PDOS-C

C-COMPILER
PROGRAMMERS REFERENCE
MANUAL

Third Edition
November 1986

PART NO. 800041

FORCE COMPUTERS Inc./GmbH
All Rights Reserved

P/N 99

January 9, 1987

PDOS C 5.Øe UPDATE

1. Please insert the attached corrected pages in your PDOS C Reference Manual (3-110 and 3-111 -- the last two arguments in the parameter list of xwfp were swapped).

2. MASM, QLINK and MLIB have been updated to be consistent with the new PDOS 3.2a release.

3. The files XLIB:SRC and XLIB:LIB have been fixed to correct the xwfp error on return as well as the xcbc and xcbp error statuses as noted on page 8 of Vol. 2 No. 1 of the PDOS Tips and Technical Notes.

4. The SYRAM:H and TCB:H files were removed from STDLIB:SRC since they were already on the disk.

5. C:BUG has been updated to note current status for this release.

N O T E

FORCE Computers Inc.
727 University Avenue
Los Gatos, CA 95Ø3Ø
U.S.A.

Phone : (4Ø8) 354 34 1Ø
Telex : 172465
FAX : (4Ø8) 395 77 18

FORCE COMPUTERS GmbH
Daimlerstrasse 9
D-8Ø12 Ottobrunn/Munich
West-Germany

Phone : (Ø89)6 Ø9 2Ø 33
Telex : 5 24 19Ø forc-d
FAX : (Ø 89)6 Ø9 77 93

FORCE COMPUTERS FRANCE Sarl
11, rue Casteja
921ØØ Boulogne
France

Phone : (1) 62Ø 37 37
Telex : 2Ø6 3Ø4 forc-f
FAX : 1 621 35 19

PDOS®

Realtime
Operating
System

C
Reference
Manual

Printed in the United States of America.
Product Number ER3550/M (for PDOS revision 3.2 and C rev. 5.0)
October 15, 1986

PDOS C Version 5.0
Update Notice

The following functional changes have been made to the compiler:

1. CC is the new control program for C. CC allows you to optionally specify libraries and/or object modules to be referenced at link time. If the first argument to CC is an object module, CC performs just the link step.

2. The C compiler now needs 100K of memory to run versus the old 85K. It will, however, report an error rather than try to execute (and crash the system) if there is insufficient memory to run.

3. CLINK has been dropped. Its function is integrated into the new CC program. To specify that additional modules must be linked with the input file, specify their names on the command line, following the options. If no file extension is specified for these additional files, ':O' is assumed. If the extension is specified as ':LIB' the files are treated as libraries and brought in with a 'LI' command in the linker; otherwise they are treated as object files and brought in with the 'IN' command.

   The first command line parameter is always the input source file. The name of this file will be used to name any output files. The second command line parameter is reserved for any options. If none are specified, the second parameter must be blank. If the first command line parameter is a filename with an extension of ":O", the CC program skips the compilation and only performs the link. Any remaining command line parameters are taken to be additional objects or libraries for the link. If no link is performed (S or C option) these parameters are ignored.

4. LOCATE is no longer called by CC to perform a post-link function. It depends on QLINK to do that function via the BITMAP instruction. This version of CC <u>must</u> use the new version of QLINK for this to work.

5. OPTIM has been eliminated. Its usefulness ceased.

6. TRANS68 has been dropped and as such, there is no longer a separate translation phase (TRANS68). Conversion to PDOS assembly syntax is performed in the code generator (C168).

   Modifications to the assembly-language translation phase have made some changes in the rules about defining/referencing global variables. Prior to version 5.0, if a global variable was referenced on the line previous to a static variable declaration, the compiler mistakenly defined the previous global variable. This usually showed up as an error at assembly time, making it more difficult to understand. Now, the compiler will correctly define/reference global variables, but it will not report another error. If the program defines a global variable twice, the compiler will pass both definitions to the assembler, causing an assembler error. (According to Alcyon, it is legal to declare a variable twice, although the space will be only allocated once -- to the largest of the sizes if there are two sizes.)

7. The following options were changed in the compiler:

   O option deleted - its use was ineffectual.
   T option deleted - 68010 support is automatically provided.
   M option changed to Q - it now corresponds with other PDOS high level languages.
   E option changed - it now allows double precision floating point as well as single precision. Both are in software and use the IEEE floating point format. The 'E' option defines the symbol 'IEEE' to the preprocessor.
   S option changed - it now outputs the source to a single file named 'xxxxx:SR' where 'xxxxx' is the source file name minus the extension.
   H option added - it allows single and double precision floating point arithmetic in hardware, using the 68881 co-processor and the 68020 processor. The format is the same as the IEEE floating point format. This also defines the symbol 'M68881'.
   D(ABC) option added - it defines symbols for the preprocessor so it can be used in statements such as: #ifdef ABC
   U(ABC) option added - it removes preprocessor definitions so it can be used in statements such as: #ifndef ABC
   B option added - it causes error messages to be saved in the file 'xxxxxx:ERR' where 'xxxxx' is the name of the source file without the extension. If the 'B' option is not specified, the errors are left in the file 'CTEMPx:ERR' where 'x' is the task number.

The following changes have been made in the libraries:

1. The library programs have been renamed so that the object modules have a ':LIB' extension:

   STDLIB has become STDLIB:LIB
   XLIB has become XLIB:LIB
   IEEEFLIB has become IEEE:LIB
   FFPLIB has become FFP:LIB

2. The following program has been added to the C disk to provide M68881 floating point support:

   M68881:LIB

3. The following functions have been added to the standard library:

   FEOF -- Check for EOF in a stream
   FFLUSH -- Flush output to file
   FREAD -- Read from stream
   FWRITE -- Write to stream
   MEMCCPY -- Copy character with break
   MEMCHR -- Locate character in memory
   MEMCMP -- Compare memory
   MEMCPY -- Copy memory
   MEMSET -- Set memory to character
   RAND -- Return random number
   RENAME -- Rename file
   REWIND -- Position to top of stream
   SRAND -- Seed random number generator
   TSTFILE -- Test for existence of a file

4. The PRINTF function in the STDLIB:LIB library has been corrected to fix a bug that occurred when a decimal number larger than 9 digits was printed. Also, octal conversion functions ("%o and %lo") have been added to PRINTF.

5. The GETC function in the standard library has been changed so that it no longer performs a sign-extend on data read from a file, which made it difficult to distinguish between a byte of 'FF' and the end of file.

6. Buffered I/O has been added to the STDLIB:LIB library. It affects functions FOPEN, FCLOSE, GETC, UNGETC, PUTC, FSEEK, FTELL, and EXIT. The additional functions _FILLBUF and FFLUSH have been added. I/O to a port is still unbuffered, but I/O to disk files is buffered 252 bytes at a time. The include file "stdio.h" has been modified to correspond with these changes. Any programs including "stdio.h" should be recompiled. Also, the file CSTART:ASM has been modified to accommodate the change. The program WC:C on the new diskette runs about four times faster with buffered I/O than without.

7. The following functions in the PDOS cross library have been enhanced:

   XSZF -- Get disk parameters
   XWFP -- Write file parameters

8. The following functions have been added to XLIB:LIB:

   XDMP -- Dump memory
   XFAC -- File altered check
   XGMP -- Get message pointer
   XPAD -- Pack ASCII date
   XPCR -- Put character raw
   XRTP -- Read time parameters
   XSMP -- Send message pointer
   XUAD -- Unpack ASCII date

9. The following functions have been added to XLIB:LIB as available by in-line code:

   X881 -- MC68881 enable
   XUSP -- Return to user mode

10. The following functions in XLIB:LIB were corrected to operate correctly:

    TESTXLIB:C
    XBFL - Build file listing
    XGML - Get memory limits

11. MC68881 floating point hardware support was added. Most of the existing floating point functions are supported by this library.

12. The following functions have been added to the floating point libraries:

    ASIN - Arc sine
    ACOS - Arc cosine

The following changes were made in the <u>PDOS C Reference Manual</u>:

1. The format has been changed for easy readability.

2. More examples have been added.

3. An appendix containing error messages and their explanations has been added.

4. An index has been included with the manual for easy reference.

The following corrections have been made to the C compiler:

1. C now handles system V style declarations of functions which return pointers to functions.

2. Structure tag names no longer conflict with structure field or variable names.

3. Occasional complaints regarding casting of expressions to void are no longer a problem.

4. Complex expressions including function return values generate correct code now (eg. var <<= func()).

5. Long and unsigned long bitfields supported.

6. Unsigned character as index into array works properly.

7. Structure field matching problems involving typedefed structures and complicated matching problems involving conditional statements have been fixed.

8. Modifications to the code generator have eliminated many of the problems with complex expressions generating "expression too complex" errors.

9. The compiler now allows an automatic which is a pointer to an array to be stored into a register. For example: register char *c[];

10. Bit field code generation has been fixed to correctly handle multiple assignments involving a bit field. For example:
    ```
    i = j = strc.bfield; or strc.bfield = i = j;
    ```

11. An error message is now generated if two structure fields in the same structure are not unique within eight characters.

12. Problems involving floating point conversion function calls overwriting temporary registers have been fixed.

13. sizeof() on an item which is greater than 32K will now generate a warning message.

14. Unsigned character cast on compile time initializer works properly now.

15. An error message is now generated if the left hand side of period operator is a pointer.

16. Structure prototypes can now have the same name as variables. For example:
```
     struct proto { int a, b, c, d; } proto;.
```

17. Structure fields can have the same name as variables. For example:
```
     typedef struct xyz { char abc; } abc;
```

18. Typedef structures can have the same prototype and typedef names. For example:
```
     typedef struct proto { int field; } proto;.
```

19. Function returns involved in shift-equal expression now work properly. For example:
```
     a >>= func();
```

20. The compiler now handles two constructions of functions returning pointers to functions. For example:
```
     (*Oldway())(arg1,arg2) {} and (*Newway(arg1,arg2)) {}.
```

21. Re-declaration of function return now generates a warning rather than an error message. For example:
```
     int func(); long func();.
```

22. The operator precedence of ?: operations involving embedded assignments has been fixed to match the system V syntax. For example:
```
     a < 0 ? a = b : "" ;   is equivalent to
     a < 0 ? (a=b) : "" ;.
```

23. Multiple structure assignments are now allowed. For example:
```
     struca = strucb = strucc;.
```

24. Structures exactly three words in length can now be assigned to one another.

25. # on left margin inside a multi-line string is no longer interpreted by the pre-processor.

26. A long variable assigned to zero as an index into an array now generates correct code. For example:
```
     array[lng=0] = 30;
```

27. The bad error message "& operator illegal" has been changed to "& operator ignored" for case of array passed as argument. For example:
```
     func(&array);
```

28. Complicated expressions which caused function returns to be stored into the wrong register or be overwritten have been fixed.

29. Correct code is now being generated when a register pointer variable is assigned to a register char. For example:
```
     regchar = regcptr;.
```

30. Sizeof on a string now generates the correct results. For example:
```
     sizeof("string");.
```

31. Initializer alignment problems involving structures with bit-fields have now been fixed.

32. Errors with structures involving long bit fields whose sizes were incorrectly calculated have been fixed.

33. Since compilers which support prototypes are becoming popular, the compiler now ignores the contents of the parentheses in an external declaration of a function. For example:
```
     extern int func(char *,int,long,long);
```

34. The #elif preprocessor command is now implemented. This acts like a combination of #else and #if. It is used between #if and #endif in the same way as the #else command, but it takes an argument like the #if command.

35. Unnamed bit-fields are now implemented.

36. Enumerated data types (keyword "enum") are now implemented.

37. The 'e' in the floating-point constant is now protected from macro expansion by the pre-processor.

38. Errors in structure/array initialization code have been fixed.

39. Initialization of multi-dimensional arrays of structures now works.

40. The expression handler has been changed to generate a warning if a period or pointer operand is used on a non-structure/union variable which is not being cast.

41. The compiler now correctly handles incrementing/decrementing of the pointer to items larger than 32K in size.

42. The compiler now performs properly handling of greater than 32K of local variable declarations.

43. The compiler now correctly generates assembly code for indexing into arrays of structures which are greater than 32K bytes in size and if the size of the structure is a power of 2.

44. If a static function is passed as a parameter, the compiler no longer generates a literal 1 instead of its address.

45. Auto-increment and auto-decrement operations on bit fields are now allowed.

46. The results of the relation operations on unsigned long items now yields a long result in all cases.

47. Unnamed bit fields now cause the following component to start on a byte/word/longword boundary as appropriate.

48. Unsigned-int or unsigned-long conversions to float or double were being assigned as if they were signed values. They have been fixed.

49. Sixteen-bit literal constants in the range 0x8000 to 0xFFFF are not sign-extended when assigned to a long int. Previously, the assignment long int a = 0x8000; resulted in containing 0xffff8000.

PDOS C REFERENCE MANUAL
TABLE OF CONTENTS

CHAPTER ONE
INTRODUCTION

This manual consists of four chapters. This first chapter provides an introduction to the compiler; the following three chapters detail the subroutines in the standard library (STDLIB:LIB), the PDOS interface library (XLIB:LIB) and the floating point libraries (FFP:LIB, IEEE:LIB, and M68881:LIB).

Chapter 1 gives an overall description of the contents of the PDOS C disk and tells how to use the compiler.

Chapter 2 describes the routines in the standard library. These routines are generally functional equivalents of routines on Unix.

Chapter 3 explains how to access the operating system interface of PDOS and use the different operating system primitives described in the PDOS Reference Manual.

Chapter 4 examines the floating-point routines that are available to the C programmer and how to use them. Floating point support consists of various I/O routines, conversion routines, transcendental functions, and arithmetic operations.

This manual is intended for someone who already understands C programming and wants to write C programs under PDOS. It is not an introduction to the C language.

To clarify the text, all examples and listings in this manual are printed in a smaller, sans-serif type, user input is bolded, and comments are printed in italics. Elipses indicate that some text (generally irrelevant or too large in quantity) that would normally appear on your screen has been omitted from the manual.

1.1 DISKETTE INSTALLATION

The PDOS C diskette that you received contains everything you need to create C programs (except an editor). A few sample programs are included on the diskette to let you test out the compiler.

Before you do anything with the compiler, back it up onto your Winchester disk or another floppy (use MBACK or MTRANS) and store the original diskette in a safe, dry place.

DON'T WRITE ON THE ORIGINAL DISK!

```
x>MBACK 0,0,,Y[CR]
PDOS Disk Backup Utility
  Source Disk # = 0[CR]
  Destination Disk # = 0[CR]
  Number of sectors = 2528[CR]
  Backup 'C 5.0..........
  Insert source disk in drive 0.  Hit <CR>....[CR]

Reading sector 0..2527
  Insert destination disk in drive 0.  Hit <CR>....[CR]

Writing sector 0..2527
  SUCCESS!  Disk Name = C 5.0..........'
```

When you have copied the disk onto your work area, you should try out the compiler by testing it on a few of the sample programs. The C compiler needs 100K of memory to run and and you will need to allow sufficient disk space for the programs as well. If HELLO:C, ECHO:C, and SIEVE:C compile and execute without errors, you may continue. If you do encounter difficulties, contact your PDOS distributor.

NOTE: **The original C disk you received does not contain sufficient free space to compile C programs. You MUST copy the compiler to a work disk -- preferably one with a few hundred extra sectors available for the temporary files.**

```
x>CC HELLO[CR]
x>HELLO[CR]
Hello, world!
x>CC ECHO[CR]
x>ECHO THIS IS A TEST[CR]
ECHO
THIS
IS
A
TEST
```

(1.1 DISKETTE INSTALLATION cont.)

Now install the appropriate help files in your HLPTX file. Edit the file HLPTX on your system disk and merge in the file C:HLP. If you already had help text in that file from an earlier revision of C, replace it with this file. That way, when you type 'HE CC', you will see how to run the current version of the C compiler.

```
x>MEDIT HLPTX[CR]
```

Use [CTRL-Z] to position to the end of file, use [CTRL-Y]C:HLP[CR] to bring in the C help, and write the result back out with [CTRL-W][CTRL-W][CR]V. Exit the editor with [ESC][CTRL-V].

