MISOSYS,Inc.
P.O. Box 239
Sterling, Virginia 22170-0239

703-450-4181

MC C-Language Compiler

Reference Manual

Published by:

MISOSYS, Inc.
P. O. Box 239
Sterling, Virginia 22170-0239
703-450-4181

*** * * A T T E N T I O N * * ***

The MC compiler can be used to generate any software product, commercial or otherwise, without payment of any royalties to MISOSYS, with the exception of the following: when MC is used to generate another compiler product, no part of the MC-supplied libraries may be included in the run-time support of the generated compiler.

The MISOSYS C Language Compiler

Introduction

Congratulations! You have purchased the finest C compiler package available for your machine environment. MC will prove itself to be a valuable investment of your software dollars. Thus, before diving into MC, we suggest that you follow the guidance expressed in the following paragraphs.

Read the entire introduction to get an idea of what MC is all about. This will provide an immeasurable insight into the C programming power available to you with the MC compiler. It may also help you understand the content of the remaining chapters.

MC requires the use of a macro-assembler that generates Microsoft compatible relocatable object module files. An assembler is not included with the compiler package. You may have purchased the MISOSYS "Relocating Macro Assembler Advanced Development System" (MRAS) or Microsoft's M-80 assembler.

If you have neither, you will not be able to use the compiler package until you obtain a copy of either MRAS or M-80. C source code is prepared using the editor which is included with your assembler package.

Make backup copies of the MC distribution diskette to use as a working master. The compiler package is released on a 40-track double density DOS data diskette. This diskette contains all of the files associated with the compiler. We suggest that you make one set of archival backups and store them away in a secure area (safe from dust, dirt, magnetic fields, etc.). Then make a working backup of the distribution diskette. The procedures for making backup copies can be located in the UTILITY section of your DOS user manual under "BACKUP". If you are going to use MC with MRAS, delete "MC/MAC" and "MCMACS/MAC" from the working disk. Conversely, if you are going to use MC with M-80, delete "MC/ASM" and "MCMACS/ASM" from the working disk.

If you are using MC on a two drive floppy system, you will have to create a DOS system diskette with a maximum of free space. Using the DOS PURGE utility on a fresh backup of DOS can create this “working system diskette”. You may remove all files except SYS0-SYS4, SYS6, SYS8 (SYS8 can be removed from TRSDOS 6.x), SYS10-SYS12.

A 40-track double density minimal system diskette per the above has about 144K free. Copy your assembler, linker, and editor from your assembler disk. Then copy the MC/JCL file and the compiler command files from the working backup to this system diskette. If there is still space left on your system disk, copy some of the other files from the work disk. You will not be able to copy all of the files; in fact, most will still be on the working disk. If your machine has 128K and you are using a RAMdisk, you will find it beneficial to copy the library /REL files to your RAMdisk. Once this is done, remove the files from the working data disk that were copied to the system disk. This should leave working space on the data diskette in your second drive for C source files and the files generated during the programming session.

Notice that MC requires a two-drive system. In fact, you may find it prudent to use MC on a three-drive system - or one using two-sided drives - or even a hard disk environment. If you have gotten this far, continue to read this chapter and discover everything that MC provides you.

MC Provided Files

MC is a complete C compiler. It adheres to the "standards" expressed by Brian Kernighan and Dennis Ritchie in their book, "The C Programming Language". MC includes an extensive UNIX System V compatible function library. All you need to generate executable CMD programs is a macro assembler that generates Microsoft compatible relocatable object modules. There are many files on your distribution diskette. In fact, in order to make room for all of the files, the header files have been merged together into a single-source archive file. The following C program (which may be on the disk, space permitting) will separate the archive into the individual header files:

```
/* unarc.ccc - 11/29/85 */
char aline[81];
main()
{
   while (gets(aline))
   {
      if(!strncmp(aline,"/*%",3) && gets(aline))
      {
         if (!freopen(strcat(aline,":3"),"w",stdout))
            exit(-1);
         else
         {
            fputs("\nGenerating: ",stderr);
            fputs(aline,stderr);
         }
      }
      else puts(aline);
   }
}
```

Once you have established your working compilation and assembly system, the unarc program should be typed into a file named "unarc/ccc". You may change the ":3" which appears in the sixth line of unarc to specify which output drive the header files should be written to. Then compile unarc with the DO command: "DO MC (N=UNARC)". After the command program has been successfully created, invoke unarc with the command:

```
UNARC <HEADERS.H
```

Here is a description of all of the files provided with the MC C compiler system. The description for the files provides their uses:

ERRNO/H

This header file supplies the constants associated with the UNIX error numbers. It also defines the exception structure used in the high-level math function error trapping.

FCNTL/H

This header file supplies the definition of the constants required for using the open() or fcntl() functions.

HEADERS/H

This is the actual file where all of the header files are stored on the release diskette. The header files are separated into distinct files by the "unarc/ccc" utility.

IN/REL

The installation function library is contained in this relocatable object module library. These functions supply graphics plotting, BASIC type string manipulation, a DOS call facility, and other hardware specific features.

LIBA/REL

This relocatable object module library contains all of the low-level routines needed to service mathematical operations and perform other maintenance functions. None of these routines are directly available to the C programmer at the function level. Any that are needed by your program are automatically added to the resulting executable command file during the linking process.

LIBC/REL

This relocatable object module library contains the standard function library. It is an implementation of the portable library available under most installations of C compilers. These functions allow programs to be written which will be directly usable under other C language systems that have the standard library available. The standard functions perform such tasks as character and block file input/output, dynamic memory allocation, system control, formatted input and output, and string handling. The standard library was designed to be compatible with the standard library documented in AT&T's "System V Interface Definition" for the UNIX operating system.

MATH/H

This header file defines the error constants used in low level math errors as well as declares all of the math functions as externs with the proper type. It also specifies the "#option MATHLIB" needed to direct an automatic search of the MATH/REL library during the linking process.

MATH/REL

This relocatable object module library contains the high level floating point function library. It includes the trigonometric library (sin, cos, tan, asin, acos, atan, sinh, cosh, tanh), functions for logarithm and exponential, square root and power, and other high level mathematical operations. Also included is the support for exception error handling as well as floating point formatted print and scan.

MC/ASM or MC/MAC

This file is the primary assembler source file assembled by MRAS or M-80. It accesses your "main" program and establishes the necessary interfacing between your program and the runtime modules needed to make a "complete" executable CMD program.

MC/CMD

This is the root segment of the MC language compiler. MC accepts, as input, C source code files, and outputs an assembly source file. In order to organize files in a structured manner, MC source code files have a file extension of "/CCC"; output assembler files have a file extension of "/ASM" or "/MAC" depending on a compiler option.

MC/JCL

This is a Job Control Language DO file which is used to turn the multi-step process of generating an executable program into a single command line invocation.

MC/OV1

This is the compiler overlay which performs the preprocessor phase of the compilation process. It will generate an intermediate file which contains a tokenized transformation of your C source file.

MC/OV2

This is the compiler overlay which reads the tokenized preprocessor output and performs major parsing of the C source. The intermediate file is then deleted.

MC/OV3

This is the compiler overlay that generates the assembler output file based on the intermediate processing of MC/OV2.

MCMACS/ASM or MCMACS/MAC

This file contains the "#option" defaults and assembly language macros referred to by the compiler and used by the assembler. The assembler always automatically accesses this file when assembling the MC/ASM or MC/MAC file.

README/TXT

This file contains last minute documentation not contained in the manuals. It should be listed via the DOS's LIST command for reference.

SETJMP/H

This header file defines the environment buffer associated with the setjmp() and longjmp() functions.

SGTTY/H

This header file defines the structure associated with the use of the ioctl(), stty(), and gtty() functions.

STAT/H

This header file defines the structure and symbolic constants associated with the fstat() function.

STDIO/H

The standard I/O header file supplies constants and definitions which are needed to use the standard I/O library.

TIME/H

This header file defines the data structure associated with the asctime() and localtime() functions.

Z80REGS/H

This header file defines an easy to use structure for passing data to or from the machine's registers for use with the call() function.

MC Environment

MC was designed to be compatible with C programs written and intended to run under UNIX. Thus many features of UNIX were incorporated into the design. These features include standard I/O devices, standard I/O redirection, device independence, command line arguments, wild-card file specifications, dynamic memory allocation, and a consistent system interface.

To make C a portable language, the interface within a program to the external world is isolated in a standard library. A program written in C using only the standard functions to perform input, output, and memory allocation can be transported in source code form to another system, recompiled, and run with minimal changes. The MC system includes a standard library that is compatible with programs developed under UNIX System V. Thus, programs developed under your DOS with MC will be compilable under UNIX as well. The reverse is also true, except in cases where features not implemented in MC are used in the program developed under UNIX.

The MC file system supports two methods of UNIX-type access. The first is conceptually called buffered I/O and concerns itself with file streams. The identification of a file stream is via a FILE POINTER. Streams are accessed with functions such as: fopen(), getc(), putc(), fprintf(), etc. The second type is conceptually called unbuffered low-level I/O and is generally a direct access to the DOS. We will term this type "block file" I/O. The identification of block file I/O is via a FILE DESCRIPTOR and the I/O is accessed via functions such as: lseek(), read(), write(), open(), etc. The DOS in this implementation generally buffers either type.

Standard Input/Output

Any program generated by MC will normally have three files automatically opened when the program begins execution. These files are standard input, "stdin" (normally the console keyboard); standard output, "stdout" (normally the console display); and standard error, "stderr" (normally the console display). The program can access these files without opening them by using standard library functions since the C standard library automatically opens these standard files. They are also automatically closed when the program is exited as well. Thus, the program which uses the standard I/O files exclusively can deal with input and output and leave the opening and closing to the C standard library.

File specifications usually have a filename field, an extension field, and a drive field. They may also carry a password. Since some operating systems use the slash "/" character to denote the beginning of the file extension while others use a dot ".", the MC file system accepts either. In this way, your programs may include file specifications such as

```
#include "stdio.h"
```

which makes your source more transparent to UNIX. If you wish, you may also enter a file specification using the slash separator. This acceptance of the dot is true everywhere a file specification can be entered throughout MC. The limitation is that a file specification must contain an extension field if it contains a password field. Thus, "YOURFILE.PASSWORD" is an invalid file specification whereas "YOURFILE.DAT.PASSWORD" is valid.

MC compiled programs can also make use of the wildcard file specification expander. The incorporation of the wildcard expander is specified at compile time by a compiler #option. With this option installed, a filespec command line entry can use the wildcard characters "*" to indicate an automatic match of the remaining portion of the field and "?" to indicate an automatic match for all characters in this character position.

Wildcards allow you to perform an operation on a group of "matched" files on a single disk drive or on all disk drives. The wildcard expander searches a designated drive or drives and expands the command line argument list (argc, argv) to include all file specifications that match your wildcard specification. This facility is detailed under compiler options.

Standard I/O Redirection

The UNIX system characterizes three standard files: standard input, standard output, and standard error output. The standard I/O files normally operate to and from the user's console. Under your DOS, these are usually defined as the "*KI" (keyboard) and "*DO" (video screen) devices. However, a facility is inherent within the MC standard library to permit you to redirect the standard I/O devices - thus the term "I/O redirection". The user can give a device or file specification that will be used in place of the normal specification when a standard file is opened. This is done on the DOS command line when the user invokes the program.

When the left angle bracket symbol, "<", appears on the command line, followed by a file specification, that file specification is used when the standard input file is opened. Similarly, the right angle bracket symbol, ">", causes substitution of the standard output file specification, the ">>" causes standard output to be appended to the redirected file/device, and the number sign symbol, "#", causes substitution of the standard error file specification. Spaces are permitted between the redirection character and the file specification.

It may not be immediately obvious how this feature can be used. Here is an example C program that illustrates the straightforward use of standard I/O redirection. The following program can be used to copy any file to any other file (remember that "file" can be any device or DOS disk file).

```
/*  CLONE  -  copy standard input to standard output  */
#include stdio.h
int c;
main ()
{
   while ((c = getchar()) != EOF)
      putchar(c);
}
```

The example program simply copies the standard input to the standard output until end of file is reached. Once this program is compiled, assembled, and linked, it can be used to copy any file to any other. For example:

```
CLONE <CLONE/CCC
```

will display the file "clone/ccc" on the system console. The command:

```
CLONE >*PR
```

lets the user type to the system printer. If disk file copying is needed, the command:

```
CLONE <INFILE/ASM:1 >OUTFILE/BAK:2
```

will copy the file "INFILE/ASM:1" to the file "OUTFILE/BAK:2". If the user wishes to have a printed log of any error messages that a program puts out, use something like :

```
MC TESTLIB #*PR
```

Any messages that MC outputs to the standard error file will be redirected to the printer device in lieu of the console display.

Command Line Arguments

When a C program is invoked, the command line is parsed into a list of arguments. A single argument is represented by a continuous string of nonwhite space characters surrounded by white space. At a minimum, a command invocation will have one argument - the program name itself. The list of arguments is passed to the executing program through the "argc" and "argv" arguments of main().

"Argc" is an integer that contains the number of arguments while "argv" is an array of elements, each element of which is a pointer to a character string. These arguments will usually be declared as:

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

providing your program with a method of recovering various data entered on the command line by the invoker. Command line arguments to a C program may be enclosed in single or double quotes. This allows the inclusion of special characters and whitespace in an argument. The quotes are stripped before the argument is passed to main(). To include a quote in an argument, precede it with a backslash "\". To include a backslash, use a double backslash "\\".

For example, in a command line such as:

```
PROCESS INPUT/DAT:3 TEMPY/TXT:5 +O=:7 -L "+C=This is a message"
```

six arguments will be passed to main() and "argc" will be equal to 6. If main() was the program:

```
main(argc,argv)
{
   while (argc--)
   puts(*++argv);
}
```

it would output the strings, "PROCESS", "INPUT/DAT:3", "TEMPY/TXT:5", "+O=:7", "-L", and "+C=This is a message". Note that the latter argument is a text string having imbedded blanks; this is permitted when the command line argument is enclosed within quotes. Any redirection specifications will be processed before the command line arguments and will not appear in the argument list.

Standard Header Files

Standard header files are files that contain definitions peculiar to a system. They usually take the form of "#define" statements and "extern" statements within the header file. In order to use certain libraries, a corresponding header file should be included (using the "#include" statement). The file extension of "/H" is used for MC header files to be consistent across versions of UNIX and other systems sporting C compilers.

A program to be compiled and linked with MC should usually have the file "STDIO/H" included to compile properly. STDIO/H also defines various system dependent parameters, such as end of file (EOF) and end of line (EOL). The FILE POINTERS <stdout>, <stdin>, and <stderr> are addresses in the standard library which do not need to be defined before use; however, the FILE DESCRIPTORS <STDIN>, <STDOUT>, and <STDERR> are defined in the header file.

MC includes many header files that are standard under UNIX. These files contain symbolic definitions of constants used in various functions in the standard library. It is essential that you use the symbols defined in the header files when noted in the documentation for the functions. Do NOT extract the symbolic constant's value and use the number in your program. Numbers are not necessarily portable across C installations; however, the symbolic NAMES are a part of the AT&T definition and are portable! Thus, by using the symbolic names defined in the appropriate header files, you will minimize any conflict in portability, not to mention compatibility with future releases of this compiler package. Any header file required by a function is documented with the function needing it.

Function Libraries

Commonly used functions are collected into FUNCTION LIBRARIES. The functions in a library can be used by the programmer without the need to rewrite, recompile, or reassemble the functions needed. Once a C program has been compiled and assembled, it can then be combined during the link phase with the functions it requires. Only those functions necessary for the execution of the program are linked to the compiled program.

Certain functions required by many programs are included in a special library called the STANDARD LIBRARY. The standard library is the common denominator among all C language installations. Programs written using functions in the standard library are easily transported to any other computer supporting a C language system with the standard library implemented. The most important aspect of the standard library is that it allows the details of each system's peculiar operating environment to be hidden from the programmer's view. The standard library provides the functions for input/output, memory allocation, and character set manipulations.

What is typical in UNIX installations is to have the standard functions "callable" from C in a library named "LIBC". In addition, a collection of subroutines used by the compiled C program to perform basic operations but not directly callable from C is contained in a library named "LIBA". MC follows these standards. MC also incorporates the high-level math functions into a "MATH" library as is also found under UNIX systems.

Users can also create their own collections of often-used functions that can be used in the same manner as the standard library. These USER LIBRARIES reduce the programming time, compilation time, assembly time, and program complexity necessary in creating new programs. Functions, once defined, written, and tested, can be added to the user library and need only be referred to by name in later programs. The linking process brings the functions into subsequent programs without the need to recompile and reassemble.

Relocatable object module libraries are created and maintained using the MLIB librarian. This facility is included with the MRAS assembler package or is available separately for M-80 users. You can even build a user library with the APPEND command in the DOS by using the (STRIP) parameter. More on library building is included in Chapter 5, "Advanced Topics".

Special purpose libraries may also be created for use in particular types of applications. For instance, the functions that are specific to the hardware provided with your C package are in the special purpose library, IN/REL. This is an example of how the C language avoids the trap of non-standard extensions being included within the language.

Runtime Error Control

MC provides certain facilities for the detection and control of four types of runtime errors. These errors may be classified as DOS I/O errors, C environment errors, low-level floating point errors, and high-level floating point mathematical errors. This section will describe the function and purpose of the control the programmer has over these errors; however, the mechanics of implementing the control will be discussed in Chapter 5, "Advanced Topics".

To begin with, the standards of UNIX System V dictate a protocol for most functions to return a value indicating that an error occurred. Chapter 4, "Function Libraries", documents the error return conditions for each function which supports an error return code. It is up to the programmer to provide appropriate code to detect and act on that error return code.

The DOS I/O errors are characterized by problems in accessing files, reading from files, or writing to files. When such an error is detected in the MC I/O package, the DOS I/O error is stored in the File Control Area assigned to the file stream, provided the file has been successfully opened. This error number may be obtained through the ferror() function. Next, the error is passed to a routine in the I/O package which will optionally display the runtime error. The option() function provided in the standard library can be used to control the behavior of this I/O error display - or suppress it entirely. Next, the DOS I/O error will be translated to an appropriate UNIX error number and stored in the global error variable, "errno". Finally, the error indication will be reported back through the highest library function invoked to return an indication of error to your program.

There are other types of errors which could be experienced. A memory allocation request could be unsatisfied because insufficient free memory was available. A request to obtain status on a file stream could be unsatisfied because the request was associated with a "character special device" (i.e. *DO) rather than a file. These types of errors have an appropriate UNIX error number assigned. Thus, any of these errors will store the designated UNIX error number in the global error variable, "errno". The error indication will be reported back through the highest library function invoked to return an indication of error to your program.

On larger computers, floating point errors such as overflow and underflow are usually trapped by hardware. When detected, they generate a hardware interrupt so software routines can be notified to take whatever action is desired. Since all of the floating point routines provided in MC are implemented in software, your computer cannot generate a hardware interrupt when a floating point error occurs. When MC does recognize a low-level floating point error, it stores an error code in "errno", the global error variable. In addition, MC provides a floating point vector, "_fltvec", which is called whenever such an error has been detected. This vector can be altered by your program to point to your function which takes whatever action you decide to implement. In its normal state, "_fltvec" does nothing but return.

The fourth facility for trapping errors is provided by "matherr()". High-level floating point errors are characterized by such things as trying to take the square root of a negative number, trying to take the log of a negative or zero number, or trying to take the Arc sine of a value which is not in the range extending from minus one through plus one. UNIX System V documents a floating point exception handler, called matherr(), which is called when such high-level mathematical errors are encountered. An exception structure contains information pertinent to the detected error at the time that matherr() is invoked. The global error variable, "errno", is also loaded with the appropriate error number. In addition, a uniform error message is optionally written to standard error. Your program may provide its own matherr() function to handle the detected error as you see fit. Illustrations of this facility appear in the documentation of those functions that support this high-level floating point exception handling.

It should have been evident from the discussion that all error types make use of the global error variable, "errno". This variable is a UNIX System V feature. All of the error numbers are represented by symbolic values and appear in the "errno" header file. The "math" header file defines the exception structure and the symbolic definitions of math errors. These header files should be "#include'd" with your source program as appropriate in order to make use of the symbolic definitions.

Closing Comments

C encourages the use of structured programming methods. Unless one uses the "goto" statement heavily, C practically demands a structured approach to program construction. This is not to say that writing programs in C will automatically make you a good, structured programmer. This is a skill that is developed by learning and applying the basics.

Some understanding of structured design concepts is necessary in order to effectively use C. Probably the first frustrating thing that novice C programmers will encounter, especially if their experience is limited to BASIC and assembly language, is the discouragement of the use of "goto". Kernighan and Ritchie, in THE C PROGRAMMING LANGUAGE, state that the "goto" is never necessary, and in practice it is almost always easy to write code without it. The concept to understand is that the "goto's" are hidden within the program statements. C provides, in a coherent, understandable form, the program constructs that you have been building out of "goto's".

Many texts are available that can be part of your library. The following list is not to be considered all-inclusive but lists those texts (alphabetically) that we have had at our disposal.

THE BIG RED BOOK OF C by Kevin Sullivan (published by Sigma Press)

THE C PRIMER by Les Hancock and Morris Krieger (published by BYTE Books).

C PRIMER PLUS by Michael Waite, Stephen Prata, and Donald Martin (published by Howard W. Sams & Co., Inc.)

C PROGRAMMER'S LIBRARY by Dr. Jack Purdum, Timothy C. Leslie, and Alan L. Stegemoller (published by Que Corp.).

C PROGRAMMING GUIDE by Dr. Jack Purdum (published by Que Corp.).

THE C PROGRAMMING LANGUAGE by Brian W. Kernighan and Dennis M. Ritchie (published by Prentice-Hall). We will refer to this book throughout this manual by the abbreviation, "K&R", for Kernighan and Ritchie.

THE C PUZZLE BOOK by Alan R. Feuer (published by Prentice-Hall).

INTRODUCTION TO C by Paul M. Chirlian (published by MATRIX)

LEARNING TO PROGRAM IN C by Thomas Plum (published by PLUM HALL).

SYSTEM V INTERFACE DEFINITION (published by AT&T)

THE UNIX PROGRAMMER'S MANUAL Volume I and Volume II by Bell Telephone Laboratories, Inc. (published by Holt, Rinehart and Winston).

Language Definition

Program Environment - Functions

The C language is, in a word, functional. The basic unit of program construction when using C is the function. Every C program is a collection of functions. Each function is a collection of statements that work together to achieve (hopefully) a useful, well-defined, purpose.

Each function can have information passed to it when it is invoked ("called"). The elements of information passed to the called function are denoted as arguments. In C, arguments are copied onto the stack. The function can then access and use the "local" (known only to the called function) arguments, leaving the original copy of the arguments unchanged. Each argument is defined at the start of the function. Functions also return values to the functions that call them. In C this value can be an integer number, a long integer, a float, a double, or a pointer. The value returned can be compared to, placed in a variable, etc. Functions can appear in an arithmetic expression anywhere that a constant can.

Here is an example of a function:

```
square(num)
int num;
{
   return num * num;
}
```

The function, square(), returns the square of a number; in other words, the argument, "num", is multiplied by itself and the result is returned. Arguments are listed in parentheses after the name of the function, separated by commas. These arguments must be passed by the calling function in the same order as they appear in this list.

The BODY of the function is the group of executable statements that are within the braces "{" and "}". Actually, the grouping of statements in between braces denotes a special kind of statement called the COMPOUND statement. The compound statement is fully explained in the section on C language statements.

Every C program has a special function called "main" which is always the entry point to the program. When referencing a function within this narrative, we will put "()" after the name to identify it as a function. This is close to the way it looks in a C program. The function, main(), calls other functions, which in turn call other functions, etc... Thus, each program is a hierarchical structure of functions, with main() at the top of the hierarchy.

The DOS command line which invokes the C program is passed to the function main() using two parameters, "argc" and "argv". One C program can invoke another program by using the system() function. When the called program finishes, a special function, exit(), is used to return a value to the calling program. Programs can call other programs, passing any information using "argc" and "argv" command line arguments as explained in Chapter 1, "Introduction". In a way, each program appears as a function to other C programs and to the DOS.

Please scrutinize the illustration of functions in the following example:

```
main()
{
   /* The "main" function ...
      execution begins here!
   */
   say_hello();
   do_work();
   say_goodbye();
   exit(0); /* a normal exit, no error code */
}


/* sorry, we can't "goto" any of the functions below. */
say_hello()
{
   puts("Hiya!!!");
}

say_goodbye()
{
   puts("Bye y'all!!!");
}

do_work()
{
   while (not_quitting_time)
   {
      attach(nut,bolt);
      pass_on(widget);
   }
}
```

Statements - Simple and Compound

To create a C function, you have to state the action to be taken using C language STATEMENTS in the desired combination. Certain special statements are built into the language to provide the necessary programming constructs (sequence, iteration, and selection). You may be surprised, at first, by the limited number of statements built into the C language. The authors of the language wished to maintain the generality of the programming statements, forcing any special features to be outside of the programming language itself. Other languages often have extensions in the form of statements to provide specialized features, leading to incompatible versions of the same language. BASIC is a well-known example of a language extended in far too many different ways. The C language avoids this situation by only providing those statements necessary for structuring the program's logical flow and by placing all special features into function LIBRARIES. Function libraries are nothing more than collections of commonly used functions.

Simple C statements always end with a semicolon ";", the STATEMENT TERMINATOR. The C compiler depends on the semicolon to tell when a simple statement ends. Any number of simple statements may be entered, one after the other, to form a SEQUENCE of statements that are executed one at a time, first to last.

The brace characters, "{" and "}", are used to enclose a sequence of statements to form a COMPOUND statement. A compound statement can be used anywhere a simple statement can be used. Thus, the body of a function (that portion enclosed in braces) is just a special form of compound statement.

For example:

```
nl = 0;
```

is a simple statement. However, the statement:

```
{
   h = h / 2; x0 = x0 + h / 2; y0 = y0 + h / 2;
   x = x0 + i * 32; y = y0 + 10; u = x; v = y;
   ++i; p( 1, i );
}
```

is a compound statement.

Data Representation - Constants

Numbers and characters must be entered in your C program in certain ways in order for the compiler to understand them properly. A fixed value to be used in a C expression is called a CONSTANT.

Where an integer number is required, you enter it just as you would write it. A leading zero indicates that the constant is in a base other than decimal. A leading zero followed by a string of digits indicates an OCTAL CONSTANT. A leading zero followed by 'X' or 'x' indicates that a hexadecimal constant follows. Thus, the decimal number, 255, can be represented as 0377 or 0xFF, as desired. A long integer constant should be terminated with the uppercase letter "L" or the lowercase letter "l" as in 1234567L or 0x2a009105L; however, an integer value greater than 65535 will be considered a long. Floating point constants have the syntax of an optional sign, followed by a string of decimal digits possibly containing a decimal point, an optional exponent field containing an 'e' or 'E' followed by an optionally signed decimal integer.

If the variable to be assigned the constant is not big enough to contain the constant, only the least significant bits (LSB) of the number are stored. This, in effect, is storing the remainder of dividing the constant by 256, 65536 or 4,294,967,296 depending on the variable size. No warning is given when this happens (except in the case of floating point overflow errors which can be trapped by the program), so the programmer must be sure that the variable can hold the number.

CHARACTER CONSTANTS supply a way to specify the code for a character that does not depend on any particular character set. A character constant is a list of characters within single quotes (apostrophes). For instance, the character constant 'A' is stored in the computer as the number 65 (in decimal). Again, it is up to the programmer to assure that the number of characters between apostrophes can fit into the variable being assigned. If more characters are specified than can fit, only the last one or two (as needed) are used.

When a sequence of characters is needed, a STRING can be specified by enclosing the characters between quotes (sometimes called "double" quotes - i.e. "This is a string"). C does not place all of these characters into a variable but rather uses the ADDRESS of the first character of the string. Thus, when the string, "testing, 1 2 3", is used in a C program, the characters between quotes are stored in memory, and the address of the first 't' is used in the expression where the string was specified. You can say that the number generated by C to represent the string really POINTS to the string. The subject of POINTER variables, which are handy for manipulating strings, will be discussed later.

There are certain control characters that are needed frequently in programs, but which differ from machine to machine. These can be represented in C programs using ESCAPE SEQUENCES, to provide a machine-independent constant. The backslash character, "\", is called the ESCAPE CHARACTER and denotes the beginning of an escape sequence. A letter following the escape character indicates which control code is being specified. Also, certain characters that would otherwise be difficult to represent in strings and character constants are generated by following the backslash with the character. These escape sequences are shown in the following table:

Escape Sequence	**Control Code**	**ASCII code used by C**
\n,\N	NEWLINE character	x'0D' CR
\t,\T	Horizontal tab	x'09' HT
\b,\B	Backspace	x'08' BS
\r,\R	Carriage return	x'0D' CR
\f,\F	Form feed	x'0C' FF
\\\\	Backslash	x'5C' backslash
\\'	Single quote	x'27' apostrophe
\0	Null	x'00' null byte
\\"	Double quote	x'22' double quote

For example, the character 'A' can be represented as '\x41' using a hexadecimal escape sequence, or as '\101' in an octal constant. Similarly, to place a carriage return at the end of a line, the following three methods could be used; however, the first is preferred:

```
   "An example of a normal escape: \n"
   "An example of a hexadecimal escape: \x0D"
   "An example of an octal escape: \015"
```

When a character escape sequence is used within a string, the actual value of the escape sequence is stored in a string (i.e., only one byte of data per escape). Thus, the string:

```
   "\n\x0d\015"
```

is only three bytes long in memory once the program is compiled and assembled.

Variable Names (Identifiers)

The names given to identify variables, functions, macros, and labels are called "identifiers" and all follow the same rules as to their format. C identifiers may be of any length (be practical) and must start with an alphabetic character ['A' through 'Z', 'a' through 'z'] or the underline ['_'], with the rest of the characters in the name consisting of upper-case or lower-case alphabetic characters ['A' through 'Z', 'a' through 'z'], numeric characters [0 through 9], or the underline character ['_'].

MC keeps all the characters of your identifier as significant; however, if the identifier is to be used as an extern, only the first seven (7) characters of an "extern" identifier will be used by the assembler and linker, so these first seven must be unique. Also, the assembler you use may limit the length of symbol names.

C is case-sensitive, i.e., recognizes the difference between lower-case and upper-case in identifiers. Thus, "EOF", "eof", and "Eof" are all different identifiers to C. However, identifiers which must be written out in assembler source code are converted to upper-case, since assemblers, in general, do not allow lower case assembly language code. A good, simple rule to follow is to use UPPER-case for macro constants only, and lower-case for all other identifiers. Since macro identifiers are not written to the assembly output file, they will not conflict with any other identifiers, which are the same, except for case differences.

The C language reserves certain "words" which it uses as keywords. These key words can not be used as identifiers. The list of reserved words is:

auto	**Break**	**case**	**char**	**continue**	**default**
do	**Double**	**else**	**entry**	**enum**	**extern**
float	**For**	**goto**	**if**	**int**	**long**
register	**Return**	**short**	**sizeof**	**static**	**struct**
switch	**Typedef**	**union**	**unsigned**	**void**	**while**

Data Declarations

C variables must always be declared before use. The standard procedure is to declare variables at the beginning of the program (globals) and at the beginning of each function (locals). Locals must be declared before any executable statements. MC supports the following variable types and adjectives:

char	- an 8-bit unsigned character
int	- a 16-bit signed integer
float	- a 32-bit floating point
double	- a 64-bit floating point
struct	- the specifier of a structure
union	- the specifier of a union
typedef	- the operand of a typedef
short	- usually applied to ints but ignored by MC
long	- used to specify 32-bit integers
unsigned	- applied to ints or long ints

Type char

Character variables are stored in eight bits, or a byte. MC always treats a char as unsigned. The declaration:

```
char c, string[81];
```

establishes a character variable named "c" and a singly dimensioned character array named "string" which can hold a string of maximum length equal to 80 characters. Arrays of one or more dimensions are allowed.

Type int [and short int]

Integer variables as well as short integer variables are stored in sixteen bits. The short declaration is provided in the interest of portability. Make no assumptions about the storage size of pointers. Although a pointer may be stored in either 16 bits or 32 bits, a pointer is not an int! The declarations:

```
int a;
short b;
short int b2;
```

are all acceptable declarations, and all result in the same size integer field. This is acceptable, since the C language does not guarantee that a "short" will be shorter than integers. Integers declared in this manner are signed, i.e., their most significant bit is regarded as a sign bit. Their values can range from -32,768 to 32,767 (decimal). Unsigned fields do not have a sign bit. They range from 0 to 65,535 (decimal) and are declared like this:

```
unsigned u;
unsigned int u2;
```

Type long int

Long integer variables are stored in thirty-two (32) bits. The declarations:

```
long int number;
long datasize;
```

are both acceptable declarations, and each result in a 32-bit integer field. Long integers declared in this manner are signed, i.e., their most significant bit is regarded as a sign bit. The values can range from -2,147,483,648 to 2,147,483,647 (decimal). Unsigned fields do not have a sign bit. They range from 0 to 4,294,967,296 (decimal) and are declared like this:

```
unsigned long lu;
```

```
   unsigned long int lu2;
```

Type float and type double

Float and double floating point variables are stored in thirty-two (32) and sixty-four (64) bits respectively. The declarations:

```
   float fvalue;
   double dvar;
```

are acceptable declarations; the first results in a 32-bit floating point field and the second results in a 64-bit double precision floating point field. Floating point variables are always signed. Their value varies from approximately -1.7e+38 through +1.7e+38. Floats maintain about 6-7 digits of precision while doubles maintain about 15-16 digits of precision. The only direct operations which may be performed on float and double variables are addition, subtraction, multiplication, division, comparison, logical not, negation, increment, decrement, and address_of. All others are illegal.

It is important to note that per the specifications in K&R pertaining to arguments of a function, "C converts all float actual parameters to double, so formal parameters declared float have their declaration adjusted to read double."

Type struct structure_tag

A structure is a method of collecting one or more variables into one grouping using a single name. Where more than one variable is grouped together, they may be of the same or of different types. The grouping is commonly conceptualized as a "record". A structure template is declared with the syntax:

```
   struct structure_tag {
      type_1 member_1;
      type_2 member_2;
      type_n member_n;
   };
```

The structure_tag is optional and is generally used when many structures will be using that structure template.

The closing brace of the structure template may be followed by a variable or list of variables, just like you can do for any of the types noted above (char, int, long, float, double). When no variable follows the closing brace, the structure template remains just a template and no space is reserved for the members. When a variable list does follow the closing brace, adequate storage for all of the structure's members is reserved for each element in that variable list. For example, the following declaration is a structure template, which has two members, one of which is an array:

```
   struct keyword {
      char *name;
      int index[2];
   };
```

The member "name" is declared as a pointer_to_char in this structure declaration while the member "index" is an integer array of length two. Since this is only a template, no storage is reserved.

The above structure template may be assigned to a structure by another structure declaration such as:

```
   struct keyword primary;
```

which declares "primary" to be a type "struct keyword". Where a structure definition is only needed in a single module, it may be declared directly without the structure tag. For example:

```
struct {
   int hours, minutes, seconds;
} clock;
```

declares a structure named "clock" which contains three ints.

Each element of the structure is termed a member. A member can only be accessed as part of the structure. For instance, in the "primary" structure defined above, the syntax "primary.name" refers to the member, "name". Likewise, the second element of the index array member is referenced by the syntax, "primary.index[1]".

A variable may be typed as an array of structures (not to be confused with an array member of a structure). Using the struct keyword template illustrated above, we can declare an array of type struct keyword with:

```
struct keyword speech[10];
```

This declares "speech" to be an array of type struct keyword. The syntax for referencing an element of the structure is to bind the structure subscript to the name of the structure. For example, the "name" and "index[0] members of the fifth structure of "speech" would be referenced with:

```
speech[4].name
speech[4].index[0]
```

A variable may be declared as a pointer_to_struct. Again following our example, if "ps" is declared via "struct keyword *ps;", then the above members would be referenced via the syntax:

```
(*ps).name
(*ps).index[1]
```

Since the structure dot operator has a higher precedence than the indirection operator, the parentheses are needed. This somewhat kludgy syntax can be replaced with the structure indirection shorthand using the "->" operator. The structure indirection operator is composed of a minus sign followed by a right angle bracket. The above two references would now be written as:

```
ps->name
ps->index[1]
```

A pointer_to_structure can be passed as an argument to a function; a structure cannot! Likewise, a function can return a pointer_to_structure but cannot return a structure. Another limitation of structures is that a member of a structure cannot be a structure of the same type (think about that); however, it can be a pointer to its type. This does not restrict members from being other structure types.

Type union union_tag

A union provides a technique for accessing elements of a record in different ways at different times. A union is declared similarly to a structure. Also, you can only access a member or take the address of a union. The members of a union all have a zero offset from the union's origin. The actual amount of memory space assigned is the space required by its largest member. The union declaration is somewhat similar to the EQUIVALENCE statement of FORTRAN.

An interesting example of a union is found in the Z80REGS header file. In this union named "REGS", two members are declared - each a structure. One member is the structure named "wordregs" and the other is a structure named "byteregs". The storage space for "byteregs" completely overlaps the storage for "wordregs" thereby providing you with a convenient method of accessing the low-order or high-order register of a 16-bit register pair. This example has a union with two structure members. Alternatively, a union could be a member of a structure.

Type void

The "void" type is used to declare a function that has no return value. It is beneficial to type "void" such functions because an attempt to use the value returned by a void function will be flagged as an error.

Arrays

Arrays of one or more dimensions are allowed for short, int, long, unsigned, float, double, or pointer types, as well as for structures and unions. An array is denoted by appending the dimension enclosed in square brackets to the identifier. For example, arrays of one, two, and three dimensions could be declared like this:

```
char buffer[81];
double grid[25][25];
char bit_plane[8][24][80];
```

The first defines a character buffer of 81 elements. The second defines a two-dimension array of doubles having 625 elements. The third defines bit_plane to be a three-dimensional character array - 8 planes of 24 rows by 80 columns.

Pointers

Pointers may be declared for any data type. Pointer variables are different from the types described so far, in that they normally contain the ADDRESS of a data item. For example,

```
char *cp;
```

declares a pointer_to_char variable named "cp". The asterisk denotes INDIRECTION, i.e. that the data item is referred to indirectly through the pointer variable "cp". The address of the data item must be stored in the variable, "cp", before it is used as a pointer to access a data item. To refer to the data itself, an asterisk is placed before the name, e.g., *cp denotes the data item. An example of practical use follows:

```
getit(cp)
   char *cp;
{ int c;
   while ((c=getchar()) != EOL && c != EOF)
      *cp++ = c;
   *cp=NULL;
   return c;
}
```

The function, getit(), inputs characters continually from the standard input until end-of-file or end-of-line characters are encountered. When getit() is called, the pointer argument, cp, contains the address of a buffer area. One by one the characters are placed in the buffer, (*cp++ = c), and the buffer pointer is incremented by the post increment operator (++).

A simple peek() and poke() set of functions can easily be written using pointers. Witness the following two functions:

```
int peek(s) char *s;
{ return *s; }
void poke(s,c) char *s; int c;
{ *s = c; }
```

When the actual argument to a function is an array, the formal argument declaration in the function may be made in two ways. It may be declared as a pointer or an array without the size in the array declaration. For example,

```
funk(arg1,arg2)
int arg1; char arg2[];
```

declares a character array, "arg2", of an unknown number of elements. This could also have been declared as:

```
   funk(arg1,arg2)
   int arg1; char *arg2;
```

to define arg2 as a pointer to a char. Pointers may be INDEXED to get to the "nth" item in an array. Using the example above, arg2 would contain the address of the beginning of an array of characters. "arg2[0]" denotes the first element in the array, and "arg2[22]" denotes the 23rd element.

No matter how a pointer is declared, either method of using the pointer may be employed as the programmer sees fit. Thus, "*arg2" and "arg2[0]" refer to the same data item and may be used interchangeably in the same program. Using "*arg2" is a little more efficient, however.

The array declaration without the number of elements within the brackets is allowed only in external declarations and in argument declarations. Statics, globals, and locals declared this way are illegal.

Pointers may point to other pointers. This bombshell of a statement is probably too much for you, after the last few paragraphs. It must be said, however. C allows pointers to have more than one LEVEL OF INDIRECTION. This can be declared several ways:

```
   shine()
   {  char *names[];
      char *(*words);
      . . . . . . . . . . . . .
```

Both of these declarations result in the same effect: a pointer that points to a pointer which points to a character field.

Pointer variables may have up to 255 levels of indirection. However, the practical limit is the ability of the programmer to keep track of all this. In general, two levels of indirection are all most folks can take.

The operations that may be performed on pointers are severely restricted. The allowed operations are:

(a) Pointer + offset or offset + pointer, offset is scaled.
(b) Pointer - offset, offset is scaled.
(c) Pointer - pointer, result is scaled.
(d) Pointer = expression.
(e) ++, --, logical not, ||, &&, and comparisons.

Initialization of Variables

C allows you to initialize variables within the declaration statement. For instance, the declarative,

```
   int var = 100;
```

declares "var" to be of type "int" with an initial value of 100. A pointer to a string may be declared and initialized with the format,

```
   char *pstring = "This is an initialized string pointer";
```

You may declare a series of variables of the same type as well as their initializations with the format,

```
   float f1=1.0, f2=10.0, f3=100.0, f4=1000.0, f5=10000.0;
```

An array can be initialized by placing the initialization values within braces. For example,

```
   int table[5] = {1, 10, 100, 1000, 10000};
```

declares table to be an integer array of dimension 5. Table[0] is initialized to the value 1, table[1] to 10, table[2] to 100, table[3] to 1000, and table[4] to 10000.

Structures may also be initialized. For instance, the format,

```
   struct { char *name; } dow[5] = {"Mon","Tue","Wed","Thu","Fri"};
```

defines the array of structures, "dow", consisting of one element, "*name". The array is of dimension 5, with each array element constituting the structure. Each structure element is initialized as a pointer to the corresponding 3-character day name. Thus, "puts(dow[0].name);" would print, "Mon".

Note that automatic structures and automatic arrays (these are considered "aggregates" by K&R) may not be initialized. You also cannot initialize a union.

Scope of Variables and Functions

Variables or functions that are declared outside of any function, i.e., which are not parameters to functions or declared within braces are called "external". They are external to all functions. External variables and functions can be accessed only from any of the functions subsequently defined within the module being compiled. They are usually used to create static variables that are to be accessed only by a function or group of functions. However, by using the "extern" statement in a separately compiled module, say module_B, an external variable or function of module_A may be accessed from that separately compiled module_B. Please do not confuse "extern" and external. External variables and functions are declared without the "extern" statement strictly by their position exterior to any function or compound statement.

Variables declared within a function are called "local". Functions may not be defined within another function, as is the case with the Pascal language. However, a function may be DECLARED "extern" so that it may be accessed within the currently defined function. Local variables may not be accessed from any other functions. They only exist for the function in which they are declared. Even within the function, a local variable can only be accessed in the block in which it is declared. Remember that a block is a section of code contained within a matching pair of braces.

Local variables can have the same name as external variables, or local variables declared in different blocks. If a local variable has the same name as an external variable then the local variable is the one accessed when used within the local block. In the following example:

```
   int same;   /* this is an external variable */
   funk(same)
   {  return same; } /* return local copy */

   hunk()
   {
      if (block_1)
      {  int same;
         /* some code could go here */
      }
      else
      {  char same;
         /* some other code here */
      }
   }
```

every declaration of "same" was a unique variable. Although legal, the declaration of local variables with the same name within the same function is not recommended. This type of trickery, as shown in hunk(), needlessly causes confusion and is easily avoided.

Storage Classes

Variables and functions may be declared as being in certain classes. These classes specify where variables are to be stored. The classes available in C are: auto, static, extern, register, and typedef. The storage class of an object is specified by placing the class name in front of the normal declaration:

```
auto char c;
static int ai[20][80];
```

Storage Class - auto

Variables that are declared “auto” are stored on the stack. This is the default for variables declared within a function, so the "auto" keyword may therefore be omitted. Local variables that are “auto” are created afresh each time the function in which they are declared is called. This allows functions to be reentrant and recursive. Functions may not be declared with class "auto" since a function must be declared outside of any other function. As K&R say, the C compiler is incapable of compiling code onto the stack!

The scope of an auto variable is the block (within braces) in which it is declared. All other portions of the code being compiled are oblivious to the existence of the auto variable, and in fact, there may exist other variables with the same name.

The auto class is illegal for functions and other external definitions (any variables declared outside of a function). In terms of speed of access, auto variables are accessed the slowest; thus, if timing is important and your program does not require recursion, use register or static variables.

Storage Class - Register

Variables declared in the register class are treated similarly to auto variables by MC. The number of register variables permitted depends on the number of extra machine registers available for use. MC makes use of the two index registers of the Z-80 (IX and IY) which can be used for ints and pointers. Register variables are usually accessed faster than autos but not as fast as statics. Any register variables declared in excess of two are stored on the stack in the same manner as an auto and are also illegal outside of a function; however, each function is permitted up to two register variables.

The scope of register variables is the same as that for auto variables. The formal arguments of a function may also be declared register. Note that the "address_of" operator may not be applied to register variables.

Storage Class - Extern

The "extern" storage class allows an external variable declared in one module to be accessed from another module. A "module" is what is processed by one invocation of MC, i.e., one set of C source input. Let's say that the following declaration:

```
int choice;
```

exists in module 1. If module 2 functions need to access this same variable, the declaration:

```
extern int choice;
```

would allow the access needed. C will not reserve any storage for "choice" in module 2, since the storage class, "extern", tells C that storage has been reserved in another module.

The programmer MUST ensure that the declarations are compatible between modules. In other words, all "extern" declarations must match the external declaration (declaration without "extern") by having the same type, size, and amount of indirection. Otherwise, C may access the variable in incorrect ways.

The extern statement may also be used to declare the return value of a function before it is defined in the program. This "forward" declaration allows a function that returns something other than a signed integer to be defined before it is used. If the forward declaration is not given and a function is as-yet-undefined, the compiler assumes that the function returns a signed integer, which may be incorrect for many functions.

Storage Class - Static

Static objects are stored in declared, fixed memory space. Their behavior is the same as that of external variables; their scope is more limited, however. Static variables declared outside of a function can only be accessed by functions within the module being compiled. Other (separately compiled) modules cannot get to them by declaring them "extern". Static variables declared outside of all functions are accessible to all functions subsequently defined within the module. Static variables declared within a function are similar in scope to auto and register variables. They can only be accessed in the block in which they are declared. Thus, two static variables with the same name may be declared in different functions.

Functions may also be defined as "static", making them only accessible from within the current module. However, since MC is a one-pass compiler, the definition of a static function must precede any reference to the static function. This is because the compiler assumes that an as-yet-undefined function is an external function. Alternatively, you may use a forward declaration.

Storage Class - typedef

Typedef is provided in the C language not as a unique storage class, but as a means for creating new data type names. Note that typedef does not create new data types, but rather provides a method for giving special names to existing data types. This facility is useful to create customized names that bear some association to the class of objects being typed. For instance, the typedef statement:

```
   typedef char *POINTER;
```

declares "POINTER" to be a synonym for the type pointer_to_character. Thus, a statement such as the following can be used to clearly denote the meaning of the declarations:

```
   POINTER first, last;
   POINTER table[SIZE];
```

A typedef is somewhat similar to the preprocessor #define. It performs some textual substitution. It is a little more flexible, though, since the actual substitution can be more complex. It has a great use in enhancing the portability across compilers by introducing a single point definition of a variable type rather than having the actual definition scattered throughout many source modules. Thus, where a declaration type must be changed, it only need be changed in one place (usually in a header file).

Storage Class - Defaults

When a variable is declared by only stating the storage class:

```
   auto x1; register x2;
   extern x3; static x3;
```

the variable type is assumed to be "int". This is a perfectly acceptable shorthand way to make integer declarations.

When the declaration of a local (declared within a function) variable or argument has no storage class, C assumes that the variable is an auto variable. A function that is declared within another function body is assumed to have a storage class of external. The compiler regards the declaration as if an "extern" statement preceded it.

External declarations which do not have a storage class declared are special entities. They belong to the implicit class, "external", and may be referenced from other (separately compiled) modules that declare the variable "extern".

Expressions

One of the most powerful features of the C language is its expression capabilities. The amount of work that can be done by one expression is sometimes overwhelming. A quick example:

```
   (end_of_file = (c=getc(file))==EOF)) ? fclose(file) : ++count ;
```

This convoluted statement will get a character from a file and place it in the variable, "c". The character is compared to the value "EOF" which indicates end of file; the result, true or false, is placed in the variable, "end_of_file". Finally, if it was the end of the file, the file is closed. Otherwise, a counter variable, "count", is incremented to provide a count of the characters read.

