Eco-C C Compiler

P.O. Box 68602
Indianapolis, IN 46268
(317) 255-6476

Software Problem Report

If you encounter any errors that you believe to be in the Eco-C compiler, please report them as soon as possible.

Eco-C License Number: _____
Please describe your:

Operating System:

Computer (including amount of memory, disk drives, etc.):

Please describe the problem as clearly as possible. If you can supply us with a copy on disk (8" SD if possible) of the program that produced the error, it will help us to correct it. Detailing the sequence of events leading up to the problem may also prove useful. If you are using some method of "getting around the problem", please describe it. We will try to correct any problems as quickly as possible.

Getting Started

Making a Working Copy

The first thing you should do is make a working copy of the Eco-C C Compiler disk. If you use the CP/M PIP utility, the sequence is to place your CP/M disk with PIP.COM on it in drive A and a blank, formatted and SYSGENed disk in B. (We have used <CR> to signify pressing the RETURN key.)

```
A>PIP <CR>
  *                    (now place the Eco-C disk in A)
                       (and enter the following line)
  B:=A:*.*[OV] <CR>
```

Note that the end of the command has a bracket ("[") followed by the letters "O" and "V". This informs PIP that you are copying a binary (non-ASCII) file (the "O" option) and verify ("V") that it is a good copy.

After the copy is completed, place the original disk you received from Ecosoft in a safe place and use the copy from now on. You will need the following files when compiling a program:

CP.COM	Macro Preprocessor
XC.COM	C Compiler
XM.COM	Code Generator
CE.COM	Error Reports
CODE.PA	Data Maps
ERR.PA	" "
STDIO.H	Standard Definitions
*.REL	Library Routines (All of them)
M80.COM	Microsoft's Macro Assembler
L80.COM	" Linker

We have supplied some of the C library functions (discussed later) in source code. These files have a ".C" file extention (e.g., STRLEN.C). These do not have to reside on your working disk and may be omitted to conserve disk space. The files listed above, however, must be on the working disk.

When you use the compiler, each compiled program generates (on its own) TOKEN.CWK and PCODE.CWK as intermediate files during compilation. These files are erased automatically as the compilation progresses. There must be sufficient disk space for them during compilation, however. Their length varies with program

size.

If you have sufficient disk capacity, you probably will want to place your text editor on the disk, too (e.g., the editor on your CP/M disk is named ED.COM). If you use a screen editor like Micropro's Wordstar, be sure you write your programs in the "non-document" mode.

Do not write programs using the file names mentioned above.

If Disk Space is a Problem

Some disk systems, especially the 5 1/4" disks, may not be able to hold all of the required files on one disk. If this is the case, the files may have a maximum split as follows:

DISK1 - CP.COM, XC.COM, XM.COM, CE.COM, CODE.PA, ERR.PA

DISK2 - M80.COM

DISK3 - L80.COM, CFF.REL, CIF.REL, CFC.REL, CIC.REL, ECC.REL, EC2.REL

Using the split above, the compiler itself resides on DISK1, the assembler (M80) on DISK2, and the linker (L80) and the necessary library functions are on DISK3. Obviously, DISK2 has a considerable amount of unused disk space. Therefore, DISK2 could be used for the source file (*.C), compiler output (*.MAC) and assembler output (*.REL). Depending on disk size, groupings of the above disks may be done. For example, on a North Star system DISK2 and DISK3 typically would be combined into one disk (e.g., a DISK2).

System Requirements

The Eco-C C Compiler requires a minimum of 52K of free memory and a minimum of 250K of disk storage. As suggested above, this disk storage may be split over two or more drives.

Having made a working copy of the Eco-C compiler, you are ready to write and compile a C program.

Features of Eco-C

Before you start using the compiler, you should know what is and is not supported in the current release (Rel. 1.41). (If you are just getting started with the C language and need an easy-to-read text on C, we would suggest the C Programming Guide by Dr. Jack Purdum (Que Corp.). If you are an experienced programmer, The C Programming Language by Kernighan and Ritchie (Prentice-Hall) is considered the technical reference manual on C. Both books are available in most B. Dalton book stores. The Purdum text is also available from Ecosoft. References to the Purdum text appear in a number of places in this manual.)

The full C syntax is supported except;

a. bitfields
b. initializers. While globals cannot be initialized at this time, they are cleared to 0 by the compiler. (Purdum, Figure 6.13, p. 135)
c. parametized macros (Purdum, p. 239. We are working on initializers and parametized macros at the present time. If your license agreement is on file at the time they are finished, you will receive the new release automatically. There is no charge.)
d. #line macro preprocessor directive
e. in compound expression following a #if, macro expansion is not done (Purdum, p.238)

Most of the above will be available in subsequent releases of the compiler. No charge to licensed users will be made for updates to the compiler during the first year it is on the market (i.e., through March, 1984). You must have a signed license agreement on file to be eligible for these updates.

Facts, Suggestions and Programming Hints

Given below is a list of specifications and suggestions that may prove useful when using the Eco-C C Compiler. Some items in the list represent syntactic-semantic rules of the C language that are not obeyed by other compilers. Because of our strict adherance to the rules of C, we may generate an error message that goes undetected by other compilers. We think you will find our strict compliance to the rules of C to be an asset.

Other items in the list are suggestions that help generate

more efficient C code. Most of these suggestions simply reflect the internal construction of the compiler and the code it generates.

1. The C keyword "unsigned" should only be used as an "adjective" with the integer (int) data type. Attemps to use unsigned long, unsigned char and unsigned short data types will generate an error message (Purdum, p. 119).

2. Some C functions [e.g., putc()] give the programmer a choice as to where program input and output (i.e., I/O) are to be directed. Normally, input comes from the keyboard (stdin) while program output (stdout) and error messges (stderr) are sent to the CRT, and the printer serves as the standard list device (stdlst). We have defined these "standard I/O devices" for use with the Eco-C compiler in the CP/M environment as follows:

```
  stdin = keyboard    (fd0)
 stdout = CRT         (fd1)
 stderr = CRT         (fd2)
stdlist = printer     (fd3)
```

(The fd's following each device is treated as a pointer variable that may be used for referencing the devices (e.g. putc(c, stdout). Details of how these pointers may be used are found in Chapter 8 and its appendix in the Purdum text, pp. 163-94. The file named INIT.ASM controls the number of files that are available for use in a program [CP/M file control blocks or fcb's], the number of I/O buffers [iob's] available, plus setting the stack.)

3. On function calls:

a. A function parameter of type float is automatically promoted to double and short or char data types become an int (Purdum, pp. 120-123).

b. Given the choice of char or int as a parameter in a function call, use an int. It avoids internal conversions, thus improving speed.

4. To initialize a pointer (e.g., x_ptr) to a function [e.g., funcl()], the syntax must be:

```
x_ptr = &funcl;
```

5. A structure or union name can only have two things done with them: 1) take its address with the address operator in front of it, or 2) have a period and a member name following its name (Purdum, p. 143).

6. Binding of structure members to a structure is absolute. If a pointer is to a structure s1 and now you want to use that pointer with a different structure named s2, you must cast the pointer to s2 or the compiler will generate an error message (Purdum, pp. 148-150).

7. If a function is not recursive (Purdum, p.6), you are usually better off to declare variables as static rather than auto or register (Purdum, pp. 59-63). It generates more efficient code.

8. When using a large "switch" statement, place the most likely case last and the least likely first.

9. Currently, we do support nested comments. The term "nested" means that we allow a comment to appear within a comment.

10. #include's may be nested only two deep. That is, a program can have a #include statement in it that calls in a second file with a #include in it. The file called in by the second file, however, cannot have a #include statement in it (Purdum, p.94).

11. The double data type variables can have up to 17 significant digits. The range for the exponent is E-38 to E+38. Only binary arithmetic is supported.

12. At the present time, floating point and long constants are combined at run time rather than compile time.

13. All program source files (e.g., TEST.C) must terminate with a carriage-return, linefeed pair (CR-LF). Failure to observe this will cause the error handler to produce strange results. In other words, when you get to the "bottom" of a program, press the RETURN key an extra time.

14. Character to integer conversions do not perform sign expansion.

15. If multiple assignments are possible, use them. For example, if i and j are integers that are to be set to 0, it is more efficient to set them with i=j=0; than to set each one separately.

16. You must explicitly call the fclose() function when using disk file I/O before the closing brace in main() is reached. The current release of the compiler does not do this automatically.

The error handler treats all errors as fatal; there is no "cascading" of false error messages. When you do get an error, you will find that it does diagnose the error correctly.