If you are short of space, the following files are not essential for using the compiler but are provided for your information. You may delete them from your working copy of the C disk, but <u>DON'T DELETE THEM FROM THE ORIGINAL DISK!</u>

@:SRC;@	Sources of the different subroutine libraries
@:DOC;@	Various help files explaining contents of the disk
@:C;@	Sample C programs and utilities
@:ASM;@	Sources of the initialization code -- only the object files are necessary to compile and load your programs

1.2 THE PDOS C COMPILER

This version of the C compiler for PDOS on the 68000 and 68010 processors or the 68020 processor with the 68881 co-processor is a complete implementation of the language defined by Kernighan and Ritchie in <u>The C Programming Language</u>. You will find it helpful to have a copy of <u>The C Programming Language</u> or a similar text available to reference. This manual is not a tutorial on the language, but rather a description of the PDOS implementation. There are a large number of good texts on the language in most bookstores.

The majority of the Unix portable C library functions have been implemented in the compiler. If one of your favorites has somehow been overlooked, contact Eyring for it to be included in a future release.

Source to the library functions has been included for the following reasons:

1. The source to a routine is the best documentation explaining that routine. If the manual leaves you unclear on a particular point, you might find the answer in the source itself.

2. The library functions do a lot of things, but you may want to write new functions. You can use these routines as examples to learn how to build your own library.

3. If you are willing to accept the responsibility of maintaining a variant library, you can modify the functions in the PDOS C library to suit your own tastes.

This version of the compiler was ported from Alcyon C68 version 5.0. It was released by Alcyon at the beginning of 1986 and is a mature product. It has been successfully ported to a large number of operating systems besides PDOS.

1.3 CONTENTS OF THE C DISKETTE

Following is a directory listing of the PDOS C diskette. Files that are found only on the 68020/68881 version of C are noted with an asterisk.

Lev	Name:ext	Description
****	HELP FILES	
0	C:HLP	Help file for compiler
0	C:BUG	Current bug list
****	COMPILER	
1	MASM	Latest version of the assembler for 68000/68010
*1	MASM20	Latest version of the assembler for 68020/68881
1	QLINK	Latest version of the linker
1	MLIB	Library manager utility
1	CC	Control program to compile C programs
1	CPP	Pre-processor phase of the compiler
1	C068	Parser phase of the compiler
1	C168	Code generator phase of the compiler
1	ROMLINK	Object module creation phase for ROM programs
****	LIBRARIES	
3	STDLIB:LIB	Standard C routines library
3	XLIB:LIB	PDOS interface library
3	IEEE:LIB	IEEE floating point library
3	FFP:LIB	FFP floating point library
*3	M68881:LIB	M68881 floating point library
3	CSTART:O	Starting module for C programs -- always linked in first
3	CEND:O	Termination module -- always linked in last

(1.3 CONTENTS OF THE C DISKETTE cont.)

**** INCLUDE FILES

4	stdio.h	Standard I/O include file <stdio.h>
4	STDIO:H	Same as <stdio.h> but with PDOS name
4	SYRAM:H	Definition of PDOS system memory record -- current rev.
4	OLDSYRAM:H	Definition of PDOS system memory record -- previous rev.
4	TCB:H	Definition of Task Control Block record
4	ctype.h	Character conversion macro definitions
4	CTYPE:H	Same as <ctype.h> but with PDOS name
4	memory.h	Definition of memory functions
4	math.h	Definition of floating point routine entry points
4	SETJMP:H	Definition of environment block for SETJMP/LONGJMP call
4	setjmp:h	Same as SETJMP:H
4	FILESLOT:H	Definition of file slot record in system
4	DIRENT:H	Definition of a directory entry on disk

**** COMPILER UTILITY SOURCE CODE

11	ROMLINK:C	Source to object module create phase of linker
11	CC:C	Source to control program of compiler
11	TESTXLIB:C	Source to suite of tests for XLIB--examples of XLIB calls

**** TEST PROGRAMS

150	ECHO:C	Sample C program -- echoes command arguments
150	HELLO:C	Sample C program -- "Hello, world!"
150	SIEVE:C	Sample C program -- prime number sieve benchmark

**** START-UP/TERMINATION SOURCE

150	CSTART:ASM	Source for initialization module
150	CEND:ASM	Source for termination module

**** LIBRARY SOURCE

150	STDLIB:SRC	Source of standard C library modules
150	FFP:SRC	Source of floating point modules
150	XLIB:SRC	Source of PDOS interface modules
150	FPERR:S	Source of IEEE run-time error handler

**** PROCEDURE FILES

151	CC:AC	Command file to compile C programs

**** PUBLIC DOMAIN C PROGRAMS (UNSUPPORTED)

152	FDIFF:C	Source to text file comparison utility
152	GREP:C	Source to pattern recognizer program
152	HANOI:C	Game source
152	SORTC:C	Source to sort utility
152	WC:C	Word/Byte/Line count utility source

Note: Some of the include files have a period embedded in the name. Because the monitor stops parsing on a period (the period normally acts as a command separator), it would seem as though these "illegal" file names were impossible to access. They are accessible from the monitor, however, if you enclose the name in parentheses.

```
x>SF stdio.h[CR]
PDOS ERROR 53 not defined

x>SF (stdio.h)[CR]
/* stdio.h -- standard defines for C under PDOS.
   Eyring Research Institute, Inc.  Copyright 1984-1986
*/
       .
       .
       .
```

1.4 RUNNING THE COMPILER

The compiler is invoked using the control program CC. This program has a number of options, described in the help file "C:HLP". You can compile a program from source to executable code, compile several modules to object code for later linking, or compile a module to assembly language source. You may optionally specify floating point in either FFP format (single precision only) or IEEE format (double and single precision), with optional 68881 support for the 68020 C. Finally, the compiler can generate object code suitable for linking into a ROM image.

Compile sieve:

```
x>CC SIEVE
x>SIEVE
100 iterations
 1899 primes
```

1.4.1 CC

The CC program handles the scheduling of the compiler phases, putting the detail of the phases of the compilation into the background. CC also allows the compilation options to be specified in any order, in upper or lower case, and allows the user to specify disk units on input files or to override the default file extensions. CC allows you to optionally specify libraries and/or object modules to be referenced at link time. If the first argument to CC is an object module, CC performs just the link step.

The source to CC is on the original C diskette. You may customize it to suit your own requirements, but you are liable for the consequences.

The CC program does not prompt interactively for its parameters. Instead, if you type 'CC' without arguments, it displays a help message to describe the format of the argument(s) it requires. You may also want to examine the file "C:HLP" for a further description. All of the arguments to CC must be specified on the command line.

```
x>CC[CR]
68K PDOS C Compiler R5.0 09/05/86
Eyring Copyright 1985-1986
Usage: CC <filename>,<options>,<object files and libraries>
  filename extension defaults to :C, but others may be specified.
  if extension is :O, only the link step is performed.
  filenames after the options are treated as libraries (extension= LIB)
    or as object files to be linked in with the first file
  options follow the filename in any order with these definitions
      V      : display each step as it executes
      D(abc) : define symbol 'abc' for preprocessor
      U(abc) : undefine symbol 'abc' for preprocessor
      F,E,H  : floating point. F = Fast Floating Point (single precision)
                               E = IEEE floating point
                               H = 68881 floating point
      B      : output error messages to 'xxxxx:ERR'
      S      : don't assemble -- leave in xxxxx:SR
      C      : don't link -- leave in xxxxx:O
      Q      : create 'SFU' link map
      Q(xyz) : create 'xyz' link map (options in parentheses)
      R      : create ROMable object instead of SY file
x>_
```

*Option H is only available on C for the 68020.

1.4.2 Compile Time Parameters

The first command line argument to CC must be the name of the file to compile. If the extension is not specified, it is assumed to be ":C". A disk volume may be specified for the source, with or without the extension.

```
x>CC HELLO          - compile HELLO:C
x>CC HELLO/2        - compile HELLO:C/2
x>CC HELLO:C        - compile HELLO:C
x>CC HELLO:C/2      - compile HELLO:C/2
x>CC hello:c        - compile hello:c (lower case!)
x>CC HELLO,CV       - compile HELLO:C to HELLO:O, verbose mode
```

The second command line argument is reserved for options. Options may appear in any order, in upper or lower case. The options are:

```
      V      : display each step as it executes
      D(abc) : define symbol 'abc' for preprocessor
      U(abc) : undefine symbol 'abc' for preprocessor
      F,E,H  : floating point. F - Fast Floating Point (single
                                   precision)
                               E - IEEE floating point
                               H - 68881 floating point
      B      : output error messages to 'xxxxx:ERR'
      S      : don't assemble -- leave in xxxxx:SR
      C      : don't link -- leave in xxxxx:O
      Q      : create 'SFU' link map
      Q(xyz) : create 'xyz' link map (options in parentheses)
      R      : create ROMable object instead of SY file
```

'V' OPTION

The compiler may take a while to run through all of its steps. If you are patient, you can sit and wait, but you may want to see what it is doing. Specifying the 'V' option tells the compiler to display each phase of the compilation as it executes.

```
x>CC HANOI,V        Compile the HANOI puzzle and display all steps of the
                    compilation
```

'F' OPTION

The default option of the compiler assumes no floating point code. Bringing in floating point when it is not needed results in a large amount of unnecessary code in your program. Specifying the 'F' switch tells the compiler to use Fast Floating Point format when compiling floating point constants and to search "FFP:LIB" during the link phase. For more information, see Chapter 4 of this manual on floating point.

x>CC TEST,F — *Compile TEST using Fast Floating Point format and link with FFP:LIB*

'E' OPTION

The default option of the compiler assumes no floating point code. Bringing in floating point when it is not needed results in a large amount of unnecessary code in your program. Specifying the 'E' switch tells the compiler to use IEEE software format when compiling floating point constants and to search "IEEE:LIB" during the link phase. For more information, see Chapter 4 of this manual on floating point.

x>CC TEST,E — *Compile TEST using IEEE floating point format and link with IEEE:LIB*

'H' OPTION

The default option of the compiler assumes no floating point code. Bringing in floating point when it is not needed results in a large amount of unnecessary code in your program. Specifying the 'H' switch tells the compiler to use IEEE 68881 hardware format when compiling floating point constants and to search "M68881:LIB" during the link phase. For more information, see Chapter 4 of this manual on floating point. This option is only available for C for the 68020.

x>CC TEST,H — *Compile TEXT using IEEE hardware floating point format and link with M68881:LIB*

<u>'S' OPTION</u>

Sometimes it is important to see the assembly language generated by the C compiler. Comparing the assembly language to the original source helps you to understand exactly what the compiler is doing with your code. If you use the PDOS debugger 'PB' to debug your C program you need to see the assembly language to trace through the program logic. If you have code that needs more optimization than the C compiler can provide, you can take the assembly language output of the compiler as a starting point and optimize it by hand. The assembly language source file created by the 'S' option has the same name as the source, but with an extension of ":SR."

```
x>CC ECHO,S        Compile ECHO:C to ECHO:SR
```

<u>'C' OPTION</u>

This option compiles your program to object code but skips the link step. The resulting file has the same name as the original source, but an extension of ":O". Thus you can compile several modules and later link them together into one program. Or, you can create your own library of commonly used C functions. For more information, see section 1.6.

```
x>CC HELLO,C       Compile HELLO:C to HELLO:O
```

<u>'Q' OPTION</u>

This option gives you access to the link map facility of QLINK. You may simply specify 'Q' as an option, which will create the default 'SFU' map (section, files, undefined), or you may request any combination of map options in parentheses after the 'Q'. See the <u>PDOS Reference Manual</u> under QLINK for more information.

```
x>CC HELLO,Q       Create HELLO:MAP on linking
x>CC HELLO,Q(S)    Create HELLO:MAP (sections only)
```

'R' OPTION

This option lets you create a completely linked, relocatable object module version of your program suitable for linking with RUNGEN and QLINK into an EPROM module. For more information, see the section 1.10.

```
  x>CC HANOI,R     Compile the HANOI puzzle to burn in ROM
```

'D' OPTION

This option allows you to pre-define symbols for the preprocessor. These symbols are then available for reference in your code via the following preprocessor commands:

```
#ifdef xxxxxx
  and
#ifndef xxxxxx
```

For instance, it is possible to put debugging statements in your program that normally are treated as comments because they are surrounded by the following lines:

```
#ifdef DEBUG
     .
     .
     .
#endif
```

Then, when you want the debugging statements to be part of the program, you can recompile it with the following instruction:

```
x>CC MYPROG,D(DEBUG)[CR]
```

The preprocessor will then include those statements in the executable code.

'U' OPTION

This option allows you to remove preprocessor definitions. The PDOS C compiler pre-defines the symbols "MC68000" and "PDOS". These symbols can be used in your program through instructions such as the following:

```
#ifdef PDOS
        printf("\nThis was compiled under PDOS");
#endif
```

You may remove these definitions by including the symbol name in parentheses after the 'U' option as follows:

```
x>CC MYFILE,U(PDOS)[CR]
```

Note: Both the 'U' and 'D' options may be specified multiple times in a single command line, interspersed with other commands, for example:

```
x>CC MYFILE,FD(DEBUG)U(PDOS)D(TESTING)U(MC68000)Q(DOU)R[CR]
```

This example will compile MYFILE using FFP format for the floating point referencing FFP:LIB during the link; defining the symbols "DEBUG" and "TESTING" for the preprocessor; removing the definitions "PDOS" and "MC68000"; and instructing the linker to create a link map "MYFILE:MAP" containing the symbol definitions, any undefined symbol references, and any overflow references. Finally, the program is passed through ROMLINK to produce an EPROM object file suitable for input to RUNGEN.

'B' OPTION

If you select the 'B' option, the compiler will provide you with a listing of errors saved to a file. If you would like error messages to be saved (as would be the case if you were compiling a large number of modules with a procedure file), specifying the 'B' option will send all error messages to a file with the same name as the source file and an extension of "ERR".

If the 'V' switch is active, all messages are copied to the user terminal as well. Otherwise, the compiler spools all messages to a file called "CTEMPx:ERR" and if there is an error on the compilation, displays that file to the screen.

The following command will capture any error messages in a file named "MYPROG:ERR":

```
x>CC MYPROG,B[CR]
```

1.4.3 Errors

When any phase of the compiler detects an error in your program, it reports it immediately. The error message has two parts:

1) the line in the file where the error occurred, and
2) a message describing the problem.

One error early in the file can cause a large number of errors to "appear" later. Usually, you should correct the early errors and see if the others go away by themselves.

The assembler may report an error if you make a mistake in an "asm" directive or if there is an error while creating the object file. For instance, the disk might be full, or there might not be room in the file directory to create a new file. If a global symbol is defined multiple times, that error may not be reported until assembly time. This error can occur if two global variables or functions have the same name or if the names are not unique in the first seven characters. See the PDOS Reference Manual for more information on the assembler and its errors.

The following errors may be reported at run time when your program is compiled with the 'E' option:

401	Double precision argument error
402	Single precision argument error
403	Double precision divide by zero
404	Single precision divide by zero
405	Integer divide by zero (not used from IEEE:LIB)
406	Double precision overflow
407	Single precision overflow

These errors may be trapped by substituting your own code for the instructions in the module FPERR:S.

1.5 PHASES OF THE C COMPILER

The C compiler runs in several phases, making use of intermediate files along the way. Currently a source file goes through a pre-preprocessor (CPP), a parser (C068), a code-generator (C168), the PDOS assembler (MASM/MASM20), and the PDOS linker (QLINK).