The example was a bit exaggerated, and expressions this complex can be quite hard to understand. Two statements must be made about the complexity of expressions in the C language.

> The programmer who does not fully know and use C's expression capabilities is seriously handicapped, unable to use the full power of the C language.

On the other hand, a quotation from THE ELEMENTS OF PROGRAMMING STYLE by Kernighan and Plaugher is appropriate:

> "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

The word "maintain" could be substituted for "debug" in the quote above, and it would still be valid. You must be able to understand later what you wrote into your program. If others are going to have to maintain your program, the principle of KISS (Keep It Simple, Stupid) should prevail. This is not intended to discourage the use of complex expressions. Just keep in mind that the more operators involved in an expression, the more difficult it is to properly place parentheses and keep the precedence of operators straight.

There are two kinds of expressions in many computer languages: logical expressions and arithmetic expressions. Logical expressions are usually for comparing things and for making choices. The result of a logical expression is either true or false. Arithmetic expressions result in a number. Usually an assignment to a variable is made to save the result of the arithmetic expression, or it is passed as an argument. In many language implementations, only one type of expression may be used in certain contexts. For instance, the BASIC program statement:

```
   1000 A = ( C <= B )
```

attempts to assign to A the result of the comparison C to B. This is not allowed in many implementations because they are expecting an arithmetic assignment. Even if some BASIC's allow it, it is best not to do this type of assignment, in order to keep programs relatively portable.

Another situation is shown in PASCAL:

```
   IF A := (B < C) THEN BEGIN
```

where the PASCAL compiler expects a boolean expression between IF and THEN. Even if A is a boolean variable this assignment is not allowed in most PASCAL compilers. This is not intended to denigrate PASCAL. There are good reasons why the authors of PASCAL did things this way. However, the C language does not draw distinctions between types of expressions within the context of the program. The distinctions are made in the types of operators instead.

Primary Expressions

The elements that are manipulated by operators in an expression are called primary expressions. The basic elements that make up a primary expression are identifiers, constants, and strings. Identifiers are the names of variables and functions. Function and array identifiers effectively resolve to the address of the function or array, while all other variable identifiers resolve to the contents of the variable. Constants are character or numeric (decimal, hex, octal) values. Strings resolve to a character pointer, which points to the first character of the string.

The operators that C provides for stating primary expressions group left to right. This means that the left-most operator is interpreted first. The five primary operators supplied by C are: isolating parentheses, subscripting, function invocation, structure/union ARROW, and structure/union DOT.

```
   (expression)            isolating parentheses
   p_ex [expression]       subscripting
   p_ex (expression_list)  function invocation
   .    DOT                obtains member of structure/union
   ->   ARROW              obtains member indirectly through
                            structure/union pointer
```

   Note: "p_ex" stands for "primary_expression"

Isolating Parentheses

When the order in which an expression is to be evaluated conflicts with the precedence of operators, the isolating parentheses provide a way around the conflict. The expression within parentheses is evaluated first, before the result of the enclosed expression is used in any expression outside the parentheses. For example, when predicting the percentage of up time for any equipment, the following formula is used:

```
                       MTBF
   availability =  -------------
                   MTBF + MTTR

        MTBF = mean time between failures
        MTTR = mean time to repair
```

When writing this formula into a C expression a conflict occurs because the division operator takes precedence over the addition operator. If the expression is written like this:

```
   up_time = mtbf / mtbf + mttr
```

the result will always be mttr plus one. This is because the division is done before the addition. To avoid this, the expression can be stated as follows:

```
   up_time = mtbf / (mtbf + mttr)
```

to achieve the correct result.

Parentheses can be used on either side of an assignment operator. At the risk of confusing the reader with as-yet-undefined operators, we nevertheless provide an example using pointers. In certain cases during the use of pointer arrays, indirection must be performed before subscripting into the data item. Since subscripting takes precedence over indirection, this kind of expression must be written as follows:

```
   example(arg)
   char *arg[]; /* pointer to a char pointer array */
   {
      /* wrong way - accesses third pointer */
      /* instead of third character. */
```

```
        *arg[3] = 0 ;

        /* right way - zero's the third character of */
        /* first string */

        (*arg)[3] = 0 ;
    }
```

Subscripting

Subscripting is denoted by a subscript in brackets following a primary expression:

```
    primary_expression [subscript]
    primary_expression [subscript_1][subscript_2]
```

If the primary expression is an array name, or a pointer to an array, the subscripted expression returns the element denoted by the value of the subscript. C arrays are subscripted from zero, i.e., the first element in an array is numbered zero. If more than one dimension is specified, the storage of array elements is such that the rightmost subscript varies fastest as elements are accessed in storage order.

Function identifiers may not be subscripted. A primary expression denoting an array of pointers to functions may be subscripted. The primary expression must indicate the size of the object being subscripted (char, int, pointer) or the subscript will produce an error message. For example:

```
    x = 25[3];
```

is invalid.

Function Invocation

A primary expression followed by parentheses will cause the function denoted by the primary expression to be called. Arguments may be passed to the invoked function by placing them in the parentheses, separated by comma's.

Any number of arguments can be passed to the called function. Care must be taken that the number of arguments passed is the number that the function expects. Otherwise, unpredictable behavior may result (certainly not correct behavior). If a variable number of parameters must be passed, then a control indicator must be passed to tell the called function how many arguments there are (for example, the fprintf() and printf() functions in the standard library). Arguments can be any valid C expression, including other function calls. The arguments are evaluated from right to left, i.e., the right-most expression is evaluated first. The programmer should not rely on this order of evaluation since some other implementations of the C language evaluate them left to right. Statements like this one:

```
   funk( arg++, arg2[arg] );
```

will cause different elements of arg2 to be passed to funk() when different C compilers are used. Stay away from this sort of trickery if you can.

. DOT [designate member of structure/union]

The structure/union dot operator is used to specify a member of a structure or union. Thus, if there exists for example, a structure of the name "time" which has members named "hour", "minutes", and "seconds", then the constructs "time.hour", "time.minutes", and "time.seconds" refer to the member objects when used in an expression. Other details concerning structures and unions are provided in the section on data declarations.

-> ARROW [designate member through structure/union pointer]

This structure/union operator is used when you have a pointer to a structure or union and you wish to reference one of its members. For example, if the identifier "pt" has been declared a pointer to the time structure noted above, then the constructs, "pt->hour", "pt->minutes", and "pt->seconds" reference the objects within the structure. The construct of "structure or union pointer -> membername" is equivalent to "(* structure or union pointer).membername" and is essentially a shorthand method of using the indirection operator.

Unary Operators

Unary operators operate on one object (hence the name). If more than one unary operator operates on the same object, the operators are evaluated right to left. The unary operators supplied by C are:

OPERATOR	OBJECT	DESCRIPTION
*****	expression	indirection; means "object at..."
&	lvalue	pointer; means "address of..."
-	expression	negates the expression; "minus expression"
!	expression	logical complement; "not expression"
~	expression	one's complement of expression
++	lvalue	increment and save in lvalue
--	lvalue	decrement and save in lvalue
(typename)	expression	cast the result to "typename"
sizeof	expression	obtain the size of the result of expression in "bytes"
sizeof	(typename)	obtain the size of "typename" in bytes

All unary operators must appear before (prefix) the object, except the increment and decrement operators, which may appear after (postfix) the object. The term "lvalue" means an expression that evaluates to the address of a data element or pointer field. Constants, function identifiers, and array names are not lvalues. The term derives from the observation that "lvalues" are the only expressions allowed on the left side of an assignment expression.

'*' [asterisk - object at]

The indirection operator can only operate on a pointer expression. Its meaning is effectively "object at..." The address contained in the pointer is the address of the object referred to by this type of expression. For example,

```
see_pointer (pointer)
char *pointer; /* a character pointer */
{
   /* first print the address passed in pointer */
   printf("address is: %u ",pointer);

   /* now print the data at that address */
   printf("data is: %u ", *pointer);
}
```

will print both the address (contents of the pointer variable) and the data at that address (result of the indirect expression).

'&' [ampersand - address of]

This unary operator effectively means "address of..." or "pointer to...". It evaluates to the address of the lvalue it precedes.

'-' [minus sign - negation]

When the unary negation operator precedes an expression, the result is the two's complement negative of the value of the expression. When the '-' precedes an unsigned or pointer expression, the one's complement of the value is taken.

'!' [exclamation point - logical NOT]]

The unary logical complement operator, or "not" operator, evaluates to FALSE if the expression is true and to TRUE if the expression is false. FALSE is defined as 0 and any non-zero value is considered to be TRUE. However, all C operators which result in TRUE or FALSE produce one (1) as the value for TRUE. Thus, the least significant bit of the result indicates TRUE or FALSE.

'~' [tilde - bitwise complement]

The one's complement operator inverts every bit in the expression. No regard is given to the type of the expression.

'++', '--' [increment, decrement]

The increment and decrement operators may be used either before (prefix) the operand or after (postfix) the operand. The operand must be an lvalue or lvalue expression. In either case the contents of the lvalue is incremented or decremented and stored back into the lvalue. The difference between prefix and postfix is how the result of the expression is produced. Prefix means that the value after the increment or decrement is the result of the expression. Postfix means that the value returned by the expression is the value before the increment or decrement.

(typename) [type casts]

The "(typename) expression" is used to explicitly force the result of "expression" to be converted to the type specified by the cast. Casts are used when the variable or expression result type does not agree with what may be needed for continued evaluation. For instance, to take the log of an integer number, it is necessary to convert it to a double. Thus, the statement:

```
   dvalue = log( (double) number);
```

converts the value of the integer "number" to a double before placing it on the stack as the argument to the log function. Note that "number" is not it- self changed, but rather the value is typed to conform to the cast. A cast is similar to the verbs: CDBL, CSNG, CINT commonly found in the BASIC language.

sizeof [obtain size of operand in bytes]

It is sometimes useful to be able to use the size of an object in an evaluation without actually knowing the physical size of the storage provided for the object. For instance, since the actual size in storage units of an int may vary from machine to machine, the programmer may need to compute a figure irrespective of the storage element size. The "sizeof" operator provides this facility. Since all sizes are known to the compiler at compile time, "sizeof object" may be used in the same way as a constant. For instance, the statements:

```
   int array[100];
   bytes = sizeof array;
```

assigns to "bytes", the actual quantity of storage elements occupied by "array". "Sizeof" may also be used in the format, "sizeof(typename)". Thus, sizeof(int) and sizeof(long) are valid operations.

There is significant use of the sizeof operator in calculating the size of a structure. Given the structure,

```
   struct functions
   {  char *name;
      double (*func)();
   } builtins[] =
   {
      "sin", sin,
      "cos", cos,
      "tan", tan,
      "asin", asin,
      "acos", acos,
      "atan", atan,
      "exp", exp,
      "log", log,
      0, 0
   };
```

the number of structure entries is simply sizeof builtins / sizeof (struct functions)

Binary Operators

Binary operators act upon two expressions together. The type of the result depends on the type of the two expressions. If both types are not identical, the lower type will be promoted to the higher type. The order from lower to higher is: char, int, unsigned int, long, unsigned long, float, double. When the operator is addition or subtraction, if only one expression is a pointer, the result of the expression is a pointer of the same type; if both expressions are pointers, the result is unsigned. These "usual arithmetic conversions" are documented in K&R page 184 (section 6.6) and are reproduced here for convenience:

> "First, any operands of type char or short are converted to int, and any of type float are converted to double. Then, if either operand is double, the other is converted to double and that is the type of the result. Otherwise, if either operator is long, the other operand is converted to long and that is the type of the result. Otherwise, if either operator is unsigned, the other operand is converted to unsigned and that is the type of the result. Else, both operands must be int, and that is the type of the result."

When several binary expressions are concatenated together (without isolating parentheses) the order in which the binary expressions are evaluated depends on the precedence of the operators in the expression. In the expression,

```
a + b * c
```

the evaluation of "b * c" precedes the evaluation of the addition, since multiplication has a higher precedence than addition. The expression is evaluated like this:

```
a + (b * c)
```

As previously described, isolating parentheses can be used to change the order of evaluation. To have the addition performed first, the expression can be written:

```
(a + b) * c
```

Each class of operands is described below in order from the highest precedence to the lowest. When all the operators in a complex expression have the same level of precedence, they are evaluated in a certain order; right to left or left to right. It can be said that a class of operators "group" left to right, or right to left. If the order of evaluation between like operators does not matter, the operator is said to be associative. Here is an example of how the order of evaluation affects an expression:

```
a / b / c / d
```

The division operator is said to group "left to right"; thus, this expression is evaluated a follows:

```
(((a / b) / c) / d)
```

The precedence of binary operators is given in the following table:

PRECEDENCE OF BINARY OPERATORS
(Highest to lowest)

MULTIPLICATIVE OPERATORS	- group left to right
expression * expression	multiplication
expression / expression	division
expression % expression	modulus (remainder)
ADDITIVE OPERATORS	- group left to right
expression + expression	addition
expression - expression	subtraction
SHIFT OPERATORS	- group left to right
expression << expression	shift left
expression >> expression	shift right
RELATIONAL OPERATORS	- group left to right
expression < expression	less than
expression > expression	greater than
expression <= expression	less than or equal to
expression >= expression	greater than or equal to
EQUALITY OPERATORS	- group left to right
expression == expression	equal to
expression != expression	not equal to
BITWISE AND OPERATOR	- associative
expression & expression	bitwise and
BITWISE EXCLUSIVE OR OPERATOR	- associative
expression ^ expression	bitwise exclusive or
BITWISE INCLUSIVE OR OPERATOR	- associative
expression | expression	bitwise inclusive or
LOGICAL AND OPERATOR	- groups left to right
expression && expression	logical and
LOGICAL OR OPERATOR	- groups left to right
expression || expression	logical or
CONDITIONAL OPERATOR	- groups right to left
expression ? expression : expression	
ASSIGNMENT OPERATORS	- group right to left
lvalue = expression	simple assignment
lvalue <op>= expression	compound assignment
(<op> is any binary operator except logical, relational, or conditional operators)	
COMMA OPERATOR	- groups left to right
expression , expression	pair of expressions

'*', '/', '%' [multiplication, division, modulus]

The multiplicative operators take precedence over all other binary operators and group left to right. When the result of integer multiplication overflows an int, or when the result of long integer multiplication overflows a long, the left-most (high-order) bits are truncated. Since integer division is used, the fractional portion of the result is lost. The result of division is always truncated toward zero. The modulus operator returns the value of the remainder in the integer division of the two expressions. Overflow and underflow floating point errors may be trapped in software by the use of a _fltvec() error trap. This is documented in Chapter 5, "Advanced Topics".

'+', '-' [addition, subtraction]

The additive operators result in the addition or subtraction of the two expressions. In subtraction, unsigned subtraction only takes place when both expressions are unsigned. If one of the expressions is a pointer and the other is not, the other value is adjusted to reflect the size of the object pointed to. Thus, if "p" is a pointer, "p + 3" returns the address of the fourth object pointed to by "p". If p points to integers, then C automatically doubles the offset to account for the two-byte elements. Likewise, if p points to long integers or to doubles, the offset is appropriately adjusted to account for the size of the object.

'<<', '>>' [shift_left, shift_right]

The shift operators shift the left-hand expression by the number of bits indicated in the right-hand expression. Zeroes are shifted in to replace the bits shifted out. The right-hand expression is always considered unsigned; thus a negative integer will be interpreted as a large unsigned integer. If the right hand expression has a zero value, no shifting takes place. If the right-hand expression contains a value that is larger than the size of the left hand object, the result is always zero.

The shift operator is a logical shift, not an arithmetic shift as in some other C implementations. Thus shifting a signed value will not preserve the sign if the shifted value is negative.

'<', '>', '<=', '>=' [less, greater, less_or_equal, greater_or_equal]

Relational operators result in a TRUE (1) or FALSE (0) value, depending on the indicated condition.

'==', '!=' [equal_to, not_equal_to]

The equality operators, "equal" and "not equal", respectively also return TRUE (1) or FALSE (0) depending on the two expressions' equality.

'&' [bitwise AND]

The bitwise AND operator does a bitwise AND with the two expressions. Each bit position in the result will be set to be one if and only if both corresponding bits in the expressions are equal to one. This is useful for isolating individual bits within a word by using a "mask" as one of the expressions. Any bit in the mask which is set to zero will cause that bit in the result to be zero. Any bit set to one will cause the bit in the other expression to remain the same.

'^' [bitwise XOR]

The bitwise exclusive OR operator. Each bit in the result of an exclusive OR is set only if the corresponding bits in the expressions are opposite, i.e., 1 and 0, or 0 and 1. If they are the same, that bit in the result will be zero. This can be used to complement bits, using a "mask" expression. Any bit which is 1 in the mask will cause the corresponding bit of the other expression to be complemented in the result. Any mask bit which is 0 will pass the corresponding bit unchanged into the result.

'|' [bitwise OR]

The bitwise inclusive OR operator. Each bit in the result will be set to 1 if either of the corresponding bits in the expressions are equal to 1. This can be used to set any particular bit in an expression to one by using a "mask" expression. If a bit in the mask is equal to 1, then the corresponding bit in the result will be set to 1. If a bit in the mask is equal to 0, then that bit in the result will be the same as in the expression being manipulated.

'&&' [logical AND]

The logical AND operator results in a TRUE (one) or FALSE (zero) condition, depending on the relationship of the two expressions. The result is TRUE only if both expressions are true (non-zero). Moreover, if the first expression is FALSE, the second is never evaluated.

'||' [logical OR]

The logical OR operator returns a TRUE (1) result if either of the expressions is true (non-zero). If the first expression is true, the second expression is not evaluated.

'? :' [conditional]

The conditional operator gives the C expression repertoire the equivalent of an if-then-else construct. It can technically be classified as a binary operator since only one of the last two expressions is evaluated. The first expression is evaluated as true (non-zero) or false (zero). Then, if the first expression was true (non-zero), the second expression is evaluated as the result of the expression. Otherwise, if the first expression was false (zero), the third expression is evaluated for the result. The conditional operator groups right to left:

```
   a ? b : c ? d ? e : f : g
```

is evaluated in the following manner:

```
   a ? b : (c ? (d ? e : f) : g)
```

Either or both of the second and third expressions can contain conditional expressions.

'=' [assignment]

The simple assignment, '=', places the result of the right-hand expression unchanged into the object denoted by the left-hand expression.

'+=', '-=', '*=', '/=', '%=', '<<=', '>>=', '&=', '^=', '|=' [assignment]

The compound assignment operators place the result of the right-hand expression into the object denoted by the left-hand expression, after performing the indicated operation with the contents of the lvalue when an assignment operator other than '=' is used. The result is the value stored into the left hand object. The compound assignment operators have the form:

```
   expression_1 <op>= expression_2
```

and is evaluated like this expression:

```
   expression_1 = expression_1 <op> expression_2
```

The first form is more efficient since expression_1 only needs to be evaluated once.

, [comma]

The comma operator sets off a pair of expressions which are evaluated left to right. Both the type and value of the result are the type and value of the rightmost operand; thus, the values of the left operands are discarded. The comma operator usually finds use in a "for" statement to initialize more than one quantity.

Statements

C statements are used to specify the action to be taken by the program. The statements given in the program are normally executed one after the other. Certain statements (conditional and looping statements) will direct the order of and conditions for execution of other statements. Some definitions of statements in the following text require that a substatement be included in the statement. Any place where a substatement is required, there may be one simple statement, or more than one statement combined in a compound statement.

Simple Statements

Simple statements are of three types: expression, declarative, and control. The declarative statements are described fully in the previous sections on functions and variables. The type, size and scope of functions and variables are declared in declarative statements.

A simple statement always ends with a semicolon. The semicolon is the STATEMENT TERMINATOR. It is not a statement separator as in the PASCAL language. It is always required at the end of a simple statement.

Compound Statements

The left and right brace characters, "{"and "}", are used to indicate the beginning and end (respectively) of a compound statement. A compound statement, also called a block, can be used anywhere that a simple statement may be used. Thus, wherever C's syntax requires a statement, more than one statement may be given by enclosing them in braces. Within the compound statement there may be any combination of simple and compound statements.

The compound statement has the format:

```
   {   <declarations>  <statements>    }
```

No declarations or statements are required, although an empty block could be used as a null statement. The declarations must appear before any executable statements. Any of the statements may also be other compound statements. No semicolon is required after the compound statement.

The only place where a compound statement is required instead of a simple statement is in the body of the switch-case statement. The body of a function is one compound statement. Here are some examples of compound statements:

```
   func()
   {
      /* the body compound statement */
      a=b;      /* simple statement */
      if (a>c)
      {
         /* another compound statement */
         c=a;
         b=a;
      }    /* end of compound statement */
      return a+b+c;
   }    /* end of function body compound statement */
```

Null Statement

A null statement is sort of a placeholder. C requires that a statement be given in certain places. If no action is needed in the place required then the null statement can be used. No action is taken by the null statement.

The null statement is simply a statement terminator (semicolon) by itself, with no preceding statement.

```
null()
{
   /* do-nothing function */
   ;    /* a null statement */
}
```

Expression Statement

A C expression followed by a semicolon is called an expression statement. The expression is performed when it is encountered. C will allow an expression that has no assignment in an expression statement, even if it does nothing. Expression statements are used to assign values to or modify values of variables, or to invoke functions. Some sample uses of expression statements:

```
retcode = call_function() ;   /* call a function */
a = b = c = 0 ;               /* make a, b, c equal to 0 */
++counter ;                   /* increment a counter */
```

If Statement

```
 ________________________________________
|                                        |
|   if (expression) statement            |
|                                        |
|   if (expression) statement            |
|   else statement                       |
|________________________________________|
```

The "if" statement gives the programmer the capability to decide whether a statement will be executed. The criterion for the decision is the result of evaluating the expression. The expression may be any valid C expression. If the expression evaluates to TRUE (non-zero), then the statement is executed. If the expression evaluates to FALSE (zero), then the statement following the "else" (if any) is executed.

"If" statements may be nested, i.e., the statement within an "if" statement may be another "if" statement. Too much nesting of "if" statements can be hard to follow, so moderation is advised.

Unless clearly established otherwise by the use of braces, an "else" will bind to the closest previous "if".

Some examples of "if" statements:

```
if (x < 0)  x = -x ;          /* absolute value of x */

if (i<=0) {  i=x;   b=a;  }   /* compound statement */
else --i;                     /* and an else clause */

if (past_twelve)              /* nested if statements */
   if (before_six)
      say("good afternoon");
   else
      say("good evening");
else say("good morning");
```

Switch-Case Statement

```
 ____________________________________________________________
|                                                            |
|   switch (expression) { <switch_statement> ... }           |
|                                                            |
|   switch_statement  =  statement                           |
|                        case constant_expression :          |
|                        default :                           |
|____________________________________________________________|
```

The switch-case statement allows program execution within the <switch_ statement> to be determined by the "case" and "default" prefixes. The expression in the switch statement is evaluated first and converted to a short integer. Then, if any statements precede the first prefix, they are executed. Finally, if any of the constant_expressions match the result of the switch expression, execution continues immediately past that "case" prefix. If none of the cases match the result and there is a "default" prefix, then execution continues at the "default" prefix. Execution will continue from then on until either the end of the switch-case statement is reached or until a "break" statement is encountered. Otherwise, when no matching case is found, no further statements in the switch_statement are executed.

The switch-case statement MUST have a compound statement as its substatement. This is the only case where this is true. The "default" and "case" statements may occur in any order within the body of the switch-case. Local declarations are acceptable preceding the first prefix or statement. The "break" statement is used to exit the switch_case statement.

```
   switch (month) {
      case january:  case october:  case december:  case july:
      case august:   case march:    case may:
         days = 31; break;
      case september:  case april:  case june:  case november:
         days = 30; break;
      case february:
         days = leap_year ? 29 : 28; break;
      default: days = 0; error = true;
   }
```

While Statement

```
 __________________________________________
|                                          |
|   while (expression) statement;          |
|__________________________________________|
```

The most basic form of looping is provided in C by the "while" statement. Simply stated, while the expression results in a TRUE (non-zero) value, "statement" (also called substatement) is executed. The expression is reevaluated before each time the substatement is executed. Therefore, the substatement may be executed from zero to any number of times depending on the expression's current value.

If more than one simple statement must be placed in the loop substatement, then the substatement must be a compound statement. The "break" statement can be used to exit the loop from within the statement. The "continue" statement can be used to continue directly on to reevaluate the test expression, skipping the rest of the substatement, from anywhere within the statement.

```
   while (driving)   watch(the_road);

   while (jogging)
   {
      take(a_step);
      breathe();
      if (too_tired)  break;
   }
```

Do Statement

```
 __________________________________________
|                                          |
|   do  statement  while (expression) ;    |
|__________________________________________|
```

"Do" differs in only one way from the "while" statement - the expression is reevaluated after the statement is executed. Therefore, the substatement will always be executed at least once. The substatement will be repeatedly executed until the expression evaluates to FALSE.

```
   do  anything();
   while (there_is_still_time) ;

   /*  shuffle routine  */
   do
   {
      cut_the_cards();
      shuffle();
   }
   while ( ! ready_to_deal ) ;
```

For Statement

```
 __________________________________________
|                                          |
|   for (expr_1;expr_2;expr_3) statement   |
|__________________________________________|
```

The "for" statement is a looping statement which provides a convenient place for initializing, testing, and incrementing loop control variables. The format shown above can be rewritten using the while statement:

```
   expr_1 ;
   while ( expr_2 )
   {
      statement
      expr_3 ;
   }
```

Expr_1 is evaluated once before the loop is entered. The test expression, expr_2, is reevaluated before each execution of the substatement. If it results in a FALSE (zero) value, the loop is not executed and execution continues to the next statement. Expr_3 is reevaluated after each time the substatement is executed.

Both expr_1 and expr_3 can be more than one expression, separated by commas. Expr_2 can only be one expression and should result in a logical TRUE or FALSE value. Note that all three expressions may be omitted. If expr_2 is omitted, the test is always TRUE.

```
   for ( c = 'A' ; c <= 'Z' ; ++c )
      putchar(c);    /* print the letter */
   /*  "Now I've said my ABC's ..."  */
```

Break Statement

```
 __________________________________________
|                                          |
|   break;                                 |
|__________________________________________|
```

"Break" is used to exit any "while", "do", or "for" loop and to exit the body of a switch-case statement. Whenever a "break" statement is encountered, execution immediately goes to the next statement past the loop or switch-case statement. "Break" is ignored outside of any loop or switch-case compound substatement. For an example of the use of "break" in a switch statement, see the section on switch-case above.

```
   strscan(c,s)
      char c,*s;
   {
      /*  find character c in string s  */
      while (*s != c)
      {
         if (*s == NULL)  break ;  /* end of string */
         ++s ;                     /* next character */
      }
      return s ;
   }
```

Continue Statement

```
 __________________________________________
|                                          |
|   continue;                              |
|__________________________________________|
```

The "continue" statement is used to skip the remaining statements in a compound loop substatement. In a "while" or "do" statement, execution continues at the test expression reevaluation. In a "for" statement, execution continues at the reinitializing expression (the third expression). The "continue" statement is ignored outside of any loop statement.

```
   /*  convert to lower case  */
   while ((c = getchar()) != EOF)
   {
      if ( c < 'A' || c > 'Z' )  /* not an uppercase character */
         continue ;             /* doesn't apply */
      c = tolower(c) ;
      putchar(c) ;
   }
```

Return Statement

```
 __________________________________________
|                                          |
|   return;                                |
|   return expression;                     |
|__________________________________________|
```

The "return" statement causes the currently executing function to end. If an expression is provided, then the result of the expression is returned as the value of the function. The returned value is undefined if no expression is provided in the "return" statement. The "return" statement is not required in order to effect a return from a function. When no statements are left (the bottom of the function body is reached), the function automatically returns as if a "return" statement with no expression were encountered. A "return" statement is needed when a value must be returned or when the return must take place before the end of the function. The result is automatically converted to the function type, if necessary.

```
   square (num)
   /*  square a number  */
   {
      return  num * num ;
   }

   getline(buf)
      char buf[] ;  /* line input buffer */
   {
      /* check for a valid file pointer */
      if (file_pointer == NULL)
      {
         buf[0] = '\0' ;  /* put a null string in buf */
         return ;         /* back to caller */
      }
      fgets(buf, bufsize, file_pointer);
   }
```

Goto Statement

```
 ___________________________________________
|                                           |
|   goto label;                             |
|___________________________________________|
```

The "goto" statement causes an unconditional branch to the statement identified by label. The labeled statement must be contained in the current function. It is illegal to attempt a "goto" to a statement in some other function. An attempt to do so will probably result in an error during the assembly phase. The following example illustrates the use of "goto" (note: it is strongly recommended that you avoid the use of the "goto" statement):

```
   rest(time)
      int time;
   {
      if ( time > 2300 ) goto sleep;
      else return;

      sleep: for ( ; ; )
         ;
   }
```

Labeled Statement

```
 ___________________________________________
|                                           |
|   label: statement;                       |
|___________________________________________|
```

Any executable statement can be prefixed with a label. This construction is usually used to target the argument of a "goto" statement. The format of a label is a valid identifier followed by a colon. The following are labeled statements:

```
   calculate: i += 10;
   bigblock:  i = j = k = l = m = n = o = p = 0;
```

MC Preprocessor

Part of the MC compiler is a preprocessor, which provides for macro substitution, conditional compilation, inclusion of additional files, and line number control. The features provided in the preprocessor extend the power of the C language by (1) controlling the flow of C source through the compilation process; (2) permitting powerful macro substitution of textual strings; and (3) offering string replacement making your program not only more readable, but easier to maintain, as well.

Some of the work performed by the preprocessor has to do with massaging the source text. In this capacity, the preprocessor provides you with the following features:

> Input lines may be terminated with a backslash "\" for continuation; this is most useful for complex expressions, function calls, and macro definitions which may take more than one line of input for their declaration.
>
> Constructions of the form "string_1"<whitespace>"string_2" are replaced with "string_1string_2". This allows substitution of macros within strings, if the macro itself is defined as a string. Restriction: the whitespace separating the strings may include an escaped (backslashed) newline but not a bare newline.
>
> Unneeded whitespace (spaces, tabs, new-lines, comments) is automatically removed; thus you have complete freedom in inserting whitespace throughout your source code for the purpose of readability.

A source text communicates to the preprocessor via directives that control various aspects of the compiler during the compilation process. One of these directives, "#include filespec", you will quickly become familiar with. All preprocessor directives begin with a number-sign character, "#". The directives supported by the MC preprocessor are:

#define	identifier text
#define	identifier(parmlist) text
#undef	identifier
#if	constant_expression
#ifdef	identifier
#ifndef	identifier
#else	
#endif	
#include	filespec

In addition to the preprocessor directives, others are used in the compilation and assembly processes. The following directives are passed through the preprocessor to downstream processes:

#option	identifier	value
#asm		
#endasm		
#line	linenumber	filespec

Simple Symbolic Substitution [#define]

The "#define" directive is a macro definition. It's syntax is:

```
 ______________________________________________________________
|                                                              |
|   #define macname text                                       |
|                                                              |
|   macname    - is the symbolic name to be assigned to the    |
|                text replacement string.                      |
|                                                              |
|   text       - is the text to replace "macname" throughout   |
|                the C source file.                            |
|______________________________________________________________|
```

Description

The "#define" directive is a macro definition. It creates a macro identifier called "macname", which is defined to be the string of characters following the "macname" (the "text"). Macro identifiers may have any length (1 - n characters), and any number of macros may be defined, subject to memory availability. Macro text replacement may be nested to a depth of 20 which means that a macro definition may itself invoke a macro, etc. Macros may also be redefined at will.

Macros may be defined in any order, even if they depend on each other; this is because preprocessor command lines are not themselves preprocessed. However, note that circular definitions will cause a fatal error.

The preprocessor will substitute the "text" string wherever "macname" is found in the C source stream except where "macname" appears within single or double quotes. For example, given a definition of the form

```
#define DRAW 1
```

there would not be a substitution in the statement, 'puts("DRAW ME A BATH");' but there would be in the statement, 'box(DRAW,x1,y1,x2,y2);'. It is recommended that macro "macnames" be defined in upper-case characters so that they become distinct when looking at your source code.

The "macname" must be a valid C identifier following all the rules of naming identifiers, whereas the "text" is anything and everything up to end of the logical input line. The "text" is substituted until "macname" is redefined via another "#define" or undefined via the "#undef" directive.

The macro "**dos**" is predefined. One of the macros **dos5** or **dos6** will be predefined based on the DOS under which MC is running. These macros have no replacement text.

The macro "**__DATE__**", when invoked, is replaced with a QUOTED STRING of the asctime() format "Tue Oct 22 10:03:10 1985\n\0", the date and time being retrieved from the system when MC is invoked.

The macro "**__FILE__**", when invoked, is replaced with a QUOTED STRING containing the name of the input file currently being read.

The macro "**__LINE__**", when invoked, is replaced with the number of the source line currently being processed, in decimal. It is NOT a quoted string.

Example

```
#include stdio.h
#define PRINTX printf("%d\n",x)

main()
{
   int x;
```

```
      x = -3 + 4 * 5 - 6; PRINTX;
      x = 3 + 4 % 5 - 6; PRINTX;
   }
```

Parameterized Macro Substitution [#define]

This powerful directive provides textual substitution with the capability of specifying substitution parameters in a parameter list that may be different for each macro invocation.

```
 ______________________________________________________________
|                                                              |
|   #define macname(parmlist) text                             |
|                                                              |
|   macname    - is the symbolic name to be assigned to the    |
|                text replacement string.                      |
|                                                              |
|   parmlist   - is zero or more parameters of the macro.      |
|                                                              |
|   text       - is the text to replace "macname" throughout   |
|                the C source file.                            |
|______________________________________________________________|
```

Description

Macro definitions may also be written to include dummy parameters, which will be substituted with actual parameters when the macro is expanded. This type of definition requires that there not be any white space between the "macname" and the opening parenthesis "(". The actual parameters are designated when the macro is expanded. The preprocessor will substitute the "text" string and the actual parameters passed in the macro call wherever "macname" is found in the C source stream except where it appears within parentheses.

For instance, a macro definition of the library function putchar() may be written as

```
   #define putchar(byte) putc(byte,stdout)
```

Thereafter, any C statement of "putchar(c)" will be replaced with the statement, "putc(c,stdout)". Notice that the actual argument, "c", replaces the dummy parameter, "byte", in the macro definition when the macro is expanded. It is also not necessary for the macro dummy parameter name to be unique across different macros, although it is recommended that it be distinct for the purpose of clarity.

A parameterized macro may have up to 128 parameters - more than you would ever have need for. Macro text replacement may be nested to a depth of 20 which means that a macro definition may itself invoke a macro, etc. Macro identifiers may have any length (1 - n characters), and any number of macros may be defined, subject to memory availability. Macros may also be redefined at will. In a macro call, commas are used to separate actual parameters, but are ignored within set(s) of balanced parentheses; a backslash may be used to escape an imbedded comma or unbalanced parenthesis within an actual parameter.

Macros may be defined in any order, even if they depend on each other; this is because preprocessor command lines are not themselves preprocessed. However, note that circular definitions will cause a fatal error.

Parameters in the definition portion of a parameterized macro are always replaced, even if they occur within a double-quoted string. Note that this can get tricky. For example, in the following series of parameterized macro definitions, when the string, "PR(cos(d))" is detected, the "x" parameter is the string, "cos(d)" resulting in the substituted statement,

```
   printf("cos(d) = %g\t",cos(d))
```

When a parameterized macro is called with one or more actual arguments missing, the missing arguments will be interpreted as if they were null strings. Note that you must still provide any comma(s) needed to specify all of the parameter positions. For instance, the macro "#define test(a,b) a b" called with the various arguments produces the following results:

```
test(123,456)      123      456
test(abc,)         abc
test(,xyz)         xyz
test()             Fatal preprocessor error - missing parameters
```

Examples

```
#define PR(x) printf("x = %g\t",x)
#define NL putchar('\n')
#define PRINT1(x1) PR(x1); NL
#define PRINT2(x1,x2) PR(x1); PRINT1(x2)
```

Forgetting a Macro Name [#undef]

This directive will cause the preprocessor to forget the operand name.

```
#undef  macname

macname    - is the symbolic name to be "forgotten".
```

Description

The "#undef" directive is used to cause the preprocessor to forget the definition of "macname". Thus, after a macro name has been forgotten, no text substitution will be performed if that "macname" is encountered in the C source code. Also, the result of an "#ifdef macname" referencing that macname identifier will be a FALSE state.

Example

```
#ifdef DEBUG
#undef BRIEF
#endif
```

Conditional IF Evaluation [#if]

The conditional "#if expression" is used for conditional compilation based on the result of an expression.

```
#if constant_expression

expression - is a simple or complex expression of the
             form permitted in the C language which will
             evaluate to a constant.
```

Description

The preprocessor will evaluate the "expression", which must evaluate to a constant, and consider the result FALSE if it is a zero value or TRUE if it is a non-zero value. If the expression is TRUE, the C source code up to the corresponding "#endif" or "#else" will be processed; otherwise it will be ignored. Conditionals may be nested 256 deep.

A constant_expression in the preprocessor can involve only integer constants and character constants. The binary operators permitted are: "+", "-", "*",

"/", "%", "&", "|", "^", "<<", ">>", "==", "!=", "<", ">", "<=", ">=", and "?:". The only unary operators permitted are: "-" and "~".

Example

```
   #if MAXLEN > 80        /* ensure external definition does */
   #define MAXLEN 80      /* not exceed our maximum limit    */
   #endif
```

Conditional test of an Identifier [#ifdef, #ifndef]

These directives will test whether a macro name has been defined or undefined. The result will be TRUE or FALSE respectively.

```
 ______________________________________________________________
|                                                              |
|   #ifdef macname                                             |
|   #ifndef macname                                            |
|                                                              |
|   macname    - is the symbolic name to be tested for its     |
|                state - defined or undefined.                 |
|______________________________________________________________|
```

Description

If the identifier noted as "macname" has been previously defined via a "#define macname" and not forgotten via an "#undef macname", the result of the "#ifdef" test will be TRUE and the result of the "#ifndef" test will be FALSE. Otherwise, the result of the "#ifdef" test will be FALSE and the result of the "#ifndef" test will be TRUE. If the test result is TRUE, the C source code up to the corresponding "#endif" or "#else" will be processed; otherwise it will be ignored.

Example

```
   #ifdef  DOS6
   #define MAXLEN  80
   #else
   #define MAXLEN  64
   #endif

   #ifdef  DBUG
   static void dsply();
   #endif
```

Alternate Conditional Block [#else]

The "#else" directive is used to establish an alternate block of C source code which is to be processed depending on the result of a previous conditional analysis.

```
 ______________________________________________________________
|                                                              |
|   #else                                                      |
|______________________________________________________________|
```

Description

The block of source between the "#else" and the "#endif" will be processed and passed to the output stream only when the test state of the preceding "#if", "#ifdef", or "#ifndef" was FALSE. This provides a construct similar to the "if (expression) statement_1; else statement_2;" in the C source language. However, in the case of the pre-processor, you can utilize this construct to conditionally insert alternative groups of C-source code into your program based on a single test. Multiple #else's are not accepted for a given #if; thus, there can be only one "#else" per "#if".

Example

```
#ifdef  DOS6
#define MAXLEN  80
#else
#define MAXLEN  64
#endif
```

Conditional Block Termination [#endif]

The "#endif" directive is used to define the terminating boundary of a "#if", "#ifdef", or "#ifndef" conditional block.

```
 ______________________________________________________________
|                                                              |
|   #endif                                                     |
|______________________________________________________________|
```

Description

The "#endif" must be used to "close" any of the "#if" directives so the preprocessor knows the extent of the conditional block.

Including Additional Files [#include]

This directive permits additional files to be included into the C source stream.

```
 ______________________________________________________________
|                                                              |
|   #include <filespec>                                        |
|   #include "filespec"                                        |
|   #include filespec                                          |
|                                                              |
|   filespec   - is the name of the C source or header file    |
|                to be inserted into the source stream.        |
|______________________________________________________________|
```

Description

This directive tells the C preprocessor to insert the file designated by "filespec" into the source stream being compiled. The "filespec" will default to an extension of "/H" if no extension is given. Includes may be nested 8 files deep. The "#include" is used quite frequently to merge the STDIO standard header file and other header files into your compilation. Note that the preprocessor will accept all three forms of the "#include"; all forms are treated alike.

Since the preprocessor makes use of the MC standard I/O package, it will accept filespecs of the form: "filename.ext" or "filename/ext".

Example

```
/* sample program to illustrate #include */
#include "stdio.h"
main()
{
   int x;
   x = -3 + 4 * 5 - 6;  printf("%d\n",x)
}
```

Forcing Assembler Options [#option]

This directive is passed through to the compiler, which turns it into an assembler instruction.

```
 ______________________________________________________________
|                                                              |
|   #option  optname  value                                    |
|                                                              |
|   optname    - is the symbol name to be assigned.            |
|                                                              |
|   value      - is the optional value to be assigned to       |
|                "optname" by an assembler DEFL instruction.   |
|______________________________________________________________|
```

Description

The "#option" directive is used to pass symbol definitions from the C source code to the assembly phase. The "optname" must be a valid C identifier. The optional "value" must be an integer or character constant expression. Escape sequences may be used in the constant. The compiler translates the "#option" directive to the form:

```
@_OPTNAME DEFL value
```

If the "value" is omitted, the DEFL statement will default to a value of negative one (-1). This indicates TRUE to the assembler.

The #option directive is used in MC to generate a "request library search" assembler directive. This results in an automatic search of a specified library during the linking phase. If your application will be using functions in either the installation library, IN/REL, or the high-level math library, MATH/REL, you will need to add the statement(s):

#option INLIB and/or **#option MATHLIB**

for the installation and high-level math libraries respectively.

MC has reserved additional option names for use with the #option directive. These are:

ARGS	specifies the generation of argc and argv for use by main();
FIXBUFS	specifies buffer pre-allocation for standard I/O;
MAXFILES	specifies the maximum number of simultaneously open files;
REDIRECT	specifies the acceptance of standard I/O redirection;
WILDCARD	specifies wildcard filespec expansion;

The use of these options and "#option" in general is documented in chapter 3, "Running the Compiler"

Including Assembler Source [#asm, #endasm]

This pair of directives permits assembler source code to be placed directly in-line.

```
 ______________________________________________________________
|                                                              |
|   #asm                                                       |
|   transparent assembly language code                         |
|   #endasm                                                    |
|______________________________________________________________|
```

Description

The directive pair, "#asm" - "#endasm", can be used to insert assembly language source code directly within the C source file. It should be used ONLY when it is ABSOLUTELY necessary to write a routine in assembly language. Remember that any C source code file that has imbedded assembly language code is generally NOT portable. The more assembly language code you imbed, the less portable your programs become and the more you have to recode when transporting your program to another machine.

All input past the "#asm" directive is passed unchanged into the output file except for preprocessor commands. Of course, since the output file is an assembly language source file, the statements following the "#asm" should be valid assembly source statements. The block of assembly statements is ended with the "#endasm" directive. Please note that the "#endasm" directive must be in column 1; otherwise the "#endasm" will not be recognized and any C source code following the "#endasm" will be passed uncompiled to the output file.

Preprocessor commands within an "#asm"-"#endasm" construct are valid. This leads to some beneficial side effects. For example, a pure ASM source file may be #include'd, or "#if/#else/#endif" may be used to conditionally pass assembler source to the compiler. Be aware that all other text between a "#asm" - "#endasm" pair is passed unchanged to the output; no macro text replacements are done. "#asm" - "#endasm" may not be nested.

This escape to assembly language is provided as a convenient kludge mechanism only. It is not intended to be the normal way of interfacing assembly language functions to a C program. The proper way to interface to assembly language is to place the assembly function in a separate module, perhaps even in a user library if it is to be used frequently. This makes the program easier to transport to other systems, as the machine-dependent code is separated from the program source. See the Advanced Topics chapter for more information on assembly language programming in the MC environment.

Line Number Control [#line]

The preprocessor provides the "#line" directive to be able to pass source line number and filespec identification to the compiler.

```
 ______________________________________________________________
|                                                              |
|   #line linenumber {filespec}                                |
|                                                              |
|   linenumber - is the number to be passed to the output.     |
|                                                              |
|   filespec   - can be used to specify an optional filename.  |
|______________________________________________________________|
```

Description

For purposes of diagnostic error messages, it is useful to be able to report on the physical line number of the specific source file where an error was detected. Since the use of "#include" provides a continuous stream of processed source to the compiler, it inserts appropriate "#line" statements which reveal when a source file switchover has occurred. The compiler expects "linenumber" to be the physical line number of the next source line. Where "filespec" is passed, that line is assumed to be originating from the identified file. If "filespec" is omitted, the filespec is assumed to be the last file identified.

