
        float   f1, f2
        printf("Enter 2 numbers: ");
        scanf("%f%f", &f1, &f2);
        printf("The largest  number is %f\n", MAX(f1,f2));
:       printf("The smallest number is %f\n", MIN(f1,f2));
}
```

12.1.3 #undef

Format:

```
#undef MACRO_NAME
```

where MACRO_NAME is an identifier

Description:

The #undef statement is used to undefine a macro. An undefined macro is no longer known to the compiler. The #undef statement can be used to free the space used by a macro definition that is no longer needed.

Example:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

main()
{
        float   f1, f2
        printf("Enter 2 numbers: ");
        scanf("%f%f", &f1, &f2);
        printf("The largest  number is %f\n", MAX(f1,f2));
:       printf("The smallest number is %f\n", MIN(f1,f2));
        #undef MAX
        #undef MIN
}
```

12.1.4 #ifdef

Format:

```
#ifdef MACRO_NAME
        statements
#endif
```

Description:

The #ifdef statement is used to conditionally compile selected statements of a program only if the named macro is defined. The conditionally compiled statements are preceded by the #ifdef and followed by a #endif. The program lines inbetween are then compiled only if the specified macro is defined. The compiler listing will show a – sign preceding a line that is not compiled.
Example:

```
#include “stdio”
#define DEBUG

writeline(f, sp)
char *s;
FILE *fp;
{
        fputs(s, fp)
        putc(‘\n’, fp)
        #ifdef DEBUG
        puts(s);            /* output to screen */
        #endif
}
```

12.1.5 #ifndef

Format:

```
#ifndef MACRO_NAME
        statements
#endif
```

Description:

The #ifndef statement is used to conditionally compile selected statements of a program only if the named macro is not defined. The conditionally compiled statements are preceded by the #ifndef and followed by a #endif. The program lines in between are then compiled only if the specified macro is not defined. The compiler listing will show a – sign preceding a line that is not compiled.

Example:

```
#include “stdio”
#include “constant”

#ifndef MAXBUF
#define MAXBUF 81
#endif

main()
{
        char   s[MAXBUF];
        while (gets(s) != NULL) puts(s);

        #include “body:
}
```

12.1.6 #if

Format:

```
#if constant-expression
        statements
#endif
```

Description:

The #if statement is used to conditionally compile selected statements of a program only if the constant-expression is non-zero (true). The conditionally compiled statements are preceded by the #if and followed by a #endif. The program lines in between are then compiled only if the expression does not evaluate to 0 (false). The compiler listing will show a – sign preceding a line that is not compiled.

Example:

```
#define DOLLAR 1
#define COMMA  1

main()
{
        int    flag;
        char s[20];
        #if DOLLAR && COMMA
        flag = 80;
        #endif
        #if DOLLAR && !COMMA
        flag = 16;
        #endif
        #if !DOLLAR && COMMA
        flag = 64;
        #endif
        #if !DOLLAR && !COMMA
        flag = 0;
        #endif
        ftoa(1000.0, s, flag, 8, 2);
        puts(s);
}
```

12.1.7 #else

Format:

```
#else
        statements
```

Description:

The #else statement may be used with the #ifdef, #ifndef, or #if statements. The #else and its accompanying statements should be placed just prior to the #endif. The #else statements are compiled only if the preceding statements (of the #ifdef, #ifndef, or #if) are not compiled.

Example:

```
#define MOD1 0
#define MOD3 0

#if MOD1 || MOD3
#define SCREEN_WIDTH   64
#define SCREEN_HEIGHT 16
#else
#define SCREEN_WIDTH   80
#define SCREEN_HEIGHT 24
#endif

main()
{
        printf("width  = %d\n", SCREEN_WIDTH);
        printf("height = %d\n", SCREEN_HEIGHT);
}
```

12.1.8 #line

Format:

```
#line constant
```

Description:

The compiler numbers lines sequentially starting at 1 when creating a program listing. The #line statement may be used to set the line number of the next line of the compiler listing. The compiler will then number subsequent lines of the listing beginning with the specified line number.

Example:

```
#include "stdio"
#line 1

main()
{
        /* main starts at line 1 */
}
```

12.2 Compiler Options

Compiler options are switches that control various characteristics of the compiler. A compiler option is specified using a comment. The format is as follows.

/*$COMPILER_OPTION*/ or /*$NO COMPILER_OPTION*/

All compiler options must be specified in upper case and there must be no spaces between the /* and the $ sign.

A compiler option may be turned on or off. The option is turned on unless preceded by NO, in which case it is turned off. Except where noted, compiler options may appear anywhere in a program. They may be turned on or off as needed. For each option, there is a default setting. If the default setting is the one desired, then the option need not de specified.