The temporary files used are CTEMPx:O, CTEMPx:L, CTEMPx:SR1, and CTEMPx:SR where 'x' is your current task number. The compiler will run faster if you pre-define these temporary files on your fastest disk. The RAM disk (disk 8) is best if you have a large enough one. For small to medium-sized programs, CTEMPx:O and CTEMPx:SR should be about 100 blocks long each. CTEMPx:SR1 and CTEMPx:L should be about 10 blocks long each.

The rest of this section describes each phase of the compiler, along with a few options that apply to each one. Normally, the operation of each phase is hidden from the user, but if you compile with the 'V' option (for view) then each phase is displayed as it occurs.

It isn't necessary to understand the different phases of the compiler in order to use it; you may skip through to section 1.6 on first reading.

```
x>CC HELLO,V[CR]
CPP HELLO:C CTEMPØ:SR
CØ68 CTEMPØ:SR CTEMPØ:O CTEMPØ:L CTEMPØ:SR1
C168 CTEMPØ:O CTEMPØ:L CTEMPØ:SR HELLO
MASM CTEMPØ:SR,CTEMPØ:O
```

Assembler messages

```
QLINK
```

Linker messages

1.5.1 CPP -- C Pre-Processor

CPP is a true pre-processor and performs two major tasks: 1) macro processing and, 2) merging include files. It resolves conditional assembly, removes comments, expands macros, and produces one simple file that just contains C program code.

```
x>CPP HELLO:C CTEMPØ:SR[CR]

x>CPP -DDEBUG ECHO:C CTEMP[CR]
```

CPP can optionally accept symbol definitions on the command line in the form '-Dxyz' occurring before the name of the input file. Alternately, pre-defined symbols can be "un-defined" on the command line by specifying '-Uxyz' before the name of the input file. Symbols may be defined or undefined on the CC command line with the 'D' and 'U' switches. CPP pre-defines the following symbols:

```
PDOS
MC68000
```

Symbols defined in CPP may be used to include or exclude sections of code that are machine dependent. For example, the following PRINTF statement will be included if the code is compiled by the PDOS compiler but not if it is compiled by other compilers.

```
#ifdef PDOS
     printf("compiled under PDOS");
#endif
```

<u>Pre-defined Macros</u>

CPP has four predefined macros that can be used by the programmer. These are "__FILE", "__LINE", "__DATE", and "__TIME". Each of these macros is expanded to the current value when it is encountered. "__FILE" is replaced by the name of the current source file, in double quotes. "__LINE" is replaced by the current line number, as a numeric constant. "__DATE" is replaced by the current date, in "MM/DD/YY" format. "__TIME" is replaced by the current time, in "HH:MM:SS" format.

CPP CONVERTS THIS	TO THIS
`char *date = __DATE;`	`char *date = "Ø8/27/85";`
`char *time = __TIME;`	`char *time = "1Ø:44:Ø4";`
`char *file = __FILE;`	`char *file = "TEST:C";`
`int line = __LINE;`	`int line = 6;`

Include Files

Under the PDOS C pre-processor, there is no difference between the two forms of the include statement.

C compilers on other systems use the angle brackets to reference files on a "system" disk area, while the quotes are used for files on the local disk area. There is no disk area on a PDOS system specifically designated as a "system" disk. The following two forms of the include statement are equivalent on PDOS:

```
#include <stdio.h>
#include "stdio.h"
```

1.5.2 C068 -- C Parser

The parser converts the C source code to an intermediate tree format. It actually produces three output files from the input file, but the third file (containing quoted strings) is concatenated to the first output file.

```
x>C068 CTEMP0:SR,CTEMP0:O,CTEMP0:L,CTEMP0:SR1[CR]
```

1.5.3 C168 -- C Code Generator

The code generator converts the intermediate files produced by the parser into 68000 assembly language.

```
x>C168 CTEMP0:O,CTEMP0:L,CTEMP0:SR,HELLO[CR]
```

The fourth argument to C168 tells which symbol to use in the IDENT statement. Normally, it is the name of the input module.

1.5.4 MASM -- PDOS Assembler For C

PDOS C is shipped with a compatible version of the PDOS compiler, either MASM for 68000/68010 systems or MASM20 for 68020 systems. Do not try to use PDOS C with earlier versions of the assembler. However, versions of MASM later than those on the C disk should work fine.

```
   x>MASM CTEMP2:SR,CTEMP2:O[CR]
   68K PDOS Assembler
```

1.5.5 QLINK -- PDOS Quick Linker For C

PDOS C is shipped with a compatible version of the PDOS linker. Do not try to use PDOS C with earlier versions of QLINK than the version that is on the C disk. Versions of QLINK later than those on the C disk should work fine.

```
   x>QLINK[CR]
   PDOS 68K Quick Linker
```

1.5.6 ROMLINK -- Post-Link Object Module Creation for RUNGEN

The self-relocation and absolute addressing mode features of a C program prevent the user from simply plugging C programs into the RUNGEN package "as-is." The 'R' option of CC allows you to create a C program file in "OB" format that can be supplied to the link phase of the RUNGEN procedure. This object file has all the absolute program references as relocatable references relative to SECTION 0 (intended to be burned in ROM) and all the absolute data references as relocatable references relative to SECTION 1 (intended to be part of the RAM space). To create this object file, the ROMlink program uses the SYfile created by QLINK, along with the link map describing all the absolute references. The name of the SYfile is the first parameter, the link map is the second, and the output object file is the third. See section 1.10 of this manual for more information.

```
x>CC HELLO,R[CR]
```

1.6 LINKING SEPARATELY COMPILED MODULES

Normally only the smallest programs can be conveniently handled as one module. Larger projects need to be broken up into modules which are compiled separately and combined at link time. The CC control program allows you to compile a module and simply create the object module, skipping the link phase by using the 'C' option after the file name. Or, you may want to write modules in assembly language, compile them with MASM, and link them together with your C program.

These object files can be linked into an executable program with the CC control program. The number of modules to be linked together is limited to the number of files you can put on a single command line (78 characters). If you need to link more files for an application, you should build your own command file for that purpose, using the 'V' option output of the compiler as an example. The file "CC:AC" is another simple example.

```
x>CC HELLO,V[CR]
CPP HELLO:C CTEMP0:SR
C068 CTEMP0:SR CTEMP0:O CTEMP0:L CTEMP0:SR1
C168 CTEMP0:O CTEMP0:L CTEMP0:SR HELLO
MASM20 CTEMP0:SR,CTEMP0:O
68020 PDOS Assembler R3.1a 08/27/86
ERII, Copyright 1983-86
SRC=CTEMP0:SR
OBJ=CTEMP0:O
LST=
ERR=
XRF=
END OF PASS 1
END OF PASS 2
QLINK
PDOS 68k Quick Linker 08/11/86
ERII, Copyright 1983-86
*ZE
*SE 0,0
*GR 0,1
*IG 2
*BITMAP BEGIN
ENTRY ADDRESS=00000000
*IN CSTART:O
ENTRY ADDRESS=00000000
*DEFINE DOX881 $4E71
*DEFINE DOXGNP $FFFFA05A
*IN CTEMP0:O
*LI STDLIB:LIB
 INPUT EXIT:O
 INPUT FFLUSH:O
 INPUT XPRINTF:O
 INPUT FPUTC:O
 INPUT CLOSE:O
*LI XLIB:LIB
 INPUT XCBC:O
 INPUT XCBX:O
 INPUT XCHX:O
 INPUT XWBF:O
*IN CEND:O
*RELINK 2,Q$H0
Q$H0=$0000077A
*BITMAP END
ENTRY ADDRESS=00000000
*OU #HELLO
*MA MUO,CTEMP0:ERR
*SY
Start address=$00000000
  End address=$000007F6
*EN
SY FILE: BASE=$00000000  LENGTH=2038
*QU
x>_
```

You may need to build your own library of modules using the PDOS utility MLIBGEN or MLIB. Give this library file an extension of ":LIB" and you may specify it in the command line along with the other modules to be linked; CC will tell QLINK to search this library (as well as the standard libraries) to resolve unresolved symbols. The full name of the library file, with the extension, must be given for CC to recognize it as a library file.

CC always uses the name of the first module in the command string to name the output SY file.

```
x>CC A,C[CR]                    create three object files
x>CC B,C[CR]                    named A:O,B:O, and C:O
x>CC C,C[CR]

x>MLIBGEN[CR]
68K LIBRARY GENERATOR 07/29/85
Copyright 1983, ERII
LIBRARY FILE=#MYSTUFF:LIB[CR]
INPUT FILE=A:O[CR]                  create library from
INPUT FILE=B:O[CR]                  object files
INPUT FILE=C:O[CR]
INPUT FILE=[CR]
ANY MORE FILES (Y/N)?N[CR]
```

In The C Programming Language it states "Only the first eight characters of an internal name are significant...for external names the number may be less" (p. 33). In PDOS C, external names are only significant to seven characters, and lower case is mapped onto upper case. Be careful to use variable names in which the first seven letters are unique.

Only a few of the CC options affect the link process. These options follow:

'V'	Displays all steps of the linkage process on the screen.
'E,F,H'	Control which floating point library (if any) will be referenced.
'R'	After the link process, ROMLINK will create an object module suitable for input to RUNGEN.
'Q'	Direct QLINK to produce a Section/File/Undefined map.
'Q(xyz)'	Direct QLINK to produce an 'xyz' map, where 'xyz' are valid map options (see the PDOS Reference Manual.)

1.7 RUN TIME LIBRARIES

The run time library support consists of five libraries -- STDLIB:LIB, XLIB:LIB, FFP:LIB, IEEE:LIB, and M68881:LIB. These are the standard I/O library, the PDOS interface library, and the floating point libraries.

1.7.1 STDLIB:LIB -- Standard Library

Since C does not intrinsically contain input/output statements, all I/O must be handled through function calls. Most implementations of C have settled on a few standard I/O routines with an established calling sequence. Adherence to this standard makes it easier to move C programs from one system to another.

The routines in STDLIB:LIB handle I/O, memory allocation, string manipulation, and a few miscellaneous sytem functions. They are functionally equivalent to the most common routines in other C implementations. A few routines vary to some degree or another. If you have problems, check the definition of the function in the library.

1.7.2 XLIB:LIB -- Interface to PDOS Primitives

The second library is a collection of routines to support calls into PDOS. They are, for the most part, straightforward implementations of the functions described in the PDOS primitives chapter of the PDOS Reference Manual. Most, but not all of the functions are supported -- some would be useless to most programs. Currently, more than 80 functions are supported by means of function calls, while an additional twelve functions are available through in-line assembly language calls. This library will be updated to keep pace with the changes to PDOS.

1.7.3 FFP:LIB, IEEE:LIB and M68881:LIB--Floating Point Libraries

The remaining libraries consist of a collection of floating point routines. Since the entry-points are the same for both sets of libraries, they are treated the same, except for one discussion of the difference in their formats. Floating point requires a considerable amount of overhead and increases the size of a task, so if you don't need floating point, don't use it.

```
x>CC FFPTEST,F[CR]
x>CC IEEETEST,E[CR]
x>CC HARDWTST,H[CR]
```

1.8 PROGRAM INITIALIZATION -- CSTART

At the head of every PDOS C program is the initialization module, CSTART. This module performs certain tasks on start-up to provide the proper environment for C programs. These tasks are collecting command line arguments, initializing the uninitialized data section, setting up task global variables, and self-relocating the task. The source to CSTART is provided for your reference in CSTART:ASM.

1.8.1 Command Line Parameters -- ARGC,ARGV

When a program is initialized, the parameters are gathered and the traditional argc,argv arguments passed to the "main" routine. See the program "ECHO:C" for an example. Remember that the parameters are gathered by XGNP, so you normally will not need or use the XGNP call. Also remember that the parameters are left in the monitor buffer, so either copy the arguments out of the buffer or beware of using other PDOS calls that modify that buffer. Specifically, the XGLM (get line in monitor buffer) function will overwrite the parameter list.

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

x>TEST This is a test[CR]
```

Execution of the above command line will result in the following variables having these values:

```
argc = 5;
argv[0] = "TEST";
arcv[1] = "This";
argv[2] = "is";
argv[3] = "a";
argv[4] = "test";
argv[5] = NULL;
```

1.8.2 Uninitialized Variable Space

A second function of initialization is to zero the uninitialized variable space. C requires that static or global variables with no initialization given be set to zero. The compiler locates all such variables in SECTION 2. (See the discussion on SECTION in the PDOS assembler and linker manuals). The initialization code then clears this space on start-up.

```
int a,b[10];            /* these are cleared */
main()
{
        static c,d[10]; /* so are these */
        int e,f[10];    /* but these are not */
```

1.8.3 Self-Relocation

In order for the program to run at any address space, the initialization code compares the current address space to the address space provided at link time. If they do not match, code provided by QLINK allows the program to modify itself to adjust to the actual address space. See the BITMAP command for QLINK in the PDOS Reference Manual.

1.8.4 Task Global Variables

The last function of the initialization module is to define the following task global variables:

_eomem : A pointer to the beginning of un-allocated memory. 'sbrk' and the memory allocation/deallocation routines use the space between the end of memory and the bottom of the stack for dynamic memory.

_fsptr : File-slot pointer. This points to the file-slot buffer. Some information on a file can only be obtained directly from the file slot. The format of the file slot buffer is described in FILESLOT:H. (Note: This pointer is no longer required and may not be supported in future versions of PDOS C).

stdin,
stdout,
stderr : Standard I/O stream pointers. Rather than have these pointers be pre-initialized array references, PDOS C lets them be pointers to streams. These are initialized to all point to the stream _strm0.

_strm0,
_strml : Head and tail of stream list. PDOS has a different way of handling open files than UNIX. The usual array of streams is inefficient in PDOS. PDOS C links all open streams together, with _strm0 at the head of the list, and then as streams are closed, unlinks them from the open list and links them to a list with _strml at the head of the list.

_tcbptr,
_syram : Task Control Block and SYRAM table pointers. Two system tables which can be very useful for system programming are the Task Control Block and the SYRAM table. The pointers are saved during initialization and can be referenced by including their definition in "TCB:H" and "SYRAM:H." (Note: the XGML function call also returns the value of thse pointers.)

_allocp : Memory allocation list head. The dynamic memory allocation routines malloc(), mfree(), and so forth maintain a linked list of blocks of memory so that they may be allocated and deallocated over and over. The head of this list is _allocp. User programs do not normally need to make direct reference to this variable.

1.9 C TO ASSEMBLY LANGUAGE INTERFACE

Although assembly language is sometimes considered a "dead language," there are some occasions when it is the best tool for the job. When the code is very long, it is best to create an assembly language subroutine and call it from C. Just be sure to preserve the appropriate registers and observe the parameter passing conventions detailed in the rest of this section. But, if the code is short, PDOS provides you a method of including it with in-line assembly.

1.9.1 In-Line Assembly Language

If you only have one or two instructions to perform, you can include them as in-line code with the "asm" pseudo-function. This statement looks like a function call with one literal string argument but it is converted by the compiler to insert the quoted string directly in-line with the rest of the C code.

Some of the calls into the PDOS interface are done more easily via this method than by calling functions from XLIB:LIB. If XCLS were implemented as a function, calling it would use a JSR instruction, an XCLS instruction, and an RTS instruction. The JSR would take 6 bytes of code for each call, plus the 4 bytes of code at the function definition. But, invoking XCLS via in-line assembly language takes up only two bytes of code and runs faster as well.

```
IN-LINE        VS     FUNCTION CALL

asm("xcls");          xcls();

A076:  XCLS           4EB9********:    JSR .XCLS
                                       .
                                       .
                                       .
                      A076:   .XCLS    XCLS
                      4E75:            RTS
```

Multiple lines can be specified in one call by embedding newlines in the string.

```
asm("move d1,d0\nxler");
```

The assembly language translator will convert lower-case text to upper case and move code out of column 1 to distinguish it from labels. If you need to put a label in your in-line assembly code, terminate it with a colon (:) and the compiler will not move it from column one.