Program Example - SORTSYM

The following C-source program makes use of many different C statements and variable types - including structures and unions. The purpose of the program is to obtain a page listing of the symbol table produced by MLINK, the linker provided with the MRAS assembler development system. A report sorted alphabetically by symbol name and numerically by symbol address is produced. A preponderance of comments has been added to explain the rationale behind the program.

```
/*
 * SORTSYM - sort MLINK-generated symbol table by symbol and by address
 * Version 0.1 - Last update - 09/21/85
 * Written by Richard N. Deglin
 *
 * Invoke via the syntax: sortsym <infile >outfile
 */

#include <stdio.h>                   /* for standard definitions */
#include <stat.h>                    /* to define the stat structure */


#define RECORD      union record
#define RECLEN      sizeof(RECORD)
#define NAMELEN     15
#define ADDRLEN     4

struct stat fdat;               /* file's stat data structure     */

unsigned short nrecs,           /* number of symbol entries       */
               fsize,           /* SYM file size bytes to read    */
               i;               /* loop variable                  */

char tempname[NAMELEN],         /* workspace for swapping members */
     tempaddr[ADDRLEN];

/*
 *   The symbol table records generated by MLINK consist of
 *   a name field (15 characters), followed by a space,
 *   followed by a 4-byte hexadecimal address, terminated
 *   by a new-line. Since we want to produce a listing
 *   ordered by symbol name and by address, we declare a
 *   union of two structures. The first structure (byname)
 *   establishes the fields in the order read from the file.
 *   The second structure shows the re-arrangement of fields
 *   that we will create by moving the data. The union only
 *   requires storage space for the largest member; thus, only
 *   space for one record is actually required for the union.
 */

union record
{   struct
    {   char name[NAMELEN];      /* symbol's name                  */
        char space;              /* interstitial space             */
        char addr[ADDRLEN];      /* symbol's address value         */
        char newline;            /* terminating new-line           */
    } byname;                    /* first member of union record   */
    struct
    {   char addr[ADDRLEN];      /* symbol's address value         */
        char space;              /* interstitial space             */
        char name[NAMELEN];      /* symbol's name                  */
        char newline;            /* terminating new-line           */
    } byaddr;                    /* second member of union record  */
} *record,        /* A pointer to type "union record". We really   */
                  /* need an array of records and would be able to */
                  /* use something like var[COUNT]; however, since */
                  /* we don't know how many records we need to     */
                  /* obtain storage for, we use a pointer and      */
                  /* allocate storage later with alloc()           */

  *base,          /* A pointer to the base of the symbol records   */

  **prec,         /* A pointer to the "array" of record pointers.  */
                  /* This is similar to the specification, *var[]; */
                  /* however, since we don't know how big to make  */
                  /* the array, we use a pointer to a pointer and  */
                  /* dynamically allocate the array space.         */

  **recp;         /* an extra one for work use                     */

/*
 *   declare the following functions as type void so that the
 *   compiler will flag an error if we try to use their
 *   non-existant return value.
 */

void abend(), puterr(), tabify();

/*
 *   the compare() function is needed by qsort(); even though
 *   MC will assume a function to return an int if not explicitly
 *   stated as such, declaring all such functions is a good habit
 *   to get into
 */

int compare();

/*
 *   declare as externs, all of the library functions used by
 *   sortsym so that the compiler knows what kind of value,
 *   if any, they return.
 */

extern int fstat(), qsort(), strncmp(), fputs();
extern char *alloc(), *move();
extern unsigned short read(), write();

/*
 *   this is the main function
 */
main()
/*
 *   Since standard output will be used to write the processed
 *   output, it will be necessary to write informative messages
 *   to standard error. That's the purpose of puterr().
 */

{   puterr("\nReading...");

/*
 *   fstat() will get the statistics of standard input since
 *   it is referencing the file descriptor, STDIN. Note that
 *   the second argument to fstat() needs to be a pointer to
 *   a structure of type struct stat. Since fdat has been
 *   defined to be of type struct stat, we use the "address of"
 *   operator to denote a pointer_to.
 */

    if (fstat(STDIN,&fdat) == EOF)   /* stat the input file (stdin) */
        exit(1);                     /* a return of EOF means an error */

/*
 *   we use the st_size member which is the file size in bytes
 *   to calculate the number of records to read. In that way,
 *   we can read the entire file in one gulp. We also store the
 *   file's size in "fsize". Note that st_size is cast to an
 *   unsigned short (16-bit integer) since st_size is declared to
 *   be a long in the stat header file. By the way, we subtract 1
 *   from the SYM file's size because MLINK adds a new-line as the
 *   last byte of the file; that's ignored by sortsym.
 */

    nrecs = (fsize = (unsigned short) fdat.st_size - 1) / RECLEN;

/*
 *   Now we get a little complicated. We want to make sure that:
 *   (1) since we cast the size to a short, that it really can
 *   fit into a short; the first test takes care of that; (2) that
 *   we can allocate enough space for the input read buffer; the
 *   second test takes care of that; and (3) that we can allocate
 *   enough space for the pointers to each record; that's the
 *   third test. Note also that since alloc() returns a pointer to
 *   type char, it is cast to a "pointer to type RECORD" for the
 *   read buffer allocation, and a pointer to a pointer of type
 *   RECORD for the "array of pointers to the records".
 */

    if (fdat.st_size > 0xffffL ||
        ! (base = (RECORD *) alloc(fsize)) ||
        ! (prec = (RECORD **) alloc(nrecs * sizeof(RECORD *))))
            abend("SYM file too large\n");

/*
 *   The file is now read. The read() function returns the number
 *   of characters actually read. If this is not fsize, then an
 *   error occurred.
 */

    if (read(STDIN, (char *) base, fsize) != fsize) /* read SYM file */
        exit(1);

/*
 *   We now generate the array of pointers to each record
 */

    for ( i=0, record=base, recp=prec; i<nrecs; ++i)
        *recp++ = record++;

/*
 *   Some status to let you know where we are
 */

    puterr("\nSorting by name...");

/*
 *   Use the library function, qsort(), to sort alphabetically
 *   by symbol name. Qsort() uses our comparison function, compare()
 */

    if (qsort((char *) prec, nrecs, sizeof(RECORD *), compare) == EOF)
        abend("Can't sort\n");

/*
 *   Since MLINK produces a 1-across symbol table when directed to
 *   a disk file, we use tabify() to convert every two out of three
 *   newlines to a tab character. This produces a 3-across table
 */

    tabify();
    puterr("\nWriting by name...");      /* more status to inform you */
```

```
/*
 *   We now start writing to the output file using stream output.
 *   fputs() returns NULL if the output was successful, else EOF
 */

    if (fputs("Alphabetic sort:\n\n", stdout))
        exit(1);

/*
 *   Each record is now written using block file writes.
 *   Similar to read(), we compare the function's return value
 *   which is the number of bytes actually written to how many
 *   bytes we told it to write to detect error conditions
 */

    for (i = 0, recp = prec; i < nrecs; ++i)
        if (write(STDOUT, (char *) *recp++, RECLEN) != RECLEN)
            exit(1);

/*
 *  We now swap names and addresses so that sortsym can
 *  produce a table sorted by address. Pay close attention
 *  to the syntax of specifying each element of the union.
 *  Since "record" is a pointer to type union, we use the
 *  "->" object_of operator to connect the union's name to
 *  its members. Also, since its members are structures, we
 *  need the dot operator to connect the structure name
 *  with its member [record->byname.name]
 */

    puterr("\nReformatting...");
    for (i = 0, record = base; i < nrecs; ++i, ++record)
    {   (void) move(record->byname.name, tempname, NAMELEN);
        (void) move(record->byname.addr, tempaddr, ADDRLEN);
        (void) move(tempname, record->byaddr.name, NAMELEN);
        (void) move(tempaddr, record->byaddr.addr, ADDRLEN);
        record->byaddr.space = ' ';
    }

/*
 *   Some more status to inform you of where we are. We then
 *   sort the array of records again with qsort() so that it
 *   is ordered by address rather than alphabetically by name
 */

    puterr("\nSorting by address...");
    if (qsort((char *) prec, nrecs, sizeof(RECORD *), compare) == EOF)
        abend("Can't sort\n");

/*
 *   A new order requires a new construction of the tabs to
 *   construct a 3-across report format
 */

    tabify();

/*
 *   We now continue writing to the output file using stream output.
 *   fputs() returns NULL if the output was successful, else EOF
 */

    puterr("\nWriting by address...");
    if (fputs("\nAddress sort:\n\n", stdout))
        exit(1);
```

```
/*
 *   Each record is again written using block file writes.
 *   As before, we compare the function's return value
 *   which is the number of bytes actually written to how many
 *   bytes we told it to write to detect error conditions
 */

    for (i = 0, recp = prec; i < nrecs; ++i)
        if (write(STDOUT, (char *) *recp++, RECLEN) != RECLEN)
            exit(1);

/*
 *   This is the informative message we were looking for! We
 *   are also leaving it up to exit() to close the output file
 *   since we did not open it - it was standard output!
 */

    puterr("\nDone\n");
}

/*
 * This function is used to abort the program execution due
 * to some error. It uses puterr() to write the error message.
 */

void abend(s)
    char *s;
{   puterr(s);
    exit(1);
}

/*
 * The library qsort() function requires us to provide a comparison
 * function which compares two arguments. It needs to return <0, 0,
 * or >0 based on the comparison. We use strncmp() to compare at
 * most "RECLEN" characters. We could also have used memcmp().
 */

int compare(a, b)
    char **a, **b;
{   return strncmp(*a, *b, RECLEN);
}

/*
 * This function is used to conveniently write a message to
 * standard error output since standard output is being used
 * to write the processed output
 */

void puterr(s)
    char *s;
{   (void) fputs(s, stderr);
}

/*
 * This function changes every two out of three "newlines" to
 * a tab in order to produce a three-across listing
 */

void tabify()
{   for (i = 0, recp = prec; i < nrecs; ++i, ++recp)
        (*recp)->byname.newline = (i + 1) % 3 ? '\t' : '\n';
    (*--recp)->byname.newline = '\n';   /* last one always has newline */
}
```

Program Example - DCAL

The next program can be used in a TRS-80 Model 4 enironment to calculate the rotational speed of a floppy disk drive. It is based on the knowledge of the actual processor clock speed to calculate the time it takes the floppy disk drive to rotate from index pulse to index pulse. The program demonstrates one form of the use of inline assembler code via the "#asm" - "#endasm" pre-processor directives.

```
/*
 * DCAL - Used to calibrate the speed of a floppy disk drive
 * Version 0.1 - Last update - 11/20/85
 * Written by Roy Soltoff - derived from DCAL/BAS by Tim Mann
 *
 * Invoke via the syntax: dcal drive#   or   dcal
 */

#include stdio.h                    /* to obtain standard constants    */
#include math.h                     /* to request a search of MATH/REL */
#option INLIB                       /* to request a search of IN/REL   */

#define CLOCK 2.02752e6             /* documented clock speed of a
                                       Model 4 in slow speed           */

#define SELECT 0xf4                 /* FDC port for drive select       */
#define COMMAND 0xf0                /* FDC port for commands           */
#define STATUS COMMAND              /* FDC port for status - Note the
                                       macro is defined with a macro   */
#define RESTORE 3                   /* Controller command to restore a
                                       disk drive to track 0           */
#define BUSY 1                      /* FDC status bit for busy         */
#define VAR_TSTATE 46.0             /* This is the number of t-states
                                       taken up by the assembler code
                                       in the tight timing loop        */
#define FIX_TSTATE 23.0             /* This is the number of t-states
                                       taken up by the assembler code
                                       executed outside of the tight
                                       timing loop but in the timing   */

int ds[4] = {1,2,4,8};              /* drive select conversion table   */

unsigned drive, ckpulse();          /* these will be unsigned ints     */

double sbar = 0.0,                  /* smoothed speed, init to 0       */
       rpm,                         /* speed at each iteration         */
       lo_rpm = 400.0,              /* the lowest speed detected       */
       hi_rpm = 200.0;              /* the hughest speed detected      */

char buffer[81];                    /* a general purpose input buffer  */

/*
 *   this is the main function - it makes use of command line
 *   arguments; thus, it specifies the two arguments of main(),
 *   argc [the count of arguments in argv] and argv [the array
 *   of argument pointers].
 */

main(argc, argv)
    int argc;                       /* declare argc an int */
    char *argv[];                   /* argv is a pointer to an
                                       array of character strings */
{
/*
 *   We first test to see if dcal was invoked with the drive
 *   specification on the command line. If so, then argc would
 *   be equal to 2; if not, then argc would be equal to 1
 */
```

```
    if (argc==2)
        drive = atoi(*++argv);
    else

/*
 *   otherwise, we request the user to enter the physical floppy
 *   number [this is not necessarily the logical drive number if
 *   the system also contains a hard drive(s). We make use of the
 *   labeled statement to provide a "branch" point for the "goto".
 *   You see, there is at least one place we can make use of "goto".
 *   Use fputs() for output since we do not want a newline.
 *   Use gets() for input as it strips the newline.
 */

        {
input:  fputs("Enter floppy physical drive number <0-3> : ",stdout);
        if (!gets(buffer))
            exit(0);               /* we do not want to show an error */
        drive = atoi(buffer);      /* convert string entry to an int  */
        }

/*
 *   Test the input for validity. If not in the range 0-3, then
 *   issue the request again (the "goto" will branch to "input"
 */

    if (drive > 3)
        goto input;

/*
 *   The timing test needs a floppy diskette in the drive
 */

    printf("Insert a disk in floppy %d and depress return",drive);

/*
 *   Index the drive select code based on the drive slot number
 *   then use getchar() to accept the go ahead entry
 */
    drive = ds[drive];
    getchar();

/*
 *   Tell how to "escape" from the continuous timing test.
 *   This time we use puts() to add the newline character.
 */

    puts("\nDepress any key to exit timing loop");

/*
 *   Model 4's vary in effective CPU speed at the high speed because
 *   some have more wait states than others; thus, use the system()
 *   function to issue a DOS "system (slow)" command to force the
 *   use of the 2 MHz clock speed. We do not have to bother to use
 *   the portable execl() or execv() functions as this program is
 *   very machine dependent and certainly not portable!
 */

    system("system (slow)");    /* clock accurate at 2Megs only */

/*
 *   Use outport() to issue the drive select command to the FDC
 *   then restore the drive to track 0; the FDC needs a type II
 *   command to be able to check index pulse status
 */
```

```
    outport(SELECT,drive);
    outport(COMMAND,RESTORE);

/*
 *   Continue to select the drive until the BUSY flag is reset.
 *   That's when we know the drive has been positioned to track 0
 */

    while (inport(STATUS) & BUSY)
        {
        outport(SELECT,drive);
        if (inkey())            /* we can poll the keyboard as a */
            exit(0);            /* means of escaping if desired  */
        }

/*
 *   This while loop continues to check the index pulse timing
 *   until such time as a keyboard character is detected
 */

    while (!inkey())            /* i.e. while no key is sensed   */
        {

/*
 *   The rotational speed is calculated from the equation:
 *
 *       60 seconds per minute times CPU speed in seconds
 *  -------------------------------------------------------------
 *   (loop t-states times pulses per loop) plus fixed t-states
 */

    rpm = 60.0*CLOCK/((double)ckpulse(drive)*VAR_TSTATE+FIX_TSTATE);

/*
 *   If speed this check is lower than the lowest already recorded,
 *   then update the lowest stored; else if speed is greater than
 *   the highest already recorded, then update the highest stored.
 */

        if (rpm < lo_rpm) lo_rpm = rpm;
        if (rpm > hi_rpm) hi_rpm = rpm;

/*
 *    Calculate a "smoothed" speed by using 90 percent of our
 *    previously calculated smoothed speed plus ten percent of
 *    the speed detected on this check. Note that if the smoothed
 *    speed has not already been established, sbar would be equal
 *    to zero - its initialized value - and the smoothed speed is
 *    just set equal to the speed this check.
 */

        if (!sbar)
            sbar = rpm;
        sbar = sbar * 0.9 + rpm * 0.1;

/*
 *    Print out the driver's speed and the other statistics gathered.
 *    The escape constant, "\x1d", is used to position the cursor
 *    to the beginning of the line; thus, the printed data constantly
 *    overprints itself rather than constantly scroll up the screen.
 *    Note the use of the backslash, "\", to tell the MC preprocessor
 *    to continue the C source line.
 */

    printf("\x1dSpeed = %3.1f-%3.1f-%3.1f -- Smoothed speed = %4.1f",\
            lo_rpm,rpm,hi_rpm,sbar);
        }                      /* this is the end of the while loop */
```

```
/*
 *    Since we slowed up the normally "fast" CPU at the beginning
 *    of the program, we now direct it to the "fast" mode.
 */

     system("system (fast)");

/*
 *    No explicit exit() is needed as the closing brace of main()
 *    always "falls through" to exit().
 */

}

/*
 *     This function contains the assembler routine which performs
 *     the actual timing test of the disk drive. It is inserted
 *     into the source stream via the "#asm" - "endasm" preprocessor
 *     directive. Note that although the identifier "STATUS" is reused
 *     here, the preprocessor does not replace it with the substitution
 *     string "0xf0" as it is within a "#asm" - "#endasm" block.
 *     We use a C function declaration to establish the function
 *     environment; we could just as well have omitted it and
 *     declared a "PUBLIC CKPULSE" and a "CKPULSE:" entry label.
 */

unsigned ckpulse(drive) int drive;
{
#asm
STATUS  EQU     0F0H            ;FDC status port
SELECT  EQU     0F4H            ;Floppy drive select port
        $GA     DE              ;Get drive number into register_DE
        DI                      ;Can't interrupt timing loop
        LD      BC,2.SHL.8+SELECT
L1      LD      HL,0            ;Init or re-init counter to zero
L2      OUT     (C),E           ;Select the drive
        INC     HL              ;Bump counter
        IN      A,(STATUS)      ;Get the status
        AND     2               ;Mask the index bit
        JP      Z,L2            ;Loop until it's reset
L3      OUT     (C),E           ;Select the drive
        INC     HL              ;Bump counter
        IN      A,(STATUS)      ;Get the status
        AND     2               ;Mask the index bit
        JP      NZ,L3           ;Loop until it's set
        DJNZ    L1              ;Do twice to check index to index
        EI                      ;Restore interrupts
#endasm

/*
 *   The closing brace will provide the needed RET statement.
 *   Since thus is an "unsigned" function, the returned value will
 *   be in the HL register pair.
 */


}
```

Running the Compiler

Keyboard Refresher

Before you begin, remember that the DOS keyboard driver has a few extra keyboard combinations to generate some of the characters needed for C. It's probably a good idea to refresh your memory as to the key combinations.

character	key combination
[	CLEAR-COMMA
\	CLEAR-SLASH
]	CLEAR-PERIOD
^	CLEAR-SEMICOLON
_	CLEAR-ENTER
{	CLEAR-SHIFT-COMMA
|	CLEAR-SHIFT-SLASH
}	CLEAR-SHIFT-PERIOD
~	CLEAR-SHIFT-SEMICOLON

Operation

The MCP preprocessor takes C source code as input and generates a preprocessed output file for the MC compiler's input. MC takes that preprocessed file as its input and generates an assembler source file as output. The assembler file is customized for either MISOSYS' MRAS or Microsoft's M80 assembler by means of a compiler option. The assembler of choice will produce a relocatable object module; this module must be combined with various modules contained in the supplied libraries in order to produce an executable program file. This combining process is called "linking"; it is performed by the linker (MLINK or L80) supplied with your assembler of choice. Thus the output of the compilation process must be assembled and linked with any required run-time library module before it can be executed.

The first stage of the C language process is, of course, to create a C source file. The editor that is a part of your assembler package may be used for this purpose or you may use any other text editor. Your assembler manual should be consulted for all operations concerning the editor or assembler functions.

Note for M80/L80 Usage

If you are using Microsoft's M80/L80 assembler/linker, you will have to add the preprocessor statement:

```
#include m80
```

in any C-source program file which includes your "main()" function. The "#include m80" must be inserted after all "#option" statements and before any global declarations. The identifier, "m80bgn", is reserved.

The second stage of the C language process is the compilation of the C source using the MCP preprocessor and MC compiler. This produces an assembly source file.

The third stage of the C language process is to convert that assembly language file into a relocatable object module. Your assembler does this.

The fourth stage of the C language process is to link the relocatable module with any other relocatable modules you have developed for that program and any modules needed from the supplied libraries. Your linker will perform this step.

Finally, your last stage is to test and debug your work. This generally requires a cycling through the above stages until you are satisfied that your program behaves as you intended.

The JCL procedure shown below can automate the stages from compilation through linking. However, if you want to take direct control of each stage of the operation, you can invoke MCP and MC directly (as well as your assembler and linker).

Using Job Control Language

A Job Control Language (JCL) file, "MC/JCL", presents the preprocessing, compilation, assembly, and linking as a job stream to the operating system. The JCL procedure requires minimal entry of commands by the programmer to create an executable CMD file. It provides eight options that are passed to the procedure in the JCL invocation. The JCL is invoked via the command:

DO MC (N=progname{,parm}{,parm}{,...})

progname	-	Specifies the name of the C source file to be compiled and assembled into a CMD file.
a	-	Specifies that the /ASM or /MAC file is to be assembled following a successful compilation.
c	-	Specifies that the input C source file is to be preprocessed and compiled (default).
cc	-	Specifies that the /ASM or /MAC file will have the C source imbedded as comments.
d	-	Specifies the drive spec of the disk drive for all file I/O (defaults to d=1).
k	-	Specifies that the intermediate files (/INT, /ASM or /MAC, and /REL) should be deleted after the /CMD file is generated.
l	-	Specifies that the /REL module is to be linked with the needed libraries (default).
list	-	Specifies that the assembly phase should produce a listing.
o	-	Specifies that the /ASM or /MAC file is to be optimized prior to assembly.

The beginning user need only be concerned with the "N=progname" parameter as this is the means to identify the name of your C source program. The optional parameters are useful to the more knowledgable and experienced user.

If you want to compile a program called "myprog" which resides on logical drive 1 and generate the finished CMD file with only one statement, then the command:

```
DO MC (N=MYPROG)
```

lets you sit back and relax while the machine does all of the work. If the /CCC file exists on another drive, use the "d" parameter.

Invoking the MCP Preprocessor

The preprocessor is invoked via the command:

MCP filespec {switch} {switch...}

filespec	-	A file specification for the input file. Only one filespec may be passed.
switch	-	Represents an optional preprocessor switch. These switches are preceded with either a plus sign (+ = ON) or a minus sign (- = OFF). The usable switches follow.
+dIDEN[=s]	-	Specifies the definition of a macro with an optional replacement text string
+o[=spec]	-	Specifies the generation of output and the output file specification.
-**o**	-	Specifies that no output file should be written.
-**p**	-	Designates that the display should not stop when error messages are emitted.
+**t**	-	Specifies the output of an untokenized text file purely for debugging macros.
+uIDEN	-	Undefines the designated identifier.

The "filespec" may optionally include a file extension; however, if it is omitted, "/CCC" will be assumed. The drivespec is optional. It is recommended that you establish your C source files with the "CCC" file extension for uniformity and standardization.

Switches:

"Switch" options are specified by a plus "+" or minus "-" sign followed by the option letter and any additional information needed by the switch(es).

+dIDENTIFIER[=TEXT]

The plus "dee" switch will define a macro with optional replacement text from the command line. If text is not given, the macro is defined as "1". The minus "dee" syntax, "-d" is an alternative form. A parameterized macro definition is acceptable.

+o[=][FILE-OR-PARTSPEC]

This plus "oh" switch instructs MCP to write output to the named file. If a partspec is given, the output filespec is constructed from the name of the main input file plus the partspec. The output file's extension will be forced to "/TOK".

-o

The minus "oh" switch specifies that no output file is to be written.

-p

When this switch is ON (+p), the preprocessor will stop when any errors are found and displayed. Any key except BREAK will continue compilation. BREAK will abort the execution of the preprocessor at any time if this switch is on.

If MCP was invoked from JCL, the JCL will also be aborted. This switch defaults to ON. If you wish to turn OFF the pause facility, enter "-p".

+t

The plus "tee" switch specifies that the output file generated by the preprocessor will be a pure text file. The normal type of file generated for input to the compiler phase is a tokenized file. Thus, the text output may be useful only for checking your macro expansions for correctness.

+uIDENTIFIER

The plus "you" switch instructs MCP to ignore the first "#define" for this macro identifier encountered in the input stream. The minus "you" syntax, "-u" is an alternative form.

Invoking the MC Compiler

MC is invoked to compile a preprocessed C source intermediate file. The command syntax is as follows:

```
MC filespec {switch} {switch...}

filespec    -  A file specification for the input file.
               Only one filespec may be passed.

switch      -  Represents an optional compiler switch(es)
               These switches are preceded with either a
               plus sign (+ = on) or a minus sign
               ( - = off). The usable switches follow.

+c          -  Specifies that the C source code will be
               written to the output file as comments.

+f          -  Specifies the FLOAT option for using the
               single precision floating point functions.

+m          -  Designates that the output should be M80
               compatible.

+o=spec     -  Designates that output should be written to
               the file identified.

-o          -  Designates that no output file should be
               written.

-p          -  Designates that the display should not stop
               when error messages are emitted.
```

The compiler is invoked by entering a command line such as:

MC CPROGRAM:2

which compiles the preprocessed C source file, "CPROGRAM/TOK:2", and generates the output file, "CPROGRAM/ASM", on drive 2. The compiler will generate an MRAS compatible output file.

The switches allow you to control certain features of the compiler. The simplest compilation command would simply be "MC PROGNAME" which compiles the file, "PROGNAME/TOK", and generates the output file, "PROGNAME/ASM".

File Specifications

There may be only one input file specification given on the command line. No extension should be given for a source file; the default extension "/TOK" is assumed.

The output file specification defaults to the same name as the input file specified. MC will append the file extension "/ASM" (or "/MAC" when the +M switch is specified) to this name. The drive specifier, if any, of the input filespec is used as the drive specifier of the output file. The drive specifier should be given if the output file must be written to the same drive as the input file. The OUTPUT option may be used to specify a different file name or change the destination drive number. Assembler source code output may be suppressed by turning off the OUTPUT option. This can be helpful for quickly checking syntax without generating an output file.

Compiler Switch Options

Compiler option switches are turned on or off by a '+' or '-', respectively, followed by the name of the switch. The compiler regards any command line argument not beginning with a plus or minus as the input file specification. Only the first letter of the switch is examined, so partial spelling (or misspelling) is accepted. Certain switches have operands which are specified by following the option name with '=' and the operand. For instance, "+o=myfile:3" will cause the output by the compiler to be written to "MYFILE/ASM:3" instead of the filespec that would have been the default.

Comment

This switch controls whether the preprocessed C source code will be written to the assembler output file as comments. The normal default is OFF. The C-source appearing as comments may be instrumental in your understanding the compiler output as it generates a minimally commented assembly source program. If you do need these "comments", then specify "+c".

Float

A compiler option has been included which allows you to change the default floating point mode of the compiler. Use of the "+f" command line option will force the compiler to do float (32-bit) arithmetic in single precision mode, rather than in double mode. Also, this forces floating point constants to be interpreted in single precision. This results in faster operation with some loss of accuracy. Note that this is an MC extension to the K&R standard, and is not portable! The default mode of the compiler is "-f". Please consult Chapter 5, "Advanced Topics" before you attempt to use this option as there is a considerable difference in the way float arguments to functions are automatically casted depending on "+f" vs "-f".

+M - Assembler Option Switch

This switch will cause the compiler to generate code suitable for the Microsoft M80 relocatable assembler. The default file extension applied to the output file will be "/MAC". Note that throughout the remainder of this chapter, the output assembly file may be referred to as /ASM.

Output=spec

This switch controls the assembly file output of the compiler. If the switch is OFF (-o), no output file is generated. However, if it is ON, but no SPEC is given (+o), MC adds the file extension "/ASM" or "/MAC" to the name of the input file in order to create the output file specification. When a file specification is given for "spec", it becomes the name given to the output file. A default extension of "/ASM" (or "/MAC") is inserted if no extension is given. If only a drive specification (":D") is given, the output file is written to that drive, with the same file name as the input file. This switch defaults to ON with no "spec" (+o).

Pause

When this switch is ON (+p), the compiler will stop when any errors are found and displayed. Any key except BREAK will continue compilation. BREAK will abort the execution of the compiler at any time if this switch is ON. If MC was invoked from JCL, the JCL will also be aborted. This switch defaults to ON. If you wish to turn OFF the pause facility, enter "-p".

Creating an Executable CMD File

Once the compiler has compiled your program into assembly language, you next need to use the MRAS or M80 macro assembler to create the relocatable (REL) object file. In order to provide control for certain MC library options, the proper initialization in the CMD file, and ensure that all necessary runtime routines are linked with your program, a special assembler file, MC/ASM (MC/H for M80 use), has been provided.

The MC/ASM file that is provided with your compiler package is generic to the environment of the operating system. The MC/ASM file is the file that is assembled in concert with your source file which contains main(). MC/ASM also includes special requests to direct the linker to search the libraries for needed modules during the link session.

For a great deal of your programs, the only MC runtime routines needed will be located in the LIBC library. Since all C programs need some of the routines in LIBC/REL, that library is ALWAYS requested (as well as LIBA). The high-level math library, MATH/REL, will be searched if your program requested it via an "#option MATHLIB" or "#include math" compiler macro.

Some useful routines are stored in the installation library, IN/REL. This library is not normally searched in order to save you linker search time when you need not refer to the IN/REL modules. However, it is very easy to force an automatic search of the installation library. All you need to do is specify an "#option INLIB" compiler macro in your C source program (similar to "#option MATHLIB"). For example,

```
#option INLIB
main()
{
   int dot;
   for ( dot=0, dot < 128, dot++ ) set( dot, 0 );
}
```

the #option INLIB statement will force a search of the installation library during the link session to resolve linkage to the set() function.

If you are adding your own relocatable library, name it "USERLIB/REL". To force an automatic search of it, add this statement to your C program:

```
#option USERLIB
```

Compile-Time Directives

Compile-time directives are used to convey information from your source file to the various stages of the program generation. These directives are initiated via the "#option" prefix. The syntax is as follows:

```
#option <optname> {value}
```

The "#option" directive is used to pass symbol definitions from the MC source code to the assembly phase of main() and the subsequent link phase. The <optname> must be a valid C identifier. Value must be a numeric or character constant or constant expression. Escape sequences may be used in the constant. The compiler translates the "#option" directive to the form:

```
@_OPTNAME DEFL  value
```

The "value" is optional (as shown above by apearing within braces). If the value is omitted, the DEFL statement will default to a value of negative one (-1). This indicates TRUE to the assembler.

The "#option" directive is used in MC to invoke an automatic search of the installation library, IN/REL, or the high-level floating point math library, MATH/REL. The automatic search is specified by the generation of "request library search" special link items in the resulting relocatable object module. This special link item is used by the linker. If your application will be using functions in either library, you will need to add the statement(s):

```
   #option MATHLIB
   #option INLIB
```

for the high-level floating point math and installation libraries respectively.

MC has reserved additional option names for use with the #option directive. These are: args, fixbufs, maxfiles, redirect, and wildcard. The following paragraphs describe their use.

ARGS

This is used to specify that your program {will}/{will NOT} be using command line arguments (argc, argv). MC will suppress the run-time code normally used to process arguments thus reducing the size of your CMD program. ARGS defaults to ON.

FIXBUFS

This is used to specify pre-allocation of buffers for standard I/O. If your program does not need dynamic memory allocation, then by specifying this option, you will inhibit the file I/O system provided in the standard library from automatically using dynamic allocation. This will result in a smaller executable program file. FIXBUFS defaults to OFF

MAXFILES

This is used to specify the maximum number of concurrently opened files permitted. The file I/O system provided in MC defaults to a maximum of 13; three additional are always provided which are needed for the standard files. Each requires memory overhead for storage. This storage space will normally be acquired dynamically (via alloc) and thus obtained only when files are opened; however, if FIXBUFS is ON, the storage space is always reserved at program execution regardless of whether files are opened or not. If your memory resources are strained and you are using FIXBUFS, then you may want to pay close attention to the maximum number of concurrently open files needed and set MAXFILES appropriately. MAXFILES defaults to 13.

REDIRECT

This is used to specify that your program {will}/{will NOT} be using standard I/O redirection (>, >>, <, #). MC will suppress the run-time code used to process I/O redirection thus reducing the size of your program. REDIRECT defaults to ON.

WILDCARD

This is used to specify that the resulting executable command program will be using command line wildcard filespec expansion; thus, the library routines needed to satisfy that request will be included during the link session. The use of this option is detailed in Chapter 5, Advanced Topics. WILDCARD defaults to OFF.

Assembly of the ASM file

The assembly file output by the compiler is converted to a relocatable object module by means of your assembler. If the file was a standalone module (not to be linked with other separately compiled modules) containing the main() function and all other user-programmed funcitons needed for main(), then a sample syntax for the invocation of MRAS or M80 would be:

```
MRAS MC +I=progname +o=progname -nl
M80 progname=progname
```

For MRAS use, this would assemble the file "MC/ASM" while it included "progname/ASM" into the source stream. For M80 use, it would assemble the source file "progname/MAC" while it included the file "MC/H" into the source stream. Note that the MC/H header file was inserted into the C-source stream by your explicit "#include m80" preprocessor statement. Using either assembler, the assembly would generate the relocatable object file named "progname/REL".

If you are developing an executable program which is composed of separately compiled modules, the procedure for accomplishing the mechanics of the compile, assemble, and link phases is covered in Chapter 5, Advanced Topics.

Linking the relocatable object module

The object module is linked with other modules supplied in the libraries by means of the linker supplied with your assembler. Sample syntax for linking the abovementioned module with your linker would be:

```
MLINK progname +n=progname -e
L80 progname-n,progname,-e:M80BGN
```

L80 users please note the "M80BGN" operand on the "-e:name" linker switch. This is mandatory! It is used to specify the transfer address of the resulting executable command file. L80 users also note that the "progname-n" output file specification must be the first specification on the argument line.

If you are developing an executable program which is composed of separately compiled modules, the procedure for accomplishing the mechanics of the compile, assemble, and link phases is covered in Chapter 5, Advanced Topics.

Function Libraries

General Information

The libraries provided with the MC compiler are a collection of useful and powerful functions, which allow the user to interface with the world external to the program without having to know the specifics of the particular environment that the program is running in. Wherever possible, each function adheres to the standards of UNIX System V; the "SYSTEM V Interface Definition" published by AT&T has served as the reference thereof.

MC provides collections of common functions in four separate libraries: LIBA/REL, LIBC/REL, MATH/REL, and IN/REL; commonly called LIBA, LIBC, MATHLIB, and INLIB respectively. The functions are stored as relocatable object modules. Each file is thusly termed a relocatable object module library. The specific relocation format used is that documented by Microsoft, Digital Research, and MISOSYS in their respective assembler development systems.

LIBA contains very low-level functions invoked by the compiler to perform basic operations such as add, subtract, multiply, divide; logic operations (or, and, exclusive or), stack access operations, and others. The functions included in LIBA are better termed routines. They are, in general, not available to the C-language programmer but are incorporated by the compiler into the compiled output as needed by the program being compiled.

LIBC contains most of the functions accessible to the C programmer. This library constitutes the "standard Library" functions as noted by most texts on the C language. You will find all of the functions associated with stream I/O and block device I/O. You will find the bulk of the character test operations, the character manipulation operations, the memory access operations, the utility operations, as well as print formatting for all variable types except floats.

The MATHLIB library includes all of the high-level floating point and double precision functions such as trigonometric operations, logarithmic operations, random number generation, and a complete print formatter for floats and doubles. In order to access any of the functions in this library, your source program must include the statement:

```
#option MATHLIB
```

In fact, if you are going to utilize any float or double, you need to specify this preprocessor directive. Most of the functions contained in MATHLIB must be declared prior to their use to inform the compiler of their "type. It's very easy to insert a bug into a program by neglecting to advise the compiler that a function returns a double such as with the declarative:

```
extern double exp();
```

The high-level math functions provide error diagnostics per System V standards. These standards dictate specific error number symbolic names as well as an exception structure for the data handling of error fixup. System V documents a header file, "math.h", which includes all of the symbolic names associated with error handling, the definition of the exception structure, and a group of mathematical constants defined as symbolic names. The "math.h" header file included with MC also adds the declarations of all functions accessible to you in MATHLIB. By adding the preprocessor directive:

```
#include <math.h>  or  #include "math.h"  or  #include math
```

in all of your modules which need access to MATHLIB, you need not worry about function declarations. You will also have the symbolic names available to you. The "math" header file also adds the "#option MATHLIB" directive so you won't have to bother with it.

The final library included with MC is INLIB. This library contains nonstandard C functions that are implementation specific. If you desire to make your program portable to all System V compilers at the source level, then do not utilize any of these functions. However, if you intend to produce a locally used machine dependent program, then INLIB will provide you access to pixel graphic functions for plotting, string manipulations similar to BASIC, and special character device operations such as keyboard scanning and video cursor manipulation. The functions included in INLIB are automatically accessible to your program when you include the preprocessor directive:

```
#option INLIB
```

within your program's source.

The pages that follow in this chapter provide detailed information on each function available to you. The chapter has been arranged purely in alphabetical order so that it may serve as a useful reference tool. Almost every function will appear on a separate page. The library where the function is stored is listed within parentheses appended to the function's name.

To assist you in your search for a function that satisfies a particular need, the following classifications may be of assistance.

Stream I/O functions:

addext	checkc	clearerr	cleareof	fclose
fdopen	fdown	feof	ferror	fflush
fgetc	fgets	fileno	fopen	fprintf
fpup	fputc	fputs	fread	freopen
fscanf	fseek	ftell	fwrite	genspec
getc	getchar	gets	getw	inkey
ioctl	isatty	printf	putc	putchar
puts	putw	rewind	scanf	ungetc
ungetch	unlink			

Block I/O functions:

addext	close	creat	dup	dup2
fcntl	fdown	fileno	fpup	fstat
genspec	gtty	ioctl	isatty	lseek
open	read	seek	stty	tell
ttyname	write			

Integer (int, long) math functions

abs	atoi	btoi	itoa	itob
itoo	itou	itox	labs	lpower
ltoa	ltob	ltoo	ltou	ltox
otoi	otol	rand	srand	xtoi
xtol				

Single precision (float) math functions

fabsf	fatn	fcos	fexp	ffix
fint	flog	fraise	frnd	fseed
fsin	fsgn	fsqr	ftan	ftoa

Double precision (double) math functions

acos	asin	atan	atan2	atod
ceil	cos	cosh	_ddv2	_ddv230
dfix	dint	dsgn	dtoa	exp
fabs	floor	fmod	frexp	hypot
ldexp	log	log10	matherr	modf
pow	sin	sinh	sqrt	tan
tanh				

Environment information functions

asctime	ctime	curpos	cursor	freemem
inkey	inport	isalnum	isalpha	isascii
isbdigit	iscntrl	isdigit	islower	isodigit
isprint	ispunct	isspace	isupper	isxdigit
localtime	outport	perror	qsort	sysdate
systime	sys_errlist	time	tolower	toupper
_xlate				

Memory access functions:

alloc	brk	calloc	fill	free
freemem	malloc	memccpy	memchr	memcmp
memcpy	memset	move	realloc	sbrk
swab	zero			

Plotting functions:

box	circle	line	pixel	ploc
pmode	point	reset	set	

Program control operations

abort	call	cmdi	execl	execovl
execv	exit	_exit	longjmp	matherr
option	setjmp	system	unlink	

String handling functions:

atod	atof	atoi	atol	btoi
dtoa	ftoa	index	itoa	itob
itoo	itou	itox	ltoa	ltob
ltoo	ltou	ltox	otoi	otol
qsort	rindex	sprintf	sscanf	strcat
strchr	strcpy	strcspn	strepl	strept
strfind	stright	strleft	strlen	strmid
strncat	strncmp	strncpy	strpbrk	strrchr
strspn	xtoi	xtol		

This function can be used to abort the executing program and optionally perform a "core dump".

void abort()

Description

Abort() will first attempt to close all open files, if possible. Abort() then issues the message:

```
Program aborted via abort at X'xxxx'
```

where "xxxx" is the program counter address when abort() was called. If the Operating System's Job Log device (*JL) is active, a formatted memory dump will be written to the job log device. The dump will include all addresses from the start of the executing program through the highest address used by the program. This will be in the format:

```
zzzz: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx aaaaaaaa aaaaaaaa
```

where "zzzz" designates the origin of a 16-byte region, "xxxx..." indicates the contents of the region in hexadecimal, and "aaaa..." indicates the contents of the region in ASCII; characters outside the range 0x20 through 0x7f will be converted to a period (0x2b).

Return Code

There is no return from abort().

Example

```
main()
{
   /* notice that puts() adds a '\n' character */
   puts("Catastropic failure in framus correctus");

   /* notice that fputs() does not add a '\n' character */
   fputs("\tDo you wish to abort <y,n> ? ",stdout);

   if (tolower(getchar()) == 'y')
      abort();
   else
      puts("You chose not to abort...");
}
```

Try the example after issuing the DOS command:

```
ROUTE *JL to *DO
```

This function is used to obtain the absolute value of an integer.

int abs(ival);
int ival;

ival is the integer whose absolute value is to be determined.

Description:

The abs() function returns the absolute value of its integer argument (i.e. abs(x) = x; abs(-x) = x;).

Example:

```
#include stdio.h
char inbuf[81];
main()
{
   puts("Enter your number: EOF to exit");
   while (TRUE)
       {
       if (!fgets(inbuf,80,stdin))
           break;
       printf("Absolute value of your number is %d\n",abs(atoi(inbuf)));
       }
}
```

See also: fabs(), fabsf(), labs()

This function is used to obtain the double precision Arc cosine.

```
#include <math.h>
double acos( xval );
   double xval;

   xval      is the double whose Arc cos is desired.
```

Description

The Arc cos function obtains the double precision principal value of its argument. The range of the result is 0 to PI. The Arc cos is calculated from the relationship:

Arc cos(x) = PI/2 + Arc sin(-x)

Return Code

If the magnitude of the argument to acos() is greater than one, zero is returned and errno is set to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output. This error-handling procedure may be changed with the function, matherr().

Example

```
#include stdio
#include math
int i; char inbuf[81]; double d1,d2;
main()
{  puts("Enter your number: EOF to exit");
   while (TRUE)
   {   if (!gets(inbuf)) break;
       errno=0;
       d1 = acos(d2=atod(inbuf));
       printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}
matherr(x) struct exception *x;
{  fprintf(stderr,"Type %d function error in %s...\n",x->type,x->name);
   fprintf(stderr,"     Args = %g,%g: ",x->arg1,x->arg2);
   return 0;
}

Enter your number: EOF to exit
0.9 |d1 = d2 0.9; = 0.451027; errno = 0
1.5 |Type 1 function error in acos...
    |     Args = 1.5,0: argument domain error
    |d1 = d2 1.5; = 0; errno = 70
```

See also: asin(), atan(), atan2(), errno

This function adds a default extension to a file specification.

```
char *addext( filespec, defextn );
   char *filespec, *defextn;

   filespec  the file specification to which the default should be added.

   defextn   the extension to add if filespec does not contain an extension.
```

Description

Addext is used to add a default file extension to a file specification if the file specification in question does not already contain an extension. It is useful for automatically adding standard extensions to files (i.e. CCC, CMD, TXT, ...) to minimize the input necessary in command lines.

Return Code

A pointer to the filespec is returned.

Warnings

The filespec argument must be a character array of at least dimension 15 to avoid writing beyond the end of the string.

See also: genspec()

This function is used to allocate a memory block.

```
char *alloc ( nbytes );
   unsigned int nbytes;

   nbytes    unsigned number of bytes needed.
```

Description

Alloc() is used to dynamically allocate memory during program execution. The complementary function, free(), is used to release memory allocated through alloc(). Alloc() may be used to get table or buffer space when the amount of memory space available is unknown, or when the program needs to dynamically allocate space for an array.

Return Code

If a memory block has been allocated, the value returned is a pointer to the memory block (pointer to char). If insufficient memory is available to satisfy the allocation, alloc() will return NULL (0).

Warnings

The program must not access memory outside of the area allocated. File access routines use alloc() and free() to establish and release File Control Areas (FCA's). The programmer cannot assume that memory not allocated is free for use, since later file opens may cause memory overlays. It is advised that the programmer always use the supplied dynamic allocation functions for memory accessing.

Example

```
symtbsz -= symtbsz % symsiz;   /* make integral */

if ((symtab = alloc(symtbsz)) == NULL)
   abend("not enough memory");

glbptr = startglb = symtab;
```

See also: brk(), calloc(), malloc(), realloc(), and sbrk()

This function is used to obtain the 26-character time string.

```
#include <time.h>
char *asctime( tm );
   struct tm *tm

   tm     is a pointer to the structure which contains the broken down time.
```

Description

Asctime() takes the broken-down time data contained in the structure pointed to by the argument and converts it to an ASCII string of the form:

```
Tue Oct 22 10:03:10 1985\n\0
```

Return Code

A pointer to the resulting string is returned.

Warning

Note that the string itself is a local static and may be valid only immediately following the function call.

Example

```
#include "time.h"
char *ctime(clock) long *clock;
{
   struct tm *localtime();
   char *asctime();

   return asctime(localtime(clock));
}
```

See also: ctime(), localtime(), time()

This function is used to obtain the double precision Arc sine.

```
#include <math.h>
double asin( xval );
   double xval;

   xval      is the double whose Arc sine is desired.
```

Description

The Arc sin function obtains the double precision principal value of its argument. The range of the result is -PI/2 to +PI/2. The Arc sin is calculated from the relationship:

```
Arc sin(x) = atan2(x,(sqrt(-x*x+1.0)))
```

Return Code

If the magnitude of the argument to asin() is greater than one, zero is returned and errno is set to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output. This error-handling procedure may be changed with the function, matherr().

Example

```
#include stdio.h
#include math.h
int i; char inbuf[81]; double d1,d2;
main()
{  puts("Enter your number: EOF to exit");
   while (TRUE)
   {   if (!gets(inbuf)) break;
       errno=0;
       d1 = asin(d2=atod(inbuf));
       printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}
matherr(x) struct exception *x;
{  fprintf(stderr,"Type %d function error in %s...\n",x->type,x->name);
   fprintf(stderr,"     Args = %g,%g: ",x->arg1,x->arg2);
   return 0;
}

Enter your number: EOF to exit
-.9  |d1 = -1.11977; d2 = -0.9; errno = 0
-1.5 |Type 1 function error in asin...
     |     Args = -1.5,0: argument domain error
     |d1 = 0; d2 = -1.5; errno = 70
```

See also: acos(), atan(), atan2(), sin(), matherr()

These functions are used to obtain the double precision Arc tangent.

```
#include <math.h>
double atan( xval );
   double xval;

#include <math.h>
double atan2( yval, xval );
   double xval, yval;

   xval, yval   is the double whose arc tangent is to be determined.
```

Description

The Arc tan function obtains the double precision principal value of its argument. The range of the result is -PI/2 to +PI/2. The Arc tan2 function uses the signs of both its arguments to determine the quadrant of the return value which will be in the range -PI to +PI. The atan() function is derived according to the algorithm given in SOFTWARE MANUAL FOR THE ELEMENTARY FUNCTIONS by William J. Cody, Jr. and William Waite.

Return Code

If both arguments of atan2() are zero, zero is returned and errno is set to EDOM. In addition, a message indicating DOMAIN error is written to standard error. This error-handling procedure may be changed with the function, matherr().

Example

```
#include stdio
#include math
int i; char inbuf[81]; double d1,d2;
main()
{  puts("Enter your number: EOF to exit");
   while (TRUE)
   {   if (!gets(inbuf)) break;
       errno=0;
       d1 = atan(d2=atod(inbuf));
       printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}
Enter your number: EOF to exit
1e15 |d1 = 1.5708; d2 = 1e+15; errno = 0
-2.5 |d1 = -1.19029; d2 = -2.5; errno = 0
0    |d1 = 0; d2 = 0; errno = 0
```

See also: acos(), asin(), tan(), errno, matherr()

atod(MATH); atoi(LIBC); atof(MATH); atol(LIBC); **ato?(MATH)**

These functions are used to convert character strings of digits to their numeric machine value.

```
#include <math.h>
double atod( string );
float atof( string );
   char *string;

int atoi( string );
long atol( string );
   char *string;

   string    is a string containing character digits valid for the number
             type.
```

Description

These functions are used to convert ASCII strings to their numeric value. Functions atoi() and atol() convert strings of decimal digits 0-9 to an int or long int. The converted integers will be modulo 65536 or modulo 4294967296 respectively. Conversion stops as soon as a character is detected which is not in the valid range. Functions atof() and atod() convert a string consisting of an optional sign followed by an optional sequence of decimal digits followed by an optional decimal point followed by an optional sequence of decimal digits followed by an optional exponent specifier ['e', 'E', 'd', or 'D'] followed by an optional exponent sign followed by an optional string of decimal digits to a float or a double respectively. The string cannot exceed 32 digits decimal digits!

Example

```
#include stdio
#include math
char inbuf[81], outbuf[81]; double dnum;
main()
{  puts("Enter your number: EOF to exit");
   while (TRUE)
   {   if (!gets(inbuf)) break;
       dnum=atod(inbuf);
       fputs("Your number is: ",stdout); puts(dtoa(dnum,outbuf));
   }
}

Enter your number: EOF to exit
1            |Your number is:  1.000000E+00
13.476       |Your number is:  1.347600E+01
-456.218e19  |Your number is: -4.562180E+21
```

See also: sscanf(), dtoa(), ftoa(), itoa(), ltoa()

This function is used to plot a rectangle.

```
int box( funcod, x1, y1, x2, y2 );
   int funcod; x1, y1, x2, y2;

   funcod    an operation code to set (1) or reset (0) the pixels involved in
             the geometric plot.

   x1,y1     the coordinate of the northwest corner of the rectangle.

   x2,y2     the coordinate of the southeast corner of the rectangle
```

Description

The "box()" function will plot a rectangle around the diagonal specified by the coordinate point pairs, (x1,y1) and (x2,y2). If the coordinates specify either equal values of x (x1 = x2) or equal values of y (y1 = y2), then the rectangle will diminish to a line. The rectangle will collapse to a point if both x1=x2 and y1=y2.

Box() supports virtual points which means that your plot does NOT have to limit itself to the CRT image area. For example, a "box(1,10,20,190,25);" function describes a rectangle partially out of the CRT image. In this example, only a portion of a rectangle will be plotted.

Return Code

A minus one (-1) indicates that the coordinate points (x1,y1), (x2,y2), or a portion of any plot is out of range (i.e. virtual and does not appear in the CRT image). A minus three (-3) will be returned if the function code passed is invalid (not in the range <0-1>.

Example

```
#option INLIB
#define X 79 /* 80 x 24 screen size */
#define Y 35
#define DRAW 1
int x1, y1, x2, y2;

main()
{
   for (x1=X,y1=Y,x2=X+1,y2=Y+1; x1 >= 0; x1--,y1--,x2++,y2++)
      box(DRAW, x1, y1, x2, y2 );
}
```

See also: circle(), line(), pixel(), ploc(), pmode(), point(), reset(), set()

This function is used to set the program break address.

```
int brk( address );
   char *address;

   address      is to be the lowest machine address not used by the program.
```

Description

The lowest memory address not used by the program (exclusive of the program stack) is termed the "break". This function will set the program break to "address". One greater than the highest address used by your program for program code and static variable storage is defined as "$PROGEND". One greater than the highest memory address allocated is defined as $LOMEM. $LOMEM is initially equal to $PROGEND. MC maintains a pointer to this location called $FREEP. The break argument, "address", may not be below $FREEP nor greater than (STACK_POINTER less 1024 bytes).

Warning

NEVER use brk() with any dynamic allocation routine such as alloc(), calloc(), free(), or malloc() or with the standard I/O package without specifying "#option FIXBUFS".

Return Code

Zero is returned if the break could be set while a -1 is returned if the break exceeded the bounds of the available memory and thus could not be set.

Example

```
#include <stdio.h>
#option FIXBUFS
char *brk(), sbrk();
main()
{  char *membuf, *lomem;
   lomem = sbrk(0);        /* Get current value of $LOMEM */
   printf("$LOMEM pointer = X'%04x'\n",lomem);
   membuf=sbrk(1024);      /* grab a 1K buffer */
   printf("$LOMEM pointer = X'%04x'\n",sbrk(0));
   if (!brk(lomem)) printf("$LOMEM pointer = X'%04x'\n",lomem);
   else puts("Brk error");
}

$LOMEM pointer = X'54cf'
$LOMEM pointer = X'58cf'
$LOMEM pointer = X'54cf'
```

See also: alloc(), calloc(), free(), malloc(), sbrk()

This function is used to convert a character string of binary digits to its integer value.

```
int btoi( string );
   char *string;

   string    is a string containing binary digits <0-1>
```

Description

This C function obtains the machine value of a string of binary digits in character form (i.e. composed of nothing but zeroes and ones). Left truncation of the integer value takes place if an excess number of digits is present (i.e. ival=btoi("10000000000000001"); would result in the integer value of 1 decimal). Conversion stops as soon as the first character not in the valid range is detected.