We think you will find the Eco-C compiler more "UNIX-like" than most on the market. Further, it performs its error checking in strict compliance with the syntax presented in K&R and Purdum. Finally, we have adhered to the function definitions presented in K&R and Purdum as much as possible in a CP/M environment. Programs written with the Eco-C compiler can be taken easily to a UNIX environment. Equally important, a program from the UNIX environment can be compiled under Eco-C with few, if any, changes.

Using the Eco-C C Compiler

Writing the Source Program

The first step is to write the C program using a text editor. (CP/M provides an editor stored on disk as ED.COM.) Be sure you do **not** use the "document mode" on some editors (e.g., Wordstar). The document mode can turn the high bit ON and produce mysterious results on occasion.

Note: when writing your source programs, keep in mind that a variable in C may contain as many characters as you wish, but the compiler currently will only recognize the first six as being significant because the assembler (M80) and linker (L80) only recognize six characters.

In producing output to the assembler, the compiler performs the following conversions on variable names:

a. All variable names (or identifiers) of one or two characters have a leading ampersand (@) added to it. This is done to avoid conflict with register names (e.g., A --> @A).

b. Lower case is converted to upper case.

c. The underscore is converted to a question mark.

The normal secondary file name, or extention, given to a CP/M C source program is ".C" (e.g., TEST.C). This is not a

requirement, however. You may use any file extention you wish. In subsequent discussion, we will assume that your program is saved on the disk as a ".C" source file, however.

Compiling the Program

Assuming the compiler and the source program (e.g., TEST.C) are on drive A, the compiler is envoked by:

```
A>CP TEST <CR>          /* Assumes a ".C" extention */
```

The program supplies the file extention of ".C" if one is not supplied when the compiler is envoked. If you supply a file extention (e.g., ".XXX"), it overrides the default (".C").

A different source disk may be specified. For example,

```
A>CP B:TEST <CR>
```

If you have a small 5 1/4" disk system, drive B might contain the disk with the M80 assembler (see the earlier discussion on the maximum split the compiler may have across disks). The output of the compiler will be written to drive B in this case, too.

The compiler automatically loads and runs the next two passes (XC and XM) if no errors are detected. The output of the final pass (i.e., XM) is an assembly language source file (e.g., TEST.MAC). This can be examined and modified if you wish.

Compiler Switches

-i This switch tells the compiler to use the integer version of the printf() function. It avoids pulling in the floating point library whenever printf() is used. If your program does not use floating point numbers, this option will produce smaller code size and may execute faster.

For example, suppose a source program is named TEST.C. To use the compiler with this option, it would be invoked with the following command line arguments:

```
A>CP TEST -I
```

causing the TEST program to use the integer version of printf(). Note that lower case letters may be used if you wish (i.e., you may use: cp test -i).

-c This switch uses the library where getchar() and putchar() do direct BDOS calls to CP/M for input/output (I/O) rather than through getc() and putc(). This prevents console I/O from going through the file handlers thus generating smaller code size. It is envoked with the following command line arguments:

```
A>cp test -c
```

Again, upper or lower case letters may be used.

-o This switch is used to change the name of the output file and the drive on which it is written. For example:

```
A>cp test -o b:test
A>cp test -ob:test
```

Both examples above are allowed. If a file type is added to the file name the default type of '.MAC' will not be used. That is, the example above causes the assembler output of the compiler to be found on drive B as TEST.MAC. If the command line arguments are:

```
A>cp test -o b:test.asm
```

the output file on drive B is named TEST.ASM.

-b This switch is used to turn off most of the messages and statistics the compiler generates during compilation.

-snnn This switch is used to control selection of the system libraries at link time. There are 10 reserved system libraries (0 through 9). Each number following the -s will cause the designated libraries to be searched in order from high to low number. That is, the option -sl35 will search SLIB5, SLIB3 and SLIB1 in that order. These libraries are reserved for Ecosoft and should not be altered by the user. (For example, the Transcendental library is stored on the disk as SLIB0.REL.) For example:

```
A>cp test -s0
```

(Note that this switch avoids having to explicitly link in the SLIB0 file via: A>l80 test,slib0,test/n/e.)

-unnn This switch is used to control selection of the user-defined libraries. These work the same as the system libraries (via the -s option), but are for use by the programmer. For proper use, these libraries must be named ULIB0.REL through ULIB9.REL. User libraries are searched before system libraries.

-nS or -nU These two options are variations on ;the above and cause a series of libraries to be searched. For example, -5U causes ULIB5.REL through ULIB0.REL to be searched, in that order. These options, therefore, search all libraries starting with the digit specified and working towards the 0 library.

NOTE: All or part of the switches may be used when compiling a program. Each switch option in the command line must be separated one from the other by a blank space; their order is not important. Example:

```
A>cp test -i -c
```

or

```
A>cp test -c -i
```

produce identical results.

Assembling the Program

The assembler output file from the compiler (e.g., TEST.MAC) becomes the input file to the assembler. Your package includes Microsoft's Macro 80 assembler. It is envoked with the following command:

```
A>M80 =TEST        /* Note blank space */
```

Since M80 has a default file extention of .MAC, the assembler may be used with or without the .MAC extention. If the input file to the assembler uses something other than .MAC, it must be supplied when M80 is run (e.g., M80 =TEST.ASM).

The output file from the assembler is named TEST.REL. Further details about using M80 are in the Microsoft manual. **NOTE:** there must be a blank space between M80 and the equal sign (=)

when the assembler is envoked.

Linking the Program

The .REL (i.e., RELocatable) file from the assembler is then linked to the standard library routines (e.g., CFF.REL, EC2.REL, etc.) by using the Microsoft linker (L80). A typical link would be:

```
A>L80 TEST,TEST/N/E
```

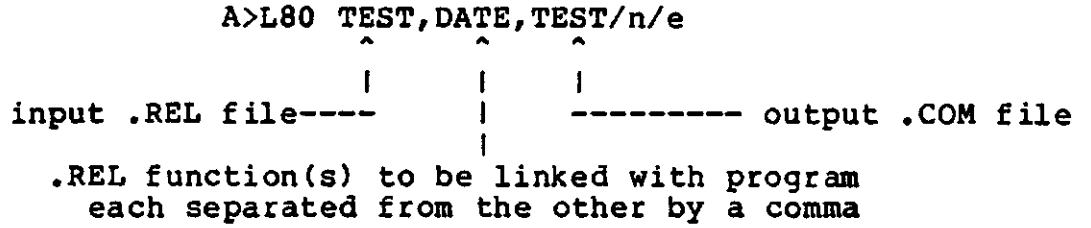
which causes TEST.REL to be the input file and produces an output file named TEST.COM. TEST.COM becomes the executable C program. The standard library routines are automatically searched.

If you wanted the input file TEST.REL to have a command file name of PRICE.COM, the syntax would be:

```
A>L80 TEST,PRICE/N/E
```

Linking in Your Own Functions

Suppose that you have compiled and assembled a function you wrote with the name DATE.C. The assembler would have generated a REL file named DATE.REL. Now you want to link the function into a program named TEST.C. The command arguments would be:



The first file name following L80 on the command line is the input file (.REL). The last file name (before the "/n/e") is the output (.COM) program file name. In between these two file names are any (.REL) function(s) you want to link in with the program. Each must be separated from the other by a comma.

Additional details on L80 and its options are found in the Microsoft manual.

Reading and Writing Data files

We have attempted to make file input and output (i.e., I/O) as consistent with the UNIX operating system as possible given the differences between it and CP/M. Below is an example of writing to and then reading from a file using the Eco-C compiler. Both programs in source are included on your distribution disk.

If all of this seems unfamiliar, consult Purdum, Chapter 8 and its appendix.

```
                /* Writing an ASCII Data File */

#include "stdio.h"      /* Include file overhead info      */
#define CLEARS 12       /* Clear screen for ADDS Viewpoint */
#define MAX 1000        /* Maximum number of characters    */

main()
{
        int i;
        char c;
        FILE *fp;

        putchar(CLEARS);        /* Clear the screen */

        if ((fp = fopen("TEXT.TXT", "w")) == NULL) {
                puts("Can't open TEXT.TXT");
                exit(-1);               /* Signals an Error */
        }

        puts("Enter line of text and press RETURN:\n");
        for (i = 0; (c = getchar()) != '\n'  && i < MAX; ++i)
                putc(c, fp);
        putc(CPMEOF, fp);           /* Must write end-of-file */

        fclose(fp);                 /* Must close */
}
```

The program begins with the #include preprocessor directive to include the file I/O information needed to work with disk files. The #defines are used to define the clear screen code (you may prefer using an octal constant '\014') for an ADDS Viewpoint and set the maximum number of characters that can be entered.