```
asm("* comment");          * comment
asm(".xyz:");              .xyz
     .
     .
     .
asm("jmp .xyz");                     jmp .xyz
```

Remember that C always prefixes symbols with a period, so the entry-point "main" will appear as ".MAIN" in assembly language, and the variable "_tcbptr" will be "._TCBPTR".

1.9.2 Calling Assembly Language From C

The calling sequence from a PDOS C program to assembly language is as follows: 1) the C function arguments are pushed in right-to-left order onto the stack, 2) a jsr is done to the function name prefaced with a dot, and 3) upon return from the jsr, the number of bytes pushed onto the stack is added back to the stack pointer.

```
/*      C code   */

main(i)
int i;
{
   register char *j;
   char k;

   i = f(i,j,k);
}
```

Assembly code generated by compiler

```
.MAIN  LINK A6,#-2                   allocate for k
       MOVEM.L D7-D7/A5-A5,-(SP)     save j register plus extra
       MOVE.B -2(A6),DØ              get k
       EXT.W DØ                      convert to int
       MOVE DØ,(SP)                  move k to stack
       MOVE.L A5,-(SP)               move j to stack
       MOVE 8(A6),-(SP)              move i to stack
       JSR .F                        call function
       ADDQ.L #6,SP                  clean up stack
       MOVE DØ,8(A6)                 move return value to i
L1     TST.L (SP)+                   throw away extra
       MOVEM.L (SP)+,A5-A5           restore j register
       UNLK A6                       deallocate local variables
       RTS
```

Every C function reserves one 32-bit word on the stack. This allows it to make one-argument function calls without cleaning up the stack afterwards. Thus, the first argument passed to the function, k, is merely moved onto the stack rather than pushed.

The variable i is offset by 8 from the current frame pointer, A6. A6 points to the old frame pointer, which is followed by the return address.

The local variable k is offset by -2 from the current frame pointer. All local variables are referenced by negative offsets from the frame pointer.

The compiler allocates register A5 for register variable j. The old value of A5 is saved on the stack upon entry and restored upon exit.

```
SP---->| EXTRA |
       | SPACE |
       ---------
       |  OLD  |
       |  A5   |
       ---------
       | k     | -2(A6)
       ---------
A6---->|  OLD  |  (A6)
       |  A6   |
       ---------
       | RETURN|  4(A6)
       |ADDRESS|
       ---------
       | i     |  8(A6)
       ---------
       |       |
```

An assembly language routine should save and restore all of the registers it uses except for D0, D1, D2, A0, A1 and A2. The C compiler does not assume that these registers will be saved across function calls.

The function return value is always in D0 for functions that return an integer or a pointer. Functions that return a double-precision floating point number (compiled with the 'E' option) return a value in registers D0 and D1. Functions using the 68881 (compiled with the 'H' option) return floating point values in FP0.

1.9.3 Calling C Functions From Assembly

Calling C functions from assembly language requires that the assembly language routine push the arguments to the C function in the previous paragraph. In addition, C functions assume that registers D0, D1, D2, A0, A1 and A2 are scratch registers. In other words, C functions do not save and restore these registers. If an assembly language routine uses any of these registers, it should save and restore them over the C function call. A C function returns integer and pointer function values in register D0. See section 1.9.2 for floating point function values.

The only possible problems with calling C functions from assembly could occur if the C functions require access to the different task global variables, or call library functions that do. You may need to replicate some of the definition code in CSTART:ASM.

Also, while a C program is self-relocating, a C function linked in with assembly functions may not be. In that case, you may need to tell QLINK the specific address where your program will run, or link using the BITMAP instructions of QLINK.

1.10 BURNING C PROGRAMS IN ROM

The 'R' option of CC allows the creation of C programs to be burned into EPROM.

```
     >CC HELLO.R[CR]
```

The 'R' option calls a special program called ROMLINK that takes the binary image produced by QLINK and produces a PDOS object file with the necessary information for a subsequent QLINK to locate the RAM and ROM sections of a C program at specific addresses.

Because C generates code with absolute instructions, the result is a program that will only run at one specific address. Normally, you can get around this limitation with the BITMAP option of QLINK, which appends relocation information to a C task to enable it to relocate itself to any address space during initialization. This is not practical for a program burned into ROM, since self-modification is not possible. In that case, the location phase must be performed by the regular linker QLINK.

Currently, burning PDOS programs in EPROM involves using the program RUNGEN. This program asks you a number of questions about your requirements for the target system, and creates a procedure file that, when executed, will build a binary image of the code to be burned into EPROM.

It does this by using QLINK to link all the task images together to make a "superprogram" that is then output in whatever format is convenient. C programs have unique requirements because they have specific section references to a RAM section as well as a ROM section. RAM is defined in SECTION 1, while ROM is in SECTION 0.

This definition is done by the program ROMLINK, which converts the SYfile normally produced by QLINK to an object file with SECTION 0 and SECTION 1 references. This object file only needs the linker to resolve the base of the two sections. There are no REFs or DEFs in this object file. Thus, you need to tell RUNGEN that you are supplying it with an "assembly language" (or "language independent") module that needs SECTION 0 to be located in the ROM area and SECTION 1 to be located in the RAM area.

When you compile with the 'R' option, the end result is a file of type "OB" with the same name as your program and the extension "ROM." The following example produces an object file named "SIEVE:ROM." You may calculate the RAM and ROM requirements by inspecting the first line of the object file. The size of the code (or ROM) section is given in eight hexadecimal digits, prefixed by the code 'E0.' The size of the data (or RAM) section is given in eight hexadecimal digits, prefixed by the code 'E1.'

```
x>CC SIEVE,R

x>SF SIEVE:ROM                          code      data
                                        size      size
ØCCTEMPØ:O   5  ØØ9Ø9861105EØØØØØØ7CCE1ØØØØ213a21ØØØØ ...
                            .
                            .
                            .
```

1.11 PROGRAMMING CONVENTIONS

Following are miscellaneous considerations to take note of when programming in C under PDOS.

1.11.1 Line Termination

Under the general conventions established by UNIX and MSDOS, a line terminator is a line-feed character (Hex 0A). Under PDOS, however, lines are terminated with a carriage return (Hex 0D). Under both operating systems, a line terminator is converted to a carriage return/line-feed combination when it is displayed to the terminal. To be consistent wtih other programs under PDOS, C programs use the carriage return character as a line terminator. Therefore, the following conventions hold:

```
    '\n' == 0x0D;          /* carriage return is a new-line */
    '\r' == 0x8D;          /* return without line-feed -- set high bit */
    '\l' == 0x0A;          /* line feed character under PDOS */
    '\t' == 0x09;          /* tab character */
    '\f' == 0x0C;          /* form feed */
    '\b' == 0x08;          /* back space */
```

In addition, the following was added for programming convenience:

```
    '\e' == 0x1B;          /* escape character */
```

1.11.2 Register Variables

Three address registers and five data registers are available for use as register variables. The address registers will be used for pointer variables. Declaring more register variables than are available does not cause an error; any register variables specified beyond the limits given above are simply defined as automatic variables (i.e. on the stack).

Access to register variables is much faster than access to automatic, static, or global variables. It also requires less code to specify a register variable than a non-register variable. It is a good idea to define heavily used variables as register variables. Registers are limited because they do not have an address. As such, the following is illegal:

```
register int x;
y=&x;
```

1.11.3 Variable Specifications

The following shows sizes of variables:

long	- 32 bits wide
pointer	- 32 bits wide
enum	- 16 bits wide
int	- 16 bits wide
short	- 16 bits wide
char	- 8 bits wide
bit field	- 1-32 bits wide

All non-pointer variables may be signed or unsigned.

1.11.4 Comments

The standard format for comments is to enclose them in "/*" and "*/". PDOS C also allows a one-line comment to be indicated by text beginning with a double slash -- "//". Text following the double slash to the end of the line is a comment.

1.12 ACCESSING MEMORY DIRECTLY

It is possible to do direct memory accessing from C, usually to memory-mapped I/O registers at a particular address. This type of code is machine-dependent and should be isolated to a few small modules if portability is desired.

Should you desire to read/write 16 bits at a time to memory address 0xFF100200, you could define a pointer as follows:

```
int *p;
int i,j;

p = 0xFF100200;

i = *p;                 /* read 16 bits from the address */
*p = j;                 /* write 16 bits to the address */
```

If you will be making many references, you may want to declare the pointer to be a register variable to give quicker access and require less code.

For only one or two references to the address, you can simply declare the code in-line as follows:

```
        i = *(int *) 0xFF100200;         /* read */
        *(int *) 0xFF100200 = j;         /* write */
```

If you need to read/write a single byte, or 32 bits, you would declare the pointer above as follows:

```
        char *p;                         /* one byte */
        long *p;                         /* 32 bits */
```

or the in-line code:

```
        c = *(char *) 0xFF100200;
        c = *(long *) 0xFF100200;
```

CHAPTER TWO
STANDARD I/O LIBRARY ROUTINES

This chapter lists and describes the functions in the standard I/O library (STDLIB:LIB). These functions are normally used with the default link of the CC program file. Specifying 'F,' 'E,' or 'H' as an option to CC tells the linker to search the appropriate floating point library before it searches STDLIB:LIB. In that case, a different version of the formatted I/O routines (printf, scanf, etc.) will be used.

In this chapter, each function call is described in terms of its name, calling sequence and parameters, and function. A limited example is given when practical. Under the heading "NOTES" any known limitations and restrictions are listed along with some practical advice and complaints.

ALLOCATE MEMORY ROUTINES

alloc --	allocate memory (another name for malloc)
malloc --	allocate memory
calloc --	allocate memory and clear it
realloc --	change size of block of memory

Format:
```
#include <stdio.h>
char *alloc(size)
    unsigned int size;
char *malloc(size)
     unsigned int size;
char *calloc(count,size)
     unsigned int count,size;
char *realloc(ptr,size)
     char *ptr;
     unsigned int size;
```

Description:
These routines all provide dynamic storage allocation functions to the user. 'malloc' returns a pointer to a block of memory of size 'size'. 'alloc' is another name for 'malloc'. 'calloc' returns a pointer to a block of memory large enough to hold an array of 'count' elements, each of size 'size'. 'realloc' takes a previously allocated block of memory and shrinks or enlarges it to be 'size'.

The memory returned by 'malloc' and 'alloc' does not have any particular initial value; it must be assumed to contain garbage. 'calloc' clears the memory it returns, while 'realloc' copies the contents of the old memory into the new memory.

Typically, the caller casts the result pointer from these routines to the appropriate pointer type. The size specification can often be computed conveniently using the 'sizeof' operator.

If the routines cannot allocate the memory, they return a null pointer (NULL).

```
unsigned count;
long *longarray;
char *calloc(),*realloc(),*malloc();
...
count = 40;
longarray = (long *) calloc(count,sizeof(long));
...

count += 20;   /* need to enlarge the array */
longarray = (long *) realloc(longarray,count*sizeof(long));

struct ABC {
    char a[20];
    int b;
    long c;
};
...
struct ABC *p;
p = (struct ABC *) malloc(sizeof(struct ABC));
...
free(p);
```

Notes:
A frequent error is to forget to declare that these routines return pointers. If a routine is not declared, C assumes that it returns an int (16 bits). The compiler hints that something may be wrong by warning that you are assigning a short to a pointer. Remember to declare the routine properly before using it.

ASCII TO INTEGER CONVERSION ROUTINES

atoi -- ASCII to int conversion
atol -- ASCII to long conversion
itoa -- int to ASCII conversion
ltoa -- long to ASCII conversion

Format:

```
int atoi(string)
     char *string;
long atol(string)
     char *string;
itoa(num, string, width)
     int num, width;
     char *string;
ltoa(num, string, width)
     long num;
     char *string;
     int width;
```

Description:
These routines perform various ASCII - integer conversions. It is useful to use these routines when the overhead of the printf function is not needed.

'atoi' converts its string argument to an int and returns the value. Leading whitespace is ignored.

'atol' converts its string argument to a long and returns the value. Leading whitespace is ignored.

'itoa' converts its numeric argument to an ASCII string of the specified width. The number is right justified in the field and padded on the left with blanks.

'ltoa' performs the same conversion as 'itoa', but the numeric argument is a long.

See also the PDOS interface functions XCBD, XCBH, XCBX, XCHX, and XCDB.

```
x=atoi("1234");
long int y,atol();
y=atol("123456");
char buf[20];
ltoa(x,buf,6);
ltoa(y,buf,10);
```

NOTES
The conversion routines stop on any non-numeric character and do not report errors on input.

CLOSE

Close a File

Format:
```
int close(fd);
int fd;
```

Description:
This routine closes a file opened by 'open', 'creat', XSOP, etc. It returns zero if there was no error; otherwise, it returns -1.

```
   if (close(fd))
        printf("error closing file");
```

COPY

Copy One File Into Another

Format:
```
int copy(source,dest)
     char *source, *dest;
```

Description:
'copy' copies a file from the source file to the destination file. If the destination file does not exist, it is created. The destination file is set to the same attributes as the source file. 'copy' works similarly to the PDOS primitive XCPY, but since it uses all available memory as a buffer, it runs faster.

The function value is zero if the copy occurred without error. An open error on the input file causes an error return of 1. An open error on the output file causes an error return of 2. Any error while writing to the output file causes an error return of 3.

```
    if (err = copy("OLDFILE:DAT","OLDFILE:BAK"))
        printf("\nError on copy file: %d",err);
```

Notes:
'copy' uses the memory between the end of memory and the bottom of the stack. When it returns, that memory may be used for anything else, but be aware that it may interact with non-standard use of 'sbrk'.

CREAT

Define a File and Open It

Format:

```
int creat(filename,mode);
char *filename;
int mode;
```

Description:
If a file by the given name exists on the disk, this call opens it for sequential output. Otherwise, it defines the file and opens it. It returns a -1 on error or it returns the file id (file slot).

```
   if ((fd = creat(filename,0)) < 0)
       printf("error opening file %s",filename);
```

Notes:
The 'mode' is currently only for UNIX compatibility, and is ignored by 'creat'. See also XDFL in XLIB. It is an error to open a driver file (like 'TTA') with this call, since 'creat' will attempt to set the End of File (EOF) pointer to the beginning of the file, resulting in a PDOS error 80, 'Driver Error.'

CTYPE

Character Class and Conversion Macros

Format:

```
#include <ctype.h>
isspace(c), isupper(c), islower(c), isalpha(c), isdigit(c),
ishex(c), isascii(c), isalnum(c), iscntrl(c), isprint(c),
ispunct(c), toupper(c), tolower(c), toascii(c)
```

These routines are all macro definitions which take a single character argument and return an integer.

Description:

isspace(c):	true if c is a space, tab, carriage return, newline, or a formfeed character.
isupper(c):	true if c is an upper case letter.
islower(c):	true if c is a lower case letter.
isalpha(c):	true if c is an alphabetic character.
isdigit(c):	true if c is is a decimal character.
ishex(c):	true if c is a hexadecimal character.
isascii(c):	true if c is in the ASCII character range.
isalnum(c):	true if c is an alphanumeric character.
iscntrl(c):	true if c is an ASCII control character.
isprint(c):	true if c is a printable character.
ispunct(c):	true if c is one of the following: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | }
toupper(c):	returns the specified character's upper case equivalent.
tolower(c):	returns the specified character's lower case equivalent.
toascii(c):	all non-ASCII bits will be masked and the resulting character returned.

```
    if (isalnum(c))
         putchar(c);
    else                    /* non-printable character */
         printf("<%d>",c);
```

Notes:

Beware of side effects when using these macros; some of them may reference their argument twice. If the argument is a function call like 'getc' then the function may be called twice instead of once per macro call.

_ERROR

Print Error Message and Exit

Format:

```
_error(format,arg1,arg2,...,argn);
char *format;
arguments can be of various types.
```

Description:

'_error' is a version of 'printf' that displays a message through 'stderr' and exits to the monitor with an error code of 1. For information on format items, see 'printf'.