Example

```
#include stdio.h
char inbuf[81]; int ival;
main()
{
   puts("Enter your binary number: EOF to exit");
   while (TRUE)
    {
      if (!gets(inbuf)) break;
      ival=btoi(inbuf);
      printf("Your number in decimal is: %d\n",ival);
   }
}

Enter your binary number: EOF to exit

10110101           |Your number in decimal is: 181
11111111           |Your number in decimal is: 255
100000001          |Your number in decimal is: 257
10000000000000001  |Your number in decimal is: 1
```

See also: atoi(), atol(), otoi(), otol(), xtoi(), xtol()

This is a generalized machine specific assembly language and system SVC interface routine.

```
#include <z80regs.h>
int call( address, reg );
   char *address; union REGS *reg;

   address   is the machine address or SVC to call.

   reg       is a pointer to a structure containing the machine's register
             contents.
```

Description

A generalized assembly language interface routine, call(), is available in the installation library. "Reg" is either an integer array of dimension 6 which should contain the quantities you want placed into the register pairs AF, BC, DE, HL, IX, and IY for regs[0]-regs[5] respectively, prior to calling the routine at location "address", or "reg" is a pointer to a structure which contains the quantities to be placed into the machine's registers. The "reg" array or structure will be loaded with the register contents that existed upon return from the called routine.

On SuperVisor Call accessible systems, an "address" value of less than 256 will be interpreted as an SVC reference in lieu of a CALL address. The call() function will then use the "address" value as the SVC number.

A union which defines the word and byte structures for the Z-80 register set is provided in the z80regs header file. Use of the union is demonstrated extensively in the example below.

Return Code

The return code will be FALSE (0) if the machine's Z-flag is set upon return from the called routine; otherwise, the return code is TRUE (1).

Example

A rather long example which illustrates the use of the REGS union in various invocations of call() appears on the following page. Note that the example is designed for an SVC-driven operating system.

```
#include <z80regs.h>
#define CKDRV  33
#define DATE   18
#define DODIR  34
#define DSPLY  10
#define TIME   19
#option INLIB

int rc,d;
char *ptr, buf[100];
union REGS reg;

main()
{
   puts("Test of call()...");
   for ( d=0; d<8; ++d )
   {
      reg.C=d;       /* Set C to the drive number */
      rc=call(CKDRV,&reg);
      printf("Drive %d %s%s\n",d,rc?"not ":"","ready");
      if (reg.F&1) puts("Drive is write protected");
   }
   wait();
   dt(DATE,"Date");
   dt(TIME,"Time");
   reg.HL="This is a message\n";
   call(DSPLY,&reg);
   wait();
   reg.B=4; reg.C=0;     /* DODIR; function=4, drive=0 */
   reg.HL=buf;
   call(DODIR,&reg);
   strmid(buf+80,buf,0,8);
   strmid(buf+90,buf,8,8);
   printf("%s %s free: %dK\n\n",buf+80,buf+90,buf[18]+(buf[19]<<8));
   reg.BC=0;             /* DODIR; function=0, drive=0 */
   call(DODIR,&reg);
   wait();
}

wait()
{
   puts("\nHit any key");
   getchar();
   clscrn();
}

dt(addr,str)
   char *addr,*str;
{
   reg.HL=buf;
   call(addr,&reg);
   *(ptr=reg.HL)=0;
   printf("%s: %s\n",str,buf);
}

clscrn()
{
   fputs("\x1c\x1f",stdout);
}
```

This function is used to allocate a zeroed memory block.

```
char *calloc ( nelem, elsize);
   unsigned int nelem, elsize;

   nelem     is the number of elements of "elsize" to be allocated.

   elsize    is the size of an element in bytes; use sizeof(elem) for
             portability.
```

Description

Calloc() is used to dynamically allocate memory during program execution. The complementary function, free(), is used to release memory allocated through calloc(). Calloc may be used to get table or buffer space when the amount of memory space available is unknown, or when the program needs to dynamically allocate space for an array and the space must be filled with binary zeroes.

Return Code

If a memory block has been allocated, the value returned is a pointer to the memory block (pointer to char). If insufficient memory is available to satisfy the allocation, calloc() will return NULL (0).

Warnings

The program must not access memory outside of the area allocated. File access routines use alloc() and free() to establish and release File Control Areas (FCA's). The programmer cannot assume that memory not allocated is free for use, since later file opens may cause memory overlays. It is advised that the programmer always use the supplied dynamic allocation functions for memory accessing.

Example

```
if ((symtab = calloc(MAXSYMS,sizeof(SYMBOL))) == NULL)
   abend("not enough memory");
glbptr = startglb = symtab;
```

See also: alloc(), brk(), malloc(), realloc(), and sbrk()

This function obtains the ceiling of a double.

```
#include <math.h>
double ceil( argx );
   double argx;

   argx      is the double whose ceiling is desired.
```

Description

The ceiling of x is the smallest integer value not less than x; it is returned as a double.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1,d2;
main()
{
   puts("Ceil: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf))
         break;
      errno=0;
      d2 = ceil(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}
```

```
Ceil: enter your number: EOF to exit
-1.7 |d1 = -1.7; d2 = -1; errno = 0
1.7  |d1 = 1.7; d2 = 2; errno = 0
```

See also: fabs(); floor(), fmod()

This function is used to detect whether or not a character is available from an input file stream.

```
int checkc( stream );
   FILE *stream;

   stream    the file pointer of the stream as returned from a successful
             fopen().
```

Description

For certain types of applications, it is useful to know if a character is available for input prior to requesting the input via getc() [or other character input function]. For instance, a terminal program usually needs to toggle between scanning for standard input and input from a communications line. Since getc() will wait until a character is available, it can lock up such a program until a character is available from the stream. With checkc(), you can refrain from invoking getc() until a character is known to be available.

Return Code

There are three possible return codes. TRUE is returned if a character is available, or an EOF is detected from the file stream. FALSE is returned if a character is not available. EOF is returned if the file pointer references a closed file or a prior EOF had been detected.

Warnings

The stream pointer passed as the argument of the function must be one obtained from fopen() or must be the standard input device [stdin]. Also, checkc() uses ungetc() to store any input character; therefore, you should refrain from invoking ungetc() immediately following a checkc() invocation.

Example

```
while (TRUE)
{
   if (checkc(stdin))
      getterm();
   else if (checkc(comm))
      getcomm();
}
```

See also: putc(), putchar(), getc(), getchar(), ungetc(), ungetch()

This function is used to plot a circle (or close to it).

```
#option INLIB
int circle( funcod, x1, y1, r1 );
   int funcod, x1, y1, r1;

   funcod    an operation code to set (1) or reset (0) the pixels involved in
             the geometric plot.

   x1,y1     the coordinate of the circle's center.

   r1        the radius of the circle in "y" units.
```

Description

The "circle()" function will plot a circle at coordinate center point (x1,y1) of radius r1. The integer value, "r1", specifies the radius of the circle. Since block graphics are generally taller than their width, it is necessary to specify the radius in units of either "x" or "y". Within these plotting functions, "r1" is a value representing the radius in "y" units.

Virtual Points

The concept of virtual points is an important one. What it means is that your plotting routines do NOT have to limit themselves to the CRT image area. For example, a circle(1,0,0,20); function describes a circle about the origin. This means that a portion of the circle would be plotted off of the CRT image. The plotting functions permit your arguments to describe such "virtual" images; however, any portion of the geometric shape that would be outside of the CRT image area is inhibited. Thus, in the above example, only a portion of a circle (an arc) will be plotted.

Return Code

A return code of minus one (-1) indicates that the coordinate point (x1,y1) or a portion of the plot is out of range (i.e. virtual and does not appear in the CRT image). A minus three (-3) will be returned if the function code passed is invalid (not in the range <0-1>).

Example

```
circles(sw)   /* concentric circles */
{
   int r;
   for (r=1;r<36;r+=3)
      circle(sw,79,35,r);
}
```

See also: box(), line(), pixel()

This function is used to clear the end-of-file condition detected on an input stream.

```
int cleareof( stream );
   FILE *stream;

   stream    the file pointer of the stream as obtained from fopen().
```

Description

When an end-of-file condition is detected on an input stream (say standard input, for instance), a flag bit is set in the File Control Area flag to indicate this condition. Subsequent requests to input a character from that file stream will always return a constant EOF. The cleareof() function can be used to reset that EOF condition flag. It is useful primarily to clear the EOF condition established by the <BREAK> key for standard input.

Return Code

If the file pointer references a closed file stream or the end-of-file bit is not set in the file control area, cleareof() will return EOF as defined in the stdio header file. A return code of NULL (0) indicates successful clearing of the flag.

Warnings

The file pointer passed as the argument of the function must be one obtained from fopen() or one of the standard devices [stdin, stdout, or stderr].

Example

```
if (getchar() == EOF )
{
   cleareof(stdin);
   break;
}
```

See also: clearerr(), errno(), ferror(), perror(), sys_errlist()

This function is used to clear any error condition detected on an input or output stream.

```
int clearerr( stream );
   FILE *stream;

   stream    the file pointer of the stream as obtained from fopen().
```

Description

When an error condition is detected on a stream or block device, a flag bit is set in the file control area to indicate this error condition and the DOS error number is stored in the File Control Area for access by ferror(). The UNIX error number associated with the error will also be stored in the global error variable, errno. The EOF flag bit is also set. Subsequent requests to input a character from that file stream will always return a constant EOF. The clearerr() function can be used to reset the ERROR and EOF condition flags.

Return Code

If the file pointer references a closed file, cleareof() will return EOF as defined in the stdio header file. A return code of NULL (0) indicates successful clearing of the flag.

Warnings

The stream pointer passed as the argument of the function must be one obtained from fopen() or one of the standard devices [stdin, stdout, or stderr].

Example

```
if (c!=putchar(c))
{
   errnum = ferror(stdout);
   clearerr(stdout);
   break;
}
```

See also: cleareof(), errno, ferror(), perror(), sys_errlist()

This function is used to close a block device which has been obtained by a call to either open(), creat(), dup(), dup2(), or fcntl().

```
int close( fildes );
   int fildes;

   fildes    the file descriptor of the block device.
```

Description

Close() is used to close an open file and to free the File Control Area (FCA) and I/O buffer for subsequent re-use. The file descriptor passed to close must have been obtained from either open(), dup(), dup2(), fcntl(), or creat(). In MC, exit() also closes files; however, the programmer should use close() on opened files to ensure compatibility and portability.

Return Code

A return code of NULL (0) indicates successful closing of the file. If any error is detected, EOF (-1) will be returned.

Example

```
creat_file(name) char *name;
{
   int i,fd;
   if ((fd=creat(name,0777))==EOF) open_error(name);
   for (i=0;i<10000;i++)
   {
      if ((write(fd,itoa(i,record),10))!=10)
      {
         printf("Error in writing %s\n",name);
         exit(-1);
      }
      cursor(10,3);            /* position cursor */
      printf("Writing record %5d\n",i);
    }
   close(fd);
}
```

See also: creat(), dup(), fcntl(), fdown(), fstat(), lseek(), open(), read(), seek(), tell(), write()

This function will exit the program and invoke a DOS command.

```
void cmdi( cmdstr );
   char *cmdstr;

   cmdstr    a pointer to the string which contains the DOS command which is
             to be executed.
```

Description

The cmdi() function will close all open files which have the "close on exec" flag set and then schedule the DOS execution of the command either pointed to by the argument or by the command contained in the string passed as the argument to the function. This could be used, for instance, to chain to another C program.

Warnings

For portability across C compilers, the UNIX functions execv() or execl() should be used in lieu of cmdi().

Example

```
char buf[81];      /* string space for command string */
main()
{
   puts("Test of cmdi()\n");
   fputs("Enter command: ",stdout);
   gets(buf);      /* Get DOS command from user */
   cmdi(buf);      /* Exit this program and execute the command */
   puts("This string will not be written");
}
```

See also: system(), execl(), execv()

This function is used to obtain the double precision cosine.

```
#include <math.h>
double cos( argx );
   double argx;

   argx      is the double expressed in radians whose cosine is to be
             determined.
```

Description

This function will obtain the double precision cosine of its argument; the argument must be expressed in radian measure.

Example

```
#include stdio
#include math
char inbuf[81]; double d1,d2;
main()
{
   puts("Cos: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      errno=0;
      d2 = cos(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}

Cos: enter your number: EOF to exit
-1.75 |d1 = -1.75; d2 = -0.178246; errno = 0
1.75  |d1 = 1.75; d2 = -0.178246; errno = 0
.5    |d1 = 0.5; d2 = 0.877583; errno = 0
```

See also: acos(), asin(), atan(), cosh(), sin(), sinh(), tan(), tanh(), fcos()

This function is used to obtain the hyperbolic cosine of a double.

```
#include <math.h>
double cosh( argx );
   double argx;

   argx      is the double expressed in radians whose hyperbolic cosine is to
             be determined.
```

Description

This function will obtain the double precision hyperbolic cosine of its argument. The hyperbolic cosine of x is defined as [[exp(x)-exp(-x)]/2].

Example

```
#include stdio
#include math
char inbuf[81]; double d1,d2;
main()
{  puts("Cosh: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      errno=0;
      d2 = cosh(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}

Cosh: enter your number: EOF to exit
0  |d1 = 0; d2 = 1; errno = 0
.5 |d1 = 0.5; d2 = 1.12763; errno = 0
10 |d1 = 10; d2 = 11013.2; errno = 0
-10 |d1 = -10; d2 = 11013.2; errno = 0
```

See also: acos(), cos(), exp(), sinh(), tanh()

This function is provided to create new files or to rewrite old files.

```
int creat( path, pmode);
   int pmode;
   char *path;

   path      a pointer to the string containing the file specification.

   pmode     the file's assigned protection mode.
```

Description

If the specified file must be created, it will be created with the protection mode specified by the "pmode" argument. If the file is existing, creat() will truncate it to a length of zero. In either case, the access "OFLAG" will be established as O_WRONLY. To maintain portability with the UNIX file system, pmode is interpreted as a nine-bit integer specifying read, write, and execute permission for the owner of the file [bits 8-6], for the owner's group [bits 5-3], and for all others [bits 2-0]. Only the "all others" 3-bit field is used to establish the file protection under LDOS/TRSDOS. This is translated as follows:

read	write	execute	LDOS/TRSDOS	read	write	execute	LDOS/TRSDOS
0	0	0	NO ACCESS	1	0	0	READ
0	0	1	EXEC	1	0	1	READ
0	1	0	WRITE	1	1	0	WRITE
0	1	1	WRITE	1	1	1	FULL

If the file specification included a password field, then this password will apply only to the OWNER's access (UPDATE password for LDOS 5.1). The USER's access field (ACCESS password for LDOS 5.1) will be changed to blanks.

Return Code

If the file is properly created, its file descriptor will be returned. If an error is detected in creating the file, creat() will return EOF (-1).

Example

```
creat_file(name) char *name;
{
   int fd;
   if ((fd=creat(name,0777))==EOF) open_error(name);
   if ((write(fd,header,RECLEN))!=RECLEN)
      printf("Error in writing %s\n",name);
   close(fd);
}
```

See also: close(), open()

This function is used to obtain the ASCII 26-character time string.

```
char *ctime( clock );
   long *clock;

   clock     is a pointer_to_long which contains the time as retrieved by
             time().
```

Description

This function converts the UNIX time stored as a long integer obtained from a function such as time() into a 26-character string of the form:

```
      Tue Oct 22 11:09:08 1985\n\0
```

All of the fields of the time string have a constant width.

Example

```
#option INLIB
main()
{
   long clock;
   long time();
   char *ctime();

   clock = time( (long *) 0);
   printf("At the tone, the time will be [%ld]: ",clock);
   puts(ctime(&clock));
   system("date");
   system("time");
}
```

```
At the tone, the time will be [500036062]: Tue Nov 05 10:54:22 1985
Tue, Nov  5, 1985
10:54:39
```

See also: asctime(), localtime(), time()

This function is used to obtain the position of the CRT's cursor.

```
#option INLIB
unsigned curpos();
```

Description

The current location of the cursor can be recovered with the curpos() function. It returns the cursor position as an encoded value. The cursor row is in the high-order byte while the cursor column occupies the low-order byte of the integer return code.

Example

```
#option INLIB
main()
{
   unsigned int crtrow, crtcol;
   puts("\x1c\x1f");                /* home, clear, CR */
   crtrow=(crtcol=curpos())>>8;     /* get curpos; calc row */
   crtcol&=0x0ff;                   /* calc column */
   printf("crt: row=%d, col=%d",crtrow,crtcol);
}

crt: row=1, col=0
DOS Ready
```

See also: cursor()

This function is used to move the cursor to a designated CRT position.

```
#option INLIB
int cursor( col, row );
   int col, row;

   col    is the column of the new position.

   row    is the row of the new position.
```

Description

To reposition the cursor on the CRT screen, use the cursor() function. The cursor is re-positioned to the location identified by the arguments.

Return Code

If the position that would result is not on the CRT screen, a range error (-1) is returned. "col" must be in the range, <0-X>, while "row" must be in the range, <0-Y>. For 64 x 16 screen sizes, the X and Y limits are 63 and 15. For 80 x 24 screen sizes, the X and Y upper limits are 79 and 23 respectively.

Example

```
#option INLIB
write_file(fildes) int fildes;
{
   int i;
   for (i=0;i<10000;i++)
   {
      if ((write(fildes,itoa(i,record),10))!=10)
         return i;
      cursor(10,3);     /* position cursor to row 3, col 10 */
      printf("Writing record %5d\n",i);
   }
   return NULL;
}
```

See also: curpos()

These functions obtain a fast divide_by_2 and a divide_by_2^30 of a double.

```
#include math.h
double _ddv2( argx );
   double argx;

double _ddv230( argx );
   double argx;

   argx      is the argument for which the divide operation is desired.
```

Description

These two functions are special and non-standard. Use them only if you are not interested in portability. They perform a fast divide operation on a double. _ddv2() divides a double by 2.0. _ddv230 divides a double by 2^30.

Example

```
#include <math.h>
double ldexp(argfr, exp) double argfr; register int exp;
{
   static double huge, fr;
   static int neg;
   static int i;
   double frexp(), _ddv230();

   fr = argfr;
   huge = HUGE_VAL;
   neg = 0;
   if (fr < 0.0) { fr=-fr; neg=1; }
   fr = frexp(fr, &i);
   while (fr < 0.5) { fr *= 2.0; --i; }
   exp += i;
   if (exp > 127) { errno=ERANGE; return neg ? -huge : huge; }
   if (exp < -127) { errno=ERANGE; return 0.0; }
   while (exp > 30)
   {   fr *= (1L << 30L);
      exp -= 30;
   }
   while (exp < -30)
   {   fr = _ddv230(fr);   /* fr /= (1L << 30L) */
      exp += 30;
   }
   if (exp > 0) fr *= (1L << exp);
   if (exp < 0) fr /= (1L << -exp);
   if (neg) fr = -fr;
   return fr;
}
```

This function obtains the truncated value of a double.

```
#include math.h
double dfix( argx );
   double argx;

   argx      is the double value to truncate.
```

Description

The truncated value is obtained by dropping all digits to the right of the decimal point so the result is a whole number. If argx is negative, dfix(argx) is equivalent to dint(x)+1.0.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1,d2;
main()
{
   puts("Dfix: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      d2 = dfix(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g\n",d1,d2);
   }
}

Dfix: enter your number: EOF to exit
-1.275 |d1 = -1.275; d2 = -1
1.275  |d1 = 1.275; d2 = 1
```

See also: dint(), floor(), ceil()

This function obtains the integer part of a double. It is identical to the standard math function, floor().

```
#include math.h
double dint( argx );
   double argx;

   argx      is the double for which the integer part is desired.
```

Description

This function obtains the largest integer not greater than "argx". The "dint()" function has been provided only for the sake of a name compatibility with other languages. For the sake of portability among C compilers, it is recommended that you use the equivalent function, floor().

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1,d2;
main()
{
   puts("Dint: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      d2 = dint(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g\n",d1,d2);
   }
}

Dint: enter your number: EOF to exit
-1.275 |d1 = -1.275; d2 = -2
1.275  |d1 = 1.275; d2 = 1
```

See also: dfix(), floor(), ceil()

This function obtains the integer sign of a double.

```
#include <math.h>
int dsgn( argx );
   double argx;

   argx      is the double for which the sign is desired.
```

Description

This function returns AS AN INTEGER, the state of the sign of the argument.

Return Code

The returned value will be minus one (-1) if argx is negative, zero (0) if argx is zero, and plus one (1) if argx is positive other than zero.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1; int sign;
main()
{
   puts("Dsgn: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      sign = dsgn(d1=atod(inbuf));
      printf("d1 = %g; sign = %d\n",d1,sign);
   }
}

Dsgn: enter your number: EOF to exit
-3.75    |d1 = -3.75; sign = -1
0        |d1 = 0; sign = 0
14.7e-15 |d1 = 1.47e-14; sign = 1
```

See also: fsgn()

This function converts a double to string form in "e" notation.

```
#include math.h
char *dtoa( argx, string );
   double argx;
   char *string;

   argx      is the double to convert to string form.

   string    is a character array of length 14
```

Description

This function will convert a double to string form using the format:

sd.ddddddEnee

where "s" represents a sign character (blank or minus), "d" represents a decimal digit, "n" represents an exponent sign (plus or minus), and "e" represents an exponent decimal digit.

Return Code

A pointer to the string is returned so that the function may be used as a "string" argument.

Example

```
#include stdio.h
#include math.h
char string[14];
main()
{
   double loop;
   for (loop=1.25e1; loop < 10.0e1; loop+=2.75e1)
      puts(dtoa(loop,string));
}
```

```
 1.250000E+01
 4.000000E+01
 6.750000E+01
 9.500000E+01
```

See also: sprintf()

These functions duplicate an open file descriptor.

```
int dup( fildes );
   int fildes;

int dup2( fildes , fildes2);
   int fildes, fildes2;

   fildes       is the file descriptor to duplicate.

   fildes2      is the file descriptor to use for the duplication of fildes.
```

Description

These two functions allocate another file descriptor synonymous with the original. The "fildes" argument must be a file descriptor obtained from an open() or creat() call or one of the standard file descriptors: STDIN, STDOUT, or STDERR.

The dup() function will search for a file descriptor to utilize, if one is available. The dup2() function will reuse the descriptor passed as the fildes2 argument. If fildes2 refers to an already opened file, that file will first be closed prior to duplicating the requested descriptor.

Return Code

EOF will be returned if the fildes argument passed is invalid or there are no more file descriptors left available. In either case, the global error variable, errno, will contain the UNIX error number associated with the error.

Example

```
#include stdio.h
char message[] = "Writing to dup'd file descriptor\n";
main()
{
   int fildes;
   if (( fildes = dup(STDOUT))==EOF)
      {
         fputs("dup() error\n",stderr); exit(-1);
      }
   write(fildes,message,strlen(message));
}

Writing to dup'd file descriptor
```

See also: creat(), fcntl(), open(), perror(), write()

This function exits the program and causes execution of a command string built from the argument list.

```
int execl(path, arg0, arg1, ..., argn, 0);
   char *path, *arg0, *arg1, ..., *argn;

   path      is a pointer to the name of the program which is to be invoked.

   argx      are pointers to character strings which will be constructed as
             arguments to the program being invoked.
```

Description

The execl() function will build a command string composed of the path name and the arguments passed to execl(). Files in the current program will remain open unless the close_on_exec flag is set [see fopen()]. The current program then terminates and the resulting command string is passed to the DOS for execution.

Conventionally, the "arg0" argument is the same as the path name.

Return Code

If the resulting command string would exceed the length supported by the DOS, execl() will return EOF and the global error variable, errno, will be set to E2BIG; otherwise, execl() will not return.

Example

```
#include stdio.h
char  path[]="dir", arg0[]="dir", arg1[]="sys:0", arg2[]="(sys,inv,a=)";
main()
{
   puts("Test of execl()\n");
   if (execl(path,arg0,arg1,arg2,0)==EOF)
      puts("DOS command line is too long");
}
```

```
Test of execl()

Drive :0 VR0005A 153 Cyl, Hard, Free =  732.00K/ 2448.00K, Date 12-Apr-85

SYS0/SYS SIP    SYS1/SYS SIP+   SYS10/SYS SIP   SYS11/SYS SIP
SYS12/SYS SIP   SYS13/SYS SIP   SYS2/SYS SIP    SYS3/SYS SIP
SYS4/SYS SIP    SYS5/SYS SIP    SYS6/SYS SIP    SYS7/SYS SIP
SYS8/SYS SIP    SYS9/SYS SIP    SYSTEM/JCL +    SYSUAF/DAT P+
```

See also: execv(), cmdi()

This function is used to invoke an overlay of the executing C program.

```
int execovl(ovnum, ovargc, ovargv);
   char ovnum; int ovargc; char *ovargv[];

   ovnum     the requested overlay number in ASCII.

   ovargc    the quantity of args to be passed to the overlay in the ovargv
             array.

   ovargv    a pointer to the argument vector which is to be passed to the
             overlay.
```

Description

The execovl() function is used to invoke an overlay where a program environment has been developed around a root module and one or more overlay modules. The arg vector "ovargv" MUST have ovargc pointers in it. The "ovargc" and "ovargv" will be passed as arguments to the overlay's ovmain() function in that order. Note that "ovargc" and "ovargv" are optional.

The generation of the root and overlay files requires the use of the MLINK linker provided with the MRAS assembler development system. Since the design of a C program environment which uses overlays is a complex procedure, its details will be discussed in the "Advanced Topics" chapter.

Return Code

A NULL will be returned if the requested overlay executed, otherwise EOF will be returned and errno will be appropriately set. Note that any return code from ovmain() is ignored; you can use public memory location(s) in the root for communications from the overlay.

Example

```
/* OVTEST/CCC - sample test root module */
extern int execovl(), printf();
int ovargc = 3;
char *ovargv[3] = { "ovtest", "this is overlay" };

main()
{
   ovargv[2] = "1";
   if (execovl('1', ovargc, ovargv)) error();

   ovargv[2] = "2";
   if (execovl('2', ovargc, ovargv)) error();
}

error()
{
   perror("ovtest");
    exit(1);
}

/* OVERLAYT/CCC - sample overlay for OVTEST */
ovmain(argc, argv) int argc; char *argv[];
{
   printf("%s: %s %s\n", argv[0], argv[1], argv[2]);
}
```

This function exits the program and causes execution of a command string built from the array of pointers to strings.

```
int execv(path, argv);
   char *path, *args[];
```

path	is a pointer to the name of the program which is to be invoked.
args[]	are pointers to null-terminated character strings which will be used as arguments to the program being invoked.

Description

The execv() function will build a command string composed of the path name and the arguments passed to execv(). The array of pointers is terminated with a NULL pointer. Files in the current program will remain open unless the close_on_exec flag is set [see fopen()]. The current program then terminates and the resulting command string is passed to the DOS for execution.

Conventionally, args[0] is the same as the path name.

Return Code

If the resulting command string would exceed the length supported by the DOS, execv() will return EOF and the global error variable, errno, will be set to E2BIG; otherwise, execl() will not return.

Example

```
#include stdio.h
char *args[] = {"dir","zzzzz","(n)",0 };
main()
{
   puts("Test of execv()\n");
   if (execv(args[0],args)==EOF)
      puts("Command line too long!");
}
```

```
Test of execv()

Drive :0 VR0005A  153 Cyl, Hard, Free =  732.00K/ 2448.00K, Date 12-Apr-85
Drive :1 VR0005B  153 Cyl, Hard, Free =  598.00K/ 2448.00K, Date 12-Apr-85
Drive :2 FIXES02   40 Cyl, DDEN, Free =  130.50K/  180.00K, Date 05-Sep-85
Drive :3 MCTEST05  40 Cyl, DDEN, Free =  169.50K/  180.00K, Date 06-Nov-85
Drive :6 VR0006A  153 Cyl, Hard, Free =  428.00K/ 2448.00K, Date 23-Sep-85
Drive :7 VR0006B  153 Cyl, Hard, Free = 1520.00K/ 2448.00K, Date 23-Sep-85
```

See also: execl(), cmdi()

This function is used to exit your program and return to DOS.

```
void exit( retcod);
   int retcod;

void _exit( retcod );
   int retcod;

   retcod    is the return code to be passed to the DOS.
```

Description

Exit() and _exit() allow the user to exit cleanly from a program and control the consequences of exiting. Passing a zero (0) for the "retcod" argument to exit() indicates normal program termination, causing exit() to take the DOS normal exit. If a non-zero retcod is passed, exit() will take the error entry into DOS, thus aborting any Job Control Language processing in effect.

If the terminating program was invoked by the system() function, the value passed to exit() will be the value returned to the calling program from system(). An exception is if a negative value is passed to exit(), in which case a negative one (-1) is returned to cmd().

Exit() closes all open files before returning to DOS; the _exit() function does not.

MC generates an automatic exit(0) when the terminating brace of main() is reached; thus, your program need not explicitly call exit() at that point.

Return Code

Exit() does not return to the caller.

Warnings

For compatibility with other C language systems, the programmer should not depend on exit() to close the program's files (other than standard I/O files).

See also: close(), fclose()

This function obtains the exponential of a double.

```
#include <math.h>
double exp( argx );
   double argx;

   argx      is the double for which the exponential is desired.
```

Description

Exp() returns the exponential function of x (e raised to the x power).

Return Code

If the resulting value would overflow, exp() returns HUGE_VAL and the global error variable, errno, is set to ERANGE. An error message indicating the overflow range error is printed to standard error output. This error handling may be changed by the use of matherr().

Example

```
#include stdio
#include math
char inbuf[81]; double d1,d2;
main()
{
   puts("Exp: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      errno=0;
      d2 = exp(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}
```

```
Exp: enter your number: EOF to exit
1.0  |d1 = 1; d2 = 2.71828; errno = 0
14.8 |d1 = 14.8; d2 = 2.67645e+06; errno = 0
150  |overflow range error
     |d1 = 150; d2 = 1.70141e+38; errno = 71
```

See also: log(), matherr(), errno

This function obtains the absolute value of a double.

```
#include <math.h>
double fabs( argx );
   double argx;

   argx      is the double for which the absolute value is desired.
```

Description

This function returns the absolute value of its argument (i.e if argx is positive, argx is returned; if argx is negative, -argx is returned).

Return Code

The absolute value is returned.

Example

```
#include stdio
#include math
char inbuf[81]; double d1,d2;
main()
{
   puts("Fabs: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      errno=0;
      d2 = fabs(d1=atod(inbuf));
      printf("d1 = %g; d2 = %g; errno = %d\n",d1,d2,errno);
   }
}

Fabs: enter your number: EOF to exit
1.675384692 |d1 = 1.67538; d2 = 1.67538; errno = 0
-13.47e-3   |d1 = -0.01347; d2 = 0.01347; errno = 0
```

See also: abs(), fabsf(), labs()

This function is used to obtain the absolute value of a float.

```
#include <math.h>
float fabsf( xval );
   float xval;

   xval      is the float whose absolute value is to be determined.
```

Description

This is a non-standard function which will obtain the absolute value of a float. For portability, use the equivalent construct:

```
      (float)fabs((double)xval);
```

See also: abs(), fabs(), labs()

This function is used to obtain the arc tangent of a float.

```
#include <math.h>
float fatn( xval );
   float xval;

   xval      is the float whose arc tangent is to be determined.
```

Description

This is a non-standard function which will obtain the arc tangent of a float. For portability, use the equivalent construct:

```
(float)atan((double)xval);
```

See also: atan(), atan2(), ftan()

This function is used to close a file opened with fopen().

```
int fclose( stream );
   FILE *stream;

   stream    the file pointer obtained from fopen().
```

Description

Fclose() is used to close an open file and to free the File Control Area (FCA) and file buffer for later use. The stream file pointer passed to fclose must have been obtained from fopen(). In the MC file system, exit() also closes files; however, the programmer should use fclose() on files opened by the program to ensure compatibility and portability.

There is a limited number of files (determined by the MAXFILES compiler option), including standard files, that may be open at one time. Fclose() is used to free FCAs so an unlimited number of files may be accessed one after the other.

Return Code

The return code will be NULL if no error was detected in the closing operation (i.e. a successful close). If an error was detected during the close operation, then EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Warnings

The return code convention under MC follows the UNIX Seventh Edition and UNIX System V convention. As such, the return codes for success and failure are opposite as that returned under LC.

Example

```
if ( lastc!=0x1a )
   putout(0x1a);
fclose(fp1);
fclose(fp2);
puts("Files now closed");
exit(0);
```

See also: fcntl(), fdopen(), fopen(), freopen(), fflush()

This function provides certain control over files.

```
#include <fcntl.h>
int fcntl( fildes, cmd, arg );
   int fildes, cmd; "varying type" arg;

   fildes    is the file descriptor of the file you wish to control.

   cmd       is the specific fcntl() command (see text).

   arg       is specific to the fcntl() command.
```

Description

Fcntl() provides for certain control over open files. The file descriptor which identifies the file is one obtained from creat(), dup(), dup2(), fcntl(), open(), or one of the standard I/O devices: 0, 1, or 2. The "cmd" argument directs fcntl() to a specific operation. The commands available to fcntl() are defined in the "fcntl" header file, and are as follows:

F_DUPFD

This command will duplicate the file associated with fildes and return a new file descriptor having the following characteristics:

- is the lowest numbered available file descriptor greater than or equal to "arg";
- is the same open file as fildes;
- has the same access mode as fildes (read, write, or read/write);
- the close_on_exec flag associated with the new file descriptor is reset to keep the file open upon execution of execl() or execv() invocations.

F_GETFD

This command will get the status of the close_on_exec flag. If the low-order bit is a 0, the file will remain open across execl() and execv() calls; otherwise, the file will be closed upon execution of execl() or execv().

F_SETFD

This command will set the close_on_exec flag associated with fildes to the state of the low-order bit of arg, 0 or 1.

F_GETFL

This command obtains the file status flags (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC) identified in the "fcntl" header file.

F_SETFL

This command permits you to set the file status flags according to "arg". You may only change the following flags: O_RDONLY, O_WRONLY, or O_RDWR.

Return Code

If fildes is invalid, or if the argument passed as "arg" to F_SETFL is invalid, or if there is no new available file descriptor to return for the F_DUPFD command, EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Example

```
#include stdio.h
#include fcntl.h
struct { int mask; char *name; } oflags[] = {
    {O_RDONLY,"read_only"},
    {O_WRONLY,"write_only"},
    {O_RDWR,"read_write"},
    {O_NDELAY,"delay"},
    {O_APPEND,"append"},
    {O_CREAT,"creat"},
    {O_TRUNC,"truncate"},
    {O_EXCL,"exclusive"}
};
char *path="default/txt:7", sbuf[81], *itob();
main(argc,argv) int argc; char *argv[];
{
    int fildes, i, oflag;
    if (argc==2) path=argv[1];
    if ((fildes=open(path,O_RDWR|O_CREAT,0777))==EOF)
        { printf("Can't open %s\n",path); exit(-1); }
    else printf("Opened %s; fildes = %d\n",path,fildes);
    oflag = fcntl(fildes,F_GETFL,NULL);
    fputs("oflags = ",stdout);puts(itob(oflag,sbuf));
    for (i=0; i<8; i++)
        if (oflag&oflags[i].mask)
            { putchar(' '); fputs(oflags[i].name,stdout); }
    printf("\nc-o-e flag is %d\n",fcntl(fildes,F_GETFD,NULL)&1);
    close(fildes);
}
```

```
Opened default/txt:7; fildes = 3
oflags = 100100
 read_write creat
c-o-e flag is 1
```

See also: dup(), dup2()

This function is used to obtain the cosine of a float.

```
#include <math.h>
float fcos( xval );
   float xval;

   xval      is the float expressed in radians whose cosine is to be
             determined.
```

Description

This non-standard function will obtain the single precision cosine of its argument. For portability, use the equivalent form:

```
(float)cos((double)xval);
```

See also: acos(), asin(), fsin(), ftan(), sin(), tan()

This function is used to associate a stream with an open file descriptor.

```
FILE *fdopen( fildes, type );
   fildes, char *type;

   fildes    is the file descriptor of the open file.

   type      is a character string which indicates the access mode chosen from
             the following:

                 "r", "w", "a", "r+", "w+", or "a+".
```

Description

Fdopen() allows the programmer to initiate stream access of a given access type to a file opened with creat(), dup(), dup2(), fcntl(), or open(). "Fildes" is the file descriptor of the already opened file. "Type" points to a null-terminated string which defines the mode of access and must agree with the access specified when the file was opened. The allowable modes of access, which may be entered in upper or lower case, are:

"r"	open for reading;
"w"	truncate or create for writing;
"a"	append; open for writing at the end of the file, or create for writing;
"r+"	open for update (reading and writing);
"w+"	truncate or create for update;
"a+"	append; open or create for update (appending) at the end of the file.

When a file is opened for update, both reading and writing may be performed on the stream; however, writing may not be immediately followed by reading without an intervening fseek() or rewind(), and reading may not be followed by writing without an intervening fseek(), rewind(), or an input operation which encounters an end-of-file [first cleared by cleareof()];

Note that fdopen() is similar to fpup() in that a stream pointer associated with a file descriptor is returned; however, fdopen() permits you to alter the access mode of the stream.

Return Code

The stream file pointer is returned if no errors are detected in the fdopen() operation. If fildes does not refer to an open file descriptor or the access type requested does not agree with the access of the open file, NULL (zero) will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Warnings

When a file is opened for append, the file is positioned so that the first output operation will write to the end of the file. The output functions do not perform this positioning operation; therefore, do not use rewind() or fseek() with streams to which you wish to append.

Example

```
#include stdio.h
extern FILE *fdopen();
char message[] = "This is written to fdopen'd stream";
main()
{
   FILE *stream;
   if (!(stream=fdopen(STDOUT,"w")))
   {
      puts("fdopen error");
      exit(-1);
   }
   puts(message,stream);
   fclose(stream);
}
```

```
This is written to fdopen'd stream
```

See also: fopen(), freopen()

This function is used to convert a stream file pointer to a file descriptor. It is identical to fileno().

```
int fdown( stream );
   FILE *stream;

   stream       the file pointer of the stream.
```

Description

Sometimes it may be useful to cross between file stream I/O functions and block device I/O functions which use file descriptors. Since file streams are referenced with file pointers and block device files are referenced with file descriptors, the fdown() function may be used to obtain the file descriptor given a stream file pointer.

Return Code

If the stream file pointer is recognized as being invalid, EOF will be returned; otherwise, the needed file descriptor will be returned.

Warning

The stream file pointer passed as the argument of fdown() must be one obtained from fopen(), fdopen(), or the file pointer of a standard device (i.e. stdin, stdout, stderr).

Example

```
if (fd2 = fcntl( fdown(stdin),F_DUPFD,3)) == EOF)
    abend(NULL);
if (fdown(stderr) != STDERR)
    abend("internal error");
```

See also: fileno(), fpup()

This function is used to determine if the end-of-file has been sensed.

```
int feof( stream );
   FILE *stream;

   stream       the file pointer obtained from fopen().
```

Description

This function obtains the status of the EOF flag maintained in the File Control Area for an open file.

Return Code

If an end-of-file had been previously encountered, feof() will return a non-zero value; otherwise, NULL will be returned.

Example

```
#include stdio.h
main()
{
   int c; FILE *fp;
   if ((fp=freopen("tfeof/ccc","r",stdin))==NULL)
   {
      perror("main: freopen error"); exit(-1);
   }
   while ((c=getchar())!= EOF)
      putchar(c);
   puts( feof(stdin) ? "EOF received" : "no EOF yet");
   printf("DOS error number is %d\n",ferror(stdin));
   if (ferror(stdin))
      perror("main");
}

EOF received
DOS error number is 0
```

See also: ferror(), clearerr(), cleareof()

This function is used to obtain the last DOS I/O error number associated with a file stream.

```
int ferror( stream );
   FILE *stream;

   stream       the file pointer obtained from fopen().
```

Description

The MC file system maintains the number of the last error code associated with a file stream or file. This error code may be recovered via the ferror() function. When an error occurs, the global error variable, errno, will contain the UNIX error number associated with the DOS error. Consult your DOS manual for the meaning of each DOS error code.

Return Code

If no error has been encountered, NULL will be returned. If the file pointer references a closed file, then EOF will be returned. Otherwise, the return code will be the DOS error code.

Warnings

The stream file pointer passed as the argument to the function must be one obtained from fopen() or the stream pointer of any standard device (i.e. stdin, stdout, stderr). The error number storage is shared with the ungetc() one character buffer; thus, an ungetc() on the stream may wipe out the error number obtainable via ferror().

Example

```
#include stdio.h
main()
{
   int c; FILE *fp;
   if ((fp=freopen("tfeof/ccc","r",stdin))==NULL)
   {
      perror("main: freopen error");
      exit(-1);
   }
   while ((c=getchar())!= EOF) putchar(c);
   puts( feof(stdin) ? "EOF received" : "no EOF yet");
   printf("DOS error number is %d\n",ferror(stdin));
   if (ferror(stdin))
      perror("main");
}

EOF received
DOS error number is 0
```

See also: perror()

This function obtains the exponential of a float.

```
#include <math.h>
float fexp( argx );
   float argx;

   argx      is the float for which the exponential is desired.
```

Description

The non-standard function, fexp(), returns the exponential function of x (e raised to the x power) of the single precision argument (i.e. a float). For portability, use the equivalent construct:

```
(float)exp((double)argx);
```

See also: flog(), exp(), log()

This function obtains the truncated value of a float.

```
#include <math.h>
float ffix( argx );
   float argx;

   argx      is the float value to truncate.
```

Description

The truncated value is obtained by dropping all digits to the right of the decimal point so the result is a whole number. If argx is negative, ffix(argx) is equivalent to fint(x)+1.0. This is a nonstandard function. For portability, use the equivalent construct:

```
(float)dfix((double)argx);
```

See also: fint(), dfix, dint(), floor(), ceil()

This function is used to force a physical write of any buffered output to an output file stream and update the DOS directory.

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

   stream    the file pointer obtained from fopen().
```

Description

When program crashes occur, some output data written to a file stream may have been buffered in the stream's I/O buffer and not written to disk. The file's directory information also may not get updated. The fflush() function can be used to complete all physical output and update the stream's directory without actually closing the file (if invoked prior to such a program crash).

Return Code

The return code will be NULL if no error occurs during the flushing operation. If an error is detected, EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Example

```
#include <stdio.h>
char message[] = "Testing, 1, 2, 3... Hello, test\n";
main()
{
   FILE *stream, *fopen();
   if ((stream=fopen("testfile/dat:7","w"))==NULL)
      { perror("fopen error"); exit(-1); }
   fputs(message,stream); fputs(message,stream);
   if (fflush(stream)==EOF) perror("fflush error");
   puts("Program terminating through _exit()");
   _exit();    /* simulate error terminate w/o closing files */
}

Program terminating through _exit()

DOS Ready
list testfile/dat:7

Testing, 1, 2, 3... Hello, test
Testing, 1, 2, 3... Hello, test
```

See also: fopen(), fclose(), putc(), fwrite()

This function is used to obtain a character from a file stream.

```
int fgetc( stream );
   FILE *stream;

   stream    is the file pointer of the stream
```

Description

Fgetc() is used to input a single byte from a file stream. The MC library treats fgetc() as equivalent to getc(): both are functions.

Return Code

The return code is the integer value of the character input from the stream. If an end of file is encountered, then "EOF" (-1) is returned.

Warning

If the value returned by the function is to be stored before testing for end-of-file, it must be stored in an integer variable and not a char. If not, the end of file value will be truncated and will remain undetected, as a char of -1 does not equal an int of -1.

Example

```
#include <stdio.h>
filecopy(fp)
FILE *fp;
{   int c;
   while ((c = fgetc(fp)) != EOF)
      if (c != fputc(c,stdout))
         abort("Output file write error");
}
```

See also: getc(), getchar(), getw()

This function is used to get a buffered line from a file stream.

```
char *fgets( buf, maxlen, stream);
   char *buf; int maxlen; FILE *stream;

   buf       is a pointer to the character buffer.

   maxlen    is the maximum length of the input string.

   stream    is the stream pointer obtained from fopen().
```

Description

Fgets() is used to obtain a buffered line from a file. A file may be the console keyboard, the RS-232 interface, or any input device or disk file. Up to (maxlen) bytes will be placed in the buffer. Input is terminated when either a newline or end of file is encountered or maximum buffer size is reached. The newline character is not stripped from the input.

For compatibility with DOS Job Control Language files, keyboard line input is performed using the @KEYIN system call. Fgets() recognizes the BREAK key as the end of file from the keyboard if the O_BREAK flag is set via the option() function. Also, the maximum number of keyboard characters accepted for input will be the minimum of (maxlen,255).

Return Code

If an error is encountered, NULL will be returned and the global error variable, errno, will contain the UNIX error number associated with the error; otherwise, a pointer to "buf" is returned.

Example

```
#include stdio.h
main()
{
   int i; FILE *stream, *fopen(); char buffer[81];
   if ((stream=fopen("testfile/dat:7","w+"))==NULL)
   {
      perror("fopen error");
      exit(-1);
   }
   for (i=1; i<9999; i*=10)
      fprintf(stream,"Record %d is here\n",i);
   rewind(stream);
   while ((fgets(buffer,80,stream))!=NULL)
      fputs(buffer,stdout);
}
```

```
Record 1 is here
Record 10 is here
Record 100 is here
Record 1000 is here
```

See also: fputs(), gets(), puts()

This function will propagate a character throughout a memory region.

```
void fill( buffer, count, cval );
   char *buffer, cval; int count;
```

buffer	is the address pointing to the start of the memory region you wish to populate.
count	is the number of bytes to populate.
cval	is the character to populate throughout the region of memory.

Description

The fill() function will propagate the character, "cval", into the memory "buffer" for "count" bytes. If "cval" is passed as an integer value, the low-order byte is used for the propagation.

Warning

Note that although the fill() function is contained in LIBC, it is not a standard function. For purposes of portability across C compilers, use the equivalent standard function, memset().

Example

```
char *getmem(size)
unsigned size;
{
    if (!(mptr=alloc(size))
        return NULL;
    fill(mptr,size,NULL);
    return mptr;
}
```

See also: memset()

This function is used to obtain the integer part of a float.

```
#include <math.h>
   float fint( xval );
float xval;

   xval      is the float whose integer part is to be determined.
```

Description

This is a non-standard function which will obtain the integer part of a float. For portability, use the equivalent construct:

```
      (float)floor((double)xval);
```

See also: floor(), ffix()

This function is used to obtain the natural log of a float.

```
#include <math.h>
float flog( xval );
   float xval;

   xval      is the float whose log is to be determined.
```

Description

This is a non-standard function which will obtain the natural logarithm (base e) of a float. For portability, use the equivalent construct:

```
(float)log((double)xval);
```

See also: log(), fexp()

This function obtains the integer part of a double.

```
#include math.h
double floor( argx );
   double argx;

   argx      is the double for which the integer part is desired.
```

Description

This function obtains the largest integer not greater than "argx".

Example

```
#include stdio
#include math
char inbuf[81]; double d1,d2;
main()
{   puts("floor: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        d2 = floor(d1=atod(inbuf));
        printf("d1 = %g; d2 = %g\n",d1,d2);
    }
}

floor: enter your number: EOF to exit
-1.275 |d1 = -1.275; d2 = -2
1.275  |d1 = 1.275; d2 = 1
```

See also: dfix(), dint(), ceil()

This function is used to perform a floating point modulo division.

```
#include <math.h>
double fmod( xval, yval );
   double xval, yval;

   xval      is the double numerator.

   yval      is the modulo divisor.
```

Description

Fmod() returns the floating point remainder of the division of xval by yval. Zero is returned if xval is zero or if xval/yval would overflow.

Example

```
#include stdio.h
#include math.h
main()
{
    double numer, denom;
    for ( numer=10.1, denom=3.0; numer>0.0; numer--)
        printf("%g   ",fmod(numer,denom));
}

1.1   0.1   2.1   1.1   0.1   2.1   1.1   0.1   2.1   1.1   0.1
```

This function is used to open a file or device for stream I/O.

