Table of Contents

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 on it in drive A and a blank, formatted disk in B. (We have used <CR> to signify pressing the RETURN key.)

```
A>PIP <CR>
  *                    (now place the Eco-C disk in A)
   B:=A:*.* <CR>       (    <---- and enter this line)
                       (Use OV option for copyinh
```

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

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

In addition, each compiled program will generate (on its own) TOKEN.CWK and PCODE.CWK as intermediate files during compilation. These intermediate files will be erased automatically as the compilation progresses. You probably will want to place your 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. Finally, do not write programs using the file names above.

If disk space is a problem, the files may have a maximum split as follows:

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

DISK2 - M80.COM

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

Using the above distribution of files, DISK2 may be used for the source file, compiler output and assembler output. DISK1 is used for the compiler and the compiler working files; while DISK3 is used for linking the output of the assembler and the compiler libraries. 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).

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 this release.

The full C syntax is supported except;

a. bitfields
b. initializers(Initializers should be available in April, 1983. Aggregate Initializers are supported.)
c. parametized macros
d. #line macro preprocessor directive
e. in compound expression following a #if, macro expansion is not done

Most of the above will be available in subsequent releases of the compiler. We also expect additional optimization (we've concentrated on longs and floats so far) to substantially shrink code size and increase speed. You must have a signed license agreement on file to be eligible for these updates.

Facts, Suggestions and Programming Hints

The following is a list of specifications and suggestions that may prove useful. Some represent "C spec" rules that are not obeyed by existing compilers and might cause an error message that would not be generated by other compilers. Others are limitations that will disappear shortly.

1. unsigned long, unsigned char and unsigned short data types will generate an error message.

2. stdin, stdout, stderr and stdlst equate to fd0, fd1, fd2, and fd3 respectively. (A file named INIT.ASM controls the number of fcb's and iob's that are generated plus setting the stack.)

3. On function calls:

a. A function parameter of float is automatically promoted to double and short or char goes to an int.

b. A function cannot return a char; only an int, unsigned, long, double, or pointer no matter how declared.

c. Given the choice of char or int, make it 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 it: 1) take its address with the address operator in front of it, or 2) have a period and a member name following its name.

6. Binding of structure members to a structure is absolute. If a pointer is to a structure s1 and now you want to use it with s2, you must cast the pointer to s2 or we will generate an error message.

7. If a function is not recursive, you are usually better off to declare variables as static rather than auto or register. If 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.

10. #include's may be nested only two deep.

11. Floating point numbers can have up to 17 significant digits. The range for the exponent is -38 to +38. At the present time, only binary arithmetic is supported (BCD is forthcoming).

12. Floating point and long constants are combined at run time rather than compile time. This, too, will be fixed.

13. All program source files must terminate with a carriage-return, linefeed pair (CR-LF). Failure to observe this will cause the error handler to produce odd results.

14. We do not yet have: lseek() ~~or unlink()~~ functions, and appending to a file. This will disappear when random files are finished.

The majority of the limitations mentioned above will disappear very soon. 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. Finally, the library is a bit meager at present, but will grow shortly.

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 defined by K&R. Finally, we have adhered to the function definitions presented in K&R as much as possible in a CP/M environment.

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 that 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.

Typically, the file 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.

Compiling the Program

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

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

The program will supply the extention of ".C" if one is not supplied when the preprocessor is envoked. If you supply a file extention (e.g., ".XXX"), it overrides the default (".C"). A different souce disk may be specified. For example,

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

The program will automatically load and run the next two passes of the compiler (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.

As an example, suppose a source program is named TEST.C. To use it with this option, the compiler 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.

-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 avoids 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
```

-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 of the above examples are allowed. If a file type is added to the file name the default type of '.MAC' will not be used.

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

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; the order is unimportant. Example:

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

or

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

both produce the same 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.

The output file from the assembler will be 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 for CP/M.

Linking the Program

The .REL file from the assembler is then linked to the standard library routines (e.g., CF.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:

```
                A>L80 TEST,DATE,TEST/n/e
                      ^    ^    ^
                      |    |    |
input .REL file-------     |    ---------- output .COM file
                           |
                           |
     .REL function(s) to be linked with program
```

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. In between these two file names are the (.REL) function(s) you want to link in with the program. Each must be separated from the other by a comma.

Again, additional details on L80 are found in the Microsoft manual.

[illegible] Page 8 and here at end of page.

If you use the /S flag in a link command line, you must explicitly specify searches for the Ec-C libraries also. One of the following libraries plus ECC and EC2 must be searched.

CFF - Floating point printf() on file I/O on getchar()/putchar().

CIF - Integer printf() and file I/O on getchar()/putchar().

CFC - Floating point printf() with BDOS calls for getchar()/putchar().

CIC - Integer printf() with BDOS calls for getchar()/putchar().

As an example, if your library were named DBASE.REL and you used compiler switche -c the link would include the following library search [illegible]:

```
l80 ....,dbase/s:cif:c:ecc/s:ec2/s:...
```

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.