12.2.1 CONVERT Option

Format:

/*$CONVERT*/ or /*$NO CONVERT*/

Default: /*$CONVERT*/

Description:

The convert option may be used to prevent automatic type conversion on the arguments in a function call. Normally, arguments are converted according to C's defined conversion rules for expressions. For example, char is automatically converted to int and float is automatically converted to double. Therefore, it is normally not possible to call a function that has char or float arguments because it is not possible to pass a char or float value. When the convert option is turned off, the compiler does not generate code to perform type conversions on function arguments. Then any type of value may be passed to a function.

Example:

```
printvalue(c, f)
char    c;
float    f;
{
        printf(%c %f", c f);
}

main()
{
        char    c  = 'a';
        float    f  =  1.0;
        /*$NO CONVERT*/
        printvalue(c f);
        /*CONVERT*/
{
```

12.2.2 LIST Option

Format:

/*$LIST*/ or /*$NO LIST*/

Default: /*$LIST*/

Description:

The list option may be used to turn the compiler listing on and off.

Example:

```
/*NO LIST*/
#include "stdio"
/*LIST*/

main()
{
        puts("Do not show listing of stdio");
}
```

12.2.3 LISTMACRO Option

Format:

/*$LISTMACRO*/ or /*$NO LISTMACRO*/

Default: /*$NO LISTMACRO*/

Description:

The listmacro option may be used to cause the compiler to generate the expansions of macros on the compiler listing. When this option is on, all lines that contain a macro will be followed by the expansion of the line.

Example:

```
/*LISTMACRO*/

#include “stdio”

main()
{
        int c;
        while ((c=getchar()) != EOF
            putchar(c);
}
```

12.2.4 NESTCMNT Option

Format:

/*$NESTCMNT*/ or /*$NO NESTCMNT*/

Default: /*$NO NESTCMNT*/

Description:

The nestcmnt option may be used to allow all comments to be nested.

Example:

```
/*NESTCNMT*/
/* /* This is a comment */ within a comment */
```

12.2.5 PAGESIZE Option

Format:

/*$PAGESIZE n*/

Default: /*$PAGESIZE 60*/

Description:

The pagesize option may be used to set the number of program lines that are printed on each page of the compiler listing.

Example:

```
/*PAGESIZE 10*/

#include “stdio”

main()
{
        /* There should be 10 program lines per page */
```

12.2.6 SIGNEXT Option

Format:

/*$SIGNEXT*/ or /*$NO SIGNEXT*/

Default: /*$NO SIGNEXT*/

Description:

The signext option may be used to cause characters to be sign extended when converted to integers in expressions. This may be useful when using variables of type char for numeric calculations. Sign extentsion means that if the char value is negative, then the conversion to int will also be negative. Without sign extension, conversion of char to int will always produce a positive result.

Example:

```
main()
{
        char  c = -1;
        printf("without sign extend: %d\n", c)
        /*$SIGNEXT*/
        printf("with sign extend: %d", c);
}
```

12.2.7 UPPERCASE Option

Format:

/*$UPPERCASE*/ or /*$NO UPPERCASE*/

Default: /*$UPPERCASE*/

Description:

The uppercase option controls the case of function names in the object code. When the object is on, the compiler converts all function names to upper case in the object code. When the function is off, the compiler outputs the function names in the same case as they appear in the program. The main function should be output in upper case letters since the C library references it in upper case. This can be changed by recompiling the C library with the uppercase case option turned off.

Example:

```
/*NO UPPERCASE*/

function()
{
}

/*UPPERCASE*/

main()
{
}
```

12.2.8 WIDELIST Option

Format:

/*$WIDELIST*/ or /*$NO WIDELIST*/

Default: /*$NO WIDELIST*/

Description:

The widelist option may be used to cause the addresses of the generated code to be printed on the compiler listing. All addresses are relative to the beginning of a function. This can be useful as a debugging aid when used in conjunction with the linkload utility. When a program terminates with a runtime error, the address of the last instruction executed is displayed on the screen. By using the symbols command of the linkload utility, that starting address of each function may be found. Separately compiled files must be loaded in the same order as they were when executing the program for this to be effective. By comparing the terminating address with the starting address of each function, the function that was executing when the termination occurred may be determined. Subtracting the starting address of this function from the terminating address gives the address relative to the beginning of the function. This can be compared to the compiler listing to determine the approximate line in the function where the error occurred.

Example:

```
/*WIDELIST*/
one()
{
   int  a = 1;
   return a;
}

main()
{
        printf("One = %d ", one());
}
```

12.2.9 ZERO Option

Format:

/*$ZERO*/ or /*$NO ZERO*/

Default: /*$NO ZERO*/

Description:

The zero option sets a runtime flag that effects two things. When the zero option is on, all local variables in a function are initialized to 0 when the function is called. Also, the memory allocated by the standard calloc function is initialized to 0. The zero option only has an effect when the main function is compiled.

Example:

```
/*ZERO*/
main()
{
        char *calloc(), *ptr;
        int    i;
        ptr = calloc(1, 30);
        for (i=0; i<30; i++) printf("%d", *ptr++);
}
```