```
FILE *fopen( path, type );
   char *fspec, char *type;

   path      points to a character string which contains the specification of
             the file.

   type      is a character string which indicates the access mode chosen from
             the following:

                "r", "w", "a", "r+", "w+", or "a+".
```

Description

Fopen() allows the programmer to initiate access to a file. Except for standard input, output, and error files which are automatically opened, all files must be opened using fopen() [or various other file opening functions]. "Path" points to a file specification string identifying the file to open. "Type" points to a null-terminated string which may be in upper or lower case and which defines the mode of access . The allowable modes are:

"r"	open for reading;
"w"	truncate or create for writing;
"a"	append; open for writing at the end of the file, or create for writing;
"r+"	open for update (reading and writing);
"w+"	truncate or create for update;
"a+"	append; open or create for update (appending) at the end of the file.

The file pointer is used whenever access to the opened file is needed. If zero is returned, an error occurred during the open process.

When a file is opened for update, both reading and writing may be performed on the stream; however, writing may not be immediately followed by input without an intervening fseek() or rewind(), and reading may not be followed by writing without an intervening fseek(), rewind(), or an input operation which encounters an end-of-file [first cleared by cleareof()];

Return Code

The stream file pointer is returned if no errors are detected in the open operation. If an error is detected during the open operation, NULL (zero) will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Warnings

Opening the same file for both input and output with two or more calls to fopen() should NOT be done. If the file is accessed in this manner, it will create unpredictable results, possibly causing loss of file integrity. A file may be "fopened" for both read and write using the "r+", "w+", or "a+" access modes.

When a file is opened for append, the file is positioned so that the first output operation will write to the end of the file. The output functions do not perform this positioning operation; therefore, do not use rewind() or fseek() on streams to which you wish to append.

Example

```
getfile(path) char *path;
{   FILE *fp;
    if ((fp=fopen(path,"r")) == NULL)
        {
            printf("Open error - %-20s\n",path);
            exit();
        }
    else return fp;
}
```

Numerous other illustrations of fopen() appear in the many other examples throughout this chapter.

See also: fclose(), fdopen(), freopen(), open()

These functions create a formatted image for output.

```
#include <math.h>            ...optional
int fprintf( stream, control, arg1, arg2, ...);
   FILE *stream; char *control; args SEE TEXT

   stream    is the stream file pointer for output.

   control   is a string containing transparent printing characters and
             conversion specifications.

   argn      are arguments to be formatted for the output print image as
             specified by the control.
```

Description

The fprintf() function is used to write a formatted output image to a file stream identified by the "stream" file pointer. The function contained in LIBC does not include support for "e", "f", or "g" translation formats. The math library package includes a version of fprintf() which supports all standard format translations and is accessible when your program includes the preprocessor directive, "#include <math.h>".

The specifications for formatting the output are determined by the character string, "control". This string will contain ordinary characters copied directly to the output image and/or specifications denoting the field conversions of all arguments. The conversion specifications take the form of:

```
%{flags}{width}{.prec}char
```

As can be noted, the specification is a sequence of sub-fields of which the percent sign (%) and the "char" are mandatory if variables are to be printed. The percent is an "escape" character signaling the start of the field specification. The "char" denotes the format of the output field image {binary, decimal, string, etc.}. The sub-field specifications are interpreted as follows:

conversion initiator

%	the translation specification initiator

flags

space	specifies that the result of a signed conversion will always start with a minus sign or space ('-' or ' ')
-	specifies that the value will be left-justified within the print field image, otherwise it will be right-justified
+	specifies that the result of a signed conversion will always start with a sign ('+' or '-')
#	specifies that the value is to be converted to an alternate form. 'c', 'd', 's', and 'u' conversions ignore this flag. For 'o' conversions, the precision is increased to force the first digit to be a zero. For 'b' or 'B' conversions, a non-zero value will be preceded with either 0b' or '0B'. For 'x' or 'X' conversions, a non-zero value will be preceded with '0x' or '0X'. For 'e', 'E', 'f', 'g', and 'G' conversions, the output will always contain a decimal point. Finally, for 'g' or 'G' translations, trailing zeroes will not be removed

width specifier

width specifies the minimum width of the print field image. If the converted value has fewer characters than this value, it will be padded right or left with a pad character which will be a blank unless the field-width begins with a zero, in which case the pad character will be a zero. If the converted value has more characters than the width, then it (the width) is adjusted to hold the full result.

precision specifier

.prec specifies the maximum number of string bytes for 's' translation, the maximum number of digits to the right of the decimal point for 'e', 'E', or 'f' translation, the number of significant digits in 'g' or 'G' translation, or the minimum number of digits in a 'b', 'B', 'd', 'o', u', 'x', or 'X' translation (the field will be left padded with zeroes); generally called the "precision".

conversion type

h	specifies that the following 'b', 'B', 'd', 'o', 'u', 'x', or 'X' translation applies to a short integer;
l	specifies that the following 'b', 'B', 'd', 'o', 'u', 'x', or 'X' translation applies to a long integer;

char

the conversion character

b=B=binary
o=octal
d=decimal
x=X=hexadecimal
s=string
c=character
u=unsigned
e=E=float]

of the format, "[-]d.dddesdd" or "[-]d.dddEsdd", with the number of d's after the decimal point according to the precision (default of 6), [f=float] of the format, "[-]ddd.dd", with the number of d's after the decimal point equal to the precision (default of 6); [g=G=float] printed with format "d", "f", 'e', or "E" whichever gives full precision in minimum space.

A field width or precision may also be indicated by an asterisk ('*') instead of a digit string. When this is the case, an integer argument must then be supplied which contains the value of the precision. A negative value will be interpreted as invoking the '-' left justification. The integer argument must immediately precede the argument corresponding to the value to be translated in the argument list.

Any portion of the control string which cannot be interpreted as a conversion specification field is considered to be transparent printing characters and will be passed directly to the print image.

Warnings

Single precision float arguments must be explicitly casted to double if you are using the "+f" compiler option as the print formatter always assumes a floating point variable to be double precision.

A string longer than 32767 characters will be truncated. A width not in the range of a short int will fail. The precision must be in the positive range of a short int.

Return Code

If an output error is detected, EOF will be returned; otherwise, the number of characters actually printed will be returned.

Example

```
fprintf(stderr,"%d characters, %d lines were copied\n",bytes,lines);
```

See also: fscanf(), printf(), sprintf()

This function obtains the stream pointer of a file descriptor.

```
FILE *fpup( fildes);
   int fildes;

   fildes    the file descriptor of the file.
```

Description

Sometimes it may be useful to cross between file stream I/O and file I/O. Since file streams are referenced with file pointers and files are referenced with file descriptors, the fpup() function may be used to obtain the stream file pointer given the file's descriptor.

Return Code

Fpup() will return NULL on an invalid file descriptor.

Example

```
   if ( seek(fd1, block, 3)) == EOF)
   {
      err_num = ferror( fpup(fd1));
      printf("Error %d detected on file %d\n",err_num,fd1);
   }
```

See also: fdown(), fdopen(), fileno()

This function will output a string to a file/device.

```
int fputs( string, stream );
   char *string; FILE *stream;

   string    is the address of the string to be output.

   stream    is the file pointer of the output file.
```

Description

Fputs() outputs to the file defined by "stream", all characters pointed to by "string", up to the first NULL byte. This function does not append any newline character to the string.

Return Code

If an error was detected, fputs() will return EOF and the global error variable, errno, will contain the UNIX error number associated with the error; otherwise, NULL will be returned.

Warning

Calling fputs() with an invalid file pointer can result in destruction of files or other havoc.

Example

```
#include stdio.h
main()
{   int i; FILE *stream, *fopen(); char buffer[81];
    if ((stream=fopen("testfile/dat:7","w+"))==NULL)
        { perror("fopen error"); exit(-1); }
    for (i=1; i<9999; i*=10)
        fprintf(stream,"Record %d is here\n",i);
    rewind(stream);
    while ((fgets(buffer,80,stream))!=NULL)
        fputs(buffer,stdout);
}
```

```
Record 1 is here
Record 10 is here
Record 100 is here
Record 1000 is here
```

See also: fgets(), puts()

This function is the float equivalent of pow().

```
#include <math.h>
float fraise( argx, argy );
   float argx, argy;

   argx      is the base value to be raised.

   argy      is the power of the base value.
```

Description

This function will raise a single precision number to a single precision power. It is equivalent to:

```
(float)pow((double)argx,(double)argy);
```

See also: lpower(), pow()

This function reads "n" items into an array from a stream input.

```
int fread( ptr, size, nitems, stream );
   char *ptr; int size, nitems; FILE *stream;

   ptr       is a pointer to the receiving array.

   size      is the size of an element in bytes.

   nitems    is the number of items to read.

   stream    is a stream pointer from which to read.
```

Description

This function will read into the array pointed to by "ptr", "nitems" elements of data from the file stream defined by "stream". Each element will be of size "size". It is sometimes convenient to use the "sizeof" operator in determining the size of an element. This is illustrated in the example.

Return Code

The number of items actually read are returned. A NULL will be returned if "size" or "nitems" is less than or equal to zero. If an error occurs and no data items are read, you can use feof() or ferror() to discriminate between an error return or a "zero" input.

Example

```
#include stdio.h
main()
{
    char days[12]; FILE *stream; int i;
    if ((stream=fopen("boot/sys.lsidos","r"))==NULL)
        { perror("fopen error"); exit(-1); }
    fseek(stream,0x202L,0);   /* seek stream to days_in_month data */
    fread(days,sizeof(char),sizeof days,stream);
    for (i=0; i<12; i++)
        printf("%d  ",days[i]);
    fclose(stream);
}

31  28  31  30  31  30  31  31  30  31  30  31
```

See also: fwrite()

This function frees memory allocated with alloc(), malloc(), calloc(), or realloc().

```
void free( ptr );
   char *ptr;

   ptr    pointer to a block previously allocated.
```

Description

Free() is called when a memory block allocated to the program by any of the memory allocating functions [alloc(), calloc(), malloc(), or realloc()] is no longer needed, and the programmer wishes to free the memory space for later use. "ptr" points to the first (lowest) byte allocated to the program and is the pointer returned by the allocating function.

Warning

Calling free() with an address other than that obtained from a call to alloc(), calloc(), malloc(), or realloc() may cause unpredictable results. If the pointer passed to free() is outside the scope of the dynamic heap, free() will report an error. Since free() is used by the standard I/O facility, the error will be reported directly to the system's output device to avoid circular errors.

Example

```
#include "stdio.h"
static int (*_compar)();
static char *_base, *pivot;
static unsigned _width;
extern char *malloc(), *move();
extern void free();
static void _sort(), _swap();
int qsort(base, nel, width, compar)
    char *base; int (*compar)(); unsigned nel, width;
{   if (! (pivot = malloc(width)))
        return EOF;
    _compar = compar;
    _base = base;
    _width = width;
    _sort(0, --nel);
    free(pivot);
    return 0;
}
```

See also: alloc(), calloc(), malloc(), realloc()

This obtains the size of the largest block of memory which can be allocated to your program immediately following the function call.

```
#option INLIB
unsigned freemem();
```

Description

Freemem() returns the maximum amount of memory which can be obtained from alloc(), calloc(), or malloc() at the time of the call to freemem(). The figure is obtained by taking the maximum of (a) the largest unused block kept in the free list available for reuse and (b) the amount of memory above the program break [reduced by approximately 1K less bytes than actual to account for reserved program stack space and function linkage overhead].

Example

```
#option INLIB
#define BLOCK 5000
char *ram[10] = {0,0,0,0,0,0,0,0,0,0};
main()
{
    int i; unsigned core_left, core_now; char *alloc();
    core_left = freemem();
    printf("Beginning core left = %u\n",core_left);
    for ( i=0 ; (ram[i]=alloc(BLOCK))&&i<10 ; i++ )
        printf("%u  ",freemem());
    putchar('\n');
    while (i)
    {
        if (ram[--i]) free(ram[i]);
        core_now = freemem();
        printf("%u  ",core_now);
    }
}

Beginning core left = 41772
36770  31766  26762  21758  16754  11750  6746  1742
6744  11748  16752  21756  26760  31764  36768  41772
```

See also: alloc(), free()

This function substitutes a named file in lieu of an open stream.

```
FILE *freopen( path, type, stream );
   char *path, *type; FILE *stream;

   path      points to a character string which contains the specification of
             the file.

   type      is a character string which indicates the access mode chosen from
             the following:

             "r", "w", "a", "r+", "w+", or "a+".

   stream    is the stream file pointer to "reopen".
```

Description

Freopen() substitutes the named file denoted by "path" in place of the already opened stream designated by the file pointer, "stream". The original stream is closed, regardless of whether the open actually succeeds. The function will return a pointer to the new stream. The freopen() function is typically used to attach the standard I/O streams to files other than the opened standard devices so that the stream I/O functions such as putchar(), getchar(), puts(), gets(), printf(), scanf(), etc., may be utilized.

"Type" points to a null-terminated string which defines the mode of access and may be in upper or lower case. The allowable modes are:

"r"	open for reading;
"w"	truncate or create for writing;
"a"	append; open for writing at the end of the file, or create for writing;
"r+"	open for update (reading and writing);
"w+"	truncate or create for update;
"a+"	append; open or create for update (appending) at the end of the file.

When a file is opened for update, both reading and writing may be performed on the stream; however, writing may not be immediately followed by input without an intervening fseek() or rewind(), and reading may not be followed by writing without an intervening fseek(), rewind(), or an input operation which encounters an end-of-file [first cleared by cleareof()];

Return Code

The stream file pointer to be used in accessing the file is returned if no errors are detected in the open operation. If an error is detected during the open operation, NULL will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Warnings

When a file is opened for append, the file is positioned so that the first output operation will write to the end of the file. The output functions do not perform this positioning operation; therefore, do not use rewind() or fseek() on a stream to which you wish to append.

Example

```
getfile(fname)
   char *fname;
{
   FILE *fp;
   extern FILE *freopen();
   if ((fp=freopen(fname,"r",stdout)) == NULL)
    {
      fprintf(stderr,"Open error - %-20s\n",fname);
      exit();
   }
   else return fp;
}
```

See also: fopen(), fdopen()

This function separates a double into its mantissa and exponent.

```
#include <math.h>
double frexp( value, eptr );
   double value; int *eptr;

   value     is the double value to separate.

   eptr      is a pointer_to_int to receive the exponent.
```

Description

Double precision floating point numbers may be written as a fractional part, x, times an integer power, n, of the base, 2; where x is in the range [0.5 <= x < 1.0]. The frexp() function separates a double into the "x" fractional part and the integer "n" exponent of base_2. The function returns the fractional part and stores the integer exponent into the location pointed to by the integer pointer, eptr. Thus, the following relationship is true:

```
mantissa = frexp(value,&exponent);
value = pow(2.0,(double)exponent) * mantissa;
```

Return Code

The fractional part of the double is returned. If "value" is zero, both the fractional part and the exponent are 0.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d, v, m; int e;
main()
{
   puts("frexp: enter your number: EOF to exit");
   while (TRUE)
   {
      if (!gets(inbuf)) break;
      errno=0;
      m = frexp(d=atod(inbuf),&e);
      v = pow(2.0,(double)e) * m;
      printf("d = %g = %g; e = %d; m = %g; errno = %d\n",d,v,e,m,errno);
   }
}

frexp: enter your number: EOF to exit
2     |d = 2 = 2; e = 1; m = 1; errno = 4
4     |d = 4 = 4; e = 2; m = 1; errno = 4
4.75  |d = 4.75 = 4.75; e = 3; m = 0.59375; errno = 4
5.8e4 |d = 58000 = 58000; e = 16; m = 0.88501; errno = 0
0     |d = 0 = 0; e = 0; m = 0; errno = 0
```

See also: ldexp(), modf()

This function obtains a float random number.

```
#include <math.h>
float frnd( value );
   float value;

   value     used to determine the range of the result.
```

Description

This function will return a floating point single precision (a float) random number. If "value" is 0.0, a number between greater than or equal to zero but less than 1.0 will be returned. If "value" is non-negative and is in the range 1-32767, the returned random number will be in the range 1.0 through fint(value) inclusive.

Random number

If "value" is negative, frnd() will return "value" and errno will be set to DOMAIN. This error condition may be trapped by the use of a _fltvec().

See also: errno, fseed(), srand(), rand()

These functions are used to scan a formatted print image, interpret the image fields according to a control string, and store the translated results in the arguments passed in the function's invocation.

```
#include <math.h>            ... optional
int fscanf( stream, control, arg1, arg2, ... );
   FILE *stream; char *control; args SEE TEXT

   stream       is the stream file pointer to read.

   control      as documented below.

   args         as required to match the control string.
```

Description

The fscanf() function is the input analog of the fprintf() function. It provides similar translations; however, the conversions are from ASCII string fields to argument values. Fscanf() inputs from the file stream identified by "stream".

The arguments identified as "arg1, arg2, ..." MUST BE POINTERS, rather than values since the scan function stores the converted results into the arguments. The control string may contain:

(1) White space characters [0x20, or 0x09 through 0x0d] which are ignored,

(2) Ordinary characters other than '%' which are expected to match the next non-white space character of the input stream,

(3) A conversion specification consisting of the character '%', an optional assignment suppression character '*', an optional decimal number specifying a maximum field width, an optional letter "l" or "h" indicating the size of the receiving variable (i.e. long or short), and a conversion character.

An input field is defined as a string of non-white space characters extending to either the next white space or until the field width is exhausted, whichever comes first. The conversion specification directs the conversion of the next input field. If assignment suppression is indicated by the '*', the input field is skipped over and no argument should be passed in the function's invocation. The scan function reads across new-lines ('\n') since they are considered to be white space.

The conversion characters supported are:

d A decimal integer is expected. It may be preceded by '+' or '-'.

o An octal integer is expected. It does not require the leading zero.

x A hexadecimal integer is expected.

b A binary integer is expected.

c A single character is expected. The normal skip-over-white-space is suppressed so that the next character is input and assigned. If you want to assign the next non-white space character, specify "%1s". If any other width is specified, that many characters will be input and assigned to the pointed to character array of sufficient size.

s A character string is expected. The assignment will be terminated with a NULL ('\0').

f A floating point number is expected. The conversion character 'e' is interpreted the same as 'f'. Preceding the 'e' or 'f' with an 'l' indicates a double rather than a float. The input format for a float is an optional sign, a string of decimal digits possibly containing a decimal point, an optional exponent field containing an 'e' or 'E' followed by a signed decimal integer. NOTE: In this implementation, the scan function will not terminate if an invalid character is detected in the float string. The assigned float value will be that which is translated up to the offending character.

n Specifies that the number of characters scanned up to that point will be returned in the corresponding argument; scanning will continue.

[Indicates string data to be matched against the following set of characters which are called the "scanset". Note also that the normal skip over whitespace is suppressed. The scanset designates the list of permissable characters to be acceptable in the input string. The scanset follows the left bracket and is terminated by a right bracket ("]").

A range of characters is entered as "abcdefg". This can also be entered as "a-g" as in 'first'-'last' where the 'first' character of the range is lexically less than the 'last' character of the range. A dash is accepted as a member of the scanset if it is either the first character in the scanset, the last character in the scanset, or the character preceding it is lexically greater than the character following it. A right bracket ("]") may be included in the scanset if it is the first character following the left bracket.

If a circumflex ("^") is the first character of the scanset, it indicates that the scanset is actually composed of all characters not in the following list; thus, it serves to complement the given list. For example, "[^a-z]" indicates that the scanset is to be composed of all characters which are not lower-case. If a right bracket or dash is to be "excluded" from such a list, it must immediately follow the circumflex. The scanset is selected from the ASCII character set only.

The input will be terminated when a whitespace character is input, a character not in the scanset is input, or the maximum field width specification is exhausted. The corresponding receiving argument must be large enough to hold the data field and the terminating NULL which will be automatically appended.

The 'f' [and likewise the 'e'] translation will not be supported unless the preprocessor statement, "#option MATHLIB" or "#include <math.h>", is inserted in your C source module or the MATH library is manually searched first during the linking process.

Any characters remaining in the input stream that were not accepted including any offending terminating character will still be available to the next scan invocation [assuming no intervening use of the input file stream].

Return Code

If end-of-file is reached during the input, EOF will be returned. Otherwise, the function's return value will be equal to the number of successfully matched and assigned input items.

Example

```
#include <stdio.h>
#include <math.h>
main()
{
    char achar, string[81]; double f1;
    int n_items, aint;
    n_items = fscanf(sfp1,"%c%3d%*2d%6s%lf",&achar,&aint,string,&f1);
    printf("Count = %d |%c %d %s %e\n",n_items,achar,aint,string,f1);
}

For input of: a12345abcdef 12.345e5
Prints      : Count = 4 |a 123 abcdef  1.2345E+06
```

See also: scanf(), sscanf()

This function seeds the random number generator fetched by frnd().

```
#include <math.h>
float fseed( seed );
   float seed;

   seed      is the new seed value.
```

Description

This function seeds the single precision random number generator with a known value.

Return Code

The function returns the old seed value.

See also: frnd(), srand(), rand()

This function is the stream equivalent of lseek().

int fseek(stream, offset, ptrname);
FILE *stream; long offset; int ptrname;

stream	is the stream file to re-position.
offset	is the signed new position of the stream.
ptrname	specifies seek from beginning, current position, or the end of the stream.

Description

This function sets the next input or output operation on the identified stream to occur at the position designated. The new position is the signed distance away from the beginning, current position, or end of the file stream depending on whether "ptrname" is 0, 1, or 2.

Fseek() will undo the effect of any ungetc() on the associated stream.

Note that you cannot fseek() a tty device!

Return Code

Fseek returns non-zero for improper seeks, otherwise NULL.

Example

```
#include stdio.h
main()
{
    char days[12]; FILE *stream; int i;
    if ((stream=fopen("boot/sys.lsidos","r"))==NULL)
        { perror("fopen error"); exit(-1); }
    fseek(stream,0x202L,0);   /* seek stream to days_in_month data */
    fread(days,sizeof(char),sizeof days,stream);
    for (i=0; i<12; i++)
        printf("%d  ",days[i]);
    fclose(stream);
}

31  28  31  30  31  30  31  31  30  31  30  31
```

See also: ftell(), isatty(), lseek(), rewind(), seek()

This function obtains the integer sign of a float.

```
#include <math.h>
int fsgn( value );
   float value;

   value     is the float whose sign is desired.
```

Description

Fsgn() obtains the status as to whether the value is negative, zero, or positive.

Return Code

Fsgn() returns an integer value less than zero, equal to zero, or greater than zero, depending on whether the argument value is less than 0.0, equal to 0.0, or greater than 0.0.

See also: dsgn()

This function obtains the square root of a float.

```
#include <math.h>
float fsqr( value );
   float value;

   value     is the float whose square root is desired.
```

Description

This function is non-standard. It obtains the square root of a float and is equivalent to:

```
(float)sqrt((double)value);
```

See also: sqrt()

This function obtains the file status of an open file.

```
#include <stat.h>
int fstat( fildes, buf );
   int fildes; struct stat *buf;

   fildes    is the file descriptor of the file to stat.

   buf       is a pointer to the structure which will be filled with the
             file's status.
```

Description

This function obtains a great deal of information concerning the open file known by "fildes". The data structure is defined in the stat header file and is as follows:

```
struct stat {
   dev_t       st_dev;     /* device # - (DEC/DRV or Device name) */
   ino_t       st_ino;     /* index node - unused in MC */
   unsigned    short st_mode;  /* modes - see #define */
   short       st_nlink;   /* # of file links - unused */
   short       st_uid;     /* User ID - MC uses 16-bit user pswd */
   short       st_gid;     /* Group ID - MC uses 16-bit owner pswd */
   dev_t       st_rdev;    /* unused */
   off_t       st_size;    /* Size of file in bytes */
   time_t      st_atime;   /* time last read - unused in MC */
   time_t      st_mtime;   /* time last written - MOD date */
   time_t      st_ctime;   /* attribute change, mtime - unused */
};
```

Masks for the "st_mode" field are defined in the header file and are:

```
S_IFMT      0170000 /* type of file - mask */
S_IFDIR     0040000 /* directory - set if DIR/SYS */
S_IFCHR     0020000 /* character special - set if device */
S_IFBLK     0060000 /* block special - set if SYS */
S_IFREG     0100000 /* regular - set for files */
S_IFMPC     0030000 /* multiplexed char special - unused */
S_IFMPB     0070000 /* multiplexed block channel - unused */
S_ISUID     0004000 /* set user id on execution - unused */
S_ISGID     0002000 /* set group id on execution - unused */
S_ISVTX     0001000 /* save swapped text - unused */
S_IREAD     0000400 /* read permission, owner */
S_IWRITE    0000200 /* write permission, owner */
S_IEXEC     0000100 /* execute/search permission, owner */
S_GREAD     0000040 /* read permission, group */
S_GWRITE    0000020 /* write permission, group */
S_GEXEC     0000010 /* execute/search permission, group */
S_UREAD     0000004 /* read permission, user */
S_UWRITE    0000002 /* write permission, user */
S_UEXEC     0000001 /* execute/search permission, user */
```

Return Code

Upon a successful completion of the function, NULL is returned; otherwise, EOF is returned and errno will contain the UNIX error number which describes the error.

Example (a piece of sortsym/ccc)

```
#include <stat.h>
struct stat stat;
unsigned short nrecs, fsize, i;
char tempname[NAMELEN], tempaddr[ADDRLEN];
union record
{
   struct
   {
      char name[NAMELEN];
      char space;
      char addr[ADDRLEN];
      char newline;
   } byname;
   struct
   {
      char addr[ADDRLEN];
      char space;
      char name[NAMELEN];
      char newline;
   } byaddr;
} *record, *base, **prec, **recp;

void abend();
extern int fstat();
extern char *alloc();
extern unsigned short read(), write();

main(argc, argv)
   int argc; char *argv[];
{
   if (fstat(0, &stat) == EOF)
      exit(1); /* stat the input file (stdin) */
   nrecs = (fsize = (unsigned short) stat.st_size - 1) / RECLEN;
   if (stat.st_size > 0xffffL
      || !(base = (RECORD *) alloc(fsize))
      || ! (prec = (RECORD **) alloc(nrecs * sizeof(RECORD *))))
         abend("SYM file too large\n");
   if (read(0, (char *) base, fsize) != fsize)
      exit(1); /* read SYM file */
}
```

See also: creat(), open()

This function obtains the trigonometric tangent of a float.

```
#include <math.h>
float ftan( value );
   float value;

   value        is the float whose tangent is desired.
```

Description

This function obtains the trigonometric tangent of its single precision floating point argument which must be in radians. It is equivalent to:

```
(float)tan((double)value);
```

See also: tan()

This function obtains the relative position of a stream.

```
long ftell( stream );
   FILE *stream;

   stream       is the stream file pointer.
```

Description

This function returns the position, as a long integer, of the stream for the next character I/O operation. The position is relative to the beginning of the stream.

Return Code

The current position relative to the beginning of the stream is returned as a long integer.

Example

```
#include stdio.h
extern int errno;
extern long ftell();
main()
{
    void abend();
    errno = 0;
    if (isatty(STDIN))
        abend("Can't fseek() a character device");
    if (fseek(stdin,0x1a05L,0))
        abend("fseek() error 1");
    if (fseek(stdin,0x333L,1))
        abend("fseek() error 2");
    printf("Current positition after seeks is %lx\n",ftell(stdin));
}
void abend(s) char *s;
{
    printf("%s: errno = %d\n",s,errno);
    exit(-1);
}

tftell:3
Can't fseek() a character device: errno = 0
DOS Ready
tftell:3 <seek/dat:7
Current positition after seeks is 1d38
```

See also: lseek()

This function converts a float to an ASCII string value.

```
char *ftoa( value, string );
   float value; char *string;

   value     is the value to convert.

   string    is a pointer to the string buffer.
```

Description

The ASCII string is constructed according to a specific "e" translation formatted as follows:

```
sd.ddddddEnee
```

where "s" is blank or '+'; "d" represents a decimal digit; "E" is the exponent indicator; "n" is the sign of the exponent ('+' or '-'); and "e" is a decimal exponent digit.

Return Code

The function returns a pointer to the ASCII value in "string".

See also: sprintf()

This function writes "n" items from an array to a stream output.

```
int fwrite( ptr, size, nitems, stream );
   char *ptr; int size, nitems; FILE *stream;

   ptr       is a pointer to the sending array.

   size      is the size of an element in bytes.

   nitems    is the number of elements to write.

   stream    is a stream pointer for which to write.
```

Description

Fwrite() will append at most "nitems" of data from the array identified by "ptr" to the file stream identified by the file pointer, "stream". The function will stop appending when it has written "nitems" of data or if it detects an error condition on the the output "stream".

Return Code

The number of data items written to "stream" will be returned. A NULL will be returned if "size" or "nitems" is less than or equal to zero. If an error occured and no data items had been written, you can use feof() or ferror() to discriminate between an error return or a "zero" output.

Example

```
#include stdio.h
#include math.h
float array[8];
main()
{   int i; FILE *stream;
    if ((stream=fopen("test/dat:7","w"))==NULL)
        { perror("main fopen() error"); exit(-1); }
    for (i=0; i<8; i++)
        array[i] = i;
    if (!fwrite(array,sizeof(float),8,stream))
        { perror("main fwrite error"); exit(-1); }
    fclose(stream);
    execl("list","list","test/dat:7","(hex)",NULL);
}

0000:00 = 00 00 00 00 00 00 00 81  00 00 00 82 00 00 40 82 ........ ......@.
0000:10 = 00 00 00 83 00 00 20 83  00 00 40 83 00 00 60 83 ...... . ..@...`.
```

See also: fread()

This function will generate a new file specification using an input and partial file specification.

```
char *genspec( inspec, partspec, extn);
   char *inspec, *partspec, *extn;

   inspec       The input file specification.

   partspec     The partial specification to expand.

   extn         The extension to add if one is omitted.
```

Description

This function will generate a new file specification from the input file specification, using the given output partial file specification and a default extension. The new file specification is written to the character array pointed to by "partspec". The following rules specify the resulting specification's name, extension, and drive of the partial filespec expansion:

(1) The expanded specification will contain all fields passed in "partspec";

(2) If the "partspec" name field is omitted, it will be filled with the name field of "inspec";

(3) If the "partspec" extension field is omitted, and "extn" is not a null string, it will be filled with "extn";

(4) If the "partspec" extension field is omitted, and "extn" is a null string, and "inspec" contains an extension field, it will be filled with the extension field of "inspec";

(5) If the "partspec" extension field is omitted, and "extn" is a null string, and "inspec" does not contain an extension field, the expanded specification will not contain an extension field;

(6) If the "partspec" drive field is omitted, it will be filled with the drive field of "inspec", if any.

Return Code

A pointer to the expanded file specification is returned.

Warning

The partial specification string must be stored in a character array of at least dimension 15 to avoid overextending the allocated string space.

See also: addext()

These functions are used to obtain a character from a file stream. Getchar() always refers to the standard input device.

```
int getc( stream );
   FILE *stream;

int getchar();

   stream       is the file pointer of the stream
```

Description

Getc() is used to input a single character from a file stream. The stream pointer must be obtained from fopen(), fdopen(), freopen(), or be a standard file pointer. MC provides getchar() as a function, not as a #define macro. Getchar() is identical in operation to getc(stdin). Any of the 256 possible binary codes may be input using getc().

On character devices [i.e. *KI, *CL], the <BREAK> key may be optioned to be interpreted as an end-of-file condition by means of the option() function. This provides a means to indicate an EOF for character devices not normally having the capability to generate the EOF indication.

Return Code

The return code is the integer value of the character input from the stream. If an end of file is encountered, then "EOF" (-1) is returned.

Warning

If the value returned by the function is to be stored before testing for end-of-file, it must be stored in an integer variable and not a char. If not, the end of file value will be truncated and will remain undetected as a char of -1 does not equal an int of -1.

Examples

```
#include <stdio.h>
filecopy(fp)    /* copy a file to the standard output */
FILE *fp;
{   int c;
    while ((c = getc(fp)) != EOF)
       if (c != putc(c,stdout)) abend("Output file write error");
}

bytes = lines = 0;
while( (c=getchar()) != EOF )
{   putchar(c); ++bytes; if ( c==EOL ) ++lines;
}
```

See also: gets(), putc(), putchar(),scanf()

This function fetches (inputs) a buffered line from standard input.

```
char *gets( buffer );
   char *buffer;

   buffer       a pointer to a character array of length 81.
```

Description

Gets() inputs a line up to 80 characters long from the standard input (stdin) and places the line in memory starting at the address given by "buffer". The input will terminate on an error or if a new-line character is input. The new-line is discarded and the string is terminated by NULL.

If you do not want to have the new-line character discarded, use the near-equivalent form, fgets(buffer,80,stdin);.

Note that if stdin is not the keyboard device and the input line is longer than 80 characters, it will be split; any part not accepted will be fetched by the next request for input from stdin.

Return Code

If an error is encountered during the input, or end-of-file is reached, the return code will be NULL (0); otherwise, buffer will be returned.

Warnings

The "buffer" must be at least 81 characters long.

Example

```
#include stdio.h
char buffer[81], string[10];
main()
{
    int i;
    fputs("Enter your string: ",stdout);
    gets(buffer);
    for ( i=0; i<strlen(buffer); i++)
        fputs(itox(buffer[i],string),stdout);
    putchar('\n');
}

Enter your string: a1b2c3d4
6131623263336434
```

See also: fgets(), puts(), fputs()

This function will read a "word" from a stream input.

```
int getw( stream );
   FILE *stream;

   stream    is a file pointer stream to read.
```

Description

Getw() inputs a word from the designated stream. A "word" is the size of an integer.

Return Code

If an error is encountered during the input, or end-of-file is reached, the return code will be EOF (-1).

Warnings

Since EOF is a valid integer value, ferror() or feof() must be used to differentiate between a -1 return and an error return. Note that since the low byte and high byte storage order of a word may vary in different machine environments, getw() is an environment dependent function.

Example

```
#include stdio.h
int array[16];
main()
{   int i; FILE *stream;
    if ((stream=fopen("test/dat:7","w+"))==NULL)
        { perror("main fopen() error"); exit(-1); }
    for (i=0; i<16; i++)
        array[i] = i;
    if (!fwrite(array,sizeof(int),16,stream))
        { perror("main fwrite error"); exit(-1); }
    rewind(stream);
    while (((i=getw(stream)) != EOF) && !feof(stream))
        printf("%d ",i);
    fclose(stream);
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

See also: getc(), putw(), swab()

This function obtains information concerning a character special device.

```
#include sgtty.h
int gtty( fildes, argp );
   int fildes; struct sgttyb *argp;

   fildes       is a descriptor of the file.

   argp         is a pointer to the data structure.
```

Description

This function obtains control information on a "files". The data obtained is stored in the data structure pointed to by the "argp" argument. The structure is defined in the "sgtty" header file and is as follows:

```
struct sgttyb {
   char sg_colctr;    /* column counter */
   char sg_control;   /* control byte */
   char sg_flag;      /* FCA flag byte */
};
```

The "sgttyb.sg_colctr" element stores a counter for the current column position of the output image for the device. This item is relative to zero. The "sgttyb.sg_control" byte stores various bit fields and is masked as follows:

```
IO_BREAK    0x10      /* mask for #option BREAK */
IO_TABSTOP  0x0f      /* mask for "tabstop-1" */
IO_FDCOE    0x80      /* mask for "close_on_exec" flag */
```

The gtty() function is equivalent to:

```
ioctl(fildes,TIOCGETP,argp);
```

TIOCGETP is defined in the "sgttyb" header file.

Return Code

If the call was successful, NULL will be returned; otherwise, an EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

See also: ioctl(), stty()

This function calculates the Euclidian distance of its arguments.

```
#include <math.h>
double hypot( argx, argy );
   double argx, argy;

   argx, argy      are the two required elements.
```

Description

The Euclidian distance is calculated according to the formula:

```
hypot = sqrt( argx*argx + argy*argy )
```

Example

```
#include stdio.h
#include math.h
main()
{
    double x, y;
    for (x=3.0,y=4.0; x<7.0; x++, y++)
        printf("x = %g, y = %g, hypot(x,y) = %g\n",x,y,hypot(x,y));
}

x = 3, y = 4, hypot(x,y) = 5
x = 4, y = 5, hypot(x,y) = 6.40312
x = 5, y = 6, hypot(x,y) = 7.81025
x = 6, y = 7, hypot(x,y) = 9.21954
```

See also: sqrt()

This function obtains the position of the first occurrence of a specified character within a string. It is identical to strchr(), its UNIX System V counterpart.

```
char *index( s, c );
   char *s; int c;

   s      is a pointer to the source string.

   c      is the character to find.
```

Description

The index() function will look for the first occurrence of character 'c' in the string pointed to by "s". The low-order byte of the integer, 'c', will be used as the character. Index will operate properly when 'c' is the NULL character.

Return Code

If the character 'c' is not found in string "s", NULL will be returned; otherwise, a pointer to the position of 'c' in "s" will be returned.

Example

```
#include stdio.h
main()
{
    char *pos, buffer[81], *fgets(), *index();
    puts("Enter your string; EOF to exit");
    while (gets(buffer))
        {
        if (!(pos=index(buffer,'r')))
            printf("'r' not found in [%s]\n? ",buffer);
        else
            printf("'r' is character %d in [%s]\n? ",pos-buffer+1,buffer);
        }
}

Enter your string; EOF to exit
abcdefghijklmnop
'r' not found in [abcdefghijklmnop]
? zyxwvutsrqponmlkjihgfedcba
'r' is character 9 in [zyxwvutsrqponmlkjihgfedcba]
? abcdefghijklmnopqrstuvwxyz
'r' is character 18 in [abcdefghijklmnopqrstuvwxyz]
?
```

See also: strcat(), strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

The inkey() function scans the DOS keyboard device (*KI).

```
#option INLIB
int inkey();
```

Description

The inkey() function makes a single scan of the keyboard and returns the ASCII value of any depressed key. It will return a zero if no key is pressed. Note that it will always scan the physical keyboard regardless of any MC standard input redirection.

Warning

Inkey() cannot return a NULL character.

Example

```
#include stdio.h
#option INLIB       /* needed for automatic search if IN/REL */
#define CTL_C 0x03
extern char *ctime();
unsigned long counter = 0L;
main()
{
    long clock;
    time(&clock);
    fputs(ctime(&clock),stdout);
    while (TRUE)    /* simulate some long processing loop */
    {
        ++counter;
        if (inkey() == CTL_C)
            break;
    }
    printf("Counter got to %ld\n",counter);
    time(&clock);
    fputs(ctime(&clock),stdout);
}
```

```
Tue Nov 19 10:30:33 1985
Counter got to 9204
Tue Nov 19 10:30:44 1985
```

See also: checkc(), getchar()

This function reads a machine port.

```
#option INLIB
int inport( port );
   int port;

   port      is the designated machine port to read.
```

Description

The function, inport(), returns as an integer, the value read from the specified port.

Example

```
#include stdio.h
#option INLIB   /* needed to automatically search IN/REL */
#define FDC 0xf0
main()
{
    int fdc[4], i;
    for (i=3; i>=0; i--)
        fdc[i] = inport(FDC+i);
    printf("FDC: data=%02x, sector=%02x, track=%02x, status=%02x\n"\
            ,fdc[0],fdc[1],fdc[2],fdc[3]);
}

FDC: data=00, sector=15, track=03, status=00
```

See also: outport()

This function performs various controlling operations on character special files (devices).

```
#include sgtty.h
int ioctl( fildes, request, argp );
   int fildes, request; struct sgttyb *argp;

   fildes       is a descriptor of the file.

   request      is the desired operation code.

   argp         is a pointer to the data structure.
```

Description

This function allows you to control certain characteristics about a file. The data acted upon is stored in the data structure pointed to by the "argp" argument and is obtained from an invocation of ioctl() using the TIOCGETP request. Once you modify the data, it is presented to the device handler via the TIOCSETP request. The data structure is defined in the "sgtty" header file and is as follows:

```
struct sgttyb {
   char sg_colctr;    /* column counter */
   char sg_control;   /* control byte */
   char sg_flag;      /* FCA flag byte */
};
```

The "sgttyb.sg_colctr" element stores a counter for the current column position of the output image for the device. This item is relative to zero. The "sgttyb.sg_control" byte stores various bit fields and is masked as follows:

```
IO_BREAK    0x10      /* mask for #option BREAK */
IO_TABSTOP  0x0f      /* mask for "tabstop-1" */
IO_FDCOE    0x80      /* mask for "close_on_exec" flag */
```

Return Code

If the call was successful, NULL will be returned; otherwise, an EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

Example

```
#include stdio.h
#include sgtty.h
#define TABSTOP 3
int c;
main()
{   static struct sgttyb sg = { 0,0 };
    if (ioctl(STDOUT,TIOCGETP,&sg))
        fputs("Error from ioctl\n",stderr);
    else
    {   sg.sg_control &= ~IO_TABSTOP;   /* strip current tab stop */
        sg.sg_control |= TABSTOP;       /* set new  tab stop */
        ioctl(STDOUT,TIOCSETP,&sg); }
    while (((c=getchar())!= EOF) && (c==putchar(c)))  {;}
}
```

See also: gtty(), stty()

These are a series of functions which perform character type tests.

```
int iswhat( c );
   char c;

   iswhat    replaced by the test function name.

   c         the character under test.
```

Description

The following functions will return TRUE when the character under test matches the test range:

function	character range for a TRUE condition
isalnum	A-Z, a-z, 0-9
isalpha	A-Z, a-z
isascii	0x00-0x7f
_isbdigit	0,1
iscntrl	0x00-0x1f, 0x7f
isdigit	0-9
islower	a-z
_isodigit	0-7
isprint	0x20-0x7e
ispunct	0x20-0x2f, 0x3a-0x40, 0x5b-0x60, 0x7b-0x7e
isspace	0x20, 0x09-0x0d
isupper	A-Z
isxdigit	0-9, A-F, a-f

Return Code

If the character does not match the test range, FALSE will be returned.

Example

```
if ( isdigit(char))
     printf("Character is <0-9>\n");
else if ( islower(char))
     printf("Character is <a-z>\n");
else if ( isupper(char))
     printf("Character is <A-Z>\n");
else
     printf("Character is none of the above\n");
```

This function determines if the designated file is character special.

```
int isatty( fildes );
   int fildes;

   fildes    is the descriptor of the file to check.
```

Description

Isatty() returns TRUE if the file descriptor refers to an open character special file (i.e. a character device). A FALSE condition is returned if the descriptor is invalid or does not refer to an open file.

Return Code

The return code is TRUE or FALSE as required.

Example

```
#include stdio.h
extern int errno;
extern long ftell();
main()
{
    void abend();
    errno = 0;
    if (isatty(STDIN))
        abend("Can't fseek() a character device");
    if (fseek(stdin,0x1a05L,0))
        abend("fseek() error 1");
    if (fseek(stdin,0x333L,1))
        abend("fseek() error 2");
    printf("Current positition after seeks is %lx\n",ftell(stdin));
}
void abend(s) char *s;
{
    printf("%s: errno = %d\n",s,errno);
    exit(-1);
}

tftell:3
Can't fseek() a character device: errno = 0
DOS Ready
```

See also: gtty(), ioctl(), stty(), ttyname()

These functions are used to convert integers to character strings of digits (binary, octal, decimal or hexadecimal).

char *itoa(ival, string);

char *itob(ival, string);

char *itoo(ival, string);

char *itou(ival, string);

char *itox(ival, string);
   int ival; char string[];

ival	is an integer value to convert.
string	is the resulting string of signed decimal, unsigned decimel, binary, octal, or hexadecimal digits for "?" equal to a", "u", "b", "o", or "x" respectively.

Description

These standard C functions are used to convert integer values to their character string image. Functions are provided to deal with character strings containing binary, octal, decimal or hexadecimal digits.

Return Code

A pointer to the character string is returned.

Example

```
#include stdio.h
char a[33], b[33], o[33], u[33], x[33];
int num = 100;
main(argc,argv) int argc; char *argv[];
{   if (argc==2) num = atoi(*++argv);
    itoa(num,a);
    itob(num,b);
    itoo(num,o);
    itou(num,u);
    itox(num,x);
    printf("num = %d: %sD %sB %sO %sU %sX\n",num,a,b,o,u,x);
}

num = 100: 100D 1100100B 144O 100U 64X
num = -32768: -32768D 1000000000000000B 100000O 32768U 8000X
num = 32767: 32767D 111111111111111B 77777O 32767U 7FFFX
```

This function is used to obtain the absolute value of a long integer.

```
long labs( lval );
   long lval;

   lval      is the long integer whose absolute value is to be determined.
```

Description

This function will obtain the absolute value of its long integer argument. If lval is negative, -lval will be returned. If lval is non-negative, lval will be returned.

Example

```
#include stdio.h
char inbuf[81];
extern long labs(), atol();
main()
{
    long num;
    puts("Enter your number: EOF to exit");
    while (TRUE)
    {
        if (!gets(inbuf)) break;
        printf("Absolute value of %ld is %ld\n",num,labs(num=atol(inbuf)));
    }
}

Enter your number: EOF to exit
-123456789 |Absolute value of -123456789 is 123456789
-276893105 |Absolute value of -276893105 is 276893105
987654321  |Absolute value of 987654321 is 987654321
```

See also: abs(), fabs(), fabsf()

This function multiplies a fractional value times two-to-an-exponent.

```
#include <math.h>
double ldexp( value, exp );
   double value; int exp;

   value     the fractional part of a double.

   exp       the integer exponent of two.
```

Description

This function calculates a double floating point number composed of a fractional "value" and an integer exponent, "exp", of base 2. It is used in combining parts of a double floating point number.

Return Code

If the resulting value would overflow, +/- HUGE_VAL will be returned and errno will be set to ERANGE. If the result would cause underflow, 0 is returned and errno is set to ERANGE.

Example

```
#include stdio
#include math
char inbuf[81]; double d, v, m; int e;
main()
{
    puts("ldexp: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        errno=0;
        m = frexp(d=atod(inbuf),&e);
        v = ldexp(m,e);
        printf("d = %g = %g; e = %d; m = %g; errno = %d\n",d,v,e,m,errno);
    }
}

ldexp: enter your number: EOF to exit
1234567   |d = 1.23457e+06 = 1.23457e+06; e = 21; m = 0.588687; errno = 0
-7.684e10 |d = -7.684e+10 = -7.684e+10; e = 37; m = -0.559085; errno = 0
1e40      |d = 1.70141e+38 = 1.70141e+38; e = 127; m = 1; errno = 3
```

See also: frexp(), modf()

This function is used to plot a line of pixels on the CRT.

```
#option INLIB
int line( funcod, x1, y1, x2, y2 );
   int funcod, x1, y1, x2, y2;
```

funcod	an operation code to set (1) or reset (0) the pixels involved in the line.
x1,y1	the coordinate of the first point defining the line.
x2,y2	the coordinate of the second point defining the line.

Description

The "line()" function will plot a line connecting coordinate point (x1,y1) with coordinate point (x2,y2).

Virtual Points

A virtual point is any pixel that is part of the plot request which cannot appear on the CRT screen as it is out of the range of acceptable coordinate values. The line function permits your arguments to describe such "virtual" images; however, any portion of the geometric shape that would be outside of the CRT image area is not plotted.

Return Code

A minus one (-1) indicates that the coordinate points (x1,y1) or (x2,y2), or a portion of any plot is out of range (i.e. virtual and does not appear in the CRT image). A minus three (-3) will be returned if the function code passed is invalid (i.e. not in the range <0-1>).

Example

```
#option INLIB
#define PLOT 1
#define ERASE 0
int x1,x2,y1,y2,t,t1;
main()
{   for ( x1=0,y1=0,x2=127,t=0; t<=47; t++)
       { line(PLOT,x1,y1,x2,t); line(ERASE,x1,y1,x2,t); }
    for ( y2=47,t=127; t>=0; t--)
       { line(PLOT,x1,y1,t,y2); line(ERASE,x1,y1,t,y2); }
}
```

See also: box(), circle(), ploc(), pmode(), point(), reset(), set()

This function obtains a pointer to the broken down time structure.

```
#include <time.h>
struct tm *localtime( clock );
   long *clock;

   clock     is a pointer to the long integer containing the UNIX time.
```

Description

This function returns a pointer to the structure, tm (defined in the "time" header file), which will contain the broken down time based on the contents of the long integer pointed to by "clock". Note that localtime() is based on the actual time set in your machine and makes no automatic allowance for time zone or Daylight Savings Time.

Return Code

A pointer to the broken down time structure will be returned.

Warning

The pointer returned by localtime() points to local static storage which may be valid only immediately following the function call.

Example

```
#include stdio.h
#include time.h
extern struct tm *localtime();
extern long time();
struct tm *p;
char *ps[4] = { "th","st","nd","rd" };
main()
{   long clock; int s;
    clock = time(NULL);
    p = localtime(&clock);
    if ((s = p->tm_yday % 10) > 3) s = 0;
    printf("The time is %d:%d:%d on the %d%s day of %d\n",\
          p->tm_hour,p->tm_min,p->tm_sec,p->tm_yday,ps[s],p->tm_year+1900);
    system("date");
    system("time");
}
```

```
The time is 15:51:12 on the 323rd day of 1985
Tue, Nov 19, 1985
15:51:13
```

See also: asctime(), ctime(), sysdate(), systime(), time()

These functions calculate the log of a double.