The main() function marks the beginning of the program and several variables are declared. The FILE typedef refers to the structure that is defined in stdio.h and is used to establish pointers to fp and fopen(). Generally, high-level file I/O will require the FILE declarations to be present in every program that works with disk files. The call to putchar(CLEARS) simply clears

the screen in preparation for entering the text.

If your terminal requires two or more characters to clear the screen, change the #define to be a string intead of a constant and change the putchar(CLEARS) statement to a puts(CLEARS). For example, if you are using a SOROC terminal which uses an escape (27 decimal = 033 octal) followed by an asterisk (*), the #define would be: #define CLEARS "\033*". You would then use puts(CLEARS) instead of putchar(CLEARS) since we are now treating CLEARS as a string.

The if statement attempts to open a text file using the name TEXT.TXT in the ASCII "write" mode. If the file cannot be opened (i.e., fp returns a NULL), a message is displayed that the file cannot be opened and the program is aborted by the call to exit(). The -1 argument in the exit() function call is used to signal that some error occurred.

If all went well, the file pointer fp serves as our link with the file that was just opened (e.g., TEXT.TXT). A prompt asks the user to enter a line of text, pressing RETURN when they have finished. Calls to getchar() take the characters from the keyboard and assign them to c. A check is made to see if the character was a newline (i.e., a '\n' which corresponds to pressing RETURN) or if i exceeds the maximum number of characters allowed (MAX).

If the tests are passed, a call to putc() places character c into the buffer associated with fp. The for loop continues until MAX - 1 characters or RETURN is entered. When that happens, a final call is made to putc() using the CP/M end-of-file (0x1A) as the character. This is necessary when using the ASCII mode for disk files.

A call to fclose() writes the buffer to the disk and closes the file and the program ends.

NOTE: you must call fclose() before the closing brace in main() is reached. The current release does not do this automatically.

Reading an ASCII Text File

The program to read the text file just created closely follows the program used to write the file. Notice that main() is called with two arguments: argc and argv. The argc variable is used to count the number of command line arguments (argument counter) used when the program was envoked. The argv[] variable is an array of pointers that points to the command line arguments

were entered. (Purdum, pp. 170-73.)

For example, to read the TEXT.TXT file, this program is envoked with:

A>READFILE TEXT.TXT<CR>

where the <CR> represents pressing RETURN.

```
                /* Reading an ASCII data file */

#include "stdio.h"      /* Pull in the overhead info again */
#define CLEARS 12       /* Clear screen for ADDS viewpoint */

main(argc, argv)
int argc;
char *argv[];
{
        int  c;
        FILE *fp;

        putchar(CLEARS);

        if (argc != 2) {
                printf("I need to know the file name.\n\n");
                printf("Use:\n\nA>READFILE FILENAME.XXX");
                exit(-1);
        }

        if ((fp = fopen(argv[1], "r")) == NULL) {
                printf("Can't open file: %s", argv[1]);
                exit(-1);
        }
        while((c = getc(fp)) != EOF)
                putchar(c);
        fclose(fp);
}
```

There are two arguments (argc = 2) and two pointers in argv[] (argv[0] pointing to READFILE and argv[1] pointing to TEXT.TXT). In the program, the argc is checked to make sure that two arguments were supplied when the program was envoked. If argc is not equal to 2, an error message is given and the program aborts.

If the argument count is correct, we try to open the file in the ASCII "r"ead mode. If the file pointer (fp) returned from the call to fopen() is a NULL, an error message is given and the program aborts.

If a valid file pointer is returned, the **while** loop does

repeated calls to getc() and assigns the character in the buffer to c. The call to putchar() displays the characters on the CRT. This continues until the end-of-file (EOF) is sensed, whereupon the file is closed and the program ends. You will notice that the variable c is defined as an integer. **This is necessary because sign expansion is not performed on char to int conversions. If the variable c were defined as a char, EOF will never be found by the program.**

It may be a worthwhile excercise to list stdio.h to see how the various symbolic constants are defined (e.g., FILE, NULL, EOF, etc.).

What to Do if You Run Out of Memory

Unlike many other C compilers that are available, the Eco-C compiler does not require the entire source code of the program to reside in memory. Even so, it is possible to "run out of memory". Usually, this is caused by overflowing the symbol table space.

Space for the symbol table is allocated dynamically. As auto variables are compiled, they are treated as temporaries and "discarded" after the function has been compiled. Very long programs with a large number of global variables (i.e., extern storage class) or a very large function with many auto-type variables are the most likely to cause the symbol table to overflow.

If this happens, all is not lost. Simply break the source program into two separate source programs and then compile and link each of them together. Keep in mind that you may have to duplicate certain parts of the program (e.g., #include STDIO.H).

For example, suppose your source program TEST.C runs out of memory during a compile. Further assume that you inspect TEST.C and find that the program can be split in half after a function definition named root().

Using your editor, create a new file called TEST1.C; this will be a new file. Now read in TEST.C and delete all of the lines from the beginning of the program to the start of the definition of root(). TEST1.C should now contain the "second half" of TEST.C. Save TEST1.C on disk. Now load TEST.C and delete everything from the definition of the root() function to the end of the program and save it on disk.

Having done the above, TEST.C contains the "first half" of the original program and TEST1.C contains the "second half". Now compile and assemble the two separately and then link them together to form one program. The sequence might look like:

```
A>cp TEST
A>cp TEST1
A>m80 =TEST
A>m80 =TEST1
A>l80 TEST,TEST1,TEST/n/e
```

which creates an executable program named TEST.COM with both "halves" linked together.

Standard Library of C Functions

Listed below are the functions that comprise the Eco-C Standard Function Library. Each function is described by name with any argument list that might be necessary for the function call. If the function returns a data type other than integer (i.e., int), the data type returned preceeds the function name. The functions are arranged in alphabetical order for easier reference.

alloc()

```
char  *alloc(u)                    /* Request for storage */
unsigned  u;
```

Returns a pointer to u bytes of storage, where each byte is able to store one char. If the request for u bytes of storage cannot be satisfied, the pointer returned equals NULL (i.e., zero). Therefore, a non-zero pointer value returned from the call to alloc() means that u bytes of consecutive storage are available.

atoi()

```
int atoi(s)                        /* Return integer of string */
char *s;
```

Returns the integer value of the string pointed to by "*s". This function permits the input string to be in decimal, hex or octal using the standard C syntax for such values (e.g., "0xff").

atol()

```
long atol(s)                       /* Return long of string */
char *s;
```

Functions in the same manner as atoi(), except the returned value is a long rather than an int.

bios()

```
int bios(offset, bc)                    /* BIOS call */
unsigned offset, bc;
```

Variable "offset" is CP/M jump table entry wanted from warm boot entry point. Returns contents of the accumulator. Variable "bc" is what is moved to the BC register pair before the call.

calloc()

```
char  *calloc(count, size)              /* Request storage */
unsigned  count, size;
```

Returns a pointer to sufficient storage for "count" items, each of which requires "size" bytes of storage. If the request is successful, each byte is initialized to NULL. If the request for storage cannot be satisfied, the pointer returned equals NULL (i.e., zero). The pointer returned, therefore, can be tested to see if the request was satisfied.

decimal()

```
long decimal(s)                         /* Return long of decimal */
char *s;
```

Returns the long value of a "decimal" string pointed to by "*s".

dint()

```
double dint(d)                          /* Integral value of double */
double d;
```

Returns the integral value of the double "d". The function "truncates" a double at the decimal point.

drand()

```
double drand()                          /* random number - double */
```

Generates a random number with a value between 0 and 1 and is returned as a double.

drandl()

```
double drandl(d)                        /* random number - log dis. */
double d;
```

Generates a random number with a value between 0 and 1 that is logrithmically distributed. The value returned is a double. Range is between 1 and exp(d).

exit()

```
int exit(i)                             /* terminate a program */
int i;
```

Used to terminate a program. A non-zero value for "i" is normally used to indicate that some error occurred when this function was called.

fclose()

```
int  fclose(fp)                         /* close a file */
FILE  *fp;
```

Function first calls fflush() to flush the contents of the buffer and then closes the file associated with the "fp" file pointer. This frees "fp" for use with another file if desired.

feof()

```
int feof(fp)                            /* Sense EOF */
FILE *fp;
```

Returns a non-zero value when end-of-file has been sensed with the input stream associated with the fp pointer.

ferror()

```
int ferror(fp)                          /* Sense error */
FILE *fp:
```

This function returns a non-zero value when an error occurs while reading or writing to the input stream associated with the pointer fp. The error condition exists until the file has been closed.

fflush()

```
int  fflush(fp)                         /* write buffer to disk */
FILE  *fp;
```

Writes the current contents of the buffer associate with "fp" to the disk (including the EOF indicator).

fgets()

```
char  *fgets(s, i, fp)                  /* get string from file */
char  *s;
int  i;
FILE  *fp;
```

Reads "i" characters from fp and places them into the character array "s". The function terminates upon reading: (1) a null character ('\0'), (2) and end-of-file indicator, or (3) i-1 characters. The character string at "s" is null terminated ('\0') upon return. The function returns a pointer to the string, or zero if end-of-file or an error occurred.

fileno()

```
int fileno(fp)                          /* Get file descriptor */
FILE *fp;
```

Returns the integer number of the "fd" associated with the file pointed to by pointer variable fp.