```
                /* 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);
}
```

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 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 and change the putchar(CLEARS) statement to a puts(). For example, if you are using a SOROC terminal which uses an escape (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, 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.

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 (**arg**ument **c**ounter) used when the program was envoked. The argv[] variable is an array of pointers that points to what command line arguments were entered.

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[];
{
        char 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 "read" 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, while **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.

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.), although you probably will not need to know the details contained therein.

What to Do if You Run Out of Memory

Unlike many other C compilers that are available, the Eco-C compiler does not require that 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 them together.

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()

```
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.

calloc()

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

Returns a pointer to sufficient storage to 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".

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.

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().

ftoa()

```
int  ftoa(s, d, prec, type)      /* Float to ASCII          */
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 18 significant digits of precision may be requested (i.e., prec = 18). The "type" variable may be 'g', 'e' or 'f' [see discussion of the printf() function]; otherwise the floating point number is free-form. Since the string is null terminated (i.e., '\0'), "s" must be large enough to hold "prec" bytes + 1 characters.

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.

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, iop)          /* get string from file */
char  *s;
int   i;
FILE  *iop;
```

Reads "i" characters from iop 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.

_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.

fopen()

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

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

"r" Open for reading. The file must already exist to use this mode. ASCII text files have thecarriage-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. (The "a", or append, option for random files is forthcoming).

fputs()

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

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

getc()

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

Reads a character from the input stream associated with "p". The character is normally returned as a positive integer although it may return EOF upon reading end-of-file or ERR if an error occurred during the read.

getchar()

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

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

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. NULL

hex()

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

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

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.

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.

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.

octal()

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

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

7. Page 20 printf() Change function definition to be

```
printf(control,arg1,arg2,.....,argn)
char *control;
```

The number of arguments passed is only limited by stack space. The number of arguments is specified in the control character array.

- 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 smallerthanthe width) blanks areusedunlessthe 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). REL Files

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).

putc()

```
int putc(c, p)                          /* Output a character */
char  c;
struct iobbuf *p;
```

Outputs the character "c" to the stream pointed to by "p" 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.)

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.

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). If 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.

Assembly Language Functions

The following is a list of assembly language functions that may be used. The are part of the EC2.REL disk file. They should be self-explanatory.

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


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


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


_exit()                         /* Close files and end program */


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


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


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


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

        /* if file i/o then n must be multiple of 128 */


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

        /* same note as write */


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


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

```
unlink(str)
char *str;

Deletes the file specified by str.
```

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.

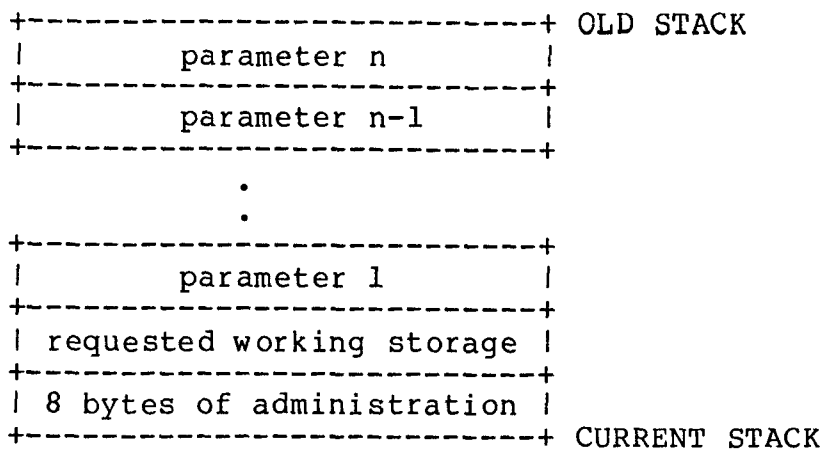
$RTN	- will return the value pointed to by the HL register pair to the calling function.
$RETVAL	- will return the value contained in the DE register pair to the calling function. If a long was requested then the value will be padded.
$RETM1	- will return a -1 to the calling program.
$FE	- calls the 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 DE for the length in BC.