```
main(argc,argv)
int argc;
char *argv[];
{
     if (argc < 2)
          _error("\n%s: must have an argument.",argv[0]);
     ...
}

x>TEST[CR]
TEST: must have an argument.
Exit Status 1
x>_
```

EXIT

Close Streams and Exit to Monitor

Format:
int exit(status);

Description:
This routine closes all files opened with 'fopen', and calls '_exit' with status. A program which executes a return at the top level or which drops out the end of the main section will also go through 'exit' but without the option of returning a status. In that case, the status is assumed to be the Last Error Number (LEN$) that was saved in the Task Control Block. The LEN$ is updated after most PDOS calls, and is set directly by the XLER call.

```
        exit(0);
        exit(1);
```

Notes:
If you open files through any method other than 'fopen', it is worthwhile to define a function called 'exit' in your own task that closes all files and performs any other necessary cleanup before calling '_exit'. The 'putc' routine calls 'exit' on detecting a [CTRL-C] from the console. Defining your own 'exit' allows you to trap the error and handle it appropriately.

_EXIT

Exit to Monitor With Status

Format:
_exit(status);

Description:
Exit to the PDOS monitor, reporting any errors. If the status is non-zero, a message "EXIT STATUS n" is printed on the console. If the Last Error Number (LEN$) was also non-zero, the PDOS error number and message is printed. This function is called by 'exit' after the files are closed. It can be called by the user if it is necessary to exit without closing files.

```
        _exit(0);
        _exit(1);
```

FCLOSE

Close a Stream

Format:

```
#include <stdio.h>
int fclose(stream);
FILE *stream;
```

Description:
'fclose' closes the stream opened by 'fopen' or 'ttyopen'. Buffers are flushed to the disk and the file block is deallocated. Returns EOF on error; otherwise it returns NULL.

```
        FILE *f,*fopen();
        f = fopen("temp","w");
        fprintf(f,"testing\n");
        fclose(f);
```

FEOF

Check for End of File in a Stream

Format:

```
#include <stdio.h>

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

Description:
'feof' returns a non-zero value if the stream is at the end of the file; otherwise, a zero value is returned.

```
while(!feof(f1))            /* copy a file */
       putc(getc(f1),f2);
```

Notes:
'feof' is implemented as a macro. The current implementation does not correspond with the ANSI standard because the EOF is true when there is no longer any data to read, not when EOF has been read.

FGETS

Get a Line From a Stream

Format:
```
#include <stdio.h>
char *fgets(buff, size, stream)
char *buff;
int size;
FILE *stream;
```

Description:
'fgets' reads characters from the stream into the buffer until it sees a newline, reaches end-of-file, or reads size-1 characters. If the read terminates with a newline, the newline is stored in the string. A null is placed after the last character read. If no characters were read (because an input error or EOF occurred immediately) the buffer is returned unmodified and the function returns a null pointer (NULL). If data is read into the buffer, the function returns a pointer to the buffer.

See also XRLF, XGLU, and XGLB for PDOS functions that perform similarly.

```
#include <stdio.h>
...
FILE *fd,*fopen;
char *fgets(),buffer[132];
fd = fopen("MYFILE:TXT","r");
if (fgets(buffer,132,fd) == NULL){
    printf("\nEnd of file");
    fclose(fd);
}
else
    printf("%s",buffer);
```

Notes:
'fgets' keeps the newline when it is read in, while 'gets' does not. The inconsistency is confusing, but standard.

FFLUSH

Flush Output to File

Format:

```
#include <stdio.h>

fflush(fbuf)
FILE *fbuf;
```

Description:
'fflush' forces all pending I/O on a stream to be sent out. It is primarily intended as an internal-use call, called by 'fclose,' 'fseek,' 'fputc,' and 'exit.'

```
FILE *f1;
f1 = fopen("MYFILE:DAT","w");
fprintf(f1,"\nThis is a test");
fflush(f1);
```

Notes:
'fflush' is not normally called directly, since the buffers are automatically flushed as necessary.

FOPEN

Open a Stream

Format:

```
#include <stdio.h>
FILE *fopen(fname,mode)
char *fname;
char *mode;
```

Description:
'fopen' opens the specified file according to the specified mode and associates a stream with it. The file modes are:

"r"	open for reading
"w"	create for writing (truncate if file already exists)
"a"	open for writing at end of file, or if non-existent, create
"r+"	open for update
"w+"	create for update
"a+"	open for update, create if necessary, position to end of file

In the last three cases, "update" means that the file is opened for random access.

'fopen' allocates a stream buffer and returns a pointer to it on success; if the file cannot be opened it returns NULL.

```
        FILE *f,*fopen();
        f = fopen("temp","w");
        fprintf(f,"testing\n");
        fclose(f);
```

Notes:
Mode "a" and mode "a+" are currently the same. To do otherwise would involve an immense amount of overhead on mode "a."

Don't open driver files (like 'TTA') in mode "w." In this mode, 'fopen' attempts to reposition the end of file (EOF) pointer to the beginning of the file and the driver returns a PDOS error 80 (driver error). Open driver files in mode "r+."

FORMATTED INPUT ROUTINES

fscanf -- formatted input from file
scanf -- formatted input from standard input
sscanf -- formatted conversion from buffer

Format:

```
#include <stdio.h>
int scanf(format, ptr1, ptr2, ...)
    char *format;
int fscanf(fid,format, ptr1, ptr2, ...)
    FILE *fid;
    char *format;
int sscanf(buffer,format, ptr1, ptr2, ...)
    char *buffer;
    char *format;
```

Description:
'scanf' reads from the standard input and returns data converted according to the format description.

'fscanf' reads from the specified stream and returns data converted according to the format description.

'sscanf' reads from the specified buffer and returns data converted according to the format description.

A non-negative number as a return status for these routines indicates the number of fields matched. A -1 indicates that no fields matched; a -2 indicates the end of file; a -3 indicates an input error.

The format string may contain spaces, tabs and newlines which are ignored, percent signs '%', and conversion characters. The conversion characters must be preceded by a percent sign. An optional maximum field width may be inserted between the two, or an asterisk '*' specifying that this item is ignored. No other characters are allowed.

```
       FORMAT STATEMENT FORMAT
'%'   'l' <max size>  's'
      '*'             'c'
                      'd'
                      'x'
                      'o'
                      'f'
                      'e'
                      '['
```

The conversion characters and their expected storage types are as follows:

'c' The next character is stored through a character pointer. It will accept blanks, tabs or carriage returns as legitimate input. If you need the first character which is not a blank, tab or carriage return specify a format of "%1s" (percent,one,s) instead and pass a pointer to a buffer.

's' The next input is a character string. It should be stored through a pointer to a character buffer which has been allocated sufficient space for the expected string.

'd' The next input is a decimal number. It should be stored through a pointer to an integer.

'o' The next input is an octal number. It should be stored through a pointer to an integer.

'x' The next input is a hexadecimal number. It should be stored through a pointer to an integer.

Note: 'f' and 'e' formats are not implemented in STDLIB. Use 'E,' 'F,' or 'H' switch when compiling or linking.

```
x>CC TEST,F[CR]
```

'f' The next input is a floating point number. It should be stored through a pointer to a float. The input may be in the format:

```
<leading digits>.<trailing digits>

or:

<digit>.<digits>e<exponent>
```

'e' Same as 'f'.

VALID 'e' or 'f' INPUT:

```
123.456
1.234560e02
-123.456e-02
```

'[' The next input is a string not delimited by space characters. The left square bracket is followed by a set of characters and a right square bracket. The set of characters in the brackets define the characters making up the string. If the first character after the left bracket is the up-arrow (^) the set of characters is all characters NOT within the brackets. Data is stored in the corresponding argument (a string array) until a character not in the set is found.

Format descriptions 'd', 'o', and 'x' must be preceded by an 'l' (el) character if the corresponding argument is a pointer to a long (32-bit) integer rather than an int (16-bit).

Double precision floating point numbers (compiler option 'E' or 'H') also require that the format description be preceded by an 'l' (el) character.

```
int i;
float f;
char name[32];
char *inptr;
...
xglu(&inptr);   /* read a line from the keyboard */
sscanf(inptr,"%d%f%s",&i,&f,name);
...
```

If the input is '123 45.678E-1 Kilroy' then 'i' will get the value '123', 'f' will receive the value '45.678E-1', and 'name' will receive the string 'Kilroy'.

Notes:
Since terminal I/O in PDOS C is currently unbuffered, the first rubout will terminate an input field, making it awkward to use 'scanf' on the keyboard. (You can't delete your mistakes). It is better to use XGLB or XGLU to read a line from the keyboard and then use 'sscanf' to parse that line.

A common programming error is to forget that 'scanf' requires the address of the return parameters.

```
int i;
scanf("%d",i);      /* this won't work */
scanf("%d",&i);     /* this will work */
```

FORMATTED PRINT ROUTINES

fprintf -- formatted print to stream
printf -- formatted print to standard output
sprintf -- formatted print to buffer

Format:

```
#include <stdio.h>
fprintf(stream,format,arg1,arg2,...);
FILE *stream;
char *format;
Arguments can be of various types.

printf(format,arg1,arg2,...);
char *format;
Arguments can be of various types.

sprintf(buffer,format,arg1,arg2,...);
char *buffer;
char *format;
Arguments can be of various types.
```

Description:
The description below applies equally to all three function calls, except that 'printf' directs its output to 'stdout', 'fprintf' directs its output to the indicated stream, and 'sprintf' outputs to the indicated buffer.

There are actually four sets of formatting routines -- one in the standard library and one in each floating point library. The one in the standard library does not handle floating point, so the %e,%f formats are not allowed. A considerable size penalty is exacted by bringing in the floating point routines, so if you don't use floating point, it is a good idea to use the simpler version of the 'printf' routines.

There are two types of items in the format string: 1) characters which are copied literally and 2) format statements which work on strings, characters and numerics. A format statement begins with a percent sign '%' and ends with one of the conversion characters. There may optionally be additional specification characters between the percent sign and the conversion letters.

'printf' expects numbers ('d','x','u','o' format) to be sixteen-bit words unless the format statement includes the lower case character el ('l').

```
        FORMAT STATEMENT FORMAT
'%'  'l'  '-'  <min size>  '.'<max size>  's'
          'Ø'                              'c'
                                           'd'
                                           'x'
                                           'u'
                                           'o'
                                           'f'
                                           'e'
```

By default, a number or string only takes as much space as is necessary to print it. If a minimum field width is specified, the number or string is printed right-justified in that field, padded with blanks if necessary. It is padded with zeroes if a '0' precedes the minimum field width. If a minus sign '-' precedes the minimum field width, the number is right justified.

A maximum field width may also be specified, with or without a minimum field width. A period '.' must precede the maximum field width. If a floating point number (conversion characters 'e' or 'f') is specified, the number following the period specifies the number of digits after the decimal point to print, rather than a maximum field width.

The conversion characters are as follows:

'c'	Print a single character.	printf("%c",65);	A
's'	Print a string.	printf("%s","testing");	testing
'd'	Print a decimal number.	printf("%d",1234);	1234
'x'	Print a hexadecimal number.	printf("%x",Øxabcd);	ABCD
'u'	Print an unsigned decimal number.	printf("%u",-1);	65535
'o'	Print an octal number.	printf("%o",64);	1ØØ

THE FOLLOWING FORMATS ARE ONLY VALID IN THE FLOATING POINT VERSIONS:

'f'	Print a floating point number. The result is in floating point notation. <leading digits>.<trailing digits> If no second width field has been specified there will be six digits to the right of the decimal point.	printf("%f",123.456);	123.455989
'e'	Print a floating point number. The result is in scientific or engineering notation. <digit>.<digits>e<exponent>	printf("%e",123.456);	1.23456Øe02

Format description characters 'd', 'o', 'x', and 'u' must be preceded by an 'l' (el) character if the argument is a long integer (32 bits) rather than a word (16 bits).

Double precision floating point numbers (compiler option 'E' or 'H') also require that the format description be preceded by an 'l' character.

Any other character following a percent sign is taken as a literal and will itself be printed. In this way you can print a percent sign. See also 'scanf'.

Notes:
Remember to search the floating point library before the standard library if you need to use the floating point, or octal conversions.

Some versions of 'printf' allow '%X' to indicate hexadecimal output with the letters A-F in upper case and '%x' to indicate hexadecimal output with the letters a-f in lower case. This convention applies to PDOS C if the 'F,' 'E,' or 'H' flag is set. The version of 'printf' from STDLIB recognizes only the lower case '%x' and puts out upper case A-F.

FPUTS

Output String to Stream

Format:

```
#include <stdio.h>

fputs(s, stream)
char *s;
FILE *stream;
```

Description:
'fputs' outputs the null-terminated string to the specified stream. The null is not transferred.

```
        FILE *fs,*fopen();
        fs = fopen("MYFILE:TX","w");
        fputs("hello, world!\n",fs);
```

Notes:
'fputs' is not implemented as a macro, but rather as a function. This may change in the future.

FREAD

Read From Stream

Format:

```
int fread(iarray,isize,icount,stream)
char *iarray;
int isize,icount;
FILE *stream;
```

Description:
'fread' reads items into 'iarray' from the input stream. The item's size is specified by 'isize.' 'fread' reads until 'icount' items are read or until EOF is encountered.

See also 'fopen,' 'fclose,' 'fwrite,' 'getc,' and 'putc.'

```
 #include <stdio.h>

 FILE *f1;
 long data_table[20];

 f1 = fopen("datfile:dat","r");
 fread(data_table,sizeof(long),20,f1);
```

Notes:
Attempting to read from a closed file may cause it to read from the standard input rather than returning an error as expected.

FREE, MFREE

Deallocate Previously Allocated Memory

Format:

```
free(ptr)
   char *ptr;
mfree(ptr)
   char *ptr;
```

Description:
'free' or 'mfree' will deallocate a block of memory previously allocated by the 'alloc/malloc/calloc/realloc' routines. The block of memory goes into a pool of similarly available blocks of memory where it may be allocated again as necessary. The argument to 'free/mfree' must be a pointer previously returned by 'alloc/malloc/calloc/realloc' and not already given to 'free/mfree.'

See the example for 'alloc/malloc/calloc/realloc.'

FSEEK

Reposition a Stream

Format:

```
#include <stdio.h>

int fseek(stream,offset,origin)
FILE *stream;
long offset;
int origin;
```

Description:

'fseek' repositions the location of the file pointer to the location 'offset' bytes from the beginning, current location or end of the file as specified by an origin of 0, 1, or 2 respectively.

Returns EOF if the seek is unsuccessful or if the stream has not been opened; otherwise it returns 0.

See also 'lseek' and XRFP.

```
        FILE *fs,*fopen()
        fs = fopen("MYFILE:SR","r+");
        fseek(fs,10L,0);
        fputs(fs,"this goes 10 bytes from the start");
        fseek(fs,100L,2);
        fputs(fs,"this goes 100 bytes from the end");
        fseek(fs,-200L,1);
        fputs(fs,"this goes 200 bytes before that");
```

Notes:

If the stream has been used last for output, 'fseek' flushes the buffer before performing the seek.

'fseek' and 'lseek' are quite similar, with the exception of the return values of the two functions. 'fseek' returns a status flag indicating success or failure, while 'lseek' returns the current byte position sought.

You cannot position beyond end of file. To extend a file, position to end of file and write data.

FTELL

File Pointer Relative Offset

Format:

```
#include <stdio.h>

long ftell(stream)
FILE *stream;
```

Description:
'ftell' returns the file pointer's current byte offset from the beginning of the stream.