_fillbuff()

```
int _fillbuff(fp)                    /* read buffer of data */
FILE *fp;
```

Used to replentish the buffer associated with fp and returns the next character or EOF.

_flushbuf()

```
int _flushbuf(c, fp)                 /* flush buffer */
char c;
FILE *fp;
```

Flushes the buffer associated with fp and then writes the character "c" into the buffer.

fopen()

```
FILE  *fopen(name, mode)             /* open file */
char  *name, *mode;
```

Open the file "name" for use in the "mode" file operation.There are three fundamental modes of operation:

"a" Open for appending. The file is opened for writing to an ASCII file. It starts appending at CPMEOF.

"ab" Open for appending. The file is opened for writing to a binary file. It starts appending at the next sector boundary.

"r" Open for reading. The file must already exist to use this mode. ASCII text files have carriage-return, line-feed (i.e., <CR><LF>) adjustment.

"rb" Open for binary reading. No <CR><LF> adjustments.

"w" Open for writing. Any existing file with the same "name" is destroyed and a new file is created. The contents of the old file are lost. Does <CR><LF> adjustment.

"wb" Open for binary writing. No <CR><LF> adjustments.

Upon a successful open, the function returns a pointer (e.g., fp) to the opened file. If an error occurred, NULL is returned.

fprintf()

```
int fprint(fp, control, arg)            /* formatted - files */
FILE *fp;
char *control, arg;
```

Formatted printing from a file, where "fp" is the pointer for the associated file to be used. See printf() for options.

fputs()

```
int  fputs(s, fp)                  /* Put a character out */
char *s;
FILE  *fp;
```

Takes the character array pointed to by "s" and puts it to the output designated by "fp".

free()

```
int free(c)                         /* Release storage area */
char *c;
```

The function call frees (i.e., de-allocates) the region of storage pointed to by "c", thus making that area of storage available for re-use. The function assumes that the pointer "c" was first obtained by a call to alloc().

fscanf()

```
int  fscanf(fp, control, arg)       /* Input from a file */
FILE *fp;
char *control, *arg;
```

Reads by calls to getc(fp) and attempts to fill the specified arguments as specified by the control string. Arguments must be pointers. (For information about the nature of the arguments and options for the control spring, see the scanf() function discussed below.) Note that fp must be a valid file pointer to a file that has been opened for reading.

The function returns: -1 if end of file is detected by getc(), a 0 if no arguments match or none were supplied, or an integer number equal to the number of valid arguments matched.

ftoa()

```
int  ftoa(s, d, prec, type)      /* Float to ASCII (or DTOA) */
char  *s;
double  d;
int  prec, type;
```

Converts a floating point number "d" into an ASCII string and stores the result in "s". Up to 17 significant digits of precision may be requested (i.e., prec = 17). Note that precision refers to the number of places after the decimal point. The "type" variable may be 'g', 'e' or 'f' [see discussion of the printf() function]; otherwise the floating point number is free-form. The string ("s") must be large enough to hold the character representation of the resulting number plus one more byte for the NULL. Since only a double may be passed to this function, it is also dtoa().

getc()

```
int  getc(fp)                    /* Read character from file */
FILE *fp;
```

Reads a character from the input stream associated with "fp". The character is normally returned as a positive integer although it may return EOF upon reading end-of-file or ERR (-1) if an error occurred during the read. If the file is opened in the ASCII mode, CPMEOF is converted to EOF by getc().

getchar()

```
int getchar()                    /* Get character from stdin */
```

Reads a single character from stdin. stdin defaults to the terminal.

getd()

```
double getd(fp)                         /* Get double */
FILE *fp;
```

Returns a double from the file associated with pointer fp.

getl()

```
long getl(fp)                           /* Get long */
FILE *fp;
```

Returns a long from the file associated with pointer fp.

gets()

```
char *gets(buff)                        /* Get a string from sdtin */
char *buff;
```

Get a string from stdin and place it in the buffer pointed to by "*buff". If input is from the console, '\r' is used to sense the end of the input string; otherwise a newline '\n' is used.

getw()

```
int  getw(fp)                           /* Get next word from file */
FILE  *fp;
```

Reads the next word, or integer, from the file associated with fp. The word is returned if a successful read is done, but may also return EOF upon reading end-of-file or ERR if an error occurred during the read.

hex()

```
long hex(s)                             /* Return long from hexs */
char *s;
```

Convert the hex string pointed to by "*s" into a long data type.

irand()

```
int irand()                          /* random number - int */
```

Generates a positive random number as an integer.

isalnum()

```
int isalnum(c)                       /* Is letter or digit */
char c;
```

Returns non-zero if variable c is a letter or a digit.

isalpha()

```
int isalpha(c)                       /* Is c alphabetic character */
char c;
```

If the character "c" is an alphabetic character, TRUE (i.e., 1) is returned; otherwise FALSE (i.e., 0) is returned.

isascii()

```
int isascii(c)                       /* Is it ASCII */
char c;
```

Returns a non-zero value if variable c is in the ASCII character set. Since ASCII does not use the high bit, c must have a value of decimal 127 or less to return a non-zero value.

iscntrl()

```
int iscntrl(c)                       /* Is control character */
char c;
```

Returns a non-zero value if the variable c is a control character.

isdigit()

```
int isdigit(i)                          /* Is i a digit */
int  i;
```

If the character "i" is a digit, TRUE (1) is returned; otherwise FALSE (0) is returned.

islower()

```
int islower(c)                          /* Is c lower case letter */
char c;
```

If the character "c" is a lower-case alphabetic character, TRUE (1) is returned; otherwise FALSE (0) is returned.

isprint()

```
int isprint(c)                          /* Is it printable char */
char c;
```

Returns a non-zero value if variable c is a printable character.

ispunct()

```
int ispunct(c)                          /* Is it punctuation */
char c;
```

Returns a non-zero value if the variable c is a punctuation character; not a space, letter, digit or control character.

isspace()

```
int isspace(c)                          /* Is c white space */
char c;
```

If the character "c" is a tab ('\t'), a space (' '), a newline ('\n') or a carriage return ('\r'), TRUE (1) is returned; otherwise FALSE (0) is returned.

isupper()

```
int isupper(c)                    /* Is c in upper case */
char c;
```

If the character "c" is an upper-case letter, TRUE (1) is returned: otherwise FALSE (0) is returned.

lrand()

```
long lrand()                      /* random number - long */
```

Generates a positive random number as a long.

ltoa()

```
int ltoa(str, lg)                 /* long to ASCII */
char *str;
long lg;
```

Converts the variable "lg" to an ASCII string, which is NULL terminated.

octal()

```
long octal(s)                     /* Return long of octal str */
char *s;
```

Returns long of the octal string pointed to by "*s".

printf()

```
int printf(control, arg)          /* Formatted printing */
char *control, arg;
```

Used for formatted printing on selected output device (e.g., stdout). The conversion character is the percent sign (%).The options available are:

- A minus sign before the conversion control character indicates that the output is to be left-justified.

```
printf("%-d", x);
```

will left-justify the contents of variable "x".

nn A digit string consisting of "nn" digits following the conversion character and preceeding the control character specifies the minimum field width to be used for printing. If padding is required (i.e., number is smaller than the width) blanks are used unless the first digit is a zero which causes padding with zeros.

```
printf("%12d", x);
```

prints the variable "x" in a field of 12 positions. It may also be used with a decimal point:

```
printf("%6.2f", x);
```

which prints in a field width of 6 places and reserves two places after the decimal point.