```
#include "math.h"
double log( argx );

double log10( argx );
   double argx;

   argx      is the double for which the log or log10 is to be determined.
```

Description

The log() function obtains the natural log (base_e) of its argument. The log10() function takes the base_10 logarithm of its argument. In either case, the value of "argx" must be positive.

Return Code

Both log() and log10() return HUGE_VAL and set errno to EDOM when "argx" is not greater than zero. A message indicating DOMAIN error is printed on the standard error output device. This error handling may be changed via the matherr() function.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d, le, l10;
main()
{    puts("log: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        errno=0;
        le = log(d=atod(inbuf)); l10 = log10(d);
        printf("d = %g; log = %g; log10 = %g; errno = %d\n" \
          ,d,le,l10,errno);
    }
}

log: enter your number: EOF to exit
100                |d = 100; log = 4.60517; log10 = 2; errno = 0
2.7182818284590452 |d = 2.71828; log = 1; log10 = 0.434294; errno = 0
-5                 |argument domain error
                   |argument domain error
            |d = -5; log = -1.70141e+38; log10 = -1.70141e+38; errno = 70
```

See also: exp(), pow(), sqrt(), matherr()

This function performs a long jump to an environment saved by setjmp().

#include <setjmp.h>
void longjmp(env, val);
   jmp_buf env; int val;

env	is the environment saved by setjmp().
val	is the value returned by setjmp().

Description

A longjmp() will restore the program to the environment as saved in "env" by a previous call to setjmp(). After the longjmp() is completed, the program execution will continue as if the corresponding call to setjmp() had just occurred; however, the returned value will be the value passed as "val" in the longjmp() call, rather than zero. The setjmp()/longjmp() companion functions can be used in error control within nested functions so as to effect an escape from a lower-level function error without back-tracking through error handling in each higher level nest.

Example

```
#include setjmp.h
jmp_buf env;
main()
{   int i, val;
    if (val=setjmp(env))
        printf("Error %d on i=%d\n",val,i);
    else
        for (i=0;i<3;++i)
            {   func1(i); printf("No error %d\n",i);    }
}
func1(i) int i;
{   func2(i);   }
func2(i) int i;
{   func3(i);   }
func3(i) int i;
{   if (i<2) return; else longjmp(env,1);   }
```

See also: setjmp()

This function is used to raise a long integer to a long integer power.

```
long lpower( argx, argy );
   long argx, argy;

   argx      is the base long integer to be raised to the argy power.

   argy      is the long integer exponent.
```

Description

Lpower() will raise a long integer, argx, to the long integer power, argy. Note that the result could overflow with relatively small values of argy; thus, exercise caution in the argument values.

Return code

If argy is not in the range 0L through 31L, the value returned will be zero and the global error variable, errno, will be set to ERANGE. Otherwise, the returned value will be argx raised to the argy power.

Example

```
#include stdio.h
#define SET 1
long john, silver;
extern long lpower();
main()
{
    option(O_KBECHO,SET);
    puts("lpower: enter your numbers: EOF to exit");
    while (TRUE)
    {
        if (scanf("%ld %ld",&john,&silver) != 2) break;
        printf("%ld to the %ld power = %ld\n", \
             john,silver,lpower(john,silver));
    }
}

lpower: enter your numbers: EOF to exit
2 20 |2 to the 20 power = 1048576
3 8  |3 to the 8 power = 6561
```

See also: fraise(), pow()

This function is used to reposition a file.

```
long lseek( fildes, offset, whence );
   int fildes, whence; long offset;

   fildes    is a file descriptor obtained from open(), creat(), dup(), or
             fcntl().

   offset    the new position relative to whence.

   whence    specifies offset from beginning, current, or end for origin equal
             to 0, 1, or 2.
```

Description

The lseek() function is used for random positioning in a file in preparation for reading or writing. Offset is a signed long integer allowing you to seek to any relative character position in the file.

Return Code

Upon successful completion, the new position relative to the beginning of the file will be returned. If an error occurs during the seeking function, EOF (-1) will be returned and the global error variable, errno, will contain the UNIX error number associated with the error. The DOS error number may be obtained via ferror(fpup(fildes));.

Warning

Remember that "offset" is treated as a SIGNED long integer. Thus, a negative offset passed with an origin of 0 is an error. It is not an error to lseek past the end of a file. You cannot lseek() a tty device.

Example

```
#include stdio.h
#include fcntl.h
char string[58]; int fildes;
main()
{   if ((fildes=open("boot/sys.lsidos",O_RDONLY))==EOF)
        { perror("open error"); exit(-1); }
    lseek(fildes,0x2c7L,0);     /* seek date mnemonics */
    read(fildes, string, 57);   /* read them into string */
    string[57] = '\0';  /* set NULL for string terminator */
    puts(string);       /* display the mnemonics */
    close(fildes);      /* close the file */
}

SunMonTueWedThuFriSatJanFebMarAprMayJunJulAugSepOctNovDec
```

See also: feof(), fseek(), isatty(), rewind(), seek(), tell()

These functions are used to convert long integers to character strings of digits (binary, octal, decimal or hexadecimal).

```
char *ltoa( lval, string );

char *ltob( lval, string );

char *ltoo( lval, string );

char *ltou( lval, string );

char *ltox( lval, string );
   long lval; char xstring[];

   lval      is a long integer value to convert.

   string    is the resulting string of signed decimal, unsigned decimal,
             octal, decimal, or hexadecimal digits for lto? equal to "a", "u",
             "b", "o", "u", or "x" respectively.
```

Description

These standard C functions are used to convert long integer values to their character string image. Functions are provided to deal with character strings containing binary, octal, decimal or hexadecimal digits.

Return Code

In each case, a pointer to the resulting string is returned.

Example

```
#include stdio.h
char a[33], b[33], o[33], u[33], x[33];
long num = 100000;
extern long atol();

main(argc,argv)
int argc; char *argv[];
{
   if (argc==2) num = atol(*++argv);
   ltoa(num,a);
   ltob(num,b);
   ltoo(num,o);
   ltou(num,u);
   ltox(num,x);
   printf("num = %ld: %sD %sB %sO %sU %sX\n",num,a,b,o,u,x);
}

num = 100000: 100000D 11000011010100000B 303240O 100000U 186A0X
num = 65537: 65537D 10000000000000001B 200001O 65537U 10001X
```

This function is used to allocate a zeroed memory block.

```
char *malloc ( size );
   unsigned size;

   size      unsigned number of bytes needed.
```

Description

Malloc() is used to dynamically allocate memory during program execution. The complementary function, free(), is used to release memory allocated through malloc(). Malloc may be used to get table or buffer space when the amount of memory space available is unknown, or when the program needs to dynamically allocate space for an array.

Return Code

If a memory block has been allocated, the value returned is a pointer to the memory block (pointer to char). If insufficient memory is available to satisfy the allocation, a NULL will be returned.

Warnings

The program must not access memory outside of the area allocated. File access routines use dynamic allocation to establish and release File Control Areas (FCA's). The programmer cannot assume that memory not allocated is free for use, since later file opens may cause memory overlays. It is advised that the programmer always use the supplied dynamic allocation functions for memory accessing.

Example

```
symtbsz -= symtbsz % symsiz;   /* make integral */
if ( !(symtab = malloc(symtbsz)) )
   abend("not enough memory");
glbptr = startglb = symtab;
```

See also: alloc(), brk(), calloc(), realloc(), and sbrk()

This function will copy "n" characters in memory until character 'c' is detected.

char *memccpy(s1, s2, c, n);
char *s1, *s2; int c, n;

s1	a pointer to the receiving memory area.
s2	a pointer to the originating memory area.
c	the copy stops after the first occurrence of this character or until n characters.
n	the number of characters to copy.

Description

This function will perform a copy of characters from the memory region pointed to by "s2" into the memory region pointed to by "s1", stopping after the first occurrence of character 'c' has been copied or after "n" characters have been copied, whichever comes first.

Warning

There is no checking on the magnitude of "n"; thus, a memccpy() with an erroneous value for "n" could overwrite a critical portion of memory. Overlapping copies may produce unpredictable results.

Return Code

A pointer to the character after the copy of 'c' in "s1" is returned if 'c' was found; otherwise, a NULL pointer will be returned.

See also: memchr(), memcmp(), memcpy(), memset()

This function finds the first occurrence of a character in memory.

```
char *memchr( s, c, n );
   char *s; int c, n;

   s      a pointer to the memory region to search.

   c      is the character to look for.

   n      is the number of characters to search.
```

Description

This function will scan the memory region pointed to by "s" for "n" characters looking for the first occurrence of character 'c'.

Return Code

A pointer to the character 'c' will be returned if that character is found within the "n" bytes; otherwise, a NULL pointer will be returned.

Example

```
#include stdio.h
extern char *memchr();
char *s = NULL, *t; int c; unsigned n = 0;
main()
{
    fuputs("Enter the character to find: ",stdout);
    c = getchar();
    while ( TRUE )
        {
        if (!(t=memchr(s,c,n)))
            break;
        printf("Found %c at %04x (%c)\n",c,t,*t);
        n = n - ( ++t - s );
        s = t;
        }
}

Enter the character to find:
Found | at 033d (|)
Found | at 0384 (|)
Found | at 03ab (|)
Found | at 058b (|)
   continued
```

See also: memccpy(), memcmp(), memcpy(), memset().

This function will compare two regions of memory.

int memcmp(s1, s2, n);
char *s1, *s2; int n;

s1	is a pointer to the first memory region.
s2	is a pointer to the second memory region.
n	is the number of characters to compare.

Description

This function will compare "n" characters of memory region "s1" to memory region "s2" and returns an integer less than, equal to, or greater than 0, depending on whether the first n-characters of "s1" are lexicographically less than, equal to, or greater than the first n-characters of "s2".

Example

```
#define SIZE 4096
main()
{
    char *p, *s;
    if (!(p=alloc(SIZE)))   /* allocate non-zeroed memory block */
        abort();
    if (!(s=malloc(SIZE)))  /* allocate zeroed memory block */
        abort();
    if (memcmp(p,s,SIZE))
        puts("Before zero(): memory compares different");
    zero(p,SIZE);
    if (memcmp(p,s,SIZE))
        puts("After zero(): memory compares different");
    else
        puts("After zero(): memory compares same");
}
```

```
Before zero(): memory compares different
After zero(): memory compares same
```

See also: memccpy(), memchr(), memcpy(), memset()

This function will copy n characters from one area of memory to another.

```
char *memcpy( s1, s2, n );
   char *s1, *s2; int n;

   s1     a pointer to the receiving memory area.

   s2     a pointer to the originating memory area.

   n      the number of characters to copy.
```

Description

This function will perform a copy of "n" characters from the memory region pointed to by "s2" to the memory region pointed to by "s1".

Warning

There is no checking on the magnitude of "n"; thus, a memcpy() with an erroneous value for "n" could overwrite a critical portion of memory. Overlapping copies may produce unpredictable results.

Return Code

The function returns a pointer to "s1".

See also: memccpy(), memchr(), memcmp(), memset()

This function sets a region of memory to a given character.

```
char *memset( s, c, n );
   char *s; int c, n;

   s      a pointer to the memory region to set.

   c      is the character value to set in memory.

   n      is the number of memory characters to set.
```

Description

This function will set "n" characters of memory starting with the address pointed to by "s" with the designated character, "c".

Warning

There is no checking on the magnitude of "n"; thus, a memset() with an erroneous value for "n" could overwrite a critical portion of memory.

Return Code

A pointer to "s" is returned.

See also: memccpy(), memchr(), memcmp(), memcpy()

This function obtains the signed fractional part of a double.

#include <math.h>
double modf(value, iptr);
double value, *iptr;

value	the number to obtain the fractional part.
iptr	a pointer to a double where the integer part of "value" is to be stored.

Description

This function returns the signed fractional part of the double, "value", and stores the integer part (as a double) in the location pointed to by "iptr". It is used to split a double floating point number into its integer part and its fractional part.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d, f_part, int_part;
main()
{
    puts("modf: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        f_part = modf(d=atod(inbuf),&int_part);
        printf("d = %g; f = %g; i = %g\n",d,f_part,int_part);
    }
}

modf: enter your number: EOF to exit
1.234e2   |d = 123.4; f = 0.4; i = 123
-1765e-2  |d = -17.65; f = -0.65; i = -17
0.56789e4 |d = 5678.9; f = 0.9; i = 5678
```

See also: frexp(), ldexp()

This function will copy a memory block in memory.

```
void move( pfrom, pto, len );
   char *pfrom, pto; int len;

   pfrom     the address of the block to be moved.

   pto       the address of the block's new starting address.

   len       the length of the block, in bytes.
```

Description

This function will perform a nondestructive move of a memory block. That means that if the "pto" address is less than the "pfrom" address, the move will start from the beginning of the block. If the "pto" address is greater than the "pfrom" address, the move will start from the end of the block. For portability across C compilers, it is recommended that you use the equivalent function, memcpy().

Warning

There is no checking on the magnitude of "len"; thus, a move() with an erroneous value for "len" could overwrite a critical portion of memory.

See also: fill(), memccpy(), memchr(), memcmp(), memcpy(), memset(), zero()

This function is used to open a file for block I/O.

```
#include <fcntl.h>
int open(path, oflag [,mode] );
   char *path; int oflag, mode;

   path      a pointer to the string containing the file specification.

   oflag     is an ORing of bit-flags defined in fcntl.h which specify the
             access of the file.

   mode      user protection level when the O_CREAT flag is specified and the
             file does not exist.
```

Description

The open() function is used to prepare an existing file or newly created file for access via read(), write(), or lseek(), or other file operation. If the "oflag" argument specifies the O_CREAT flag, then open requires the third argument, "mode", which should contain the creation access mode to be applied to the designated file if it is non-existing [see creat()].

The "oflag" field is entered by ORing the desired conditions defined in the fcntl header file. These symbols are as follows:

```
O_RDONLY        /* Open for reading only */
O_WRONLY        /* Open for writing only */
O_RDWR          /* Open for reading and writing */
O_NDELAY        /* Non-blocking I/O - unsupported */
O_APPEND        /* Append (initial writes guaranteed at the end) */
O_CREAT         /* Open with file create */
O_TRUNC         /* Open with truncation */
O_EXCL          /* Exclusive open - unsupported */
```

Exactly one of the first three must be specified.

Return Code

If an error is detected in opening the file, open() will return EOF (-1) and the global error variable, errno, will contain the UNIX error number which describes the error; otherwise, the file's descriptor will be returned.

Warning

If the "path" field references a character special device, you will not be able to use lseek() with the returned file descriptor.

Example

```
#include <stdio.h>
#include <fcntl.h>

main(argc, argv)
   int argc; char *argv[];
{
   int fd1, fd2, n; char buf[BUFSIZE];

   if (argc != 3)
      error("Usage: cp from to", NULL);

   if ((fd1=open(argv[1],O_RDONLY))==EOF)
      error("Can't open %s\n",argv[1]);
   if ((fd2=open(argv[2],O_WRONLY||O_CREAT,0777))==EOF)
      error("Can't create %s\n",argv[2]);

   while ((n = read(fd1, buf, 512)) > 0)
      if (write(fd2,buf,n)!=n)
         error("Write error\n",NULL);
}

error(s1, s2)
   char *s1, *s2;
{
   printf(s1, s2);
   exit(1);
}
```

See also: creat()

This function will set the state of and/or return the state of an option flag of MC.

int option(flag, switch);
   int flag, switch;

switch	is the designated operation on flag; 0 to reset; 1 to set; 2 to test
flag	is the designated flag to test or alter.

Description

This function can be used to alter any of the run-time option settings. The option masks are defined in the stdio header file and are as follows:

O_ERRORMSG	When reset, the I/O system will be inhibited from displaying the usual DOS error message on an I/O error. This is the default.
O_KBECHO	When set, and keyboard input will be echoed to stdout. The default is "reset".
O_BRIEF	When set, this will force the I/O system to display brief error messages if O_ERRORMSG has also been set. MC defaults this to "reset".

Return Code

The function will return FALSE if the designated flag is reset, TRUE if the designated flag is set, or EOF if the designated flag is out of range. For "switch" equal to 0 or 1, the returned flag state is the state before the reset or set (i.e the state prior to the option() request).

Example

```
#include "stdio.h"
main()
{   FILE *stream;
    option(O_ERRORMSG,1);
    if (!(stream=fopen("nosuchf.ile:7","r"))) perror("fopen error");
    else fclose(stream);
}
```

```
** Error code = 24, Returns to X'363F'
** File not in directory
File = NOSUCHF/ILE:7
Last SVC = 102, Returned to X'1A19'
fopen error: File not in directory
```

These functions are used to convert character strings of octal digits to their integer or long integer value.

```
int otoi( string );
   char *string;

long otol( string );
   char *string;

   string    is a pointer to a string containing octal digits <0-7>.
```

Description

These standard C functions are used to convert strings containing octal character digits to their (long) integer equivalent. Conversion stops as soon as the first character not in the valid range is detected. The result is returned as the function value.

Example

```
#include stdio
char inbuf[81]; int ival;
main()
{   puts("Enter your octal number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        ival=otoi(inbuf);
        printf("Your number in decimal is: %d\n",ival);
    }
}

Enter your octal number: EOF to exit
1777    |Your number in decimal is: 1023
111     |Your number in decimal is: 73
1777777 |Your number in decimal is: -1
```

See also: btoi(), btol(), xtoi(), xtol()

This function outputs a character to a machine port.

```
#option INLIB
void outport( port, value );
   int port, value;

   port      is the designated machine port.

   value     the low-order byte of value is output.
```

Description

The outport() function outputs the integer value to the port. The value is truncated to its low-order byte.

Example (partial illustration of dcal/ccc)

```
#include stdio.h
#define SELECT 0xf4
#define COMMAND 0xf0
#define STATUS COMMAND
#define RESTORE 3
#define BUSY 1
#option INLIB
int ds[4] = {1,2,4,8};
unsigned drive, ckpulse();
char buffer[81];
main()
{
    input:
    {
        fputs("Enter floppy physical drive number <0-3> : ",stdout);
        if (!gets(buffer)) exit(0);
        drive = atoi(buffer);
    }
    if (drive > 3) goto input;
    printf("Insert a disk in floppy %d and depress return",drive);
    drive = ds[drive];
    getchar();
    outport(SELECT,drive);
    outport(COMMAND,RESTORE);
    while (inport(STATUS) & BUSY)
    {
        outport(SELECT,drive);
        if (inkey()) exit(0);
     }
}
```

See also: inport()

This function produces an error message on standard output.

void perror(s);
char *s;

s is a pointer to a message string printed as a prefix to the generated message.

Description

The perror() function writes an error message corresponding to the error number contained in the global error variable, errno, to standard error output. The argument string, "s", is first written. Then a colon and blank are next written. Finally, a message describing the error number contained in "errno" is written followed by a new-line character.

The "errno" message written by perror() is obtained from sys_errlist(). Note that in order to keep program-occupied memory to a minimum, sys_errlist is implemented in MC as a function which generates the message as required rather than as an array of message strings.

Example

```
#include stdio.h
#option MAXFILES 5
char message[] = "Writing to dup'd file descriptor\n";
main()
{
    int fildes;
    if ((fildes=dup2(STDOUT,10))==EOF)
        {
        perror("main: dup2() error");
        exit(-1);
        }
    write(fildes,message,strlen(message));
}
```

```
main: dup2() error: Illegal logical file number
```

See also: errno, sys_errlist()

This plotting function supports the block graphics mode available to the CRT screen to turn on, turn off, or determine the status of any point (pixel) in the screen image.

```
#option INLIB
int pixel( funcod, xval, yval);
   int funcod, x, y;

   funcod    specifies whether the pixel is reset (0), set (1), or pointed
             (2).

   xval      specifies the horizontal coordinate.

   yval      specifies the vertical coordinate.
```

Description

The "pixel()" function can be used to point, reset, or set the pixel depending on the function code supplied as the argument. The function code (funcod) specifies the operation to be performed on the pixel. It is an integer value in the range <0-2>. These codes are used as follows:

1 Indicates "reset", which will turn off (make dark) the pixel.

2 Indicates "set", which will turn on (make light) the pixel.

3 Indicates "point", which will return the status of the specified pixel. The status will be zero (0) for reset, one (1) for set, negative one (-1) if (x1,y1) is not in the CRT image, or negative two (-2) if the specified pixel does not contain a graphic character.

The "xval" and "yval" are integers which specify the pixel position along the x-axis (horizontal) or y-axis (vertical). The value is a virtual pixel, which means that it does not have to be a position in the CRT image. The direction away from the origin is always considered to be in the positive direction (for more information on this subject, see the pmode() function).

Return Code

Pixel() returns a zero (0) to indicate that the pixel is reset; a one (1) to indicate that the pixel is set; a minus one (-1) to indicate that the point (x1,y1) is out of range (i.e. virtual and does not appear in the CRT image); a minus two (-2) to indicate that the pixel does not contain a graphics character; or a minus three (-3) to indicate that the function code passed to pixel() is invalid (not in the range <0-2>); otherwise, for functions 0 and 1, a NULL will be returned.

See also: box(), circle(), line(), point(), reset(), set()

This function is used to sense the state of a pixel in the screen image.

```
#option INLIB
int point ( xval, yval );
   int x, y;

   xval      specifies the horizontal coordinate.

   yval      specifies the vertical coordinate.
```

Description

The "point()" function will obtain the state of the specified pixel according to the designated point, (x,y). The "xval" and "yval" values specify the pixel position along the x-axis (horizontal) and y-axis (vertical) respectfively. The value is a virtual pixel, which means that it does not have to be a position in the CRT image.

"point(xval,yval)" is equivalent to "pixel(2,xval,yval)".

Return Code

Point() returns a zero (0) to indicate that the pixel is reset; a one (1) to indicate that the pixel is set; a minus one (-1) to indicate that the point (x1,y1) is out of range (i.e. virtual and does not appear in the CRT image); and a minus two (-2) to indicate that the pixel does not contain a graphics character.

See also: box(), circle(), line(), pixel(), reset() set()

This function establishes the starting address of the CRT image area.

```
#option INLIB
char *ploc( buffer );
   char *buffer;

   buffer    specifies the starting address of the plotting image area.
             Plotting functions use the CRT address unless changed by ploc().
```

Description

The ploc() function can be very powerful in creating dynamic displays. By establishing an off-CRT buffer equal in length to the CRT image area, its address can be passed via ploc() so that the plotting functions plot into the buffer. The buffer could be subsequently moved to the CRT image area with the move() function on memory-mapped video machines or via a SuperVisor Call request on banked video machines.

On machines which do not have user-accessible memory-mapped video, an address of zero is used to reference the CRT. Ploc() is initialized to reference the video region. On machines with memory-mapped video, "address" is initialized to the video memory address. Thus, if you pass some other address via ploc(), you can restore plotting to the CRT by passing this CRT memory address via another ploc() invocation. An address of zero also will reference the video RAM; thus, denoting video reference via a ploc() argument of zero will be portable across all MC releases.

Passing a zero as the address will also return the current ploc() address as the function value while it resets the referencing to the video.

Warning

The buffer must be large enough to contain the screen image.

Example

```
#option INLIB
int r; unsigned regs[6]; char *p1;
main()
{   putchar('\x0f');
    if (!(p1=alloc(1920))) exit(-1);
    memset(p1,' ',1920);
    ploc(p1);
    for (r=1;r<40;r+=3) circle(1,79,35,r);
    regs[1]=0x500; regs[3]=p1;
    call(15,regs); getchar();
}
```

See also: pixel(), pmode()

The "pmode()" function establishes the CRT image area as one of the four quadrants in the cartesian coordinate system.

```
#option INLIB
int pmode( quadrant );
   int quadrant;

   quadrant     sets the plotting image to quadrant <1-4> of the x-y plane;
                initialized to quadrant 4. If quadrant = 0, then the current
                quadrant number in effect will be returned.
```

Description

The pmode() function is quite useful when your application concerns the graphing of mathematical functions in the standard cartesian coordinate system. Since most functions are graphed in the first quadrant, a "pmode(1)" will establish the image area for that purpose. Please note that any graphics or characters currently on the screen at the time pmode() is invoked are left undisturbed; pmode() does NOT refresh the current screen contents to the revised quadrant but prepares the plotting functions for the new quadrant.

"Quadrant" is used when changing the base origin of the plot image area with the pmode() function. The image area is considered to represent only one quadrant of the x-y plane in the cartesian coordinate system. The quadrants are numbered as follows:

```
             |
     2       |     1
   ________O|O_________
           O|O
     3       |     4
             |
```

with the point 0,0 (the origin) appearing at the corner identified with the letter "O". The standard quadrant used by the plotting functions will be quadrant 4 unless changed with a pmode() function call. Remember that the direction away from the origin is always considered to be positive.

Return Code

A return code of 1, 2, 3, or 4 indicates the current quadrant in effect when a pmode(0) function is invoked. A value of EOF (-1) is returned if "quadrant" is not in the range <0-4>.

This function raises a double value to a double power.

```
#include "math.h"
double pow( argx, argy );
   double argx, argy;

   argx      is the base value to raise.

   argy      is the power to raise argx.
```

Description

This function will raise a double precision number to a double precision power and return a double precision result.

Return Code

Pow() returns 0 and sets errno to EDOM when argx is zero and argy is non-positive, or when argx is negative and argy is not an integer value. In both of these cases, a message is printed to standard error indicating DOMAIN error. When the correct value for the function would overflow or underflow the machine maximum or minimum for a double, pow() returns HUGE_VAL or zero respectively and sets errno to ERANGE. This error handling procedure may be changed via the matherr() function.

Example

```
#include stdio.h
#include math.h
double d,e,r;
main()
{  option(O_KBECHO,1);
   puts("pow: Enter your x and y values: EOF to exit");
   while (TRUE)
   {
      if (scanf("%lf %lf",&d,&e) != 2) break;
      errno=0; r = pow(d,e);
      printf("%g to the %g power = %g; errno = %d\n",d,e,r,errno);
   }
}
matherr(x) struct exception *x;
{  fprintf(stderr,"Type %d function error in %s...\n",x->type,x->name);
   fprintf(stderr,"     Args = %g,%g: ",x->arg1,x->arg2); return 0;
}

pow: Enter your x and y values: EOF to exit
2 32    | 2 to the 32 power = 4.29497e+09; errno = 0
-2 4    | -2 to the 4 power = 16; errno = 4
-5 4.7  | Type 1 function error in pow...
        |      Args = -2,4.7: argument domain error
        | -2 to the 4.7 power = 0; errno = 70
```

See also: exp(), log(), log10(), matherr(), sqrt()

These functions create a formatted image for standard output.

```
#include "math.h"            ... optional
int printf( control, arg1, arg2, ...);
   char *control; args SEE TEXT

   control   is a string containing transparent printing characters and
             conversion specifications.

   argn      arguments to be formatted for the output print image as specified
             by the control.
```

Description

The printf(...) function is used to write an output image to the standard output device and is identical to fprintf(stdout,...).

The specifications for formatting the output are determined by the character string, "control". This string will contain ordinary characters copied directly to the output image and/or specifications denoting the field conversions of all arguments. The conversion specifications are detailed under the documentation for fprintf().

Including the preprocessor statement, "#include math", will force an automatic search of the MATH/REL library during the link process and link the module which supports double precision floating point formatting.

Return Code

If an error is detected during printing, EOF will be returned; otherwise, the number of characters actually printed will be returned.

Example

```
printf("%d characters, %d lines were copied\n", bytes, lines);
```

See also fprintf(), sprintf(), scanf(), fscanf(), sscanf()

These functions are used to output a character to a file stream.

```
int putc( c, stream);
   char c; FILE *stream;

int putchar( c );
   char c;

   c         is the character to be output.

   stream    is the file pointer for the output file.
```

Description

Putc() is used to output single characters to the file stream identified by the file pointer, "stream". 'c' is any of the 256 possible character codes. If an integer value is passed it is left-truncated, so that only the least significant byte is output.

Putchar() outputs the character 'c' to the standard output file and is functionally equivalent to putc(c,stdout). Although some C implementations support putchar() as a #define macro, MC supports putchar() as a function.

Return Code

The return code is the character passed in 'c' if no errors are detected; otherwise, it will be different from the character passed in 'c'. When an error is detected, the global error variable, errno, will contain the UNIX error number which describes the error. Ferror() can also be used on the stream to obtain the DOS error number.

Example

```
if ( putc( c, fp ) != c )
     return(1);
else return(0);
```

See also: fputc(), puts(), putw()

This function is used to output a string to the standard output.

```
int puts( string );
   char *string;

   string    is the address of the string to be output.
```

Description

Puts() outputs "string" to the standard output file. All characters up to the first zero byte are output. A new-line character is then output. This function is similar to fputs(string,stdout); however, puts() adds a new-line character whereas fputs() does not.

Return Code

A NULL (zero) will be returned if no error was detected in the I/O operation, otherwise, the function will return EOF (-1) and the global error variable, errno, will contain the UNIX error number associated with the error. The DOS error may be recovered from the stream by invoking ferror().

Example

```
if (argc!=3)
{   puts("Format error:  compare file1 file2");
    exit();
}
```

See also: putc(), putw()

This function will write a "word" to a stream output.

int putw(word, stream);
int word; FILE *stream;

word	is the integer value to write.
stream	designates the output file stream.

Description

Putw() outputs a word to the designated stream. A "word" is the size of an integer.

Return Code

If an error is encountered during the output, the return code will be EOF (-1); otherwise, the word output will be returned.

Warnings

Since EOF is a valid integer value, ferror() or feof() must be used to differentiate between a -1 return and an error return. Note that since the low byte and high byte storage order of a word may vary in different machine environments, putw() is an environment dependent function.

Example

```
#include stdio.h
main()
{   FILE *stream, fopen(); int ii, i, array[10];
    if (!(stream=fopen("test.dat:7","w+"))) exit(-1);
    for ( i=0; i<10; i++)
        if (putw((ii=i*i),stream) != ii) fputs("fwrite error\n",stderr);
    rewind(stream);
    read(fileno(stream),array,sizeof(array));
    for ( i=0; i<10; i++)
        printf("%d ",array[i]);
    fclose(stream);
}
```

```
0 1 4 9 16 25 36 49 64 81
```

See also: getc(), getw(), swab()

This function implements a recursive quicker sort algorithm.

```
int qsort( base, nel, width, compar );
   char *base; unsigned int nel, width; int (*compar)();

   base      a pointer to the base of the table to sort.

   nel       is the number of elements in the table.

   width     is the size of an element in bytes.

   compar    is a pointer to a comparison function which returns -1, 0, or +1
             according to a comparison of two elements.
```

Description

This function implements Hoare's quicker sort algorithm. The implementation orders the table via a recursive invocation of the ordering function. The comparison function must return a value less than zero, equal to zero, or greater than zero depending on whether a comparison of element_a to element_b finds element_a less than, equal to, or greater than element_b.

Return Code

If qsort() cannot obtain dynamic storage for the pivot element, an EOF is returned; otherwise, a NULL is returned to indicate success.

Warning

An attempt to qsort() a very large table may result in a stack overflow with unpredictable results.

Example

```
/* excerpt from sortsym/ccc */
if (read(0, (char *) base, fsize) != fsize) exit(1); /* read SYM file */
for ( i=0, record=base, recp=prec; i<nrecs; ++i)
    *recp++ = record++;
puterr("\nSorting by name...");
if (qsort((char *) prec, nrecs, sizeof(RECORD *), compare) == EOF)
    abend("Can't sort\n");
tabify();
puterr("\nWriting by name...");
if (fputs("Alphabetic sort:\n\n", stdout)) exit(1);
for (i = 0, recp = prec; i < nrecs; ++i)
    if (write(1, (char *) *recp++, RECLEN) != RECLEN) exit(1);
```

This obtains a pseudo-random integer number in the range 0 to 32767.

```
int rand();
```

Description

This function obtains a pseudo-random number in the range 0 to 32767. It is seeded by srand() and initially seeded with a value of 1.

The following functions define the implementation of rand() and srand():

```
static unsigned long int next = 1;
int rand()
{
    next = next * 1103515245 + 12345;
    return ((unsigned int)(next/65536) % 32768);
}

void srand(seed) unsigned seed;
{
    next = seed;
}
```

Example

```
main()
{   int loop;
    for (loop=0; loop < 12; loop++)
        printf("%d ",rand());
    putchar('\n');
    srand(1);
    for (loop=0; loop < 12; loop++)
        printf("%d ",rand());
}
```

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086 2749 12767
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086 2749 12767
```

See also: frnd(), fseed(), srand()

This function is used to read a block of bytes from a file.

int read(fildes, buffer, n);
   int fildes, n; char *buffer;

fildes	is the file descriptor of the file.
buffer	a byte buffer of at least length n.
n	the number of bytes (block size) to read.

Description

The read() function will read a number of bytes equal to the "n" argument or until the file's EOF is reached. In either case, the exact number of bytes read will be returned as the function's return code. If the file is at it's EOF, zero will be returned. Use the lseek() or seek() function to position to where you want the reading to start. For sequential reads, neither lseek() nor seek() is needed. The block size which is optimum for speed of throughput will be a multiple of the file's sector size which is defined as BUFSIZ in the stdio header file.

Return Code

The number of bytes actually read will be returned. The file will be at its EOF when NULL is returned.

Example

```
#include <stdio.h>
#include <fcntl.h>
main(argc, argv) int argc; char *argv[];
{   int fd1, fd2, n; char buf[BUFSIZE];
    if (argc != 3) error("Usage: cp from to", NULL);
    if ((fd1=open(argv[1],O_RDONLY))==EOF)
        error("Can't open %s\n",argv[1]);
    if ((fd2=open(argv[2],O_WRONLY||O_CREAT,0777))==EOF)
        error("Can't create %s\n",argv[2]);
    while ((n = read(fd1, buf, 512)) > 0)
        if (write(fd2,buf,n)!=n) error("Write error\n",NULL);
}
error(s1, s2) char *s1, *s2; { printf(s1, s2); exit(1); }
```

See also: close(), open(), write()

This function changes the length of a previously allocated block.

char *realloc(ptr, size);
char *ptr; unsigned int size;

ptr	is a pointer to the current block.
size	is the size of the new block in bytes.

Description

This function may be used to increase or decrease the size of a block of memory previously allocated via malloc(), calloc(), alloc() or a previous realloc(). If the new block size is larger than the old block size, the existing contents of the block will be unaltered and the remainder of the block will be zeroed. If the new size is smaller than the old size, the contents of the block will be unaltered up to the new size.

Return Code

If no additional memory is available, realloc() returns a NULL pointer; otherwise, a pointer to the new block is returned.

Warning

If insufficient additional memory is available, the contents of the old block may be lost.

Example

```
mrealloc(ptr, newsize, oldsize) /* reallocate data space of size bytes */
    char *ptr;
    unsigned newsize, oldsize;
{   static char *p;
    static int i;
    printf("attempting to reallocate %d bytes at %#X\n", newsize, ptr - 4);
    if (! (p = realloc(ptr, newsize)))  /* couldn't do it */
        return NULL;
    printf("reallocated %d bytes at %#X\n", newsize, p - 4);
    printf("data = ");
    for (i = 0; i < newsize; ++i)
        printf("%2.2X ", p[i]);
    printf("\n");
    cfill(p, newsize - oldsize, oldsize);
    return p;
}
```

See also: alloc(), brk(), calloc(), free(), malloc()

This function is used to turn off any point (pixel) in the screen image.

```
#option INLIB
int reset ( xval, yval );
   int xval, yval;

   xval      specifies the horizontal coordinate.

   yval      specifies the vertical coordinate.
```

Description

The "reset()" function will turn off the specified pixel according to the designated point, (x,y). The "xval" and "yval" values specify the pixel position along the x-axis (horizontal) and y-axis (vertical) respectively. The value is a virtual pixel, which means that it does not have to be a position in the CRT image.

"reset(x,y)" is identical in function to "pixel(0,x,y)".

Return Code

A return code of minus one (-1) indicates that the point (xval,yval) is out of range (i.e. virtual and does not appear in the CRT image).

Example

```
for ( x=0, y=40; x<128; x++)
    set( x, y );
for ( x=0, y=40; x<128; x++)
    reset( x, y );
```

See also: box(), circle(), line(), pixel(), point() set()

This function positions a stream to its beginning.

```
void rewind( stream );
   FILE *stream;

   stream    is the stream file pointer.
```

Description

This function positions a stream to a zero offset relative to the beginning. The stream must be associated with a block file device rather than a character special device (i.e. isatty(fileno(stream)) must return FALSE). It is equivalent to "fseek(stream,0L,0);" except that no value is returned.

Example

```
#include stdio.h
main()
{   FILE *stream, fopen(); int ii, i, array[10];
    if (!(stream=fopen("test.dat:7","w+"))) exit(-1);
    for ( i=0; i<10; i++)
        if (putw((ii=i*i),stream) != ii) fputs("fwrite error\n",stderr);
    rewind(stream);
    read(fileno(stream),array,sizeof(array));
    for ( i=0; i<10; i++)
        printf("%d ",array[i]);
    fclose(stream);
}

0 1 4 9 16 25 36 49 64 81
```

See also: fseek(), ftell()

This function obtains the position of the last occurrence of a specified character within a string. It is identical to strrchr(), its UNIX System V counterpart.

```
char *rindex( s, c );
   char *s; int c;

   s     is a pointer to the source string.

   c     is the character to find.
```

Description

The rindex() function will look for the last occurrence of character 'c' in the string pointed to by "s". The low-order byte of the integer, 'c', will be used as the character for which to look. Rindex() will operate properly when 'c' is the NULL character.

Return Code

If the character 'c' is not found in string "s", NULL will be returned; otherwise, a pointer to the position of the last 'c' in "s" will be returned.

See also: index(), strcat(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strchr(), strspn()

This function is used to obtain a memory block.

```
char *sbrk( nbytes );
   unsigned int nbytes;

char *sbrk( 0 );

   nbytes    an unsigned integer number of bytes needed.
```

Description

Sbrk() reserves memory for use by a program from the system memory pool. The memory allocated by sbrk() cannot be deallocated until the program finishes execution. Alloc() uses sbrk() to request blocks of memory as needed but maintains a linked list of blocks available but unallocated. If the memory requested will only be needed for part of the execution of the program, it is recommended that either alloc(), calloc(), or malloc() be used.

If the argument passed to sbrk() is zero, then the current value of the program break ($LOMEM) will be returned. This may be useful with brk().

Return Code

The return code, "ptr", is the address of the allocated block of memory if the sbrk() was successful. If not enough memory is available to satisfy the request, "ptr" is set to NULL (0).

Warnings

Only memory allocated by sbrk(), alloc(), calloc(), malloc(), or realloc() should be used by the programmer for dynamic space. File opens and closes, including standard files, use these functions for setting up File Control Areas (FCA's). These FCAs can be clobbered if the program accesses unauthorized memory. If you inadvertantly pass a negative int as "nbytes", it will be treated as a large unsigned request.

See also: brk()

This function is used to scan a formatted print image from standard input, interpret the image fields according to a control string, and store the translated results in the arguments passed in the function's invocation.

```
int scanf( control, arg1, arg2, ... );
   char *control; SEE TEXT for type of arg1, arg2, ...

   control      is the decoding control string.

   args         as required to match the control string.
```

Description

The scanf() function is the input analog of the printf() function. It provides similar translations; however, the conversions are from ASCII string fields to argument values. Scanf() inputs from standard input [stdin].

The arguments identified as "arg1, arg2, ..." MUST BE POINTERS, rather than values since the scan function stores the converted results into the arguments. For an argument defined as an array, its name will be a pointer. For all scaler arguments, use the "address of" operator. The control string is detailed under the documentation for fscanf().

The scanf() function contained in the LIBC library supports all variable types except floats and doubles; thus "e", "f", and "g" translations will be ignored unless #option MATHLIB is specified to request the MATH/REL scanf.

Return Code

If end-of-file is reached during the input, EOF will be returned. Otherwise, the function's return value will be equal to the number of successfully matched and assigned input items.

Warnings

Arguments receiving strings must be large enough to contain the string.

Example

```
#include <stdio.h>
#include <math.h>
main()
{   char achar, string[81], float f1; int n_items, aint;
    n_items = scanf("%c%3d%*2d%6s%f",&achar,&aint,string,&f1);
    printf("Count = %d |%c %d %s %e\n",n_items,achar, \
            aint,string,(double)f1);
}

For input of: a12345abcdef 12.345e5
Prints      : Count = 4 |a 123 abcdef  1.2345E+06
```

See also: fscanf(), sscanf()

This function is used to randomly move around a file without actually reading or writing (unflushed output buffered by the system will be physically written when a seek() is performed).

```
int seek( fildes, offset, origin );
   int fildes, offset, origin;

   fildes    the file descriptor of the file.

   offset    the new position relative to origin.

   origin    specifies offset from beginning, current, or end for origin equal
             to 0, 1, or 2. For origin values of 3, 4, or 5, offset is
             multiplied by 512 prior to the seek and origin is then
             reinterpreted as 0, 1, or 2.
```

Description

The seek() function is used for random positioning in a file. Since offset is treated as a signed integer, you can only seek within 32K of a position (beginning, current, or end). For files of size greater than 64K, it is best to consistently use a 512-byte block seek followed by a block offset seek to postion to a location. Thus for large files, you can use origins of 3, 4, or 5 to seek to any given 512-byte block then invoke a subsequent seek to position to the offset from that block.

It is recommended that you use the long form of the function, lseek().

Return Code

On origins of 0, 1, or 2, the positive offset from the current 512-byte block prior to the seek will be returned. On origins of 3, 4, or 5, the number of the current 512-byte block will be returned. These numbers may be used to calculate the current record position or the number of records in a file.

If an error occurs during the seek, EOF (-1) will be returned.

Warning

The file descriptor passed as the fildes argument of the function must be one obtained from open(), creat(), dup(), dup2(), or fcntl(), or be a standard file descriptor: STDIN, STDOUT, STDERR. Also, remember that the offset is treated as a SIGNED integer. Thus, a negative offset passed with an origin of 0 or 3 is an error. You can't seek a tty device!

See also: isatty(), lseek(), fseek()

This function is used to turn on any point (pixel) in the screen image.

```
#option INLIB
int set ( xval, yval );
   int xval, yval;

   xval      specifies the horizontal coordinate.

   yval      specifies the vertical coordinate.
```

Description

The "set()" function will turn on the specified pixel according to the designated point, (x,y). The "xval" and "yval" values specify the pixel position along the x-axis (horizontal) and y-axis (vertical) respectively. The value is a virtual pixel, which means that it does not have to be a position in the CRT image.

"set(x,y)" is identical in function to "pixel(1,x,y)".

Return Code

A return code of minus one (-1) indicates that the point (xval,yval) is out of range (i.e. virtual and does not appear in the CRT image).

Example

```
for ( x=0, y=40; x<128; x++)
    set( x, y );
for ( x=0, y=40; x<128; x++)
    reset( x, y );
```

See also: box(), circle(), line(), pixel(), point() reset()

This function saves the environment for a "far" return via longjmp().

```
#include <setjmp.h>
int setjmp( env );
   jmp_buf env;

   env       is the environment saved by setjmp().

   val       is the value returned by setjmp().
```

Description

The initial call to setjmp() always returns zero. A subsequent longjmp() using the same environment will restore the program to the environment as saved in "env" by the previous call to setjmp(). After the longjmp() is completed, the program execution will continue as if the corresponding call to setjmp() had just occurred; however, the returned value will be the value passed as "val" in the longjmp() call, rather than zero. The setjmp()/longjmp() companion functions can be used in error control within nested functions so as to effect an escape from a lower-level function error without backtracking through error handling in each higher level nest.

Warning

The values of automatic and register variables are unpredictable following a longjmp(). Do not use setjmp() in any expression other than an immediate assignment or relational test.

Example

```
#include setjmp.h
jmp_buf env;
main()
{   int i, val;
    if (val=setjmp(env))
        printf("Error %d on i=%d\n",val,i);
    else
        for (i=0;i<3;++i)
            {   func1(i); printf("No error %d\n",i);    }
}
func1(i) int i;
{   func2(i);   }
func2(i) int i;
{   func3(i);   }
func3(i) int i;
{   if (i<2) return; else longjmp(env,1);   }
```

See also: longjmp()

This function obtains the trigonometric sine of a double.

```
#include <math.h>
double sin( value );
   double value;

   value     the double whose sine is desired.
```

Description

The sin() function obtains the triginometric sine of the double argument entered in radians. The result is a double.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1,d2;
main()
{   puts("sin: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        errno=0;
        d2 = sin(d1=atod(inbuf));
        printf("sin of %g = %1.14f; errno = %d\n",d1,d2,errno);
    }
}

sin: enter your number: EOF to exit
1.5707963 | sin of 1.5708 = 1.00000000000000; errno = 0
1         | sin of 1 = 0.84147098480790; errno = 0
-1        | sin of -1 = -0.84147098480790; errno = 0
```

See also: asin(), cos(), tan(), fsin()

This function obtains the hyperbolic sine of a double.

```
#include <math.h>
double sinh( value );
   double value;

   value     a double whose hyperbolic sine is desired.
```

Description

The sinh() function obtains the hyperbolic sine of the double argument entered in radians. The result is a double.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1,d2;
main()
{   puts("sinh: enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        errno=0;
        d2 = sinh(d1=atod(inbuf));
        printf("sin of %g = %1.14f; errno = %d\n",d1,d2,errno);
    }
}

sinh: enter your number: EOF to exit
.5  | sin of 0.5 = 0.52109530549375; errno = 0
1.0 | sin of 1 = 1.17520119364380; errno = 0
10.0| sin of 10 = 11013.23287470339369; errno = 0
```

See also: cosh(), tanh()

These functions create a formatted image for output to a string.

```
int sprintf( buffer, control, arg1, arg2, ... );
   char *buffer, *control; args SEE TEXT

   buffer    the string area to receive the output.

   control   is a string containing transparent printing characters and
             conversion specifications.

   argn      arguments to be formatted for the output print image as specified
             by the control.
```

Description

The sprintf() function performs the same formatted output operations as the printf() and fprintf() functions; however, the output is placed in a string buffer and is terminated by a NULL ['\0'].

The specifications for formatting the output are determined by the character string, "control". This string will contain ordinary characters copied directly to the output image and/or specifications denoting the field conversions of all arguments. The conversion specifications take the form of:

```
%{flags}{width}{.prec}char
```

and are detailed under the documentation for fprintf().

Sprintf() may be used when more flexible control over floating point formatting is required than that provided by dtoa() and ftoa().

Warning

The buffer must be large enough to contain the entire formatted string.

Return Code

The number of actual characters printed excluding the terminating NULL is returned.

Example

```
sprintf(buf,"%d characters, %d lines were copied",bytes,lines);
puts( buffer );
```

See also: dtoa(), fprintf(), fscanf(), ftoa(), printf(), scanf(), sscanf()

This function obtains the square root of a double.

```
#include <math.h>
double sqrt( value );
   double value;

   value     a double whose square root is desired.
```

Description

The sqrt() function obtains the double precision square root of its double precision argument. The result is a double.

Return Code

Sqrt() returns 0 and sets the global error variable, errno, to EDOM when "value" is negative. A message indicating DOMAIN error will also be written to standard error. These procedures may be changed via matherr().

Example

```
#include stdio.h
#include math.h
int i; char inbuf[81]; double d1,d2;

main()
{   puts("sqrt: Enter your number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        errno=0; d2 = sqrt(d1=atod(inbuf));
        printf("The square root of %g is %g; errno = %d\n",d1,d2,errno);
    }
}

matherr(x) struct exception *x;
{   fprintf(stderr,"Type %d function error in %s...\n",x->type,x->name);
    fprintf(stderr,"     Args = %g,%g: ",x->arg1,x->arg2);
    return 0;
}

sqrt: Enter your number: EOF to exit
170e6 | The square root of 1.7e+08 is 13038.4; errno = 0
-9    | Type 1 function error in sqrt...
      |      Args = -9,0: Argument domain error
      | The square root of -9 is 3; errno = 70
```

See also: exp(), log(), pow()

This function seeds the pseudo-random number generator, rand().

```
void srand( seed );
   unsigned int seed;

   seed      a seed for the rand() number generator.
```

Description

The srand() function may be invoked anytime to seed the random number generator, rand(). Rand() is initially seeded to a value of 1 by the system.