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   DE,@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>180 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.

```
            .Z80
            INCLUDE FCB.MAP
NFCB        EQU     10              ;SET FILES TO 10
            INCLUDE FCB.ASM
            INCLUDE IOB.ASM
            CSEG
$INIT::     POP     BC
            LD      HL,0        ;SET STACK TO TOP OF MEMORY
            LD      SP,HL
            PUSH    BC
            LD      HL,$FCB
            LD      E,L
            LD      D,H
            INC     DE
            LD      BC,NFCB*FCBL
            LD      A,B
            OR      C
            RET     Z
            DEC     BC
            LD      (HL),0
            LDIR
            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

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 will cannnot 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.

To implement the file I/O as it exists in UNIX would require double buffering of all file I/O. This approach was considered too costly given the memory space available. Only single buffering in the user's program is used. This buffer is the one created by fopen() and pointed to by the associated _iob.

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. In the CP/M environment the count must be a multiple of a CP/M sector size (i.e. 128). 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>.

32 Change Note: to read

In the non-binary mode, the CP/M EOF indicator (0x1a) is translated by getc() to EOF and all characters are stripped to 7 bits. The CP/M EOF indicator must be written explicitly before closing a non-binary mode file.

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 was 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.

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 ___.
3	A comma or semicolon was expected instead of ___.
4 1009 1015	An identifier, (or * was expected instead of ___.
5	A semicolon was expected instead of ___.
6	A closing parenthesis was expected instead of ___.
7	An opening parenthesis was expected instead of ___.
8	A closing brace was expected instead of ___.
9	A closing bracket was expected instead of ___.
10	A parameter list was expected instead of ___.
11 1011 1012	An opening brace was expected instead of ___.
12	A parameter declaration list or opening brace was expected instead of ___.

13

The parameter declaration list is in error.

14

An array size was expected but found ___.

15

A structure or union tag or opening brace was expected but found ___.

16

A type specifier was expected but found ___.

17

A statement or declaration was expected but found ___.

18

A type specifier or storage class specifier was expected but found ___.

19

A storage class specifier was expected but found ___.

20

An attempt was made to "goto" an illegal label name.

21

An attempt was made to perform an illegal "break" or "continue"

22

The "while" is missing from a "do{ }while" statement.

23

A multiply defined "default" was found in the "switch".

24

Multiply defined label.

25

An identifier, opening parenthesis or constant was expected in the expression instead of ___.

26

Variable is undefined.

27

Illegal indirection or array reference.

28

Expected a comma or closing parenthesis instead of ___.

29

Internal Error - illegal multiplier.

30

Illegal bitwise operand.

31

Illegal pointer arithmetic.

32

Illegal floating point operation.

33

Illegal negation.

34

Illegal logical not operation.

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.

38

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

39

Input file not found.

40

A colon was expected in ternary "?_:" but was not found.

41

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

42

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

43

Out of memory.

44

Illegal address of (&id).

45

WARNING - An extra opening brace was found and ignored.

46

Illegal use of struct or union as source operand.

47

Attempted assignment to a constant.

48

Illegal assignment to structure, union, function name or array name.

49

Attempt to "type" a parameter which is undefined in the function declaration.

51

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

52

A referenced structure or union tag has not been defined.

53

An external data definition and the current data definition do not match.

Pass 3B Error 54 Change to read

54

Label ... was jumped to but was not defined.

60

Initializers not currently supported.

1000

Expected type specifier, [{ ;) = or : instead of ___.

1001

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

1002

Expected type specifier, typedef name, [(; ,) = or : instead of ___.

1003

Expected type specifier, typedef name, [{ ; ,) = or : instead of ___.

1006

Expected identifier, * (or : in structure or union declaration instead of ___.

1007

Expected : ; or , instead of ___.

1010

Expected = ; or , instead of ___ in declaration list.

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;

Quick Function Reference

```
char        *alloc(u)
unsigned    u;

double      atof(s)
char        *s;

int         atoi(s)
char        *s;

long        atol(s)
char        *s;

int         _bdos(val,call)
int         val,call;

char        *calloc(count,size)
unsigned    count,size;

            chain(s)
char        *s;

int         close(fd)
int         fd;

int         creat(s)
char        *s;

long        decimal(s)
char        *s;

int         exit(i)
int         i;

            _exit()

int         free(s)
char        *s;

int         ftoa(s,arg,prec,type)
char        *s;
double      arg;
int         prec,type;

int         _ftoa(s,dbl)
char        *s;
double      dbl;

int         fclose(fp)
FILE        *fp;

int         fflush(fp)
FILE        *fp;
```

```
char        *gets(buff)
char        *buff;

unsigned    getw(fp)
FILE        *fp;

long        hex(s)
char        *s;

int         isalpha(c)
char        c;

int         isdigit(c)
char        c;

int         islower(c)
char        c;

int         isspace(c)
char        c;

int         isupper(c)
char        c;

long        octal(s)
char        *s;

int         open(s)
char        *s;

int         printf(control,arg,..
char        *control,*arg;

int         putc(c,fp)
char        c;
FILE        *fp;

int         putchar(c)
char        c;

int         putw(u,fp)
unsigned    u;
FILE        *fp;

int         read(fd,buf,n)
int         fd,n;
char        *buf;

char        *sbrk(u)
unsigned    u;

int         strcat(d,s)
char        *d,*s;
```

```
char        *fgets(s,i,fp)
char        *s;
int         i;
FILE        *fp;

int         _fillbuff(fp)
FILE        *fp;

FILE        *fopen(name,mode)
char        *name,*mode;

int         fputs(s,fp)
char        *s;
FILE        *fp;

int         getc(fp)
FILE        *fp;

int         getchar()
```

```
int         strcmp(d,s)
char        *d,*s;

int         strcpy(d,s)
char        *d,*s;

int         strlen(s)
char        *s;

int         tolower(c)
char        c;

int         toupper(c)
char        c;

int         write(fd,buf,n)
int         fd,n;
char        *buf;
```

Operator Precedence
(Highest to lowest)

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

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.