The field width specifier may also be used with string data.

```
printf("%25s", str);
```

which prints the first 25 characters of the string variable "str" (right-justified).

l States that the data item is a long rather than an int. (This is the letter "l", not a one).

```
printf("%ld", x);
```

The conversion characters available are:

d The data item is printed in decimal notation.

o The data item is printed in octal notation (unsigned).

x The data item is printed in hexadecimal notation (unsigned).

u The data item is printed in unsigned decimal notation.

c The data item is printed as a single character.

s The data item is a string. The item must be null terminated or have a width specification equal to or less than the length of the string.

e The data item is a float or double and is printed in scientific notation. Default precision is 6 digits, but may be modified with an "nn" specifier.

f The data item is a float or double and is printed in decimal notation with a default precision of 6 digits. The precision may be changed with an "nn" specifier.

g Select the shorter of options e or f (i.e., use the one with the shortest width).

The conversion characters may be in upper or lower case letters (they are converted to lower case during the parse of the control string). For additional details, see Purdum, pp.103-05.

putc()

```
int putc(c, fp)                          /* Output a character */
char  c;
FILE *fp;
```

Outputs the character "c" to the stream pointed to by "fp" and returns the character "c".

putchar()

```
int putchar(c)                          /* Send character to stdout */
char c;
```

Outputs the character "c" to stdout which is normally the console. (This function calls putc() with "c" and stdout as its arguments.)

putd()

```
int putd(d, fp)                          /* Write a double */
double d;
FILE *fp;
```

Writes the double variable d to the file associated with fp.

putl()

```
int putl(l, fp)                          /* Write a long */
double l;
FILE *fp;
```

Writes the long variable l to the file associated with fp.

puts()

```
int puts(s)                              /* Prints string */
char *s;
```

Displays the string pointed to by "s" on the console. It assumes the string is NULL terminated.

putw()

```
int putw(u, fp)                          /* Output a word */
unsigned  u;
FILE  *fp;
```

Outputs the unsigned integer word "u" to the stream pointed to by "fp". This function is accomplished by calls to putc() with the lower byte sent first followed by the high byte. "Word" is taken to be two bytes in length.

scanf()

```
scanf(control, arg)                      /* Formatted input */
char *control, *arg;
```

Reads the input from calls to getchar() and places the input into the arguments (arg) supplied with the results determined by the control string. It serves a purpose similar to printf(), but uses input instead of output.

The control string uses the percent sign (%) for conversion purposes, just as printf() does. A second special conversion character is the asterisk (*) which indicates that an input is to be ignored, or "skipped over", with no assignment made to the associated variable. The control string may also contain digits to indicate the maximum field width for a given input.

In addition, the following permissible conversion characters are permitted in the control string.

d = a decimal integer

o = an octal integer

x = a hexadecimal integer

h = a "short" integer (treated as an int)

c = a single character

s = a character string

f = a floating point number

If the first three conversion characters in the list (d, o, x) are preceeded by an ell (l), the variable is taken to be a long data type rather than an int.

Note that arguments must be pointers to their respective variables and that their data type must match that specified in the control string. That is:

```
scanf("%5d", &num);
```

places a maximum of five digits entered by the user into the integer variable named num. If num is a float, all kinds of problems can arise. Arrays do not need the "address of" operator (&) in front of their name, since arrays are not copied in function calls (i.e., functions receive the address of the array variable, not a copy). For additional details on scanf(), see Purdum, pp.105-07.

The function returns: -1 if end of file is returned by getchar(), a 0 if the first argument is invalid or no arguments are supplied, or the number of valid arguements that were matched.

Keep in mind that scanf() is a complex function and, hence, generates quite a bit of code. If a simpler input function will do, using it will probably save code space.

sprintf()

```
int sprintf(s, control, arg)        /* format to string */
char *s, *control, arg;
```

Function is similar to printf(), but the output is placed in the string pointed to by "s" instead of the console. See printf() for options available.

srand()

```
int srand(lg)                       /*Seedrandomnumber generator*/
long lg;
```

Seeds the random number generator. If the argument "lg" equals 0, a prompt is issued that tells the user to press any key. When the key is pressed, the value held in the generator at that point becomes the seed for the random number generator. (Numbers are being generated while the user thinks about pressing a key.) If the argument "lg" is a value other than 0, that number becomes the seed for the random number generator. This allows pseudo (repeatable) random numbers to be generated. If "lg" is 1, random seeds are generated until the key is pressed, at which time the seed is set and returned. (This option lets a program seed be generated without a prompt.)

sscanf()

```
int sscanf(s, control, arg)         /* Input from string */
char *s, *control, *arg;
```

Gets the characters from the string pointed to by s and attempts to place the input into the arguments pointed to by arg according to the conversions specified in the control string (control). As will all members of the "scanf()" family, the type of pointer must match the associated data type specified in the control string. The options available are detailed in the discussion of the scanf() function above.

The function returns an integer equal to: -1 if NULL (machine 0) is reached in the input string (s), 0 if none of the arguments are matched or none were supplied, otherwise the number of valid arguments that were matched.

strcat()

```
int strcat(s, t)                    /* Concatenate strings */
char  *s, *t;
```

The string pointed to by "t" is concatenated (i.e., added) onto the end of the string pointed to by "s". It is the programmer's responsibility to ensure that the character array pointed to by "s" is large enough to hold both "s" and the appended string at "t" (including the NULL terminator '\0').

strcmp()

```
int strcmp(s, t)                    /* Compare strings */
char  *s, *t;
```

Compares the two strings "s" and "t" character-by-character. If the two strings match, a value of zero is returned. Otherwise, the function returns the result of the subtraction of the character in "t" from the character in "s". For example, if the match fails on the fifth character and s[4] = 'A' and t[4] = 'B', -1 is returned (i.e., 'A' in ASCII = 65, 'B' in ASCII = 66; therefore, -1 = 65 - 66). It follows that a negative value is returned if the ASCII value in "t" is greater than the ASCII value in "s". Positive values are returned when the ASCII value in "t" is less than the ASCII value in "s".

strcpy()

```
int strcpy(dest, src)               /* Copy a string */
char *dest, *src;
```

Copies the string pointed to by "src" (i.e., the source string) into the location pointed to by "dest" (i.e., the destination string). It is the programmer's responsibility to ensure that the destination is sufficiently large to hold a copy of the source string (including the null terminator '\0').

strlen()

```
int strlen(p)                          /* Find string length */
char  *p;
```

Determines the length of the string pointed to by "p". It returns an integer number equal to the number of bytes read before reading the null terminator ('\0'). The null terminator is not included in determining the length of the string.

tolower()

```
int tolower(c)                         /* Convert to lower case */
char c;
```

Returns the lower-case equivalent of "c" if "c" was in upper case. Otherwise it returns "c" unchanged.

toupper()

```
int toupper(c)                         /* Convert to upper case */
char c;
```

Returns the upper-case equivalent of "c" if "c" was in lower case. Otherwise it returns "c" unchanged.

ungetc()

```
int ungetc(c, fp)                      /* Character pushback */
char c;
FILE *fp;
```

Pushes back the character "c" to the file associated with pointer "fp". Only one is allowed by access.

Assembly Language Functions

The following is a list of functions written in assembly language for maximum speed and are available to the user. These are part of the EC2.REL disk file.

atof()

```
double atof(s)                  /* Convert ASCII to double */
char *s;
```

Converts the ASCII string referenced by s, to a double which is returned.

_bdos()

```
int _bdos(call,val)             /* CPM BDOS call */
int val,call;
```

Does a CP/M bdos call and returns the answer.

chain()

```
chain(s)        /* Load and run a program   chain("filename") */
char *s;
```

Loads and executes the program stored on disk pointed to by s. You may, of course, use the filename in quotes.

close()

```
int close(fd)                   /* Close file associated with fd */
int fd;
```

Close the file associated with the fd file descriptor.

creat()

```
int creat(s,mode)              /* Create a file named str */
char *s;
int mode;
```

Creates a file with the name pointed to by s. It can only be created for the write mode (mode =1).

_exit()

```
_exit()                        /* Warm Boot */
```

Causes the program to execute a jump to memory location 0 (JP 0).