Example

```
main()
{   int loop;
    for (loop=0; loop < 12; loop++)
        printf("%d ",rand());
    putchar('\n');
    srand(1);
    for (loop=0; loop < 12; loop++)
        printf("%d ",rand());
}

16838 5758 10113 17515 31051 5627 23010 7419 16212 4086 2749 12767
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086 2749 12767
```

See also: frnd(), fseed(), rand()

These functions are used to scan a formatted print image stored in a string, interpret the image fields according to a control string, and store the translated results in the arguments passed in the function's invocation.

```
int sscanf( buffer, control, arg1, arg2, ... );
   char *buffer, *control; args SEE TEXT

   buffer       the string area containing the input image.

   control      a string containing translation control.

   args         as required to match the control string.
```

Description

The sscanf() function is the input analog of the sprintf() function. It provides similar translations; however, the conversions are from ASCII string fields to argument values.

The arguments identified as "arg1, arg2, ..." MUST BE POINTERS, rather than values since the scan functions store the converted results into the arguments. The format of the control string is detailed under the documentation for scanf().

If the end of the string is reached during the input, a newline character will be automatically supplied to serve as a terminating whitespace.

A typical use for sscanf() is to provide reformatting of variables under controlled conditions using internal memory buffers along with the complementary function, sprintf().

Return Code

When the end of the string is reached during the input, sscanf first treats it as a new-line character for the purpose of terminating the scanned field by whitespace. Subsequent requests for "input" from the string will return EOF. Otherwise, the function's return value will be equal to the number of successfully matched and assigned input items.

See also: fscanf(), scanf(), sprintf()

This function appends one string to another.

char *strcat(dest, source);
char *dest, *source;

dest	is a pointer to the destination string.
source	is a pointer to the source string.

Description

The strcat() function will concatenate (append) the source string to the destination string. The "dest" argument is a pointer to a character array which must be of sufficient size to contain the appended string.

Return Code

A pointer to the "dest" string is returned.

Example

```
char string[81], lilbuf[9];
main()
{
    strcpy(string,"Today is ");
    strcat(string,sysdate(lilbuf));
    strcat(string," and the time is ");
    strcat(string,systime(lilbuf));
    puts(string);
}
```

```
Today is 11/21/85 and the time is 14:50:56
```

See also: strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function finds the first occurrence of a character in a string.

```
char *strchr( s, c );
   char *s; int c;

   s         is a pointer to the source string.

   c         is the character to find.
```

Description

The strchr() function will look for the first occurrence of character 'c' in the string pointed to by "s". The low-order byte of the integer, 'c', will be used as the character. Strchr() will operate correctly when 'c' is the NULL character.

Return Code

If the character 'c' is not found in string "s", NULL will be returned; otherwise, a pointer to the position of 'c' in "s" will be returned.

Example

```
char *p, string[81];
main()
{
    puts("Enter your string; EOF to exit");
    for (;;)
    {
        if (!gets(string)) break;
        p=string;               /* point p to string[0] */
        while (p=strchr(p,'.')) /* until no more dots */
            *p='?';             /* replace '.' with '?' */
        puts(string);
    }
}

Enter your string; EOF to exit
this is a test.                  | this is a test?
This "..." is an elipsis of dots. | This "???" is an elipsis of dots?
```

See also: strcat(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function compares two strings

```
int strcmp( string_1, string_2 );
   char *string_1, *string_2;

   string_1     is a pointer to the first string.

   string_2     is a pointer to the second string.
```

Description

This function compares "string_1" to "string_2" and returns an integer less than zero, equal to zero, or greater than zero, depending on whether "string_1" is lexicographically less than, equal to, or greater than "string_2".

Return Code

The return code of strcmp() is <0, 0, or >0 as noted above.

Example

If you are unfamiliar with how ASCII strings are "ordered", perhaps a strong example will clarify this discussion. The following is an ordered list of strings in ascending order:

```
a8bcde
abc
abcd
jim
karl
rich
roy
this_is_a_long_string
```

Keep this ordered list in mind in the following examples. The statement:

```
if (strcmp("abc","a8bcd") < 0) ? printf("above") : printf("below");
```

should print the word, "below" since the string, "abc" is below the string "a8bcd" in an ascendingly sorted list. The statement:

```
if (strcmp("abc","abcd") < 0) ? printf("above") : printf("below");
```

should print the word, "above" since the string, "abc" is above the string "abcd" in an ascendingly sorted list.

See also: strcat(), strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function copies one string into another.

```
char *strcpy( dest, source );
   char *dest, *source;

   dest      is a pointer to the destination string.

   source    is a pointer to the source string.
```

Description

The strcpy() function copies an image of the source string to the destination string buffer. Strcpy() is useful to initialize an allocated but unzeroed character array to known string value. You cannot use strcat() to copy a string into a buffer unless either the first element of the buffer is known to be NULL or the buffer contains a string; thus, strcpy() is useful in order to "prime" a string buffer.

Warning

The destination character array must be large enough to contain the source string.

Return Code

A pointer to the "dest" string is returned.

Example

```
char string[81], lilbuf[9];
main()
{
    strcpy(string,"Today is ");         /* initialize output string */
    strcat(string,sysdate(lilbuf));     /* add date to string */
    strcat(string," and the time is ");
    strcat(string,systime(lilbuf));     /* add time to string */
    puts(string);
}
```

```
Today is 11/21/85 and the time is 14:50:56
```

See also: strcat(), strchr(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function obtains the length of the initial sub-string of a first string composed entirely of characters not found in a second string.

```
int strcspn( string1, string2 );
   char *string1, string2;

   string1      is a pointer to the first string.

   string2      is a pointer to the second string.
```

Description

This function will scan "string_1" for a substring of characters beginning with the first character of "string_1" which is composed entirely of characters not found in "string_2". Both the "string_1" and "string_2" arguments should be pointers to character arrays.

Return Code

The length of the resulting substring is returned.

Example

```
char string[81], *field = "aeiou";
main()
{   int count;
    puts("Enter your string; EOF to exit");
    for (;;)
    {
        if (!gets(string)) break;
        if (count=strcspn(string,field))
        {
            *(string+count)='\0';
            printf("Substring <%s> contains no vowels\n",string);
        }
        else
            puts("A vowel starts your entry");
    }
}

Enter your string; EOF to exit
"rhythm" is a great word for the game of hangman!
Substring <"rhythm" > contains no vowels
```

See also: strcat(), strchr(), strcpy(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function replaces a substring of one string with another string.

```
char *strepl( source, dest, pos, count );
   char *source, *dest; int pos, count;

   count        is the destination sub-string length.

   dest         is a pointer to the destination string.

   pos          starting index position or array subscript.

   source       is a pointer to the source string.
```

Description

The strepl() function replaces that portion of the destination string starting at relative position "pos" and continuing for "count" characters [the destination substring] with the source string.

The argument "pos" represents a starting position relative to the beginning of the destination string. It is essentially used as an index or subscript into the "dest" character array.

The argument, "count", controls how much of the destination string is to be replaced. If "count" is zero (0), then an insert operation is performed without deleting any characters of the destination string.

The length of the replacement string is the length of the entire source string. If the source string is null (i.e. of zero length), then only the identified destination sub-string is deleted.

Warning

The destination character array must be of a size sufficient to contain the generated string.

Return Code

If "dest+pos" exceeds the bounds of the destination string, a NULL (0) will be returned and the string operation will be aborted; otherwise, a pointer to the destination string will be returned. For instance:

```
if (!strepl(s, "destination_string", 24, 3)) puts("String error!");
```

will result in the error message display since the position, 24, is not in the range of the destination string.

See also: strcat(), strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function replicates a string.

```
#option INLIB
char *strept( dest, source, repeat );
   char *dest, *source; int repeat;

   dest      is a pointer to the destination string.

   source    is a pointer to the source string.

   repeat    a repetition counter.
```

Description

The strept() function replicates the source string into the destination string the number of times indicated by "repeat". Both "source" and "dest" represent pointers to character arrays. Note that the replication uses the entire source string and not just the first source string character.

Return Code

The function returns a pointer to the destination string.

Warning

The destination character array must be of a size sufficient to hold the resultant string generation.

Example

```
#option INLIB
char string[81], *repeats[5] = {"*","*=.","*_* ","{[(<>)]}","=="};
main()
{   int loop;
    for (loop=0; loop<5; loop++)
        puts(strept(string,repeats[loop],5));
}

*****
*=.*=.*=.*=.*=.
*_* *_* *_* *_* *_* 
{[(<>)]}{[(<>)]}{[(<>)]}{[(<>)]}{[(<>)]}
==========
```

See also: strepl(), stright(), strleft(), strmid(), strfind()

This function will obtain the position of a substring in a string.

```
#option INLIB
char *strfind( dest, source, pos );
   char *dest, *source; int pos;

   dest      is a pointer to the destination string.

   pos       starting index position or array subscript.

   source    is a pointer to the source string.
```

Description

The strfind() function will search the destination string for the first appearance of the source string. The destination string will be searched starting at the position "dest+pos". Both "source" and "dest" represent pointers to character arrays containing the respective strings. "Pos" is the starting position relative to the beginning of the destination string.

Return Code

If the source string is a null string, the value of "dest+pos" will be returned. If the destination string is a null string or if the source string is not found (i.e. is not a sub-string of the destination), a NULL (0) will be returned. If the source string is found in the destination string, a pointer to its position in the destination string will be returned.

Example

```
#option INLIB
char *p, *string = "This is string one";
char *find[5] = {"hello","is","is","is","is"};
int loop, pos[5] = { 0,0,3,8,20 };

main()
{   for (loop = 0; loop < 5; loop++)
        if(p=strfind(string,find[loop],pos[loop]))
            printf("Found %s[%d]: %s\n",find[loop],pos[loop],p);
        else
            printf("Could not find %s[%d]\n",find[loop],pos[loop]);
}

Could not find hello[0]
Found is[0]: is is string one
Found is[3]: is string one
Could not find is[8]
Could not find is[20]
```

See also: strepl(), strept(), strleft(), strmid(), strfind()

This function obtains the rightmost substring of a string.

```
#option INLIB
char *stright( dest, source, count );
   char *dest, *source; int count;

   count     is the integer sub-string length.

   dest      is a pointer to the destination string.

   source    is a pointer to the source string.
```

Description

The stright() function will copy the rightmost "count" characters (the sub-string) of the source string to the destination string. This is NOT an append operation. The destination string is replaced with the sub-string. "Count" indicates the length of desired substring. If "count" is zero, the destination becomes a null string. If the "count" is greater than the source string length, the entire source string is copied.

Warning

The destination character array should be dimensioned large enough to contain the entire sub-string and terminating NULL.

Return Code

The function returns a pointer to the resulting string.

Example

```
#option INLIB
char *string = "This is a test string", buf[100];
main()
{   int loop;
    for (loop=5; loop<30; loop+=5)
        puts(stright(buf,string,loop));
}

tring
est string
s a test string
his is a test string
This is a test string
```

See also: strepl(), strept(), strleft(), strmid(), strfind()

This function obtains the leftmost substring of a string.

```
#option INLIB
char *strleft( dest, source, count );
   char *dest, *source; int count;

   count     is the integer sub-string length.

   dest      is a pointer to the destination string.

   source    is a pointer to the source string.
```

Description

The strleft() function will copy the leftmost "count" characters (the sub-string) of the source string to the destination string. This is NOT an append operation. The destination string is replaced with the sub-string. "Count" indicates the length of the desired substring. If "count" is zero, the destination becomes a null string. If the "count" is greater than the source string length, the entire source string is copied.

Warning

The destination character array should be large enough to contain the entire substring and terminating NULL.

Return Code

The function returns a pointer to the resulting string.

Example

```
#option INLIB
char *string = "This is a test string", buf[100];
main()
{   int loop;
    for (loop=5; loop<30; loop+=5)
        puts(strleft(buf,string,loop));
}

This
This is a
This is a test
This is a test strin
This is a test string
```

See also: strepl(), strept(), stright(), strmid(), strfind()

This function obtains the length of a string.

```
int strlen( source );
   char *source;

   source    is a pointer to the source string.
```

Description

The strlen() function returns the length of the source string. The length of a string is the number of characters up to but not including the first NULL.

Return Code

The return code of strlen() is the length of the target string.

Example

```
#include stdio.h
char buffer[81];
main()
{
    puts("strlen: Enter your string; EOF to exit");
    while (TRUE)
        {
        if (!gets(buffer)) break;
        printf("Your string length is %d\n",strlen(buffer));
        }
}

strlen: Enter your string; EOF to exit
this is a test string                   |Your string length is 21
when in the course of human events      |Your string length is 34
she sells sea shells down by the seashore|Your string length is 41
to err is human; to forgive is divine   |Your string length is 37
```

See also: strcat(), strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function will obtain a substring from a string.

```
#option INLIB
char *strmid( dest, source, pos, count);
   char *dest, *source; int pos, count;
```

count	is the integer sub-string length.
dest	is a pointer to the destination string.
pos	starting index position or array subscript.
source	is a pointer to the source string.

Description

Strmid() replaces the destination string with a substring of "count" characters starting at the source string position, "source+pos". "Pos" is the starting position relative to the beginning of the source string. Both "dest" and "source" represent pointers to character arrays. "Count" indicates the length of the desired substring. If "count" is zero, the destination string will be null. "Count" may be greater than the length of "source+pos".

Warning

The destination array must be a size sufficient to hold the sub-string.

Return Code

If "source+pos" exceeds the bounds of the source string, a NULL (0) will be returned and no string replacement will occur; otherwise, strmid() will return a pointer to the destination string. For instance:

```
if ( !strmid(s,"error",6,3) ) puts("String error!");
```

will result in the error message display since the position, 6, is not in the range of the string, "error".

Example

```
#option INLIB
char *string = "This is a very long test string", buf[100];
main()
{   int loop;
    for (loop=0; loop<30; loop+=5)
        puts(strmid(buf,string,loop,11));
}
```

```
 This is a v |
 is a very l |
 very long t |
 long test s |
 test string |
 string
```

See also: strepl(), strept(), stright(), strleft(), strfind()

This function appends at most "n" characters of one string to another.

```
char *strncat( dest, source, n );
   char *dest, *source; int n;

   dest      is a pointer to the destination string.

   source    is a pointer to the source string.

   n         the maximum number of characters to append
```

Description

The strncat() function will concatenate (append) at most "n" characters of the source string to the destination string. If "n" is greater than strlen(source) then the entire source string will be appended.

Warning

The "dest" character array must be large enough to contain the resulting string.

Return Code

A pointer to the "dest" string is returned.

Example

```
char *init = "Now is the time for all good men ",
     *string = "to come to the aid of their country",
      buf[100];
main()
{   int loop;
    for (loop=10; loop<40; loop+=5)
    {
        strcpy(buf,init);   /* init buf */
        puts(strncat(buf,string,loop));
    }
}

Now is the time for all good men to come to
Now is the time for all good men to come to the
Now is the time for all good men to come to the aid o
Now is the time for all good men to come to the aid of the
Now is the time for all good men to come to the aid of their co
Now is the time for all good men to come to the aid of their country
```

See also: strchr(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function compares at most "n" characters of one string to another.

```
int strncmp( string_1, string_2, n );
   char *string1, *string_2; int n;

   string_1  is a pointer to the first string.

   string_2  is a pointer to the second string.

   n         the maximum number of characters to compare.
```

Description

This function will compare at most "n" characters of "string_1" to "string_2" and return and integer less than zero, equal to zero, or greater than zero, depending on whether "string_1" is lexicographically less than, equal to, or greater than "string_2".

Return Code

The return code of strcmp() is <0, 0, or >0 as noted above.

Example

If you are unfamiliar with how ASCII strings are "ordered", perhaps a strong example will clarify this discussion. The following is an ordered list of strings in ascending order:

```
a8bcde
abc
abcd
rich
roy
this_is_a_long_string
```

Keep this ordered list in mind in the following examples. The statement:

```
if (strncmp("abc","a8bcd",2)<0) ? printf("above") : printf("below");
```

should print the word, "below" since the string, "ab" is below the string "a8" in an ascendingly sorted list. The statement:

```
if (strncmp("abc","abcd",4)<0) ? printf("above") : printf("below");
```

should print the word, "above" since the string, "abc" is above the string "abcd" in an ascendingly sorted list.

See also: strcat(), strchr(), strcmp(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strrchr(), strspn()

This function copies at most "n" characters of one string into another.

char *strncpy(dest, source, n);
   char *dest, *source; int n;

dest	is a pointer to the destination string.
source	is a pointer to the source string.
n	the maximum number of characters to copy.

Description

The strncpy() function copies an image of at most "n" characters of the source string to the destination string buffer. If the length of "source" is equal to or greater than "n", the result will NOT be null-terminated. If the length of "source is less than "n", NULLs will be added to "dest".

Return Code

A pointer to the "dest" string is returned.

Example

```
char  *string = "123456789012345678901234567890", buf[100];
main()
{   int loop;
    for (loop=25; loop; loop-=5)
    {
        strncpy(buf,string,loop);   /* copy ast most n chars */
        *(buf+loop)='\0';           /* add NULL, in case! */
        puts(buf);                  /* display the result */
    }
}
```

```
1234567890123456789012345
12345678901234567890
123456789012345
1234567890
12345
```

**See also: strcat(), strchr(), strcpy(), strcspn(), strncat(), strpbrk(),
strrchr(), strspn()**

This function finds the first occurrence in one string of any character in another.

```
char *strpbrk( string_1, string_2 );
   char *string_1, *string_2;

   string_1     is a pointer to the first string.

   string_2     is a pointer to the second string.
```

Description

This function will scan through "string_1" and return a pointer to the first character in "string_1" which is also found in "string_2".

Return Code

A NULL will be returned if no character of "string_1" is contained in "string_2"; otherwise, a pointer to the matching character in "string_1" is returned.

Example

```
#option INLIB
char *init = "the quick brown fox jumped over the lazy dogs back",
     *find[4] = {"aeiou","abc","xyz",",.;:?"}, *p;
main()
{   int loop;
    puts(init);
    for (loop=0; loop<4; loop++)
    {
        if (p=strpbrk(init,find[loop]))
            printf("Found char in set [%s] at init+%d\n", \
                   find[loop],p-init);
        else
            printf("No chars from set [%s] in init\n", \
                   find[loop]);
    }
}
```

```
the quick brown fox jumped over the lazy dogs back
Found char in set [aeiou] at init+2
Found char in set [abc] at init+7
Found char in set [xyz] at init+18
No chars from set [,.;:?] in init
```

See also: strcat(), strchr(), strcmp(), strcpy(), strcspn(), strncat(), strncpy(), strrchr(), strspn()

This function finds the last occurrence of a character in a string.

```
char *strrchr( s, c );
   char *s; int c;

   s      is a pointer to the source string.

   c      is the character to find.
```

Description

The strrchr() function will look for the last occurrence of character 'c' in the string pointed to by "s". The low-order byte of the integer, 'c', will be used as the character for which to look. Strrchr() will operate properly when 'c' is the NULL character.

Return Code

If the character 'c' is not found in string "s", NULL will be returned; otherwise, a pointer to the position of the last 'c' in "s" will be returned.

Example

```
int find[4] = {'a','e','i','o'}; char *p, *rule =
"0123456789012345678901234567890123456789012345678901234567890123456789012345",
*init =
"this function finds the last occurrence of a character in a string";
main()
{   int loop;
    puts(rule); puts(init);
    for (loop=0; loop<4; loop++)
        {
        if (p=strrchr(init,find[loop]))
            printf("Found last char %c at init+%d\n",find[loop],p-init);
        else
            printf("Char %c not in init\n",find[loop]);
        }
}
```

```
0123456789012345678901234567890123456789012345678901234567890123456789012345
this function finds the last occurrence of a character in a string
Found last char a at init+58
Found last char e at init+52
Found last char i at init+63
Found last char o at init+40
```

See also: strcat(), strcpy(), strcspn(), strncat(), strncpy(), strpbrk(), strchr(), strspn()

This function obtains the length of the initial substring of a first string composed entirely of characters found in a second string.

```
int strspn( string_1, string_2 );
   char *string_1, string_2;

   string_1     is a pointer to the first string.

   string_2     is a pointer to the second string.
```

Description

The "strspn" function will scan "string_1" for a substring of characters beginning with the first character of "string_1" which is composed entirely of characters found in "string_2". The "string_1" and "string_2" arguments represent pointers to character arrays which contain the strings.

Return Code

The length of the resulting substring is returned.

Example

```
char string[81], substr[81], *field = "asdfghjkl";
main()
{   int count;
    puts("Enter your string; EOF to exit");
    for (;;)
    {
        if (!gets(string)) break;
        if (count=strspn(string,field))
        {
            *(string+count)='\0';   /* terminate the substring */
            printf("Substring <%s> on home row\n",string);
        }
        else
            puts("String does NOT start on home row");
    }
}

Enter your string; EOF to exit
dash   |Substring <dash> on home row
chaff  |String does NOT start on home row
shaft  |Substring <shaf> on home row
```

See also: strcat(), strchr(), strcmp(), strcpy(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr()

This function sets control information concerning a file.

```
#include <sgtty.h>
int stty( fildes, argp );
   int fildes; struct sgttyb *argp;

   fildes    is a descriptor of the file.

   argp      is a pointer to the data structure.
```

Description

This function sets control information on files. The data to be set is stored in the data structure pointed to by the "argp" argument and has been previously obtained via a call to gtty() or ioctl(). The structure is defined in the "sgtty" header file and is as follows:

```
struct sgttyb {
   char sg_colctr;        /* column counter */
   char sg_control;/* control byte */
   char sg_flag;          /* FCA flag byte */
};
```

The "sgttyb.sg_colctr" element stores a counter for the current column position of the output image for the device. This item is relative to zero and is reset to zero when a new-line is sent to the fildes. It is initialized to zero when the filedes is opened. The "sgttyb.sg_control" byte stores various bit fields and is masked as follows:

```
IO_BREAK    0x10      /* mask for #option BREAK */
IO_TABSTOP  0x0f      /* mask for "tabstop-1" */
IO_FDCOE    0x80      /* mask for "close_on_exec" flag */
```

The stty() function is equivalent to:

```
ioctl(fildes,TIOCSETP,argp);
```

TIOCSETP is defined in the "sgtty" header file. The function is typically used to establish tab stops at other than the default of every 8 columns or to establish the BREAK key value as EOF on an input device. Note that for tabs, IO_TABSTOP is one less than the stop value (i.e. for every eight columns, use 7).

Return Code

If the call was successful, NULL will be returned; otherwise, an EOF will be returned and the global error variable, errno, will contain the UNIX error number associated with the error.

See also: ioctl(), gtty()

This function copies bytes from one region to another with swapping.

void swab(from, to, nbytes);
char *from, *to; int nbytes;

from	is a pointer to the region to copy.
to	points to the region to copy into.
nbytes	is the number of bytes to swap.

Description

This function copies "nbytes" from the location pointed to by "from" into the location pointed to by "to", while it exchanges odd and even bytes during the copy operation. The value, "nbytes", should be even and non-negative. The function is typically used to prepare integer data on the host machine for transmission to another machine which uses a reverse order for storing 2-byte words.

Example

```
int loop, flipped[5], words[5] = {1,10,100,1000,10000};
char *p;
main()
{
    swab(words,flipped,sizeof(words));
    fputs("Before swab: ",stdout); p = words;
    for (loop=0; loop<sizeof(words); loop++)
        printf("%02x ",*p++);
    fputs("\nAfter swab : ",stdout); p = flipped;
    for (loop=0; loop<sizeof words ; loop++)
        printf("%02x ",*p++);
}

Before swab: 01 00 0a 00 64 00 e8 03 10 27
After swab : 00 01 00 0a 00 64 03 e8 27 10
```

See also: getw(), putw()

This function returns a pointer to an error message for a UNIX error.

```
char *sys_errlist( errnum );
   int errnum;

   errnum    is the error number of the desired message.
```

Description

The sys_errlist() function is normally used by perror() to post an error message to standard error. It can be used to obtain the error message string associated with any UNIX error. In this implementation, sys_errlist() uses the DOS error message dictionary to produce the error message string for selected errors which have a near-UNIX counterpart - although the actual text may not be that which is printed in some manuals covering the UNIX system.

Warning

UNIX implements sys_errlist as an array of strings; however, in order to keep program-occupied memory to a minimum, sys_errlist is implemented in MC as a function which generates the message as required rather than an array of message strings. This is non-standard!

Note that the returned character pointer references static data which is subject to change.

Example

```
extern char *sys_errlist();
main()
{
   int i;
   for (i=0; i<75; i++)
       printf("%d = %s\n",i,sys_errlist(i));
}
```

See also: errno, perror()

This function returns the DOS 8-character date string.

```
char *sysdate( sdate );
   char *sdate;

   sdate     is a pointer to your 9-character buffer.
```

Description

This function obtains the 8-character date string from the system and places it into your buffer appending a NULL following the eighth character. The string is in the format:

```
MM:DD:YY\0
```

Warning

Your buffer must be a minimum of 9 characters in length.

Example

```
char buf[9];
main()
{
    printf("The date is %s and from buf: %s\n",sysdate(buf),buf);
    printf("The time is %s and from buf: %s\n",systime(buf),buf);
}

The date is 11/25/85 and from buf: 11/25/85
The time is 13:40:19 and from buf: 13:40:19
```

See also: asctime(), ctime(), localtime(), systime(), time()

This function is used to invoke a DOS command from within a program and then return to your program when the DOS command completes.

```
int system( cmdstr );
   char *cmdstr;

   cmdstr    a pointer to the string which contains the DOS command which is
             to be executed.
```

Description

If you want to execute a command and return to your program, use the system() function. This function will pass the command string pointed to by the pointer argument or the command string passed as the function's argument to the DOS command interpreter. Upon completion of the command, control will be returned to the running program. Your program and variables will be saved during the execution of the command.

Return Code

A return code of zero (0) will be retrieved If the executing command returns through @EXIT. If the @ABORT exit is taken, the return code generated will be obtained from the value contained in register pair "HL". If this value is positive (i.e. bit 15 reset), it becomes the return code. If the value is negative (i.e. bit 15 set), then a negative one (-1) will be returned.

Example

```
#include stdio.h
char buf[81];
main()
{   puts("Test of system()\n");
    system("LIB");
    puts("Hit EOF to exit back to DOS");
    while (TRUE)
    {   puts("Enter command: ");
        if (!gets(buf)) break;
        printf("\nReturn code is %d\n",system(buf));
    }
    puts("Now leaving system()...");
}
```

See also: cmdi(), execl(), execv()

This function returns the DOS 8-character time string.

```
char *systime( s_time );
   char *s_time;

   s_time    is a pointer to your 9-character buffer.
```

Description

This function obtains the 8-character time string from the system and places it into your buffer appending a NULL following the eighth character. The string is in the format:

```
HH:MM:SS\0
```

Warning

Your buffer must be a minimum of 9 characters in length.

Example

```
char buf[9];
main()
{
    printf("The date is %s and from buf: %s\n",sysdate(buf),buf);
    printf("The time is %s and from buf: %s\n",systime(buf),buf);
}

The date is 11/25/85 and from buf: 11/25/85
The time is 13:40:19 and from buf: 13:40:19
```

See also: asctime(), ctime(), localtime(), sysdate(), time()

This function obtains the double precision trigonometric tangent of an angle given in radians.

```
#include <math.h>
double tan( argx );
   double argx;

   argx      is the angle in radians for which the tangent is desired.
```

Description

The function obtains the double precision trigonometric tangent of a double precision argument entered in radians. The result is double precision.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1, d2;
main()
{
    puts("tan: Enter your number: EOF to exit");
    while (TRUE)
    {
        if (!gets(inbuf)) break;
        d2 = tan(d1=atod(inbuf));
        printf("The tangent of %g is %e\n",d1,d2);
    }
}

tan: Enter your number: EOF to exit
1    |The tangent of 1 is 1.557408e+00
1.57 |The tangent of 1.57 is 1.255766e+03
1.6  |The tangent of 1.6 is -3.423253e+01
```

See also: ftan(), sin(), cos()

This function obtains the hyperbolic tangent for an angle in radians.

```
#include <math.h>
double tanh( argx );
   double argx;

   argx      is the angle in radians for which the hyperbolic tangent is
             desired.
```

Description

This function obtains the double precision hyperbolic tangent of its double precision argument entered in radians. It returns a double precision result.

Example

```
#include stdio.h
#include math.h
char inbuf[81]; double d1, d2;

main()
{
    puts("tanh: Enter your number: EOF to exit");
    while (TRUE)
    {
        if (!gets(inbuf)) break;
        d2 = tanh(d1=atod(inbuf));
        printf("The hyperbolic tangent of %g is %e\n",d1,d2);
    }
}

tanh: Enter your number: EOF to exit
0.5 |The hyperbolic tangent of 0.5 is 4.621172e-01
1.0 |The hyperbolic tangent of 1 is 7.615942e-01
3.0 |The hyperbolic tangent of 3 is 9.950548e-01
```

See also: sinh(), cosh()

This function obtains the current position of a file.

```
long tell( fildes );
   int fildes;

   fildes    is the file descriptor of the file.
```

Description

Tell() obtains the current position of the file identified by the file descriptor, "fildes". The function is identical to:

```
lseek( fildes, 0L, 1)
```

and is considered obsolete.

Return Code

The file's position relative to the beginning of the file is returned as a long integer.

See also: lseek(), open(), creat(), fseek()

This function obtains the UNIX time measured in seconds.

```
long time( 0 );
long time( tloc );
   long *tloc;

   tloc      is a non-zero pointer where the returned value will also be
             stored.
```

Description

The UNIX time is defined as the number of seconds since 00:00:00 standard local time January 1, 1970. As long as "tloc" is not zero, the return value is also stored at the location pointed to by "tloc".

Example

```
main()
{
    long tod;
    time(&tod);
    printf("Time now is %ld seconds since 1/1/70\n",tod);
    printf("  in ASCII is %s",ctime(&tod));
}
```

```
Time now is 501780054 seconds since 1/1/70
  in ASCII is Mon Nov 25 15:20:54 1985
Time now is 501780066 seconds since 1/1/70
  in ASCII is Mon Nov 25 15:21:06 1985
```

See also: asctime(), ctime(), localtime(), systime(), sysdate()

This function is used to convert a character to ASCII.

```
char toascii( c );
   char c;

   c      is the character under test.
```

Description

Toascii() converts the character under test to ASCII by stripping bit_7 from the character passed as the argument.

Return Code

The function will return the converted character, as required.

Example

```
char *p,*input="ThIs Is A f\xf5NnY \xD3tRinG";
main()
{
    p=input-1;
    while (*++p)
        *p=toascii(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=tolower(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=toupper(*p);
    puts(input);
}
```

```
ThIs Is A fuNnY StRinG
this is a funny string
THIS IS A FUNNY STRING
```

See also: tolower(), toupper()

This function is used to convert a character from upper or lower case to lower case.

```
char tolower( c );
   char c;

   c     is the character under test.
```

Description

Tolower is used to convert an upper-case character <'A' through 'Z'> to a lower-case alphabetic <'a' through 'z'>. The function affects only alphabetic characters; numbers, special symbols, etc., are returned unaltered.

Return Code

The function will return the converted character, as required.

Example

```
char *p,*input="ThIs Is A f\xf5NnY \xD3tRinG";
main()
{
    p=input-1;
    while (*++p)
        *p=toascii(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=tolower(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=toupper(*p);
    puts(input);
}
```

```
ThIs Is A fuNnY StRinG
this is a funny string
THIS IS A FUNNY STRING
```

See also: toascii(), toupper()

This function converts a character from lower or upper case to upper case.

```
char toupper( c );
   char c;

   c     is the character under test.
```

Description

Toupper() performs the function opposite to tolower(); a lower case character is converted to upper case. The function affects only alphabetic characters; numbers, special symbols, etc., are returned unaltered.

Return Code

The function will return the converted character, as required.

Example

```
char *p,*input="ThIs Is A f\xf5NnY \xD3tRinG";
main()
{
    p=input-1;
    while (*++p)
        *p=toascii(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=tolower(*p);
    puts(input);
    p=input-1;
    while (*++p)
        *p=toupper(*p);
    puts(input);
}
```

```
ThIs Is A fuNnY StRinG
this is a funny string
THIS IS A FUNNY STRING
```

See also: toascii(), lower()

This function obtains the file name of the terminal device associated with an open file descriptor.

char *ttyname(fildes);
int fildes;

fildes is the file descriptor for which the terminal name is desired.

Description

This function returns a pointer to the name of the character special file (i.e. device) identified by the file descriptor, "fildes".

Warning

The returned pointer references static data which is subject to change.

Example

```
/* cktty/ccc */
extern char *ttyname();
main()
{
    int fildes;
    for (fildes=0; fildes<3; fildes++)
        printf("The name of fildes %d is %s\n",fildes,ttyname(fildes));
}

cktty #*pr

The name of fildes 0 is *KI
The name of fildes 1 is *DO
The name of fildes 2 is *PR
```

See also: isatty()

These functions can be used to return a single character to a file stream.

```
int ungetc( c, stream );
   char c; FILE *stream;

int ungetch( c );
   char c;

   c         the character or byte to un-get.

   stream    the file pointer obtained from fopen().
```

Description

The opportunity may present itself where a character that has been obtained via getc() is not needed and should be left available for the next getc() invocation. The ungetc() function can be used to store this character in a one-character buffer associated with the file stream.

The ungetch() function is identical to the construct, ungetc(c,stdin);. Note, the scanf() function uses ungetc().

Return Code

If the one-character buffer is already storing a character from a previous ungetc(), EOF will be returned; otherwise, NULL will be returned.

Warnings

The file pointer passed as the stream argument of the function must be one obtained from fopen(). Do not use ungetch(fpup(fildes)) as the block file I/O functions do not utilize the ungetc() buffer. The ungetc() storage is shared with the storage of the last DOS error number encountered.

Example

```
/* after newline, strip line feed if present */
if (( c=getchar()) != 'x0a') ungetch(c);
```

This function is used to delete a file from disk.

```
int unlink( path );
   char *path;

   path      the file specification of the file that is to be removed.
```

Description

This function performs the same operation as the DOS function to delete a file [LDOS - KILL, TRSDOS - REMOVE]. The file must not currently be open in the MC file system.

Return Code

The return code will be NULL if the file is unlinked without error. A return code of EOF indicates that an error has occurred and the global error variable, errno, will contain the UNIX error number associated with the error.

Warning

It is an error to try to unlink a file which is currently open [via fopen(), open(), dup(), dup2(), or creat()] in the MC file system and unlink() will not permit the deletion.

Example

```
#include <stdio.h>
main(argc,argv)  int argc; char *argv[];
{
    FILE *fp;
    if ( argc==1 )
        { puts("Usage: testunlk filespec [inhibit]\n"); exit(-1); }
    if (argc == 3)
        if (( fp=fopen(argv[1],"R"))==NULL )
            { printf("Error in opening %s\n",argv[1]); exit(-1); }
    printf("Unlink's return code = %d\n",unlink(argv[1]));
}
```

Write() is used to write a block of bytes to a file.

int write(fildes, buffer, n);
int fildes, n; char *buffer;

fildes	the file descriptor of the file.
buffer	a buffer containing the bytes to write.
n	the number of bytes (block size) to write

Description

The write() function will write a number of bytes equal to the "n" argument or until an error in writing is detected. Use lseek() or seek() to position to the start of where writing is to occur. If you are writing sequentially, a seek operation is not needed. The block size which is optimum for speed of throughput will be a multiple of the file's sector size which is defined as BUFSIZ in the "stdio" header file.

Return Code

The exact number of bytes written will be returned as the function's return code. If this number does not equal "n", then an error has occurred and the global error variable, errno, will contain the UNIX error number which describes the error. The DOS error number may be obtained with ferror().

Example

```
creat_file(name) char *name;
{   int i,fd;
    if ((fd=creat(name,0777))==EOF) open_error(name);
    for (i=0;i<10000;i++)
    {
        if ((write(fd,itoa(i,record),10))!=10)
            { printf("Error in writing %s\n",name); exit(-1); }
        cursor(10,3);            /* position cursor */
        printf("Writing record %5d\n",i);
    }
    close(fd);
}
```

See also read(), open(), lseek()

This function obtains the equivalent UNIX error number of a DOS error.

```
int _xlate( errnum );
   int errnum;

   errnum    is the DOS error number for which the UNIX equivalent is desired.
```

Description

This is not a standard UNIX function; thus, it is not portable across C implementations. Of course, DOS error codes are not portable, either. The error number may be one obtained from ferror(). The UNIX errors are defined in the "errno" header file. A Unix error may be displayed via perror() or the error message obtained via sys_errlist().

Example

```
main()
{
    int i;
    for (i=0; i<25; i++)
        printf("%02d ",i);
    putchar('\n');
    for (i=0; i<25; i++)
        printf("%02d ",_xlate(i));
}

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
00 04 02 04 04 04 06 04 08 04 02 04 04 04 04 15 16 04 04 16 04 04 04 04 24
```

See also: perror(), ferror(), errno, sys_errlist()

This function is used to convert character strings of hexadecimal digits to their integer or long integer value.

```
int xtoi( string );
   char *string;

long xtol( string );
   char *string;

   string    is a string containing hexadecimal digits <0-9, a-f, A-F>.
```

Description

This C function obtains the machine value of a string of hexadecimal digits in character form (i.e. composed of nothing but characters in the range 0-9, a-f, and A-F). Left truncation of the integer value takes place if an excess number of digits is present (i.e. ival=xtoi("10001"); would result in the integer value of 1 decimal). Conversion stops as soon as the first character not in the valid range is detected.

Return Code

The obvious return code of the xtoi() function is the integer value of the string. The xtol() function returns a long integer.

Example

```
#include stdio.h
char inbuf[81]; int ival; long lval;
extern long xtol();     /* to show that function returns a long */

main()
{   puts("Enter your hexadecimal number: EOF to exit");
    while (TRUE)
    {   if (!gets(inbuf)) break;
        ival=xtoi(inbuf);
        lval=xtol(inbuf);
        printf("Number in decimal (int|long)is: (%d|%ld)\n",ival,lval);
    }
}

Enter your hexadecimal number: EOF to exit
1234   |Number in decimal (int|long)is: (4660|4660)
12345  |Number in decimal (int|long)is: (9029|74565)
fffff  |Number in decimal (int|long)is: (-1|1048575)
abcd   |Number in decimal (int|long)is: (-21555|43981)
8000   |Number in decimal (int|long)is: (-32768|32768)
```

See also: atoi(), atol(), btoi(), btol(), otoi(), otol()

This function will fill a memory block with binary zeroes.

void zero(address, length);
char *address; unsigned length;

address	is the starting address of the memory region to zero.
length	is the number of memory cells to zero.

Description

This function is equivalent to "fill(address, length, 0);".

Example

```
#define SIZE 4096
main()
{
    char *p, *s;
    if (!(p=alloc(SIZE)))   /* allocate non-zeroed memory block */
        abort();
    if (!(s=malloc(SIZE)))  /* allocate zeroed memory block */
        abort();
    if (memcmp(p,s,SIZE))
        puts("Before zero(): memory compares different");
    zero(p,SIZE);
    if (memcmp(p,s,SIZE))
        puts("After zero(): memory compares different");
    else
        puts("After zero(): memory compares same");
}
```

```
Before zero(): memory compares different
After zero(): memory compares same
```

See also: fill(), memset()

Advanced Topics

Runtime options and I/O control

Runtime options provide flexibility in customizing some aspect of the program that you are developing. Note that the option support provided in MC is not something usually found in other C compilers; or where provided, it would typically be implemented in some other manner (such as forcing you to edit some library or invoke some function other than a standard function). What this means is that when you utilize these options, a little lack of portability is introduced into your C source program. You therefore should adequately document your use of these options so that you may recognize their effects when you choose to port your program to some other system.

MC provides a few different facilities for invoking options. The specific facility used depends on the scope of the desired effect. For instance, the "#option" preprocessor directive is usually used to change which library modules are to be included during the link session. The "option()" function is used to alter the behaviour common to the entire I/O package while the "ioctl()" function is used to alter the behaviour of a specific device. In addition, compiler options, such as "+f", alter a particular aspect of the compiler's behaviour.

The following sections include material on various options. Some of the material enhances the information provided elsewhere in this document while other material specifically documents the use of an option facility.

ERRORMSG

The I/O package provided in LIBC has the capability of invoking the DOS I/O error handler when any error is indicated by a DOS service call. This handler uses the "@ERROR" DOS facility. Since the DOS error facility always issues its error message display through the "*DO" device, any display will be irrespective of standard output or standard error output redirection. The default state of "errormsg" is FALSE; thus, any detected error will not initiate a request to the DOS error handler.

There will always be some error returned from a library function when a DOS I/O error has been detected regardless of the state of "errormsg". Using the function's error indication, along with any DOS error available via ferror() or the UNIX error variable, "errno", and perror(), your program can obtain some "controlled" error message display. A controlled display is usually more appropriate for an installed program. The error return codes are documented for each function in Chapter 4, "Library Functions", where applicable.

During program development, you may desire to have the DOS I/O error message displayed automatically upon detection of an I/O error. All you need do is set the "errormsg" flag to TRUE. The state of this flag is altered by coding the option() function as follows:

```
option(O_ERRORMSG,TRUE);
```

within your C source program. The symbolic constant "O_ERRORMSG" is defined in the stdio header file. When an I/O error occurs, the DOS error handler will report an error diagnostic similar to the following:

```
** Error code = 24, Returns to X'363F'
** File not in directory
File = NOSUCHF/ILE:7
Last SVC = 102, Returned to X'1A19'
```

BRIEF

This option works in concert with "errormsg". The DOS provides a facility for specifying a long or a short (brief) error diagnostic display message. If the long message identified above is too lengthy, it can be shortened to:

```
File not in directory
```

by invoking the option() function as follows:

```
option(O_BRIEF,TRUE);
```

The symbolic constant "O_BRIEF" is defined in the stdio header file. Note that neither the long error diagnostic nor the short error diagnostic will be displayed unless "errormsg" has been optioned TRUE.

KBECHO

In certain types of programs, such as screen and graphics editors, the programmer may choose to control the display of characters typed at the keyboard. However, for most programs it is desirable to be able to see what is being typed without overt action on the part of the program, even if standard output has been redirected. The "kbecho" option allows this flexibility. When set TRUE, "kbecho" will cause the getc() function (and likewise the getchar() function) to echo all characters input from the keyboard to the video display device (*DO). This holds true for ANY file opened as "*KI", not just the standard input. Since "kbecho" defaults to FALSE, if you desire its effect, you need to set it TRUE by invoking the option function as follows:

```
option(O_KBECHO,TRUE);
```

The symbolic constant "O_KBECHO" is defined in the stdio header file. The state of "kbecho" (i.e. FALSE to TRUE and TRUE to FALSE) may be freely altered while your program is running.

BREAK

The TRSDOS 6 keyboard driver provides a facility for generating an end of file indication by a specific combination of keystrokes (specifically, <CTRL><SHIFT><@>). Thus, an EOF can be generated from the keyboard just like an EOF is generated when reading a file stream and the end of the file is reached. Unfortunately, LDOS 5.1 does not have such a capability. Therefore, an option has been provided to enable the <BREAK> key to be used as an EOF indicator. This option is device specific; thus, it is enabled or disabled via the ioctl() function. The option is operational on all MC releases. You should read the documentation on the ioctl() function contained in Chapter 4, "Function Libraries". The "break" option appears in numerous examples throughout chapter 4.

WILDCARD

The "wildcard" facility in MC provides an automatic search of a disk directory so as to expand a command line wildcard file specification argument into a vector of all file specifications found in the directory which match the wildcard file specification found on the command line. The wildcard file specification is interpreted as containing three fields: a file name, a file extension, and a drive specification.

A wildcard file specification uses two special characters significant for matching purposes: the asterisk "*" and the question mark "?". The question mark will match any character in that character position. The asterisk is used to match all trailing characters in that field. When the asterisk is used, it should be the last character in a field (the "last" might also be the "first"). The drive specification field does not accept the two "matching" characters but is considered to be an asterisk if the field is omitted.

Since the design of a program must specifically take into account the use of wildcards, the wildcard expander module contained in LIBC is not normally linked to your program. You specify the use of wildcards by adding an "#option" directive to your program such as the following:

```
#option wildcard
```

As is typical with other #option directives, the operand may be upper or lower case. The "wildcard" option directive can also be used to inform the wildcard expander that you want it to ignore checking the first "n" arguments for wildcards. This is accomplished by specifying the directive as:

```
#option wildcard nskip
```

The "nskip" operand is the quantity of arguments that should be skipped. Note that the first argument, argv[0], is always the program name and is never included in the "nskip" count. If no arguments are to be skipped, the "nskip" operand MUST be omitted; do NOT enter it as a ZERO. This argument skipping facility allows you to accept pattern-matching arguments on the command line without worrying about them being interpreted as wildcards. The appearance of the "#option wildcard" directive in either form will automatically alter the conditions established in the MC/ASM file so as to link the wildcard expander module to your program during the link session.

The typical command line would use one or more wildcard specifications to designate a collection of files for an input stream. Any output generated by the program would need to use the standard output stream which may be redirected on the command line or reopened within the program by the freopen() library function. A command line could also include "switches" which are usually prefixed by a plus or minus sign.

The following rules are used by the wildcard expander in determining whether a command line argument is to be interpreted as a wildcard specification:

1) The first command line argument is always interpreted as the invoked program name; thus, it is not treated as a wildcard.

2) The next "nskip" arguments encountered on the command line are passed unchanged to the argument list; they are never treated as wildcards.

3) Any argument enclosed in either single quotes or double quotes is NOT treated as a wildcard, regardless of the presence of any wildcard characters (*,?) within the text of the argument.

4) If the first character of an argument is a question mark, the argument is interpreted as a wildcard.

5) If the first character of an argument is an asterisk and the following character is not alphabetic, the argument is interpreted as a wildcard.

6) If the first character of an argument is an asterisk and the following character is an alphabetic, the argument is not interpreted as a wildcard but as a possible devicespec.

7) If the first character of an argument is an alphabetic, then the entire argument is scanned for wildcard characters (*,?). If any are found, the argument is considered a wildcard, otherwise it is passed unchanged to the argument list.

Non-wildcard arguments are included in the argument list in their position relative to adjacent arguments. Wildcard arguments force a search of the appropriate directory for matches to the wildcard specification. If the drive specification is omitted from the wildcard specification, the directories of all connected disk drives will be scanned for file specifications that match the wildcard specification. Note that a device specification (the typical device specification is an asterisk followed by two alphabetic characters) is still interpreted as a device specification since an alphabetic following an asterisk will not be interpreted as a wildcard file specification. Note that arguments enclosed within single or double quotes will not be expanded; they are not treated as wildcards.

If the command line includes one or more wildcard specifications and no match is found during the directory search(es), the diagnostic message:

```
No files matched wildcard(s)
```

will be displayed and the program will abort. This acts as a safety valve to a program that could not function since no input file specification was provided.

The following test program can be used to illustrate the expansion effects of various command lines that include one or more wildcards.

```
/*
 * WILDTEST/CCC - test "built-in" wildcard arg expander
 */
#include <stdio.h>
#option WILDCARD
int i;
main(argc, argv)
   int argc; char *argv[];
{
   printf("argc: %d, argv: %#04X\n\n", argc, argv);
   for (i = 0; i < argc; ++i, ++argv)
      printf("argv[%4d]: %#04X is '%s'\n", i, *argv, *argv);
}
```

The following program reads a collection of assembler source files and writes to the standard output, all lines that begin with a comment indicator (the semicolon). The wildcard expander makes it easy to specify the collection of "*/ASM" files on a particular drive for use as input.