It returns EOF on error, or if the specified stream has not been opened for file I/O.

```
        long here,ftell();
        FILE *fs,*fopen();
        fs = fopen("MYFILE:TXT","r+");
        fputs(fs,"Remember where this goes");
        here = ftell(fs);
                .
                .
                .
        fseek(fs,here,Ø);        /* reposition to where we were */
```

FWRITE

Write to Stream

Format:
```
int fwrite(iarray,isize,icount,stream)
char *iarray;
int isize,icount;
FILE *stream;
```

Description:
'fwrite' writes 'icount' items from 'iarray' to the output stream. The size of each item is specified by 'isize.'

See also 'fopen,' 'fclose,' 'fread,' 'getc,' and 'putc.'

```
#include <stdio.h>

FILE *f1;
long data_table[20];

f1 = fopen ("datfile:dat","w");
fwrite(data_table,sizeof(long),20,f1);
```

GETC

Get a Character From a Stream

Format:
```
#include <stdio.h>
int getc(fd);
FILE *fd;
```

Description:
'getc' gets a single character from a file. It returns a character or EOF.

It echoes on input if *fd -- _strm0 and fd->fslot -- 0.

To avoid echo, open your console with 'ttyopen' and perform the input from that stream.

```
#include <stdio.h>
FILE *fd,*fopen(),*ttyopen();
int c;
fd = fopen("DATA:FIL","r");
while ((c = getc(fd)))
   putchar(c);

FILE *p;
char c;
p = ttyopen(0);
printf("\nEnter a character.  It should echo:");
c = getc(stdin);
printf("\nYou typed %c.  Now enter a character.  It won't echo:",c);
c = getc(p);
printf("\nYou typed %c",c);
```

Notes:
Versions of PDOS C prior to 5.0 implemented 'getc' as a function instead of a macro. 'getc,' now a macro, buffers file I/O. Terminal I/O is still unbuffered. The macro is defined in "stdio.h."

GETCHAR

Get a Character From Standard Input

Format:

```
#include <stdio.h>
int getchar();
```

Description:
'getchar' reads a character from the stdin stream and returns it. This is implemented as a macro in stdio.h as follows:

```
#define getchar() getc(stdin)


        c = getchar();
```

Notes:
Versions of PDOS C prior to 5.0 implemented 'getchar' as a function instead of as a macro. 'getchar,' now a macro, buffers file I/O. Terminal I/O is still unbuffered. The macro is defined in "stdio.h." You must include "stdio.h" to use 'getchar'.

GETS

Read Line From Standard Input

Format:

```
#include <stdio.h>

char *gets(s)
char *s;
```

Description:
'gets' reads a string of characters from the 'stdin' into the buffer that is passed in. It returns a pointer to the data if it succeeds; if an end of file is found before data is read, then NULL is returned. The newline at the end of the line is deleted and replaced with a null.

```
        char buffer[132],*gets();
        if (gets(buffer) != NULL){
            .
            .   /* process the line of data */
            .
        }
```

Notes:
'gets' is not consistent with 'fgets' in the handling of the newline at the end.

GLOB

Expand Ambiguous Filename to List of Filenames

Format:

```
glob(argcptr, argvptr, filespec)
    int *argcptr;
    char ***argvptr;
    char *filespec;
```

Description:
This routine takes an ambiguous filename specification (one which may have wild cards in it) and expands it to a list of the files on the disk that match that specification. If no disk unit is given, the current SY unit is taken as a default. If no level is specified, the current level is taken as a default. It returns a count of the matching names in the first parameter, and a pointer to the array of strings in the second parameter.

```
        int rargc;
        char **rargv;
        glob(&rargc,&rargv,"@:C;@/2");
        for (i=0;i<rargc;i++)
            printf("\n%s",rargv[i]);
```

Notes:
The storage for the filenames is allocated from the memory between the end of memory (_eomem) and the bottom of the stack. This memory is not deallocated, so that the strings are not over-written. Repeated calls to 'glob' will eventually use up all the available memory. Programs needing to make repeated calls to 'glob' should preserve the value of '_eomem' before the call and restore it after all the strings have been seen.

Don't perform any other dynamic memory allocation after saving the value of '_eomem' or the restoration of '_eomem' could disrupt that activity as well.

The description/execution of this function is somewhat awkward. It conforms to no known standard.

INDEX

Return Position of Character in String

Format:
```
int index(str,c)
char *str,c;
```

Description:
'index' searches for the character 'c' in the string 'str'. If the character is found, the position in the string is returned. -1 is returned if the character is not in the string. Compare with 'rindex'. See also 'strchr' in STRINGS.

```
printf("char %c is at location %d in string %s",c,index(str,c),str);
```

Notes:
UNIX defines one version for System III and another for Version 7. This routine is compatible with Version 7.

LSEEK

Position File Slot

Format:

```
long lseek(fd,offset,flag)
int fd;
long offset;
int flag;
```

Description:
The 'lseek' system call allows the user to position the file pointer for an open file. The file pointer references the place in the file where the next 'read' or 'write' is performed.

If the 'flag' is 0, the 'offset' is taken to be absolute, or relative to the beginning of the file.

If the 'flag' is 1, the 'offset' is taken to be relative to the current position in the file.

If the 'flag' is 2, the 'offset' is taken to be relative to the end of file.

The value returned by the function is the current position within the file, or -1 on error.

See also 'fseek' and XRFP.

```
int fs;
fs = open("MYFILE:SR",2);
lseek(fs,10L,0);
puts(fs,"this goes 10 bytes from the start");
lseek(fs,100L,2);
puts(fs,"this goes 100 bytes from the end");
lseek(fs,-200L,1);
puts(fs,"this goes 200 bytes before that");
```

Notes:
You cannot position beyond end of file. To extend a file, position to end of file and write data.

LONG DIVISION ROUTINES

ldiv -- long division
ldivr -- long remainder
ldivu -- unsigned long division

Format:

```
long ldiv(num1,num2)
long num1,num2;

extern long ldivr;

unsigned long ldivu(num1,num2)
unsigned long num1,num2;
```

Description:
'ldiv' computes the division of 'num1' by 'num2' and returns the quotient. The remainder from the division is stored in the global variable 'ldivr'.

'ldivu' performs an unsigned division on the two operands and returns the quotient.

These routines are generally not explicitly called by the programmer but are automatically generated by the compiler as needed.

```
printf("%ld",ldiv(2147393664,17394L));
```

LONG MULTIPLICATION ROUTINES

lmul -- long multiplication
lmulu -- unsigned long multiplication

Format:

```
long lmul(num1,num2)
long num1,num2;

unsigned long lmulu(num1,num2)
unsigned long num1,num2;
```

Description:
'lmul' computes the long multiplication on 'num1' and 'num2' and returns the product.

'lmulu' computes the unsigned product of the two numbers.

These routines are generally not explicitly called by the programmer but are automatically generated by the compiler as needed.

```
printf("%ld",lmul(17394L,123456));
```

Notes:
These routines don't check for 32-bit overflow.

LONG REMAINDER ROUTINES

lrem -- long remainder
lremu -- unsigned long remainder

Format:

```
long lrem(num1,num2)
long num1,num2;

unsigned long lremu(num1,num2)
unsigned long num1,num2;
```

Description:
'lrem' does long division on the 'num1','num2' pair and returns the remainder from 'num1' divided by 'num2'.

'lremu' does the operation using unsigned arithmetic.

These routines are generally not explicitly called by the programmer but are automatically generated by the compiler as needed.

See 'ldiv' and 'lmul'.

```
printf("%ld",lrem(7654321,123456));
```

MEMORY HANDLING ROUTINES

memccpy -- copy character with break
memchr -- locate character in memory
memcmp - compare memory
memcpy -- copy memory
memset -- set memory to character

Format:

```
#include <memory.h>

char *memccpy(dst,src,bc,cnt)
char *dst,src;
int bc,cnt;

char *memchr(str,ch,cnt)
char *str;
int ch,cnt;

int memcmp(m1,m2,cnt)
char *m,*m2;
int cnt;

char *memcpy(dst,src,cnt)
char *dst,*src;
int cnt;

char *memset(m,ch,cnt)
char *m;
int ch,cnt;
```

Description:

'memccpy' copies from 'src' to 'dst' 'cnt' bytes or until the break character ('bc') is copied. A pointer to the character following 'bc' in 'dst' is returned if 'bc' is encountered. If 'bc' is not encountered, a NULL is returned.

```
char buf[50]
memccpy(buf,"Hello world\n\0!",'!',20);
printf(buf);
Hello world
```

'memchr' returns a pointer to the first occurrence of 'ch' in a block of memory with size of 'cnt.' If the character is not found, a NULL is returned.

```
printf("The 9 is at addr $%lx\n",
       memchr("0123456789ABCDEF",'9',16));
The 9 is at addr $E08D
```

'memcmp' compares two blocks of memory for 'cnt' bytes to determine if they are equal. An integer value less than, equal to, or greater than 0 is returned if block 1 is less than, equal to, or greater than block 2.

```
if (!memcmp("Hello there world","Hello world",6))
        printf("Match\n");
else
        printf("No match\n");
Match
```

'memcpy' copies 'cnt' bytes from 'src' to 'dst.' The address of 'dst' is returned.

```
char buf[50];
memcpy(buf,"Hello world\nHow are you?\n",12);
buf[12] = '\0';
printf(buf);
Hello world
```

'memset' sets 'cnt' bytes in the memory block to a given character.

```
char buf[50];
memset(buf,'\0',50);
memset(buf,'\n',49);
memset(buf,'x',48);
printf(buf);
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Notes:
The include file 'memory.h' contains the definitions of these functions.

NON-LOCAL GOTO ROUTINES

```
setjmp --  save current environment for later restoration
longjmp -- restore previously saved environment (and jump)
```

Format:

```
#include <SETJMP:H>
int setjmp(env)
    jmp_buf env;
longjmp(env,ret)
    jmp_buf env;
    int ret;
```

Description:
'setjmp' saves the current execution environment (all important registers) in the jmp_buf passed as a parameter and returns with a value of 0 to the calling routine. At a later point in the program, another function can call 'longjmp' with the same jmp_buf and restore the execution environment to the state it was when 'setjmp' was first called. At this point, 'setjmp' will return with a non-zero value. The return value is specified as the second parameter of 'longjmp' with the condition that it may not be zero.

The routine that called 'setjmp' must <u>not</u> have returned before the 'longjmp' is executed.

These functions define an error trap that can be called when lower-level routines detect some sort of error and need to restore context to a known state.

This code is taken from the 'xeq' function in the standard library.

```
#include <TCB:H>
#include <SETJMP:H>
static jmp_buf env;
char *oldext,*olderr;
int extrap(),errtrap();
.
.
.
retval = setjmp(env);
if (retval == 0){          /* first time through */
   .
   .
   .
   oldext = tcbptr->_ext;      /* save old tcb exit vectors */
   olderr = tcbptr->_err;
   tcbptr->_ext = extrap;      /* load new tcb exit vectors */
   tcbptr->_err = errtrap;

/* call xldf, which only returns on an error; otherwise, it exits through */
the exit vectors */

   retval = xldf(1,lowptr,hiptr,filename, &lowload,&hiload);
   xler(retval);
}
tcbptr->_ext = oldext;  /* restore task exit vectors */
tcbptr->_err = olderr;
.
.
.
extrap()                   /* normal exit trap */
{
       longjmp(env,1);        /* return from setjmp with status of 1*/
}

errtrap()                  /* error exit trap */
{
        asm("move.l d1,d0\nxler");
        longjmp(env,2);        /* return from setjmp with status of 2*/
}
```

Notes:
These routines are complicated and awkward. Be sure you understand them before using them.

OPEN

Open a File

Format:

```
int open(filename,mode);
int mode;
char *filename;
```

Description:
'open' associates a file on the disk with a file id for further access. The mode determines the kind of access:

```
0 - Read Only (XROO)
1 - Sequential Access (XSOP)
2 - Read/Write Random (XROP)
3 - Shared Random (XNOP)
```

'open' returns a valid PDOS file slot or an EOF on error.

See also XROO, XSOP, XROP, and XNOP.

```
        fd = open("MYFILE:SR",0);
```

Notes:
The mode might not be exactly compatible with UNIX programs. Check carefully any programs using this call.

PUTC

Output a Character to a Stream

Format:
```
#include <stdio.h>

putc(c,stream)
char c;
FILE *stream;
```

Description:
'putc' outputs the specified character to the output stream.

The 'putc' routine checks for a break character from the keyboard after outputting a newline. That way, you can abort runaway programs. On detection of a [CTRL-C], the program will call 'exit' with a status of -1. You may define your own exit routine to trap this call and close all your files before calling '_exit' to return to the monitor.

'putc' currently aborts on detection of a file error.

```
#include <stdio.h>

FILE *fs,*fopen();
fs = fopen("MYFILE:SR","w");
putc('\n',fs);
```

Notes:
The 'putc' macro simply calls the 'fputc' function. Versions of C prior to 5.0 made an operating system call for each character. The latest versions of 'putc' and 'fputc' buffer output to files for greater efficiency.

PUTCHAR

Output a Character

Format:

```
#include <stdio.h>
putchar(c);
char c;
```

Description:
'putchar' performs a 'putc' on the 'stdout' file variable. This function is defined as a macro in <stdio.h> as follows:

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

See 'putc'.

```
 c = "\nHello, world!";
 while (*c)
     putchar(*c++);
```

Notes:
You must include "stdio.h" to use 'putchar'.

PUTS

Output a String to the Standard Output

Format:

```
#include <stdio.h>

puts(s)
char *s;
```

Description:
'puts' outputs the null terminated string 's' to the standard output. A carriage return is appended to the string, and the null is not output.

```
#include <stdio.h>
puts("\nHello, world!");
```

Notes:
While this call is implemented as a function, it could probably be implemented as a macro.

'puts' appends a newline to the string but 'fputs' does not.

RANDOM NUMBER GENERATION

rand -- return random number
srand -- seed random number generator

Format:

```
int rand()

int srand(seed)
int seed;
```

Description:
'rand' returns a random signed value of type int.

'srand' may be used to set the seed. If the seed is not set, 1 is used.

```
int old,rnum;

old = srand(33);          /* seed generator and save old seed */
rnum = rand();            /* get a new random number */
```

READ

Read From a File

Format:
```
int read(fd,buffer,bytes);
int fd,bytes;
char *buffer;
```

Description:
'read' reads in data from a file or, if 'fd' is less than or equal to 0, from a port. 'bytes' is the count of data to be read. If the read is terminated by an end of file, the count of bytes actually read is returned as the function value. A value of -1 (EOF) is returned on an error. An end of file is detected when the count read is less than the bytes requested -- usually zero. When the input comes from a port, an [ESC] or [CTRL-C] terminates the input.

If 'fd' equals 0, the data comes from the current input port for the task. If 'fd' is less than 0, the number is negated and used for the port number.

```
char buffer[100];
int fd,count;
fd = open("MYFILE",0);
count = read(fd,buffer,25);
```

RENAME

Rename File

Format:

```
int rename(old,new)
char *old,*new;
```

Description:
'rename' changes the name of a file from old to new. If a file with the new name already exists, it is deleted first. Both old and new files must be on the same logical disk unit.

A return value of 0 indicates success; otherwise, a PDOS error code is returned.

See also XRNF.

```
 rename("object:old","object:new");
```

REWIND

Position to Top of Stream

Format:
```
int rewind(stream)
FILE *stream;
```

Description:
'rewind' repositions the location of the file pointer to the beginning of the file. 'rewind' is implemented as a macro calling 'fseek.' For diagnostics, see 'fseek.'

```
#include <stdio.h>

FILE f1;

f1 = fopen("mydat","r");
c = getc(f1);
. . .
rewind(f1);
```

RINDEX

String Position

Format:

```
int rindex(s,ch)
char *s,ch;
```

Description:
'rindex' searches for the character 'ch' in the string 's'. If the character exists, the position of the last occurrence of the character in the string is returned. Otherwise, it returns a -1 (EOF).

Compare with 'index'. Also see 'strrchr' in STRINGS.