_ftoa()

```
int _ftoa(s,db)                /* Convert double to ASCII */
char *s;
double db;
```

Used by atof() to do conversions.

inp()

```
int inp(port)                  /* Read port */
int port;
```

Returns the value read from the port specified.

lseek()

```
int lseek(fd, offset, origin)          /* Random access */
int fd, origin;
long offset;
```

Move to position "offset" within the file associated with "fd" relative to the location given by "origin". The variable "origin" may assume three values: 0 specifies that "offset" is measured from the beginning of the file, 1 measures from the current position in the file, and 2 measures from the end of file. No error checking is done.

open()

```
int open(s,mode)                /* Open file named str */
char *s;
int mode;
```

Opens the file named is the character array pointed to by s for one of the following modes: 0 = reading, 1 = writing, 2 = reading or writing.

outp()

```
int outp(port,val)              /* Send to port */
int port, val;
```

Outputs the dat in "val" to the specified "port".

read()

```
int read(fd, buf, n)              /* Read device */
char *buf;
int fd, n;
```

Attempts to read "n" bytes of data from the file associated with "fd" into the buffer ("buff"). It will return -1 if an error occured or EOF if end-of-file is read. If fd equals 0, the console is read to string length or a CR is sensed and returns the number of characters found. It does not do NULL termination and CR is not included in the count. fd's 1, 2, and 3 are illegal.

rsvstk()

```
int rsvstk(size)                /* space betwen stack and sbrk */
unsigned size;
```

Returns the number of bytes between last used segment of memory and the stack.

sbrk()

```
char *sbrk(u)                    /* Return u bytes of storage */
unsigned u;
```

Returns a pointer to u bytes of storage if available, zero otherwise.

setmem()

```
int setmem(ptr,len,val)          /* Fill memory with value */
char *ptr;
int val;
unsigned len;
```

Fills memory from "ptr" for length "len" with the value of "val".

write()

```
int write(fd,buf,n)              /* Write to device */
char *buf;
int fp, n;
```

Write n bytes of data from buffer buf to the file associated with fd. (Purdum, p. 181). If file i/o then n must be multiple of 128.

Machine Constants

What follows is a list of constants that have been defined in the compiler and are available to the user. All have been set to machine precision and, hence, are doubles. To use them in a program, they should be declared as:

```
extern double CONSTANT_NAME;
```

before used. Obviously, no other variables should use these names.

Constant Name	Meaning
_FPMAX	Maximum floating point number (machine infinity).
_FPONE	Floating point one (1.0).
_10E1	10 (Ten to the first power).
_10E2	100
_10E4	10,000
_10E8	100,000,000
_10E16	10,000,000,000,000,000
_10E32	100,000,000,000,000,000,000,000,000,000,000
*** The following are in SLIB0.REL; use -s0 switch option ***	
_pi	pi to machine infinity
_pi_2	pi divided by 2
_pi_4	pi divided by 4
_16	16
_inv2	1 divided by 2
_inv4	1 divided by 4
_inv16	1 divided by 16

_loge	log(e)
_tsqr3	2 minus the square root of 3
_sqrt3	square root of 3
_sqrt5	square root of .5

Assembler Language Function Interface

Assembly language routines may be written and called by the "C" program or other assembler language functions. There exist several support routines that may be used to simplify assembler language interface.

$RTN	- will return the value pointed to by the HL register pair to the calling function.
$RETVAL	- will return the value contained in the DE register pair to the calling function. If a long was requested then the value will be padded.
$RETM1	- will return a -1 to the calling program.
$FE	- calls the function pointed to by the HL register pair using parameters following the call and information contained in the called function.
$PPARM	- pushes the parameter addressed in HL for the length in BC.

All variables of length 2 are processed by value while all others are processed by address. If a 2 byte integer value is to be passed as a parameter, its value is pushed on the stack. If the length is greater than 2 bytes, the address of the variable is loaded into the HL register pair and $PPARM is used. On return from a function, if the return value was length 2, its value is in the HL register pair. If it is longer than 2 bytes, it has been moved to the variable specified at call time and the address of that variable is in the HL register pair.

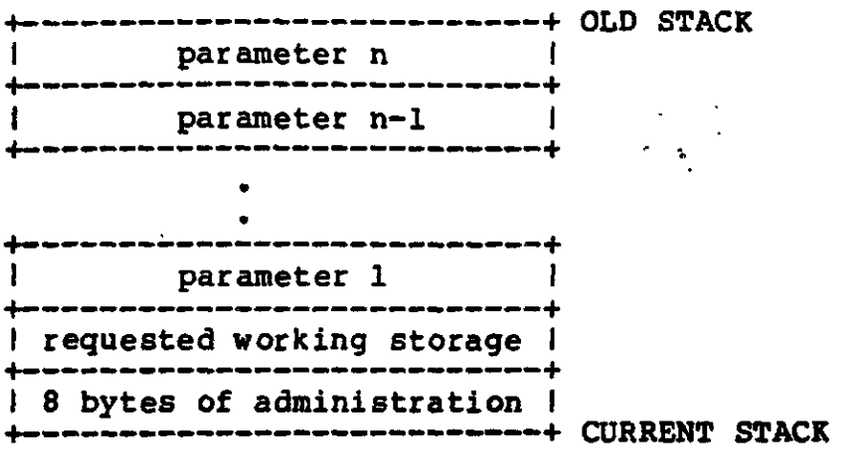
When a function is defined, the first word in the function following the function name is the amount of stack space required for working storage by the function. This is expressed as a negative value. This stack space is the equivalent to auto variables. The code for the function immediately follows this word. For example:

```
FAKE::  DEFW    -8
        LD      IX,0
        etc.
        LD      HL,ANSWER
        JP      $RTN##
```

states that 8 bytes of working stack storage are required by the function FAKE. The stack will always contain 8 bytes of

administrative data prior to the requested storage.

The stack looks like this:



The adminstrative data on the stack consists of the following:

```
STACK     - Address of where to return value.
STACK+2   - Length to return.
STACK+4   - Address of calling program.
STACK+6   - Address of old stack.
```

Invoking a function in Assembler Language

To invoke a function in assembler language the following convention is used.

```
LD        HL,FUNCNAME
CALL      $FE##
DEFW      old stack offset
DEFW      return value address offset
DEFB      return value length
```

All offsets are positive. All return values must first be stored on the stack. For example, to do an assembler language call equivalent to the following "C" call

```
double    atof();
     /*  code of some kind... */
a=atof(b);
```

the equivalent assembler code is:

```
LD    HL,@B          ;address of B
LD    BC,8           ;length of B
CALL  $PPARM##       ;push it
LD    HL,ATOF##      ;address of ATOF
CALL  $FE##          ;call it
DEFW  8              ;offset for return value
DEFW  8              ;old stack offset
DEFB  8              ;length to return
```

Using INIT.ASM

The file INIT.ASM is used to change the number of iob's and fcb's and to set the stack pointer. To change the number of iob's and fcb's, change the number in the EQU statement to the desired number. To change the stack pointer, change the code that initializes the stack. After INIT.ASM has been assembled, it must be linked in explicitly. For example:

```
A>l80 test,init,test/n/e<CR>
```

Below is a copy of INIT.ASM modified for 10 files in addition to stdin, stdout, stderr, and stdlst. The stack has also been altered to reside at high memory rather than bottom of bdos.

```
            .Z80
            INCLUDE FCB.MAP
NFCB        EQU     10              ;SET FILES TO 10
            INCLUDE FCB.ASM
            INCLUDE IOB.ASM
            CSEG
$INIT::     POP     BC
            LD      HL,0            ;SET STACK TO TOP OF MEMORY
            LD      SP,HL
            PUSH    BC
            RET
            END
```

Remember that, when INIT.ASM is assembled, the following files must be available on the disk:

```
FCB.MAP
FCB.ASM
IOB.ASM
```

Naming Conventions

In order to prevent conflicts with the assembler registers all variables of two characters or less have a prefix character of @. For example:

```
A    ->   @A
BC   ->   @BC
A1   ->   @A1
```

In addition to the above convention, all underscores are translated to question marks. For example:

```
_AB  ->   ?AB
A_B  ->   A?B
```

Creating Libraries

Custom libraries may be generated for use with the Eco-C Compiler. What is presented here is an outline of the procedure necessary to establish your own library of C functions. Complete information is contained in your MACRO 80 documentation.

First, each function to be included in the library should be compiled and assembled individually just as you would with any C program. The output from the assembler (M80) will be a series of REL files; one for each function.

At this point LIB80 is used to consolidate these files into a library. The following sequence of commands illustrates the creation of a library called OWN, containing two modules (.REL files) called ONE.REL and TWO.REL respectively.

```
A>LIB80 <CR>

*OWN=ONE,TWO <CR>

*\E <CR>

A>                    /* <--- Control returned to CP/M */
```

Note: in creating a library, the order of the modules is important due to a linker restriction. Any module which references an external label in another module must be included **before** the module containing the reference. For example, if file ONE.REL references a label in file TWO.REL, the above library generation is correct.

On the other hand, if TWO.REL contained a reference to a label contained in file ONE.REL, the above library construction would cause an error message to be generated by the linker (L80). The linker makes only one pass through the libraries and therefore cannot find any external label referenced in a file "in front of it" in the link sequence.