```
#include stdio.h
#option wildcard
char buffer[81];
main(argc,argv)
   int argc; char *argv[];
{
   while (--argc)  /* while there is still an input spec */
   {
      if (!freopen(*++argv,"r",stdin)) /* attempt to open it */
      {
         fprintf(stderr,"Unable to access %s\n",*argv);
         continue;
      }
      while (gets(buffer)) /* while the file still has input lines */
      {
         if (*buffer == ';')  /* if a comment, then output it */
            if (puts(buffer)==EOF) /* continue if no output error */
               exit(-1);          /* error exit */
      }
   }
}
```

Compiler "float" option

K&R is quite specific concerning the operational characteristics of floating point numbers. In paragraph 6.2, they state, "All floating point arithmetic is carried out in double-precision; whenever a float appears in an expression it is lengthened to double by zero-padding its fraction." What this means is that double precision floating point arithmetic is the norm for a standard C compiler. The type of float is available only to limit the storage of floating point numbers; arithmetic is carried out in double precision. Thus, C dictates an automatic upward conversion of floats to doubles for calculations. Furthermore, in paragraph 10.1 concerning the parameter list of a function, they state, "C converts all float actual parameters to double, so formal parameters declared float have their declaration adjusted to read double."

What these preceding statements mean is that a standard C compiler has no facility for calculating with single precision floating point; double precision is the standard. MC follows this standard; however, since MC is targeted for a machine environment which is necessarily slow in performing double precision floating point calculations, an option has been provided to specify a non-standard single precision floating point mode. Bare in mind, this option is not without headaches. Great care must be exercised in its use. You also must thoroughly understand the behaviour of the compiler when dealing with the arguments of a function.

Since any float argument to a function is always converted up to a double, none of the single precision functions provided in the math library [fabsf(), fatn(), fcos(), fexp(), ffix(), fint(), flog(), fraise(), frnd(), fseed(), fsin(), fsgn(), fsqr(), ftan(), and ftoa()] may be used in the normal compiler mode. You cannot explicitly cast a float argument to a float and override this upward conversion.

The default compiler mode for floats is to treat them as doubles in three places:

(1) When used in a complex expression.
(2) When passed as actual arguments to functions.
(3) When declared as formal arguments in function definitions.

The "+f" option disables all three. It also causes floating point constants to be treated as float, not double, thus causing a (possible) loss of precision. It also allows you access to the much faster single precision floating point functions (with a resultant loss of accuracy). The "+f" compiler option does not prevent you from using any double precision function or performing any complex expression evaluation that mixes floats and doubles. Floats will still be upgraded to double when in an expression with a double (see the Arithmetic conversion specifications noted under "binary operators" in Chapter 2). Explicit casts to float or double may thus be necessary in certain places. For instance, the various print() functions ALWAYS expect a double for a floating point argument. If you have invoked the "+f" compiler option and pass a float as a print() parameter, you must explicitly cast it to a double since the compiler will no longer do this for you. All this boils down to a couple of observations:

(1) Use this option sparingly; only when lower precision is acceptable and/or when faster execution is desired and/or when code/data space is at a premium.
(2) The generated code is obviously non-portable.
(3) Mixing of doubles and floats is not recommended.
(4) The need for casts forces the code to get somewhat tricky in places.

Call: DOS SVC interface

You will usually find the C langauge useful in writing DOS utility programs which extend the capabilities of the DOS just like the DOS provided library commands. Such utility programs may need direct access to DOS service calls. Since these types of programs are DOS specific in nature, there is usually no need to worry about portability aspects; however, it is always wise to provide some standardized method of isolating the non-portable DOS interface. Thus, the programmer has been provided with a function in the installation library that provides a level of standardization. This is the call() function.

The call() function has been provided in the installation library to standardize the invocation of machine language routines and DOS service calls. The use of call() is thoroughly documented in Chapter 4, "Function Libraries". The call() function saves the state of the index registers, IX and IY, during its execution; thus, there is no restriction on its use. Note that the function will either invoke a system SVC or directly CALL a machine address based on the "address" parameter value passed in the function's invocation. That means that even in an SVC interfaced DOS, it is still possible to use call() to execute and return from a memory resident machine language module which is external to the C program's environment.

Separate Compilation of Modules

MC supports separate compilation: functions and modules can be compiled at different times, assembled separately, then linked together to produce one program. This facilitates the creation of relocatable module function libraries, and results in great time savings. Commonly used functions can be compiled and assembled once, then only linked into new programs, without recompiling or assembling. Large programs may be segmented and each segment compiled and assembled separately, then linked as a whole by the linker. With the "extern" and "static" statements, the variables used in a module may be specified as external or local.

When separately compiling modules that reference variables in other modules, there is only one approach that may be taken to supply declarations for the shared variables. That approach, which is the proper method and results in better structure in programs, is to define variables as extern when referenced by all but one of the modules with that one module defining the "public" variables outside of all functions that it contains.

To illustrate the mechanics of separate compilation, consider a large program which has been broken up into four separate C source files. The files are: "progmain", "progio", "progmenu", and "progwork". Each of these files may be compiled with a JCL command line such as:

```
DO MC (N=progname,C)
```

The JCL parameters specify the name of the file to be compiled (N=progname) and the compile option (C). After the JCL has been invoked for each of the C source files, this results in four assembly output files. Following the compilation phase, the three subordinate files ("progio", "progmenu", and "progwork") can each be assembled to a separate relocatable module with a command such as:

```
MRAS progname -NL
M80 progname=progname
```

which results in the generation of a /REL file with the same filename as "progname". The "progmain" file which includes the main() function would be assembled with a command line such as:

```
MRAS MC +I=PROGMAIN +O=PROGMAIN -NL
M80 prognameprogname
```

The difference in this MRAS invocation is that the MC/ASM file is the primary file to be assembled with "progmain" as an "INCLUDE" file. The "+O=PROGMAIN" is used to override the default output file name with a specific designation of "PROGMAIN". The "progmain" file could also be compiled and assembled in one command step via the JCL command string previously shown by adding the JCL parameter, "A", as in:

```
DO MC (N=progname,C,A)
```

The subordinate files cannot be compiled and assembled in one JCL command step with the MC/JCL file supplied; however, it could be done by altering the MRAS invocation command line in the JCL file to appear like the first illustration.

Once all of the four files have been compiled and assembled, you generate the executable command program with the linker. To continue with this illustration, an appropriate link command would be:

```
MLINK PROGMAIN,PROGIO,PROGMENU,PROGWORK-N=PROG:d-E
L80 PROG:d-N,PROGMAIN,PROGIO,PROGMENU,PROGWORK,-E:M80BGN
```

where the ":d" would be replaced with the output drive specification. The resulting executable command file would be named "PROGMAIN/CMD".

Obviously, the rationale for using this separate compilation approach would be to limit your development efforts to more managable "chunks" of code. You would rarely be working on all of the files simultaneously but would most likely complete the development of a subordinate module (compile and assemble steps) prior to continuing with the remaining modules. This provides a scheme of partitioning your program into a sequence of small development efforts.

An alternative method of partitioning a large program into small "chunks" would be to isolate every function into a separate file. Compile and assemble each function individually. Write small test programs to exercise each function separately, then build a library, say PROGLIB", with the relocatable function modules. Your "progmain" could then just declare an "#option PROGLIB" to search the library during the link process. See the section on "Building and Maintaining Relocatable Libraries" in this chapter.

Using extern and static

When writing a large program, it is best to try to logically structure your program into modules containing related functions with the data structures they use within the same module. Any functions or non-automatic data structures in a module that need not be accessed by any external function can be declared as "static". These static functions and variables will be unique in name when compiled and assembled, and will not be accessible to other modules; thus, there will be no conflicts in naming.

Those functions and non-automatic data structures declared in the module which need to be accessed by functions in other modules should be declared without any storage class. This causes these functions and data structures to become "public", meaning that they are defined in this module, and can be accessed from other modules.

When a module uses a function or data structure declared in another module, the "extern" statement is used to declare the type of the object. "extern" is required for accessing variables outside the module. However, if a function is used without an "extern" declaration, the compiler will assume that the function returns an integer value. If any other result is returned, the function must be declared "extern" with the appropriate type.

A few examples may help you appreciate the distinctions being addressed here. Spend a little time studying the C source code declarations in the following module until you understand the accessibility of each variable.

```
int var1;           /* public to all functions of all modules */

static int var2;    /* accessible to functions which follow in
                       this module: main(), func1(), func2() */

main()
{   static int var3;/* local to main only */

    int var4;       /* local to main only */
}

int var5;           /* public to all functions of all modules */

static int var6;    /* accessible to functions which follow in
                       this module: func1(), func2() */


func1()
```

```
        {   int var7;       /* local to func1 only */

            static int var8;/* local to func1 only */
        }

        static int var9;    /* accessible to functions which follow in
                               this module: func2() */

        func2()
        {   ;   }
```

The variables "var1" and "var5" become public to all modules because these variables are defined outside of all functions of this module and are also not defined with a storage class. Thus, some other module may access these two variables if that other module includes the statement,

```
   extern int var1, var5;
```

Now even though "var2", "var6", and "var9" are variables defined outside of all functions in this module, since they have the storage class "static", they become accessible only to the functions within this module. In fact, note that a static variable defined outside of a function becomes accessible only to the functions that follow the declaration. Thus, "var2" is accessible to main(), func1(), and func2(); but "var6" is accessible only to func1() and func2() since it is declared following main().

Static variables declared inside of a function are local only to that function. The primary difference between a static and an auto is that a static has a specific memory storage assignment whereas an auto's storage is strictly on the stack. Statics are generally accessed faster than autos; thus, if speed is important, consider the use of statics. The contents of such static variables also persist across repeated calls to the same function.

Building and Maintaining Relocatable Libraries

We encourage C users to create libraries of commonly used functions. This increases your productivity, since functions need not be rewritten for each program. A library should contain self-contained functions; i.e, they do not require the calling function to know about the library module's internal structure, and do not assume anything about data structures that the calling function declares. In structured programming lingo, library functions should be data-coupled and functionally cohesive. Also, functions should be tested and be well debugged before being placed in a function library.

The first point to make concerning relocatable libraries is that the filename field of a library file specification cannot exceed seven characters. The second point is if you name your library "USERLIB", you need take no further action after you have built the library.

If you are using the MRAS assembler package, building a library is very easy since you have the MLIB relocatable module librarian. MLIB is used to build and maintain relocatable libraries. If you have no library of your own already built, you start just by using the "load library" MLIB command and load a compiled and assembled function (a module). You can continue to add additional compiled and assembled function modules via the "add module" MLIB command. The library is "built" in memory as the modules are added; then you write it to disk via the "save library" MLIB command.

If you are using the M80 assembler package, you may not have access to a convenient librarian such as MLIB. Thus, you must build your library sequentially by appending modules together using the APPEND library command provided with your DOS. Any number of functions can be added in this fashion; however, since you have no facility to extract or rearrange the library modules, remember to save a copy of each module individually on an archive disk. When you use the APPEND command to add each module, use the STRIP parameter of APPEND to remove the end of file byte (X'9E') from the library file when you are appending a module to the library. For instance, if you want to build a NEW library composed of three modules named "eenie", "meenie", and "minie", the sequence of commands would be like the following:

```
COPY EENIE/REL USERLIB/REL
APPEND MEENIE/REL USERLIB/REL (STRIP)
APPEND MINIE/REL USERLIB/REL (STRIP)
```

The first command is used to copy the first module into a new file called "userlib/rel". The subsequent APPEND commands will append the second and third modules to the "library" while the use of the "strip" parameter tells APPEND to backspace over the last byte of the library file (the X'9E) before appending the new module.

A library is normally searched by the linker sequentially from beginning to end unless the library is constructed as an "indexed relocatable" library (IRL) by the MLIB librarian. Therefore, the order in which functions are placed in the library becomes important. If a function of a module in the library is called by another function in another module within the library, then the calling function's module must appear first. This is because the external reference for the called function will not be sensed until the calling function is linked. So the general rule is: Calling functions first, called functions last.

Note that in the above discussion, it was recommended that you name your library "USERLIB/REL". That's because there are two files provided with the MC package which need to be adjusted depending on the name and quantity of user provided libraries. The first file which needs modification is the MCMACS/??? file (where ??? could be ASM or MAC depending on the assembler you are using). For each user provided library you want to have searched, the definition,

```
@_LIBRARYNAME DEFL 0
```

must be added to the list of "established defaults". In the following excerpt of the provided file,

```
; establish defaults for MC options
@_MATHLIB DEFL  0
@_INLIB DEFL 0
@_ARGS DEFL  -1
@_REDIRECT DEFL -1
@_FIXBUFS DEFL  0
@_MAXFILES DEFL 13
@_USERLIB DEFL  0
```

note that the declaration for a "userlib" library has already been inserted (the seventh line). There is no requirement for any specific position of the declaration in this file. It can be placed anywhere.

The other file which needs to be altered is the MC/??? file. This file must include the statements:

```
IF    @_LIBRARYNAME
$REQ  LIBRARYNAME
ENDIF
```

Contrary to the insertion into the MCMACS/??? file, the positioning of these statements is very important in the MC/??? file. The following excerpt of the provided MC/??? file shows that the statements needed for a library named "userlib" have already been included.

```
LD      HL,0        ;Set return code
PUSH    HL
CALL    EXIT        ;Back to DOS
IF      @_USERLIB   ;TRUE if #option USERLIB
$REQ    USERLIB
ENDIF
IF      @_MATHLIB   ;TRUE if #option MATHLIB
$REQ    MATH
ENDIF
IF      @_INLIB     ;TRUE if #option INLIB
$REQ    IN
ENDIF
$REQ    LIBC        ;Standard lib always!
$REQ    LIBA        ;Run-time lib always!
```

The "$REQ" assembler macro statement passes a request to the linker to search the designated library. Note that the "userlib" request precedes those for MATH, IN, LIBC, and LIBA. This means that your "userlib" is searched before any of the MC provided libraries. It was stated above that calling functions must precede called functions in a library. Likewise, it is also mandatory that a library that has functions that call functions in another library must be searched prior to the library that contains the called functions. This means that the functions in your library have access to all of the functions in MATH, IN, LIBC, and LIBA but that no function in those libraries can "call" any of your user provided functions.

The order of library requests is significant when you want to provide a second user library, say MYLIB. If the request statements for MYLIB are inserted into MC/??? prior to the statements for USERLIB, then no function in "userlib" can call a function in "mylib" but functions in "mylib" can call functions in "userlib". These restrictions could be overcome by adding multiple library search requests during a manual invocation of the linker; MLINK or L80.

Programs with overlays

MC provides a support function and interface control for the development of a complex program environment involving a root program and one or more overlay programs. The generation of the root and overlay files requires the use of the MLINK linker provided with the MRAS assembler development system. Depending on the total size of the relocatable modules making up the root and the overlays, you may need to invoke the virtual memory facility of the linker. This is discussed in the linker documentation. The design of a C program environment that uses overlays is a complex procedure. It will certainly be difficult to provide an in-depth analysis of all details concerning the design of such a complex program environment. This section will cover only the mechanics for the procedure, not the underlying rationale behind the design of such a complex program.

In general, a root module contains all of the functions and external variables that must be accessible to the root and all overlay modules. The root module contains the function, main(), and also must contain the overlay handler provided in the IN/REL installation library, execovl(). Each overlay module must contain a similar "main" function called ovmain(). Your job is to decide how your program environment is to partition itself into root and overlay modules. In order to accomplish this, you need to understand exactly what overlay support is provided.

To begin with, there is absolutely NO facility provided for the automatic loading of an overlay when a function is called. Your program MUST know it wants to access a given overlay. An overlay is loaded and executed only as the direct invocation of the C overlay handler, execovl(). Next, the root module does not have knowledge of any variable that is a part of any overlay module. Neither does any overlay module have any knowledge of any variable that is a part of another overlay module. On the other hand, every overlay module has total knowledge of "public" variables declared in the root module provided that they are declared "extern" in the overlay. Remember from the discussion of external variables, in the section on separate compilation, that "public" variables are variables declared outside of all functions of a module and declared without a storage class. The functions of an overlay must still declare such variables "extern" to be able to access them.

The execovl() function will call an overlay from the root program. This function is used to invoke an overlay of the executing C program and optionally pass an argument list to the called overlay. Duplicating somewhat the information on execovl() from Chapter 4, it is defined as follows:

int execovl(ovnum, ovargc, ovargv);
char ovnum; int ovargc; char *ovargv[];

ovnum	-	the requested overlay number in ASCII.
ovargc	-	the quantity of args to be passed to the overlay in the ovargv array.
ovargv	-	a pointer to the argument vector which is to be passed to the overlay.

The argument vector "ovargv" MUST have ovargc pointers in it. The "ovargc" and "ovargv" will be passed as arguments to the overlay's ovmain() function in that order. Note that "ovargc" and "ovargv" are optional.

NULL will be returned if the requested overlay loaded and executed, otherwise EOF will be returned and errno will be appropriately set. The options O_BRIEF and O_ERRORMSG will be in effect, if set, when such an error occurs. Note that any return code from ovmain() is ignored; you can use public memory location(s) in the root for communications from the overlay.

Your root program can be preprocessed, compiled, and assembled as usual with two exceptions. First, all library routines (i.e. functions provided in MATH, IN, or LIBC as well as runtime routines in LIBA) which are to be common across the overlays MUST be declared "extern" in this root program regardless of whether or not the root makes use of them. The library routines needed by a program may be ascertained by manually linking (i.e. not via Job Control Language) the program with the libraries, specifying the library searches explicitly, then obtaining a symbol table listing which will include all public symbols. Second, the C source module must include the directive,

```
#option root
```

This statement passes information to the MC/ASM file during the assembly phase of your root program. The file, OVTEST/CCC, which follows, contains a fully functional sample root program.

```
/* OVTEST/CCC - overlay test root */
#option INLIB       /* used to obtain execovl() */
#option root
extern int execovl(), printf();
int ovargc = 3; char *ovargv[3] = { "ovtest", "this is overlay" };
main()
{
   ovargv[2] = "1";
   if (execovl('1', ovargc, ovargv))
      error();
   ovargv[2] = "2";
   if (execovl('2', ovargc, ovargv))
      error();
}
error()
{
   perror("ovtest");
   exit(1);
}
```

The overlay programs MUST each have a public function called ovmain(), which will be the entry point to each overlay. They each can be preprocessed, compiled, and assembled as usual with one exception. The directive,

#option overlay

must be included in the C source module of each overlay. The following file, OVERLAYT/CCC, contains a sample overlay program.

```
/* OVERLAYT/CCC - overlay for OVTEST */
#option overlay
ovmain(argc, argv) int argc; char *argv[];
{
   printf("%s: %s %s\n", argv[0], argv[1], argv[2]);
}
```

Obviously, with an understanding of the preceding discussion, it becomes obvious that you must compile and assemble the program and its overlay modules separately. The procedure produces a relocatable module for the root and one for each overlay. The link session will then utilize the overlay generating capabilities of the MLINK linker to produce the proper executable root command and overlay files. Thus, the linker will need to be operated manually (without the benefit of the provided MC/JCL Job Control Language file). The link session used to generate a root program from ovtest and two overlay programs using overlayt (using the same module twice just for the purpose of illustration), is as follows:

```
    . Link OVTEST etc. to OVTEST/CMD,OV1,OV2
    mlink -l=2600 -a=y
    ovtest
    -o overlayt
    -o overlayt
    -n=ovtest -e
    //exit
```

In general, the link session must be in something like the following order:

```
    MLINK
    ? -v=:4 ..................> optional; preferably a RAM drive!
    ? root_program_rel_file(s)
    ? -S=necessary_library_1
    ? -S=necessary_library_2
    ...
    ? -S=necessary_library_n
    ? -O overlay_1_rel_file(s)
    ? -O overlay_2_rel_file(s)
    ...
    ? -O overlay_n_rel_file(s)
    ? -N=program_name
    ? -E
```

When Things Go Wrong

C is a language that offers great flexibility, but not without a price. The price of C's freedom is the programmer's ability to make catastrophic errors with ease. The programmer is not protected from himself when using C. Your best protection is to carefully check your programs when you write them, for any evident errors before you try to run them. Of course, any time you test a program you should not have any disks in your drives that you would care about if they were suddenly erased. This is not to say that you should limit experimentation; quite the contrary. However, always be prepared for the worst.

With MC, you have an advantage over some other compilers. MC generates an assembly language source file. You can debug the program without trying to second-guess the compiler, or having to disassemble the compiled output. The modularity of the program also helps, since there are clear interfaces (functions) to breakpoint at. It would be helpful, though not essential, for the programmer to have familiarity with the Z-80 instruction set and with the debug facility of DOS.

Debugging Techniques

One of the most direct methods of debugging a C source program which compiles, assembles, and links without error, yet produces incorrect or unexpected results, is to liberally sprinkle the C source code with printf() statements. These "debugging" statements can usually be contained within a preprocessor "#if DBUG ... #endif" directive. By using this method, you can conveniently eliminate the debugging printf() statements after you have corrected the problem by undefining the DBUG symbol. This saves you a lot of editing. In fact, it is highly useful to design such debugging printf() statements into your program to begin with, rather than wait until you need to debug unexpected results.

If you feel the need to step through the program's execution with the DEBUG utility provided with the DOS, it is recommended that you obtain the appropriate DD&T package to complement DEBUG. This package adds an online disassembler to DEBUG's display. With a good debugger at your disposal, a copy of the symbol table produced by the linker, and the assembly listings appropriate to the modules you wish to debug, you should be ready to investigate the behaviour of your program. Since machine code debugging is a lengthy and brutal experience, it is strongly recommended that you first go back to your C source listings and triple check your coding and algorithms. Also verify the proper checking for error codes returned by functions. Your C source listing is the best place to start your efforts.

Compilation Errors

MC generates an error message whenever it finds something in the input file that cannot be recognized, or that doesn't fit the syntax of the C language. When MC outputs an error message, it should print the line in error and point to the particular character where the error was recognized. It also displays the line number of the file currently being processed as well as the line number of the function currently being processed. This helps you locate what statement in your source stream is being flagged as containing an error. The actual programming error may be earlier in the program, depending on the type of error.

Some errors may not be detected until many lines later. For example, if a closing brace is missing in the input file, MC will not be able to detect the error until the next function declaration, which will then be flagged as a function call without an ending semicolon. This is because MC thinks the previous function has not been completed. Similarly, if an opening brace is missing, MC may not find out until the last closing brace is encountered, with no match.

In the appendix of this manual is a list of the error messages which MC can generate, and some likely causes for each. Most errors are usually typographical, but the user should be well versed in the C language and the MC implementation. Learn where to find information regarding syntax and capabilities of the language. The language definition chapter of this manual and the appendix of the K&R text are good places to look when you are not sure of syntax.

Assembly errors

There are few assembly errors which can occur if you assemble your programs using MC/??? with your compiled C modules. Your safest bet is to use MC/JCL when compiling, assembling, and linking. If you have written portions of your program in assembly language, you may have a few more errors to deal with. Please read the section on assembly language interfacing for hints in debugging your assembly language.

If you are creating a CMD file directly from the MC/JCL file, you will be aware of an assembly error if the JCL aborts from MRAS. To specifically isolate the assembly error, you will need to invoke MRAS without the "-NL" switch. The most direct way to accomplish this is to list the SYSTEM/JCL file (the one that is generated from the MC/JCL invocation) and examine the line that invoked MRAS. It may look something like this:

```
MRAS MC +I=YOURPROG/ASM +O=YOURPROG/CMD -NL
```

At the DOS Ready prompt, all you need do is input the line modified as follows:

```
MRAS MC +I=YOURPROG/ASM -WE -NC
```

When the assembler is invoked in this manner, it will pause when an error is detected. One of the following two errors should usually prevail.

UNDEFINED SYMBOL - A symbol that you referenced in your program was undefined in any module in your program. This can be caused by omitting the definition, misspelling the identifier, or defining them incorrectly. A look at the name that is undefined will give you a clue as to which situation was the cause. Misspelling should be obvious. Look out for upper-case versus lower-case names. Remember that C is case sensitive.

MULTIPLE DEFINITION - A symbol was defined twice with the same name. If you are assembling separately compiled functions, you may have named two external variables or functions the same. To correct this, make one or both of the identifiers static within the module. If separate compilation is not being used, you have defined two external objects in your program with the same name.

It is useful to familiarize yourself with the section on MC assembly language output structure. Also, if you are assembling separately compiled functions, a good understanding of the "extern" statement and external versus static variables is essential. Refer to the section on storage classes in Chapter 2, MC Language Definition.

Linker errors

The linker errors which you may experience may be limited to the following three errors. A "Symbol table overflow" error could be experienced when you are linking a large C program. If you are using MLINK, you should attempt to relink your program using the virtual memory switch. You also could experience a "Multiply defined symbol" error. This means that you have either defined the same public symbol in two different modules or that you have defined as public, a symbol already defined in one of the provided library functions. This is an easy problem to correct in your program. The third error would be "Undefined symbol". This would arise from a symbol declared as "extern" in one of your C source files but no module defined the symbol. It may be a spelling error or a programming oversight.

Runtime Error trapping

As stated in the section on Runtime Error Control in Chapter 1, MC provides certain facilities for the detection and control of four types of runtime errors: DOS I/O errors, C environment errors, low-level floating point errors, and high level floating point mathematical errors. The facilities at your disposal to detect and trap the first two types should be obvious to you after even a brief perusal of Chapter 4, Library Functions. That's because DOS I/O errors and C environment errors are indicated by an error return code passed back from a function call.

The following methods are available to your program for displaying I/O errors:

1) Enable the ERRORMSG option via "option(O_ERRORMSG,TRUE);" as discussed earlier under Runtime Options. This provides automatic display of an error diagnostic message when a DOS I/O error occurs.

2) Use the ferror() function to obtain the DOS error code from the stream and act on that code accordingly. You can introduce your own messages as appropriate. With ERRORMSG set to FALSE, your program controls the error diagnostic display.

3) Use the perror() function to obtain the UNIX error message associated with the error. Remember that every DOS I/O error code is automatically translated to a corresponding UNIX error number, which is stored in "errno". If you desire portability, you should restrict your error handling to the use of "errno" and perror().

A discussion covering specific methods of dealing with these errors is beyond the scope of this document.

The C environment errors will only be reported via an error return code from a function call with "errno" set to the appropriate UNIX error number. Thus, these types of errors can be displayed by method (3) shown above for DOS I/O errors. You don't have to use perror(). Your program can still invoke its own messages in its own manner based on the error code contained in "errno".

Since the detection and control of low-level floating point errors is an implementation specific entity, this topic will be discussed in a little more detail. All of the low level math routines that operate on floats or doubles, (i.e. those that perform addition, subtraction, multiplication, division, etc.) exit through a common routine, which sets "errno" to an appropriate error number if the routine's return code is non-zero. The only math routines which provide a non-zero return code are the single and double precision floating point routines. When errno is set, control is passed to a floating point vector, called "_fltvec". This normally vectors to an assembler RET op code. Thus, when such a low-level floating point error is detected, nothing unusual happens other than the setting of "errno".

If your C program wants to take control when such an error is detected, your program only needs to replace the default "_fltvec" vector with the address of your handler function. The following header file illustrates a low-level floating point handler called "floaterr()" which can display an error message appropriate to the detected error. It includes a function which is called from main() named "initflterr()". This function replaces the "_fltvec" vector with the address of the "floaterr()" handler. Examine closely the behaviour of the various parts of the header file.

```
/*
 * FLOATERR/H - general floating point error routine - 01/07/86
 */
#include <math.h>               /* needed for definitions */
void floaterr(), initflterr();
extern int _fltvec;             /* kludge to declare _fltvec */
static char codes[] =           /* error codes returned by math routines */
{   0,
    UNDERFLOW,
    DOMAIN,
    OVERFLOW,
    DIVBY0
};

static char *msgs[] =          /* appropriate error message strings */
{   "No error",
    "Underflow",
    "Illegal function call",
    "Overflow",
    "Division by zero",
    "Unknown error"
};

/*
 * Initialize floating point error routine - call from main()
 */
initflterr()
{   _fltvec = floaterr;        /* set error vector */
}

/*
 * Routine to trap floating point error
 */
floaterr(retaddr)
    unsigned retaddr;
{   unsigned *sp;
    int i;
    sp = &retaddr;                  /* get stack pointer */
    for (i=0; i< sizeof codes; ++i) /* search table, if not found i = 5 */
        if (codes[i] == errno)
            break;
    fprintf(stderr, "\nFP error %2d (%s), Trace: %04x, %04x, %04x\n",
           errno, msgs[i], *(sp - 1), *sp, *(sp + 1));
}
```

There is no way to recover the name of the routine that returned the error; you can only detect the error number. It is beyond the scope of this document to provide any additional material for such purposes. The following example illustrates a program that uses this header file. The diagnostic error messages displayed upon running the example are also noted.

```
#include <stdio.h>
   #include <floaterr.h>
   main()
   {
       initflterr();                   /* set error vector */
       1. / 0.;                        /* division by zero */
       1.701411e+38 * 1.701411e+38;    /* overflow */
       1.701411e-38 * 1.701411e-38;    /* underflow */
   }

   FP error  7 (Division by zero), Trace: 5455, 31d8, 0000

   FP error  3 (Overflow), Trace: 5455, 31ea, 4986

   FP error  4 (Underflow), Trace: 5455, 31fc, 4c82
```

MC also supports the high-level mathematical function error exception reporting as documented for UNIX System V. In this facility, when errors are detected by certain high-level functions, they will invoke the matherr() function in a structured manner. The supplied matherr() function is a NULL function; it does nothing. The high-level error handler, however, will print an appropriate diagnostic message to the standard error file when an error occurs. If you supply your own matherr() function, you can disable the system's message display as well as provide your own error handling.

When matherr() is called, a pointer to the exception structure as defined in the math header file is passed as an argument. This structure includes the following members:

```
        int type;      - type of error
        char *name;    - pointer to name of function where error ocurred
        double arg1,   - first argument with which *name was invoked
               arg2,   - second argument with which *name was invoked
               retval; - default value returned by the function
```

The member named "type" contains the error number that describes the error. The following list of symbolic constants, defined in the math header file along with the system's standard error messages, shows the possible error numbers.

DOMAIN	- argument domain error
SING	- argument singularity
OVERFLOW	- overflow range error
UNDERFLOW	- underflow range error
TLOSS	- total loss of significance
PLOSS	- partial loss of significance
DIVBY0	- divide by zero

The member "name" points to the actual name of the function that detected the error. The following list of functions report errors through matherr(): exp(), log(), log10(), pow(), sqrt(), asin(), acos(), atan(). This list may not be all inclusive.

The members "arg1" and "arg2" contain the first two values of the arguments of the function which was invoked. Obviously, if the invoked function has only one argument, then "arg2" is irrelevant.

The "retval" member will contain the value that the invoked function will return to the calling function. If you want to change that value, all you need do is assign a new value to the structure member.

The matherr() function has the following invocation syntax:

```
#include <math.h>
int matherr( x );
struct exception *x;

x           - is a pointer to the exception structure
              defined in the math header file.
```

The return code from matherr() is used by the system. If matherr() returns a non-zero value, the normal error message printed to standard error by the math error handler will be suppressed and the global error variable, "errno", will not be altered. On the other hand, if your matherr() function returns a zero value, the system's error message will be output normally and "errno" will be set to the appropriate error number.

Some of the example programs provided as illustrations of functions in Chapter 4, Function Libraries, depict various implementations of a matherr() function provided by the program. The examples for exp(), log(), and log10() provide no matherr() function so the system supplied error messages, "Overflow range error" and "argument domain error" are displayed. Make note of the matherr() functions provided in acos(), asin(), and sqrt(). An example of a more involved and complex matherr() function is illustrated here. This version traps domain errors which would occur from passing a negative argument to sqrt(). The example "fixes up" the error by assuming that the argument to the square root function should have been positive. Pay attention to the method of altering the value to be returned by the high level math function by assigning a new value to "x->retval".

```
matherr(x)
    struct exception *x;
{
    fprintf(stderr,"Type %d function error in %s...\n",x->type,x->name);
    fprintf(stderr,"     Args = %g,%g: ",x->arg1,x->arg2);
    if ((x->type==DOMAIN) && (!strcmp(x->name,"sqrt")))
        x->retval= sqrt(-x->arg1);
    return TRUE;
}
```

Assembly Language Interfacing

Since we encourage the programmer to use 100% C source code in the generation of a program, no attempt will be made to document any assembly language interface to library modules. The barest minimum is provided to assist the experienced assembly language programmer who desires to include some assembly language control into his C programming. The programmer who wishes to learn more about the interface between the C language and the provided libraries would be best advised to examine the compiler's assembly language output from compilations of various C source statements. The caution to be observed is that this output may not be consistant across future releases of the compiler.

While it is possible to insert assembly language source code directly into your C program using the "#asm-#endasm" construct, it is much cleaner to interface by placing your assembly language code into a separate module. This keeps all the non-portable code separate from the portable C code. It is best to call assembly language as a function, rather than including it directly into a C function by mixing C and assembly source code in line.

Program Memory Map

MC programs, once assembled and linked, have the following utilization of available memory:

$DBEG [or $CBEG]	- Established as the program's LINK ORIGIN during the link process. This will be the lowest machine address used by the program.
$START	- Program execution begins here. This is an ENTRY within the MC/ASM file.
	- MC generated modules.
	- In-line assembly language modules.
	- Static variables and strings intermixed within modules.
	- Library modules from USERLIB/REL, IN/REL, MATH/REL, LIBC/REL, and LIBA/REL.
($MEMRY)	- Highest machine address used by the program and the static variable storage.
($FREEP)	- Contains first machine address available for dynamic memory allocation.
	- Memory allocated by sbrk(); in use by the program, or maintained by alloc(), malloc(), calloc(), and free().
($LOMEM)	- Unused memory available from alloc(), calloc(), malloc(), or sbrk().
(SP-1024)	- Memory reserved from dynamic allocation for use as program stack space. Local variables are stored here. MC always reserves space for the program stack when requests for dynamic allocation of memory are made.
(HIGH$)	- Original CPU Stack Pointer.

MC Identifier Output

The following table outlines the format of label generation from MC:

Identifier Class	MC Output
Local labels	$?#
External identifiers	NAME
-longer than 3 characters	NAME
-3 characters or less	NAM$
Static identifiers	
-external to functions	Same as externals
-internal to functions	NAME@?+
Goto labels	NAME$?*

NAME, NAM	MC identifier, 1 to n characters, upper case [external identifiers should be limited to a length of seven significant characters].
#	The local label number
*	The function number (within the module)
+	The compound block number (within the module)

MC generates labels in this fashion so that static variables, external variables, and labels will not conflict with each other. Thus, there can be an external variable named x, a static named x in two different modules, a static named x in two different functions in the same module, and a (goto) label named x, all within the same program, with no conflicts. A dollar sign, '$', is appended to external or static labels which are three characters long or less. This prevents possible conflicts with register names and logical operators in your assembler.

Local labels are used to implement strings, double precision constants, conditional statements and operators, and loops. They are assigned numbers starting with 1 and incremented by 1.

Runtime variable storage format

Characters occupy eight bits and are stored in single 8-bit bytes; they are considered unsigned.

Integers and short integers occupy sixteen bits and are stored in two 8-bit bytes. The leftmost bit (which is the sign bit of signed integers) is the most significant; the rightmost bit the least significant. Integers are stored in memory with the low-order byte occupying the lower machine address.

Long integers occupy thirty-two bits and are stored in four 8-bit bytes. The leftmost bit, bit 31 (which is the sign bit of signed integers), is the most significant; the rightmost bit the least significant. Long integers are stored in memory with the lowest-order byte occupying the lower machine address, the next higher-order byte occupying the next higher machine address, and so forth.

Single precision floating point numbers occupy thirty-two bits and are stored in four 8-bit bytes, a three-byte fractional part in bit positions 0-23 and a one-byte exponent in bit positions 24-31. The leftmost fraction bit (which is the sign bit of the number) is the most significant; the rightmost bit the least significant. Floats are stored in memory with the lowest-order fraction byte occupying the lower machine address, the middle-order fraction byte occupying the next higher machine address, the highest-order fraction byte occupying the next higher machine address, and the exponent occupying the highest address.

Double precision floating point numbers occupy sixty-four bits and are stored in eight 8-bit bytes, a seven-byte fractional part in bit positions 0-55 and a one-byte exponent in bit positions 56-63. The leftmost fraction bit (which is the sign bit of the number) is the most significant; the rightmost bit the least significant. Double precision floating point/ numbers are stored in memory with the lowest-order fraction byte occupying the lower machine address, the next higher fraction byte occupying the next higher machine address, etc., and the exponent occupying the highest address.

The exponent of a floating point number is stored in "excess 128" notation. Using the syntax "**" to indicate "raise to the power", this means that an exponent of zero (2**0 = 1) is represented by 128D (80H); positive exponents are denoted by values of greater than 128, and negative exponents by values of less than 128. Thus, by subtracting 128 from the value, the true exponent is obtained.

The fractional part of a floating point number is always in BINARY normalized form, which means that the values lie within the range:

```
   2**-1 <= X < 2**0
```

A little simple arithmetic will show that this range, base 10, is (.5<=X<1). All this means is that there will always be a one (1) immediately to the right of the binary point when the binary normalized mantissa is shown in its binary form. Since there will always be a one bit in this high-order bit of the normalized fraction, it is not necessary that it be kept. In fact, the bit is dropped entirely and the position in the normalized fraction is used to store the SIGN of the entire number; a one indicating a negative number and a zero indicating a positive number.

Register Utilization

All registers are available for use by an assembly language function with the exception of the index registers, IX and IY. If you want to use the index registers, they must be saved on entry to your function and restored on exit. The stack pointer must be returned in the same condition as it was upon entry.

For math operations, the following CPU registers are used, in general, to contain the various variable types:

Type	primary	secondary	notes
char	L*	E*	* high-order byte zeroed
int	HL	DE	
long	BCDE	stack	
float	BCDE	stack	
double	(HL)*	stack	* 8 bytes moved to memory storage

Argument passing

C passes arguments on the machine stack. Each argument is pushed onto the stack according to its memory storage format [characters take two bytes of stack space of which the higher order byte is ignored]. Arguments are pushed in an order opposite of the order they are specified in the function call. Here is the assembly language which C generates to perform a function call (it is assumed that "x", "a", "b", and "c" are each static integers and the function returns an integer; we will ignore the use of macros):

```
        ;       x=func(a,b,c);
                LD      HL,(C$)
                PUSH    HL
                LD      HL,(B$)
                PUSH    HL
                LD      HL,(A$)
                PUSH    HL
                CALL    FUNC
                POP     AF
                POP     AF
                POP     AF
                LD      (X$),HL
```

This process generates the following appearance on the Z-80 machine stack:

```
        (SP+6)  ==>     <c>
        (SP+4)  ==>     <b>
        (SP+2)  ==>     <a>
        (SP+0)  ==>     return address
```

That is how the arguments appear to the called function when first entered. Several methods can be used within the called function to obtain the arguments. The simplest method is to POP each argument off the stack. This is not suitable for large numbers of arguments, but most efficient for 3 operands or less. Using the example above, the arguments could be retrieved as follows:

```
        FUNC    POP     AF      ;return address saved
                POP     BC      ;argument <a> in BC
                POP     DE      ;argument <b> in DE
                POP     HL      ;argument <c> in HL
                PUSH    HL      ;restore argument <c>
                PUSH    DE      ;  "        "     <b>
                PUSH    BC      ;  "        "     <a>
                PUSH    AF      ;stack is same as at entry
```

Note that the stack is returned to its original condition. It is always important to keep track of the stack pointer. However, the contents of the stack, i.e., the arguments, are "owned" by the called function and can be used like any local variable. A better method to use when dealing with large numbers of arguments is shown below:

```
                LD      HL,2    ;offset to <a>
                ADD     HL,SP   ;HL = address of <a>
                CALL    @GINT   ;get contents of <a>
```

@GINT is a run-time library function which gets the integer pointed to by HL into HL.

Another method is to utilize the Z-80 index registers. The stack pointer must be placed into the index register first, then index offset values can be used to get and store the arguments:

```
                PUSH    IX              ;Must save register IX
                LD      IX,0
                ADD     IX,SP           ;Get SP into IX
                LD      L,(IX+2+2)      ;Get LSbyte of a
                LD      H,(IX+3+2)      ;Get MSbyte of a
                POP     IX              ;Restore IX
```

If an argument is intended to be a character variable, only the least significant byte (LSbyte) is needed, so a single indexed load is used.

Returning a value from a function

MC programs use the primary register associated with the function type in order to return the function's value (see "register utilization" above). For instance, a short int should use the HL register pair for a 16-bit accumulator. Any short int value to be returned by a called function must be placed in HL before returning to the calling function. Take care that a full 16-bit value is returned. If a character or 8-bit value is being returned, then H should be loaded with zero. If a TRUE or FALSE indication is to be returned, HL should be set to 1 or 0, accordingly.

Appendix

Error Messages

Preprocessor errors - Warning

The following messages may occur during the preprocessor phase of compilation and are only warnings; processing continues. Preprocessor warnings are unnumbered messages.

"Conditionals nested too deeply"

> The maximum conditional nest depth is 256. How you reached that limit is beyond imagination.

"#else without #if"

> This obviously occurs when a "#else" is detected without any previously corresponding "#if".

"#endif without #if"

> This is emitted when a "#endif" is detected without a previously corresponding "#if".

"Extra macro parameters, ignored"

> A parameterized macro was invoked with more arguments than when the macro was defined. Double check your argument list.

"Illegal preprocessor directive"

> The input stream provided a "#something" when "something" was not one of the legal directives.

"Multiple #else's"

> Only one #else per #if is allowed.

"Redefining <macro_identifier>"

> This informs you that a previously defined macro is being redefined. It is not an error.

"Undefining nonexistent macro"

> This informs you that a macro is being undefined which had not previously been defined. It is not an error.

"Unterminated conditional(s)"

> The end of the input file stream was reached while one or more "#if" conditionals was left unclosed by a corresponding "#endif".

"Unterminated input line"

> The end of the input file stream was reached with a missing newline at the EOF.

"Unterminated string or char constant"

> A string or character constant was being read from the input stream and the end of the line (EOL) was reached without the closing corresponding double quote or single quote.

Preprocessor errors - Fatal

The following errors which may occur during the preprocessor phase of compilation are considered "fatal errors"; they result in the termination of the process.

"-? argument too long"

A command line -d or -u option is longer than 64 characters.

"Duplicate macro arguments"

A parameterized macro definition has two dummy arguments with the same name.

"Illegal constant expression"

The operand field of a "#if" or "#option" must be a valid constant expression.

"Illegal floating point constant"

The floating point constant does not follow the K&R rules.

"Illegal hex constant"

A hexadecimal constant is improperly formed according to the rules for such constants as noted in K&R.

"Illegal macro definition"

A parameterized macro dummy argument list was improperly formed.

"Include filespec missing"

A "#include" directive was encountered which had no file specification in the operand field.

"Includes nested too deeply"

The "#include" statement would have nested too deeply, if not ignored. Up to eight (8) nesting levels are available in MC.

"Invalid command line option"

One of the options specified in the command line was not a valid MC option.

"Invalid #line"

A "#line" preprocessor statement was improperly formed.

"Macro calls nested too deeply"

The maximum macro nest depth is 20 which has been exceeded. You will have to redesign your use of macros.

"Missing ')'"

The operand field of a "#if" or "#option" used an open parenthesis in a constant expression but the corresponding closing parenthesis was omitted.

"Missing ':'"

The conditional operator "?:" was used in the constant expression operand of a "#if" or "#option" directive but the ":" was omitted.

"Missing macro parameters"

A parameterized macro was invoked with fewer formal arguments than when the macro was defined. Double check your argument list.

"Missing or illegal identifier"

The syntax required an identifer, but the input text did not conform to C's rules for identifiers.

"Must be short int or char constant"

#option requires a short integer or character constant if a constant expression is present.

"No input file given"

No input files were specified on the command line that invoked MCP.

"Out of memory"

No more memory space is available. Either decrease the amount of memory reserved in high memory or split the module being compiled into smaller modules with fewer external variables. You may also consider decreasing the amount of modules in high memory (filters, KSM, SYSRES'ed overlays, etc.) and try to execute MCP again.

"Too many macro arguments"

A parameterized macro was defined with more dummy arguments than allowed (128). Double check your argument list.

"Unterminated macro call"

A macro call was missing the closing parenthesis.

Compiler errors - fatal

The following messages may occur during the compiler phase. The errors attributing to the emittance of such messages are considered fatal errors. These messages are unnumbered.

Compilation aborted

A fatal error was detected which prevented the compiler from continuing the process of compilation.

"No input filespec"

No input file was specified on the command line that invoked MC.

"Out of memory"

No more memory space is available. Either decrease the amount of memory reserved in high memory or split the module being compiled into smaller modules with fewer external variables. You may also consider decreasing the amount of modules in high memory (filters, KSM, SYSRES'ed overlays, etc.) and try to execute MC again.

"Too many input files"

Only one input file may be passed on the command line to the compiler.

"Unrecognizable option"

This results from a bad command line option.

Compiler errors - warning

The following messages are warnings and do not normally result in an immedate abort of the compiler's processing. The messages fall into broad classifications. These messages are emitted along with the error number shown.

General syntax errors (10-19)

10 - "Missing ':'"

A "?" operator was found without a matching ":" operator. This may also arise from an omission of the colon from a structure/union bit field declaration.

11 - "Missing '('"

An opening parenthesis was omitted from where it was required.

12 - "Missing ')'"

The syntax of the statement being parsed required a closing parenthesis, which was omitted.

13 - "Missing ']'"

The field specifying the subscript of an array object was not terminated with the closing square bracket.

14 - "Missing '{'"

The opening brace required to designate a compund block was omitted.

15 - "Missing '}'"

The end of the input stream was encountered without a closing brace for the current function being found.

16 - "Missing ';'"

No semicolon was found at the end of a statement. The ';' character is the statement terminator, and must be placed at the end of a simple statement. It's also required in a "for" statement [i.e. for (; ;)].

17 - "Missing ','"

The context of the input required a comma, but none was found.

Constant expression errors (20-29)

20 - "Illegal constant expression"

A constant expression did not follow the K&R rules.

21 - "Must be short int or char constant"

An object of case, array dimension, or initializer requires a short integer or character constant.

Declaration errors (30-49)

30 - "Illegal declaration or typename"

A catchall - Some sort of bad syntax in a declaration or cast or sizeof, not otherwise mentioned.

31 - "Declaration too complex"

More than 16 levels of "array of", "pointer to", or "function returning" occurred in a declaration.

32 - "Struct/union includes self"

A structure/union may not include an instance of itself as a member; however, a pointer to an instance of itself may be included as a member.

33 - "Cannot return struct/union"

It is impossible for a function to return a structure or union; however, a function can return a pointer to a structure or union.

34 - "Function not declared"

The context of the input demanded that a function be declared, i.e., the input did not match anything that could be a compiler directive or a variable declaration, so it was assumed that a function was being declared.

35 - "Zero size illegal"

An array dimension is zero or missing where required, or sizeof returned zero when invoked.

36 - "Redefinition"

The object being declared has already been defined in the module.

37 - "Argument list illegal"

A function argument list was found in a function FORWARD DECLARATION. Only allowed in a function DEFINITION.

38 - "Illegal argument name"

An argument name in the argument list of a function definition is not a valid C identifier.

39 - "Unmatched argument"

The function argument being declared did not match any of those in the argument list for the function.

Initialization errors (50-59)

50 - "Illegal initializer"

This is a catchall emitted when something wrong is detected with the syntax of an initializer.

51 - "Undefined"

The object being referenced in an initializer has not been defined. You may wish to check the spelling.

52 - "Must be static or external"

An initializer using the address of an automatic or register variable is illegal.

Statement errors (60-69)

60 - "Compound statement required"

The switch-case statement must have a compound statement as its sub statement. The body of a function must also be a compound statement (i.e. one enclosed in curly braces).

61 - "Declaration follows executable statement"

All declarations in a block must precede the first executable statement in that block. Resequence your statements.

62 - "No while after do"

A "do" statement was compiled, but no "while" statement followed it.

63 - "Void function return"

A function declared as void contains a return statement with an expression following.

64 - "Bad label"

The label specified in a "goto" statement or as a statement label was not a valid C identifier.

65 - "Extra default"

The body of a "switch" statement included more than one default sub-statement.

Expression errors (70-99)

70 - "Illegal expression"

The input could not be recognized as an expression when the context of the program required an expression.

71 - "Illegal floating point operation"

A float or double expression is using an operator that's not allowed. See the language specifications in Chapter 2 for valid floating point operators.

72 - "Illegal pointer operation"

An operation was attempted on a pointer other than that documented on page 2-12.

73 - "Illegal address"

The "&" (address of) operator was used with an expression that was not an object in memory.

74 - "Not a pointer expression"

The indirection operator, "*", was used with an expression that does not result in the address of an object in memory.

75 - "Using void result"

An expression is attempting to use the non-existent return value of a function that was defined to be of type void.

76 - "Function name illegal"

The name of a function was used improperly. Check for misspelling or missing variable definition.

77 - "Must be lvalue"

The expression being processed specifies that a value be placed into an object, but no object which could be stored into was found. This occurs in assignments, "++", and "--". Usually it's the result of a spelling error.

78 - "Must be struct/union and not array"

The left side of a "." or "->" requires this.

79 - "Unknown struct/union"

The structure/union object being declared or referenced has not had a template defined for it.

80 - "Must be struct/union pointer"

The designated operation requires a pointer, not the object itself.

81 - "Member name required"

The operand on the right hand side of a "." or "->" must be a member of the structure/union on the left hand side.

82 - "No such member"

The referenced member of a structure/union has not been defined in the structure/union template.

83 - "Must be array or pointer"

Square brackets "[]" were applied to an object which was neither a pointer nor an array.

84 - "Illegal function call"

A function call is being attempted, but the object being called is not a function name or dereferenced function pointer.