```
 printf("char %c is at location %d in string %s",c,rindex(str,c),str);
```

Notes:
UNIX defines one version for System III and another for Version 7. This function is compatible with Version 7.

SBRK

Add 'n' Bytes to Task Break

Format:

```
char *sbrk(size);
int size;
```

Description:
'sbrk' allocates memory from the task space between the end of the variable space and the stack. The global variable '_eomem' points to the current limit on the variable space; it is initialized during start-up and then maintained by this routine. If an allocation wrote over the stack area, 'sbrk' returns a null pointer (NULL). The memory is cleared before the pointer returns.

```
char *newbuf,*sbrk();
newbuf = sbrk(1024);
```

Notes:
You should probably try to get memory via XGUM if no space remains in the task. You can also deallocate memory by passing a negative value to 'sbrk', but there is no checking for valid values. There currently is no routine called 'lsbrk' which would allocate large amounts of memory (greater than 32767 bytes).

Be sure and define 'sbrk' as a function returning a pointer before you use it.

STRING HANDLING ROUTINES

strcat, strncat, strchr, strrchr, strcmp, strncmp, strend, strcpy, strncpy, strlen, strpbrk, strspn, strtok

Format:

```
char *strcat(s,t)         /* string concatenation */
char *s,*t;

char *strncat(s,t,n)      /* limited string concatenation */
char *s,*t;
int n;

char *strchr(s,c)         /* find character in string from left */
char *s,c;

char *strrchr(s,c)        /* find character in string from right */
char *s,c;

int strcmp(s1,s2)         /* compare strings */
char *s1, *s2;

int strncmp(s1,s2,n)      /* compare strings within limit */
char *s1, *s2;
int n;

int strend(s1,s2)         /* string end comparison */
char *s1, *s2;

char *strcpy(s1,s2)       /* copy string */
char *s1, *s2;

char *strncpy(s1,s2,n)    /* copy limited string */
char *s1, *s2;
int n;

int strlen(s)             /* string length */
char *s;

char *strpbrk(s1,s2)      /* find character set in string */
char *s1, *s2;

int strspn(str,pat)       /* string segment pattern count */
char *str, *pat;

char *strtok(str,toksep)    /* string token manipulation */
char *str, *toksep;
```

Description:
'strcat' appends the second string to the end of the first string and returns a pointer to the first string. 'strncat' does the same thing, but makes sure that the result is no more than 'n' characters long. The following example defines a 10-byte array, initializes it to the string "bob," then appends "cat" to it. The resulting string is "bob cat."

```
char b[10];
strcpy(b,"bob");
strcat(b," cat");
```

'strchr' searches for a character in a string and returns either a pointer to it (if found) or a NULL (if not found). 'strrchr' does the same thing, but searches from the right instead of from the left, thus searching for the last occurrence instead of the first. Compare these functions to 'index' and 'rindex'.

```
char *p,*strchr();
if (p = strchr("abcde.fghij",'.'))
    printf("%s",p);
.fghij
```

'strcmp' compares two strings and returns 0 for equality, a positive number if the first is greater than the second, and a negative number if the second is the greater. 'strncmp' does the same thing, but compares at most 'n' characters.

```
if (strncmp("abc123","abcxyz",3) == 0)
    printf("same first three letters");
```

'strend' returns 1 if the end of the first string matches the second string, 0 otherwise.

```
if (strend("abcdefghij","ghij"))
    printf("same ending");
```

'strcpy' copies the second string into the first string and returns a pointer to the first string. 'strncpy' does the same thing, except that exactly 'n' characters are copied. This means that if the source string is too long, it is truncated, and if it is too short, it is padded with nulls.

```
char c[5];
strncpy(c,"testing",4);
c[4] = 0;
printf("<%s>",c);
<test>
```

'strlen' returns the length of the string it has passed. The null at the end is not counted in the length of the string.

```
char *c = "TESTING";
printf("length = %d",strlen(c));
length = 7
```

'strpbrk' examines the first string for any character occurring in the second string and returns a pointer to the first such character. NULL is returned if none are found.

```
char *p,*strpbrk();
if (p = strpbrk("abcde.fghij",",?!."))
    printf("%s",p);
.fghij
```

'strspn' returns a count of the number of characters in the first string which also occur in the second string, without a break.

```
char *pat = "cabbage";
char *str = "abcdefg";
printf("span = %d",strspn(str,pat));
span = 3  ('d' is not in 'cabbage')
```

'strtok' parses the first string passed for tokens separated by the token specified in the second string. It returns a pointer to the first such token, terminated by a null (the token is over-written). 'strtok' retains a pointer to the current position in the string in a static variable, so that subsequent calls with NULL in the first parameter will continue parsing the original string. NULL is returned when no more tokens remain.

```
char *strtok();
char *tokens = ", ;.";
char *c = "THIS,IS A:TEST!";
printf("\nFirst token=%s",strtok(c,tokens));
while ((c = strtok(NULL,tokens)) != NULL)
    printf("\n<%s>",c);

First token=THIS
<IS>
<A:TEST!>
```

SYSTEM

Send a Command to the PDOS Monitor

Format:
```
int system(str)
char *str;
```

Description:
'system' creates a task in the memory between end of memory (indicated by '_eomem') and the bottom of the stack. It gives the command string to the PDOS monitor, modifies SYRAM so that the new task gets the parent task's input as well as output port, and puts the parent task to sleep while the new task executes. The parent task waits for the sub-task to set an event to notify it of the completion of the function. On termination of the sub-task, the parent checks the Last Error Number (LEN$) of the sub-task's Task Control Block (TCB) and returns the resultant status as the status of the call.

```
system("LT");          /* print task list on screen */
system("SIEVE");       /* run sieve benchmark */
```

The parent task expects the sub-task to set event flag number 64+n where 'n' is the task number of the parent task. This information is automatically added to the command line passed to the sub-task, but it is lost if the command line invokes a command file. In the latter case, the command file should contain an EV instruction to set the proper event to notify the parent task of completion.

Notes:
A sub-task can abort without setting the event to signal the parent task. In that case, you may find yourself at the monitor level in the sub-task with no idea of how you got there. To get back into the parent task, type something on the monitor to cause a PDOS error such as ;lkj;ljk[CR]. The error will trap back to the parent task, killing the sub-task.

```
system("ACFILE"); /* run a procedure file */
...
x>???
```

The use of global events 64..95 by the 'system' call may conflict with their use by application programs. In that case, the user may need to modify the 'system' routine to use a different set of signals.

If someone on a different terminal kills the sub-task, havoc may result. The parent task hangs until someone manually sets the event; and if the task was killed without saving memory, the system will allow other tasks to be created in the parent task's memory space. To kill a sub-task created by the system call, set the event for the parent task and it will kill the sub-task.

```
system("KT");  /* hang up! */

x>LT
Task   Prt Tm  Event   Map Size
*0/0   64  1            0   78
 1/0   64  1   65       0   622
x>EV 65
```

TSTFILE

Test for Existence of a File

Format:

```
int tstfile(fname,fsize)
char *fname;
long int *fsize:
```

Description:
'tstfile' may be called from C programs to determine if a file exists and, if it exists, its size. The filename is passed in as a null-terminated string. The status returned is zero if the file exists; otherwise, it is a PDOS error number. If the file exists, the size of the file in bytes is returned as a long integer through the address passed as the second parameter.

```
long size;
if (tstfile("MYFILE:DAT",&size))
   printf("File doesn't exist");
else
   printf("File contains %ld bytes",size);
```

Notes:
'tstfile' is not a standard function on other systems, but can be very useful at times.

TTYOPEN

Connect a PDOS Port to a Stream for Input/Output

Format:

```
#include <stdio.h>
FILE *ttyopen(port)
    int port;
```

Description:
'ttyopen' creates a stream for I/O to a port. Then 'fprintf', 'getc', 'putc', etc. can function on that stream, sending and receiving characters to that port. 'fclose' deallocates the stream but performs no I/O to the port. 'fseek' and 'ftell' are not valid operations on a stream that connects to a port.

```
#include <stdio.h>
FILE *ttyopen(),*port;
.
.
.
port = ttyopen(3);      /* open port 3 for I/O */
fprintf(port,"\n\lThis goes out to port three");
.
.
.
fclose(port);
```

Notes:
The newline is left as a carriage return on output to a port, instead of being converted to carriage return/line feed.

Input from the port does not echo. Port I/O is not buffered.

UNGETC

Put Back Character on a Stream

Format:

```
#include <stdio.h>

int ungetc(c,fs)
FILE *fs;
char c;
```

Description:
'ungetc' puts back a character onto a stream so that it can be read again by 'getc'. It returns EOF on an error; otherwise, it returns the value of the character passed.

```
FILE *fd;
char c;
if ((c = getc(fd)) == 'B')
       ungetc('C',fd);
```

Notes:
Only one character can be put back. EOF (-1) cannot be put back.

UNLINK

Delete File

Format:
```
int unlink(filename);
char *filename;
```

Description:
'unlink' deletes a file from the disk. It returns 0 if the file is successfully deleted. If it is unsuccessful, it returns -1. This routine is identical to the XDLF call except for the status returned.

```
unlink("JUNKFILE");
```

WRITE

Write to File ID

Format:

```
int write(fd,buffer,bytes);
int fd,bytes;
char *buffer;
```

Description:

'write' transfers data to a file or, if 'fd' is less than or equal to 0, to a port. The file must have been opened previously with the 'open' call, or XSOP, etc. The number of bytes actually written is returned. If this number is not the same as the number of bytes requested, an error occurred during the write. The data may contain nulls and control characters.

It writes to a port if 'fd' = -(port number). If 'fd' = 0, it writes to the task's current output port.

```
fd = open("DATA:DAT",1);
write(fd,"hello, world!",13);   /* write to file */

write(0,"hello, world!",13);    /* put message to terminal */

write(-3,"hello, world!",13);   /* put message to port 3 */
```

XEQ

Run Another Program and Return Its Status

Format:

```
int xeq(program)
    char *program;
```

Description:
'xeq' makes use of the XLDF primitive to load a program into available memory and execute that program as if it were a subroutine. The 'ext' and 'err' vectors in the Task Control Block (TCB) are diverted to trap the normal and error exits of the loaded program, so when it exits the calling routine regains control.

The parameter that is passed must begin with a program name, followed by optional parameters. These parameters are set up as the command line for the called program so that XGNP will work, then the original command line is restored when the program exits.

'xeq' returns as a value any errors encountered on the call, or the Last Error Number (LEN$) in the TCB after the called program exits.

```
 if ((err = xeq("MASMC TEST:SR1,TEST:O")) != 0)
     printf("\nError on assembly = %d",err);
```

Notes:
Only files with attributes 'SY' or 'OB' can be executed, because of the limitations of the XLDF call. Some PDOS system programs won't run if they are not loaded immediately after their own TCB. Such programs cannot make use of this function.

SECTION THREE
INTERFACE TO PDOS SYSTEM CALLS

The fundamental interface to the PDOS operating system is a collection of approximately 100 system calls. Access to these calls is provided to the C programmer through the XLIB:LIB library. The following is a list of the PDOS primitives which have been implemented in that library. Each of these primitives is listed alphabetically and includes the format, description, an example, and notes which may include bugs, restrictions, and cautions. Many of the examples are taken directly from the test program used to check out these primitives, TESTXLIB:C. In that test program there are two functions, GETSTR and GETNUM, that prompt the user for input and return that input to the calling routine. These functions are in the test program if you would like more information. Although there is generally a sufficient explanation for each function in this manual, a more detailed description of the call can be found in the PDOS Reference Manual. The functions and their descriptions are as complete as possible. Please notify Eyring if you discover any problems other than those listed under the "notes" section of the function.

UNIMPLEMENTED PDOS CALLS

XLIB does not have a function for every PDOS call in the operating system. There are some primitives which do not apply in C, and others which are either difficult or hazardous to use as functions. Some calls have not been implemented because they are easily called with in-line code.

The compiler allows you to insert assembly language instructions directly into your program by the pseudo-call 'asm("literal text string");'. For example, if you want to cause an XBUG (debugger) trap at a place in your program, insert the line 'asm("xbug");' at that location. The compiler will pass the line directly to the assembler and the XBUG trap will be called when the program executes to that point.

Primitives called with in-line code

XBUG -- drop into debugging tool
XCLS -- clear console screen
XEXT -- exit to monitor without closing files
XPBC -- dump the user buffer to screen
XPCL -- output carriage return/line feed to screen
XPSP -- output one space character to screen
XRCN -- close AC command file
XRDM -- dump registers to screen

(UNIMPLEMENTED PDOS CALLS cont.)

XSUP -- enter supervisor mode
XSWP -- swap to next task
XULT -- unlock task
XUSP -- return to user mode

The remaining unimplemented PDOS primitives are listed below with an explanation of why they are excluded.

X881 --	Save 68881 enable. X881 automatically executed in CSTART if code compiled/linked with 'H' option.
XCBM --	Convert to decimal with message. This procedure is done better with a 'printf' call. XCBM requires that the pointer to the message be in-line with the call.
XEXC --	Execute a PDOS call. This primitive could be implemented, but there is no apparent reason to use it. If there is a demand for it, it will be placed in the library.
XISE --	Initialize sector. Not enough general application.
XLFN --	Look for name in slots. Not enough general application.
XLSR --	Load status register. Not enough general application.
XPCB --	Push command to buffer. Normally, this is a function only the operating system would need.
XPEM --	Put encoded message to console. Same problem as XCBM.
XPMC --	Put message to console. Same problem as XCBM.
XRDN --	Read directory entry by name. Not enough general application.
XDTV --	Define trap vectors. This procedure can be performed just as easily by modifying the vector table directly in the TCB.
XRSZ --	Read sector zero. This routine is the same as the XRSE call, with the two parameters pre-defined. Use XRSE instead.

If there is sufficient user interest in having a specific call in the library, it can be added to the list. In any case, since the sources to the current list of PDOS library functions are included in the distribution, most users should not have too much trouble implementing any functions they need.

XAPF

Append File

Format:

```
int xapf(fname1,fname2);
char *fname1,*fname2;
```

Description:

XAPF appends file1 to file2. It returns the error status or 0 if no error.

```
char fname1[20],fname2[20];
getstr("Name of file to append",fname1);
getstr("Name of file to append onto",fname2);
error = xapf(fname1,fname2);
```

XBCP

Baud Console Port

Format:

```
int xbcp(port,baud,porttype,portbase);
int port,baud,porttype;
char *portbase;
```

Description:
XBCP initializes an I/O port and binds a physical UART to a character buffer. It sets handshaking protocol, receiver and transmitter rates, and enables receiver interrupts.

'port' is the port number, 1-15. The right byte of 'port' is used to store the port number. The left byte of 'port' is composed of flags to control the protocol.

Receive and transmit rates are specified in the second parameter, 'baud.' Valid entries are 0,1,2,3,4,5,6,7 or 19200,9600,4800, 2400,1200,600,300,110.

If the third parameter, 'porttype,' is nonzero, then the port type is selected and the fourth parameter specifies the port base address. These parameters are system-defined and correspond to the UART module. The return value nonzero indicates an error -- invalid baud rate or invalid port.