When creating a library, you can request LIB80 to inform you of any unresolved errors the linker might encounter in searching the library. If you wish to check OWN for possible unresolved references, the sequence is:

```
A>LIB80 <CR>

*OWN/U <CR>
```

LIB80 will then list all unresolved references (i.e., globals).

If you want a listing of the modules in a library with information concerning entry points and external references, the following command will cause that listing to be generated on the screen.

```
A>LIB80 <CR>

*OWN/L <CR>
```

The LIB80 manual contains further information on how to use other commands to alter or create libraries.

CP/M I/O Interface

Since the standard C library was written to exist in the UNIX environment, implementing the same library in a CP/M environment presents several problems. In implementing the library for the Eco-C Compiler the following approach was used.

The system routines, such as read(), write(), open(), and create() are contained in the user program but should be viewed as part of the operating system. In the UNIX environment, read() and write() may have a count specification of any size when dealing with files. The fcb's are maintained by these "system" routines and contain information in addition to the fcb proper. It is intended that the fcb's be transparent to the user and therefore cannot be referenced by the user.

The _iob's maintain a UNIX format and are defined in "stdio.h". An additional flag (_BFLAG) is used to determine if I/O is to do conversions on carriage-return, line-feed (CR LF). If the _BFLAG is set, no translation takes place. If the "_BFLAG" is cleared, the following translations take place.

1. All input with the exception of _fd==0 will strip all <CR>'s from the input stream; it does not matter if a <LF> follows. The other choice was to look for a <LF> and then do an ungetc() if it is not an <LF>. This would make ungetc() unreliable for use by the program. The ungetc() function is only insured to work once, and this once was used by getc(), not the user.

It was decided to make ungetc() available for the user program at the expense of an extra <CR> being stripped. In a file where this is critical, the program may be opened in binary mode (e.g., "rb" and "wb") which causes all <CR>'s processed by the user's program.

2. When input is from _fd==0, a <CR> is taken as the end of input. At this point a <LF> is echoed to the screen and '\n' returned to the user program.

3. On all output, '\n' translates to <CR> <LF> and '\r' remains as <CR>.

Note: In the non-binary mode, the CP/M EOF indicator (0x1a) must be checked for explicitly on getc() and written explicitly before closing the file. EOF is only returned at **physical** end of file.

Floating Point Notes

In the Eco-C compiler, a floating point number consists of two parts: 1) the mantissa and 2) the exponent. The mantissa uses 56 bits. All floating point numbers are normalized. (Therefore, since the high bit is always 1, if the Most Significant Bit [MSB] of the mantissa is 0, it is a negative number. If it is a negative number [i.e., MSB=0], it is set to 1 before exponent adjustment is done.) The mantissa is expressed as a fraction, which means the most significant bit is interpreted as 1/2, the next bit is 1/4, then 1/8 and so on. The series formed by the mantissa bits may be viewed as (from most to least significant):

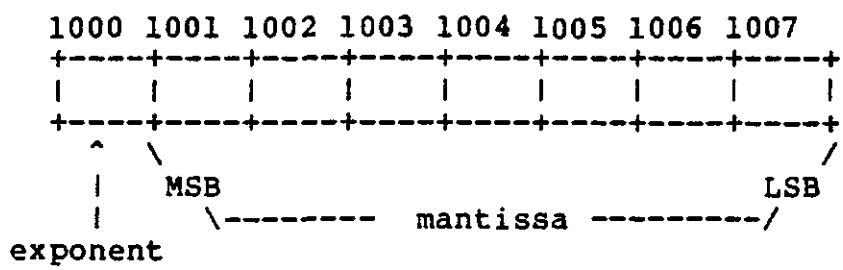
1/2^1, 1/2^2, 1/2^3, . . ., 1/2^55, 1/2^56

The exponent is 8 bits and is an integer with a 128 offset and represents powers of 2 (i.e., binary). A zero exponent is used for the number 0.0. To illustrate, 2 raised to the 0 power is 128, 2 to the 1st becomes 129, and 2 to the -1 is 127. The range for the exponent is 2 to the -127 through 2 to the +127.

Taken collectively, a floating point number is represented as:

.mantissa * ((2 raised to the power of exponent) - 128)

If we assume that a floating point number is stored in memory starting at memory address 1000h, its representation might be depicted as:



Using the scheme immediately above, the following represent

floating point numbers (in hexadecimal):

Value	Hex
1.0	81 80 00 00 00 00 00 00
-1.0	81 00 00 00 00 00 00 00
.5	80 80 00 00 00 00 00 00
85.0	87 aa 00 00 00 00 00 00
-85.0	87 2a 00 00 00 00 00 00

The range of the floating point representation is:

2.9387358770557187 E-39

to

1.7014118346046922 E+38

For those that want to interface with the floating point package, the following assembly language functions should prove useful. In each case, the function is called with the address of the first operand (op1) in the HL register pair and the address of the second operand (op2) in the DE register pair. The function returns with the answer in the floating point accumulator ($FPACC) which is pointed to by the HL register pair.

$ADDD	Add ($FPACC = op1 + op2)
$SUBD	Subtract ($FPACC = op1 - op2)
$MUD	Multiply ($FPACC = op1 - op2)
$DVD	Divide ($FPACC = op1 / op2)

(Note that $FPACC is the address of the 8-byte floating point accumulator. $FPMAX is the address of the constant that is equal to the maximum floating point number and $fpone is the address of a constant equal to a floating point 1 and may be used to increment a floating point number.)

Error Messages

Any error in the source program is detected by the Syntactic Parser pass. To illustrate how this works, suppose you tried to compile the following "do-nothing" program:

```
main()
{
     char wrong          /* need semicolon at end of line */
}
```

Assume further that we have named this program ERROR.C. The error message generated will look like:

```
Error in File: ERROR.C  Line: 4  Char: 1  Error: 1002  Token: }
   Expected type specifier, typedef name, [ ( ; ,
   ) = or : instead of }.
```

If you look at the program, we forgot to add the semicolon after the character variable named "wrong". The compiler found the character "}" when it expected to find something else. (A list of possibilities is given as part of the error message.)

Since predictive parsing is used, we must look "backwards" from the point where the error was detected to find the error. Since the error was the **first** character in line 4, the error must have been caused by something near the end of line 3. Inspection of line 3 shows that we forgot the semicolon in the declaration of "wrong".

Many of the messages will tell you what should have been found in the program where the error occurred. You will also note that some of the error messages have more than one number associated with them. This is so we can tell exactly where the messages is generated within the error handler.

You might want to write a few programs with known errors in them to "get a feel" as to how they are handled by the error

handler. All errors are treated as fatal so there will be no cascading of false error messages. We think you will find that the error handler pinpoints the source of the error. We have listed pages references from the **C Programming Guide** (Purdum) for further information about the nature of the error.

Error code Number(s)	Meaning
1	Internal Error - an attempt was made to create a temporary variable of an illegal type. Check source code for legality of statement.
2	A type specifier, storage class specifier or function declaration was expected instead of ___. (p.46, p.53, 128)
3	A comma or semicolon was expected instead of ___. (p.34, p.242)
4 1009 1015	An identifier, (or * was expected instead of ___. (p.132, p.242, p.241)
5	A semicolon was expected instead of ___. (p.8, p.242)
6	A closing parenthesis was expected instead of ___. (p.9, p.241)
7	An opening parenthesis was expected instead of ___. (p.10, p.241)
8	A closing brace was expected instead of ___. (p.10, p.242)
9	A closing bracket was expected instead of ___. (p.65, p.242)
10	A parameter (argument) list was expected instead of ___. (p.9, p.46)

11 1011 1012

An opening brace was expected instead of ___.
(p.10)

12

A parameter declaration list or opening brace was expected instead of ___.
(pp.9-10, p.242)

13

The parameter (argument) declaration list is in error.
(p.46)

14

An array size was expected but found ___.
(p.65)

15

A structure or union tag or opening brace was expected but found ___.
(p.149, p.158, p.10)

16

A type specifier was expected but found ___.
(p.46)

17

A statement or declaration was expected but found ___.
(p.249, p.244)

18

A type specifier or storage class specifier was expected but found ___.
(p.46, p.53)

19

A storage class specifier was expected but found ___.
(p.53)

20

An attempt was made to "goto" an illegal label name.
(p.112)

21

An attempt was made to perform an illegal "break" or "continue"
(p.36, p.37)

22

The "while" is missing from a "do{ }while" statement.
(p.29)

23

A multiply defined "default" was found in the "switch".
(p.116)

24

Multiply defined label.
(p.113)

25

An identifier, opening parenthesis or constant was expected in the expression instead of ___.
(p.132, p.10, p.29)

26

Variable is undefined.
(p.13)

27

Illegal indirection or array reference.
(p.76, p.87)

28

Expected a comma or closing parenthesis instead of ___.
(p.34, p.9)

29

Internal Error - illegal multiplier.

30

Illegal bitwise operand.
(p.25)

31

Illegal pointer arithmetic.
(p.71)

32

Illegal floating point operation.
(p.117)

33

Illegal negation.
(p.25)

34

Illegal logical not operation.
(p.25)

35

Internal Error - illegal constant.

36

Symbol multiply defined.

37

Pointer is not to a structure or union in p->x or type initial field is not a structure or union in i.x.
(p.150, p.143)

38

A subfield of the name referenced does not exist in the structure or union in i.s or p->s.
(p.140)

39

Input file not found.

40

A colon was expected in ternary "?_:" but was not found.
(p.125)

41

The results of the two expressions in ternary "?_:" were not of a legal combination.
(p.125)

42

A non-zero integer constant was mixed with a pointer in the ternary "?_:" expression.
(p.125)

43

Out of memory.

44

Illegal address of (&id).
(p.74)

46

Illegal use of struct or union as source operand.
(p.137, p.158)

47

Attempted assignment to a constant.
(p.86)

48

Illegal assignment to structure, union, function name or array name.
(p.137, p.158, p.86)

49

Attempt to "type" a parameter which is undefined in the function declaration.
(p.46)

51

A structure, union or function data type was specified illegally.

(p.137, p.158)

52

A referenced structure or union tag has not been defined.
(p.149, p.158)

53

An external data definition and the current data definition do not match.
(p.58)

60

Initializers not currently supported.
(p.140)

1000

Expected type specifier, [{ ;) = or : instead of ___.
(p.46, pp.241-42)

1001

A declarator delimiter (e.g, = , or ;), parameter declaration for a function or { was expected instead of ___.
(p.242)

1002

Expected type specifier, typedef name, [(; ,) = or : instead of ___.
(p.46, p.241-242)

1003

Expected type specifier, typedef name, [{ ; ,) = or : instead of ___.
(p.46, p.241-242)

1006

Expected identifier, * (or : in structure or union declaration instead of ___.
(p.150, p.158)

1007

Expected : ; or , instead of ___.
(p.46, p.241-242)

1010

Expected = ; or , instead of ___ in declaration list.
(p.46, p.241-242)

1016

Compiler doesn't handle function name as pointer yet. For now:

Define variable as pointer to function
then equate it to address of function name.
i.e. a=&getc;
(p.71)

Operator Precedence
(Highest to lowest)

```
1     ->    .     ()    []
2     ++    --    -     !     ~     (cast)     *     &     sizeof
3     *     /     %
4     +     -
5     <<    >>
6     <     <=    >     >=
7     ==    !=
8     &                 /* bitwise AND */
9     ^
10    |
11    &&
12    ||
13    ?:
14    =     +=    -=    *=    /=    %=    <<=   >>=   &=    ^=    !=
15    ,
```

Appendix A

Trancendental Library

This appendix describes the trancendental functions that are included in the Eco-C compiler. These functions are found in Standard Library Zero (SLIB0.REL) and are envoked at compile time with the -S0 option as described earlier. Note that all values returned are doubles. The calling function, of course, must be aware of this fact (Purdum, pp.205-06).

sqrt(x)

```
double  sqrt(x)                         /* Square root of x */
double  x;
```

Returns the square root of the (positive) value of x. If x is not a valid argument (e.g., a negative number), the return value is 0.

ln(x)

```
double  ln(x)                           /* Natural log of x */
double  x;
```

Returns the natural logarithm of the value of x.

log(x)

```
double  log(x)                          /* Base 10 log of x */
double  x;
```

Returns the base 10 logarithm of the value of x.

exp(x)

```
double  exp(x)                          /* e to power x */
double  x;
```

Returns (the constant) e raised to the power of the value of x.

power(x,y)

```
double  power(x,y)                      /* x to power y */
double  x,y;
```

Returns the value of x raised to the power of y.

sin(x)

```
double  sin(x)                          /* sin of x */
double  x;
```

Returns the sine of x, where the value of x is expressed in radians.

cos(x)

```
double  cos(x)                          /* cosine of x */
double  x;
```

Returns the cosine of x, where the value of x is expressed in radians.

tan(x)

```
double  tan(x)                          /* tangent of x */
double  x;
```

Returns the tangent of x, where the value of x is expressed in radians.

cotan(x)

```
double  cotan(x)                        /* cotangent of x */
double  x;
```

Returns the cotangent of x, where the value of x is expressed in radians.

asin(x)

```
double  asin(x)                         /* arc sine of x */
double  x;
```

Returns the arc sine of x, where the returned value of x is expressesd in radians.

acos(x)

```
double  acos(x)                          /* arc cosine of x */
double  x;
```

Returns the arc cosine of x, where the returned value is expressed in radians.

atan(x)

```
double  atan(x)                          /* arc tangent of x */
double  x;
```

Returns the arc tangent of x, where the returned value is expressed in radians.

sinh(x)

```
double sinh(x)                           /* hyperbolic sine of x */
double  x;
```

Returns the hyperbolic sine of x where x is expressed in radians.

cosh(x)

```
double  cosh(x)                          /* hyperbolic cosine of x */
double  x;
```

Returns the hyperbolic cosine of x, where x is expressed in radians.

tanh(x)

```
double  tanh(x)                          /* hyperbolic tangent of x */
double  x;
```

Returns the hyperbolic tangent of x, where x is expressed in radians.

atan2(x,y)

```
double atan2(x,y)                        /* arctangent of x over y */
double x, y;
```

Returns arctangent of x divided by y, where, if y equals 0 and x not equal to 0, it returns pi divided by 2. If both are 0, it returns 0.

SOFTWARE LICENSE AGREEMENT

BETWEEN: Ecosoft Inc. License No. EC-139-M
P.O. Box 68602
Indianapolis, IN 46268
("Ecosoft")

and

("Licensee")

Ecosoft grants to licensee a nontransferable, nonexclusive license to use the _____TM program (the "Software") subject to the terms and conditions of this Agreement.

1. Licensee may not transfer, assign, or sublicense either the Software, manuals or documentation supplied with the Software.

2. Licensee may make backup copies of the Software provided EACH BACKUP COPY CONTAINS THE ECOSOFT COPYRIGHT NOTICE. IN NO EVENT SHALL LICENSEE MAKE COPIES OF ANY PORTION OF THE MANUALS AND/OR DOCUMENTATION ACCOMPANYING THE SOFTWARE WITHOUT PRIOR WRITTEN CONSENT OF ECOSOFT.

a. Licensee may use the Eco-C C Compiler for commercial use, provided that the Ecosoft copyright notice (i.e., Eco-C, Copyright 1983 by Ecosoft Inc.) appears near the beginning of the program source code and documentation.

3. Violation of the terms of this Agreement are cause for this license to be terminated by Ecosoft without notice. In the event Ecosoft elects to terminate, by sending written notification to Licensee at the address indicated on Ecosoft's records, by registered or certified mail, Licensee shall, within five (5) days following receipt of such notification, return to Ecosoft the original media upon which the Software is recorded along with all manuals and/or documentation relating to the Software. Licensee shall provide a written statement attesting that no copies of the Software have been retained in any form.

4. This license is granted by Ecosoft on an "AS IS" basis. ECOSOFT MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE, UNINTERRUPTED OR THAT ANY PROGRAM DEFECT WILL BE CORRECTED. ECOSOFT MAKES NO WARRANTY WITH RESPECT TO THE MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR USE OF THE SOFTWARE.

IN NO EVENT SHALL ECOSOFT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY THE SOFTWARE, INCLUDING BUT NOT

LIMITED TO LOSS OF BUSINESS, ANTICIPATION OF PROFITS OR INTERRUPTION OF SERVICES.

5. This Agreement shall be construed according to the laws of the State of Indiana and contains the entire Agreement of the parties. No representations, promises, agreements or understandings, written or oral, not contained herein shall be of any force or effect. No change or modification of this Agreement shall be valid or binding unless it is in writing and signed by the party to be charged.

IN WITNESS WHEREOF, Ecosoft and Licensee have executed this Agreement this _____ day of _____, 198___.

ECOSOFT, INC.

"Ecosoft"

By:_____
Jack J. Purdum, President

_____
"Licensee"

By:_____

Its:_____

Dealer:

_____

_____

_____

_____