```
        xbcp(2,300,0,0L); /* set port two to 300 baud */
```

```
              PORT FLAGS
fwpi 8dcs
 \\\\ \\\\_ 0 = Enable ^S^Q software handshake
  \\\\ \\\_ 1 = Control character disable
   \\\\ \\_ 2 = Enable DTR hardware handshake
    \\\\ \_ 3 = 8-bit character enable
     \\\\_ 4 = Receiver interrupt enable
      \\\_ 5 = Even parity enable
       \\_ 6 = *Reserved (High/low water)
        \_ 7 = **Reserved (^S^Q flag bit)

               *Used to clear all bits
              **Used to set U2P$
```

(XBCP cont.)

Speed table:	0 =	19200 baud
	1 =	9600 baud
	2 =	4800 baud
	3 =	2400 baud
	4 =	1200 baud
	5 =	600 baud
	6 =	300 baud
	7 =	110 baud

Notes:
While reading and writing the port flags is straightforward, you must go to SYRAM to read the baud rate once it has been written.

Since XBCP resets the physical I/O port, any data waiting in hardware buffers may be lost. For instance, on many of the Force CPU boards, the I/O port buffers up to eight characters on both input and output. If you transmit data and immediately perform an XBCP, several bytes of data may be lost. It is best in this case to wait a few tics fter the last transmt before performing an XBCP.

See Also:
XRPS - Read port status
XSPF - Set port flag

XBFL

Build File Listing

Format:
```
int xbfl(mask,buffer,endbuffer);
char *mask,*buffer,*endbuffer;
```

Description:
XBFL builds a directory listing in the buffer. The first parameter is a pointer to a filename specification that may have file name wildcards in it. The second parameter points to the start of a buffer where the corresponding file names will be written. The third parameter marks the end of the buffer. The function returns a status of zero, or an error number. The possible errors are 73-Not enough memory, 67-Invalid parameter, and various disk errors.

```
char filespec[20];
extern char *_eomem;     /* pointer to end of memory */
int err;
register char *bufptr = _eomem;  /* use un-allocated RAM */
getstr("Enter filespec",filespec);
if (err = xbfl(filespec,bufptr,bufptr+2000))    /* assume 2000 max */
   return err;
printf("\nfiles found:");
while(*bufptr){          /* loop until a double null */
   asm("xpcl");
   while(xpcc(*bufptr++))      /* print out a line */
        ;
}
```

Notes:
Earlier versions of this call (prior to PDOS 3.0) had one less parameter.

See Also:
XFFN - Fix file name
XLST - List file directory
XRDE - Read next directory entry

XBUG

Enter Debugger

Format:
asm("xbug");

Description:

XBUG enters the debugger in trace mode. You then can use the regular debugging commands to check your program or data, continue execution, set breakpoints, trace, single-step, and so forth.

See the PDOS Reference Manual and associated documents for a description of the debugger functions. Type [ESC] to drop into command mode on the debugger. 'H' displays the legal debugger commands. The 'G' command resumes program execution.

LEGAL DEBUGGER COMMANDS

A0-7	A-reg	#	Mem IAC
B{#,a}	Lst/def break	#,#	Mem dump
D0-7	D-reg	#,#+	Disassemble
{#}G	Go & break	#,#,#{WL}	Find B/W/L
M	Last dump	#(0-7	d(Ax)
N#	0-W,1-B,+2-w/o read	#{+-}#	Hex +/-
O	Offset		
P	PC		
Q	Exit	-	Open previous
R	Reg dump	LF	Open next
S	Status	+#	# + offset
T	Trace		
U	Unit		
W{s,e}	Window	^D	Disassemble
X	Set breaks & exit		
Z	Reset		

Trace Options:

F/R/M	Dump
G	Go
T	Running

```
putstr("\nthis is before the debug call\n");
asm("xbug");
putstr("\nthis is after the debug call");
```

the following is displayed...

```
this is before the debug call
T> 100608/0108: 2EBC00100638  MOVE.L  #$00100638,(A7) SR=.....0.....Z..[spacebar]
T> 10060E/010E: 4EB9001006AA  JSR     $001006AA       SR=.....0........[spacebar]
T> 1006AA/01AA: 226F0004      MOVE.L  $0004(A7),A1    SR=.....0........G
this is after the debug call
```

Notes:
Prior to PDOS 3.0, you had to initialize the debugger by first entering it from the monitor (PB) before executing an XBUG call. This procedure is no longer necessary.

See Also:
XRDM - Dump registers

XCBC

Check For Break Character

Format:
int xcbc();

Description:
XCBC checks for a break character. The status returned indicates what was found:

```
 0 : No break character.
-2 : [ESC] detected.
-3 : [CTRL-C] detected.
```

```
int err;
switch(err = xcbc()){
  case -2: printf("\nescape\n"); break;
  case -3: printf("\nctrl c\n"); break;
  case Ø: xpcc('.');break;
}
```

Notes:
If the control character disable flag is set on the port, XCBC does not detect break characters and always returns a status of 0.

See Also:
XCBP - Check for break or pause

XCBD

Convert Binary to Decimal String

Format:

```
char *xcbd(i);
long int i;
```

Description:

XCBD converts a 32-bit signed binary value to a decimal string. The return value is the address of the string. The buffer used is in the PDOS monitor.

```
 char *xcbd();
 xplc(xcbd(1234L));
```

Notes:

The data from this call is overwritten by other calls that also use the monitor work buffer, such as XRTM. If this is a problem, use XCBX instead and provide your own buffer.

See Also:

XCBH - Convert binary to hex
XCBX - Convert to decimal in buffer
XCDB - Convert decimal to binary
XCHX - Convert binary to hex in buffer

XCBH

Convert Binary to Hexadecimal String

Format:
```
char *xcbh(i);
long int i;
```

Description:
XCBH converts a 32-bit binary value to a hexadecimal string. The return value is the address of the string. The buffer used is the PDOS monitor work buffer in the TCB.

```
char *xcbh();
xplc(xcbh(0xabcd));
```

Notes:
The data from this call is overwritten by other calls that also use the monitor work buffer, such as XRTM. If this is a problem, use XCHX instead and provide your own buffer.

See Also:
XCBD - Convert binary to decimal
XCBX - Convert to decimal in buffer
XCDB - Convert decimal to binary
XCHX - Convert binary to hex in buffer

XCBP

Check for Break Character/Pause

Format:
int xcbp();

Description:
XCBP checks for a break character or pause. The status returned is:

0	The user pressed a key and paused the program.
-1	The user has not pressed a key since last check.
-2	The user pressed an [ESC].
-3	The user pressed a [CTRL-C].

When this call is executed, the program checks the input buffer. If the user has pressed a key, the program pauses until another key is pressed. An [ESC] returns status immediately.

If the task is already paused and an [ESC] is entered, a -2 is returned. If a [CTRL-C] is entered, a -3 value is returned. Any other character results in a return of 0.

```
while(1)
    switch(xcbp()){
        case  0: xplc("Paused..");      break;
        case -1: xpcc('.');             break;
        case -2: xplc("Saw escape");    return 0;
        case -3: xplc("Saw control C"); return 0;
    }
```

Notes:
If the port is in control character disable mode, XCBP does not treat the [ESC] and [CTRL-C] any differently than the other characters. A status of -1 or 0 is returned from XCBP.

See Also:
XCBC - Check for break character

XCBX

Convert Binary to Decimal in Buffer

Format:

```
char *xcbx(buffer,i);
long int i;
char *buffer;
```

Description:
XCBX converts binary to a decimal string in buffer provided. The return value is the address of the string.

```
char buffer[20];
char *xcbx();
putstr(xcbx(&buffer[10],123L)); /* convert the num at 10th and print */
putstr(buffer);  /* print the entire string  */
```

See Also:
XCBD - Convert binary to decimal
XCBH - Convert binary to hex
XCDB - Convert decimal to binary
XCHX - Convert binary to hex in buffer

XCDB

Convert Decimal String to Binary

Format:

```
char *xcdb(buffer,iptr);
long int *iptr;
char *buffer;
```

Description:
XCDB converts a numeric string to a 32-bit binary value. The string that 'buffer' points to is converted to binary and stored in '*iptr'. If no conversion is possible, a -1 value is returned and '*iptr' is not altered. If a partial conversion is possible, *iptr is altered and the function returns a pointer to the next character after the bad terminator. If there are no errors, XCDB returns a 0 (NULL) and *iptr is altered. The string should be null terminated. No error is returned for overflow of a 32-bit integer.

Hexadecimal numbers are preceded by a '$' and binary numbers by a '%'. Otherwise, numbers are assumed decimal. A leading '-' indicates a negative number. There can be no embedded blanks.

```
 char buff[80];
 char *c,*xcdb;
 long i;
 getstr("Enter a number",buff);
 switch ((int)(c = xcdb(buff,&i))){
    case -1: printf("no conversion possible ");break;
    default: printf("partial conversion -- remaining is %s\n",--c);
    case 0:  printf("xcdb returns %ld",i);
  }
```

Notes:
No errors on 32 bit overflow.

See Also:
XCBD - Convert binary to decimal
XCBH - Convert binary to hex
XCBX - Convert to decimal in buffer
XCHX - Convert binary to hex in buffer

XCFA

Close File With Attributes

Format:
```
int xcfa(filid,attribute);
int filid,attribute;
```

Description:
XCFA closes a file. It returns an error number or 0 if there is no error. The first parameter given is the 'filid' obtained from the open. The second is the attribute to give the file after the close.

The file attributes are coded in the following way:

0x80	AC - Procedure file
0x40	BN - Binary file
0x20	OB - Object file
0x10	SY - Memory Image of machine code
0x08	BX - BASIC token file
0x04	EX - BASIC ASCII file
0x02	TX - Text file
0x01	DR - System I/O driver
0x00	Clear file attributes

```
err = xcfa(filid,0x20); /*close the file as object file */
```

Notes:
If the file has not been altered since it was opened, XCFA does not alter the file attributes.

See Also:
XRFA - Read file attributes
XWFA - Write file attributes

XCLF - Close file

XNOP - Open non-exclusive random
XSOP - Open sequential
XROO - Open random read only
XROP - Open random

XCHF

Chain File

Format:

```
int xchf(filename);
char *filename;
```

Description:
XCHF chains to another file. It allows your task to end and another task to begin. The returned value (if it comes back) is a PDOS error. The file named can be of type 'OB', 'SY', 'BX', 'EX', or 'AC'.

```
 xchf("SIEVE");
 xplc("this shouldn't print");
```

See Also:
XEXT - Exit to monitor
XEXZ - Exit to monitor w/ command
XLDF - Load file

XCHX

Convert Binary to Hexadecimal String in Buffer

Format:
```
char *xchx(buffer,i);
long int i;
char *buffer;
```

Description:
XCHX converts a 32-bit binary value to a hexadecimal string. The returned value is the address of the string.

```
char buffer[] =  "the magic number is xxxxxxxxxxx";
char *xchx();
xchx(&buffer[20],123L);
xplc(buffer); /* now print the number */
```

See Also:
XCBD - Convert binary to decimal
XCBH - Convert binary to hex
XCBX - Convert to decimal in buffer
XCDB - Convert decimal to binary

XCLF

Close File

Format:

```
int xclf(filid);
int filid;
```

Description:
XCLF closes a file. It returns an error number or 0 if no error. The parameter given is the 'filid' obtained from the open.

```
 err = xclf(filid); /*close the file I opened earlier */
```

See Also:
XCFA - Close file w/attribute

XNOP - Open non-exclusive random
XSOP - Open sequential
XROO - Open random read only
XROP - Open random

XCLS

Clear Screen

Format:
```
asm("xcls");
```

Description:
XCLS clears the screen. It returns no status. XCLS depends on the CSC$ field in the Task Control Block (_tcbptr->_csc) to tell what characters to print on a given terminal. These are initialized by the MTERM utility, or can be set under program control.

Notes:
Some manufacturer's terminals cannot clear the screen at the advertised baud rate. You may have to delay after giving this command so that the terminal can finish. In that case, try the following:

```
     asm("xcls");
     xpdc(5,"\0\0\0\0\0"); /* send some nulls to wait a bit */
```

See Also:
XPSC - Position cursor
XRCP - Read port cursor position
XTAB - Tab to column

XCPY

Copy File

Format:

```
int xcpy(fname1,fname2);
char *fname1,*fname2;
```

Description:
XCPY copies file1 to file2. It returns the error status or 0 if no error.

```
 char list1[20];
 char list[20];
 getstr("source file",list);
 getstr("enter destination file",list1);
 return (xcpy(list,list1));
```

Notes:
XCPY is very slow since it only copies 252 bytes at a time and uses the user buffer in the TCB for the work space. The 'copy' call in STDLIB uses the un-allocated memory in your program and runs faster.

See Also:
COPY (STDLIB)

XCTB

Create Task Block

Format:

```
int xctb(mem,priority,port,low,high,comstring,sontasknoptr);
int mem,priority,port;
char *low,*high,*comstring;
int *sontasknoptr;
```

Description:

XCTB creates a task block and a new task for PDOS to run. The task number of the new task is written in location 'sontask-noptr'. The first parameter, 'mem', is the task size in 1K byte increments. If 'mem' is a negative number, then 'low' and 'high' input parameters specify the memory range and 'comstring' points to the task's command line. If 'mem' is 0, then 'low' and 'high' specify bounds and 'comstring' points to the task entry address. If 'mem' is positive, then 'low' and 'high' have no meaning. Also, when 'mem' is nonzero, a 'comstring' value of 0 indicates that the PDOS monitor is the default task to start.

'mem'	'low/high'	'comstring'
neg	memory limits	command pointer
0	memory limits	starting address
pos (Kbytes)	ignored	command pointer

'Priority' is the priority of the new task and ranges from 1 to 255, with 1 as the low priority. 'Port' is the I/O port of the task. Any or all tasks on the system may share an output port, but input ports are given to only one task at a time. If the port value is negative, then the port is for output only. If 'port' is 0, then no port is assigned.

Any errors are returned. A zero is returned if there are no errors.

```
extern long *_eomem;
long unsigned lowmem,highmem;
int taskno;
                         /* round up to a 2K boundary */
lowmem = (_eomem + 2048L) & 2047L;
highmem = lowmem + 4096;        /* create a 4K task */
error = xctb(   -1,  /* execute monitor command line */
                64,             /* priority 64 */
                -_tcbptr->_prt, /* use my port for output */
                lowmem,highmem, /* use this memory for task */
                "LT.EV 65.PB",  /* execute these commands */
                &taskno);       /* tell me what task it is */
if (error)
    return error;
xsui(65);                    /* wait on that event */
xktb(-taskno);               /* kill the task without */
                             /* deallocating memory */
while (xrts(taskno)){
    asm("xswp");             /* wait for task to die */
}
return 0;

x>FM 2[CR]       Make sure there is system memory for tasks!
...
int err;
int sontask;
err = xctb(2,64,0,0L,0L,"DF TSTFILE",&sontask);
/* 2k  of memory,  priority 64,  no port,  memory to  come from system (hope-
fully!), a monitor task to define a new file */
```

Notes:
This call is easy to foul up! Be prepared to crash your system with your experiments.

An invalid port assignment does not result in an error. The command line has a maximum of 64 characters. If you don't specify the low and high memory, the new task is created at the end of your task, wiping out your stack and return address! Use 'sbrk' to allocate memory for the new task or put it at a known available address. If the task dies or you kill it with a positive task number, the memory is deallocated to the system and available to other users. If you got the memory with XGUM there should be no problem, but if you allocated it from your own stack, it could be a disaster!

See Also:
XLDF - Load file
XKTB - Kill task
SYSTEM (STDLIB)
XEQ (STDLIB)

XDEV

Delay Set/Reset Event

Format:

```
int xdev(time,event);
long int time;
int event;
```

Description:

XDEV causes an event to be set/reset after a delay. A time interval is specified, and the system counts down the time in tics. When 0 is reached, the specified event is set. If the 'event' is negative, then it is reset.

If the 'time' is 0, then any pending timed event equal to the second parameter is deleted from the system stack.

An error status is returned. 0 specifies no errors.

```
xplc("waiting 10 seconds...");
err = xdev(1000L,128);
if(err)
  xerr(err);
xsui(128); /* sleep until event 128 */
xplc("time is up");
```

See Also:

XSEF - Set event flag w/swap
XSEV - Set event flag
XSUI - Suspend until interrupt
XTEF - Test event flag

XDFL

Define File

Format:

```
int xdfl(filename,sectors_allocated);
char *filename;
int sectors_allocated;
```

Description:

XDFL defines a file. 'filename' is any valid PDOS filename. 'sectors_allocated' is the number of contiguous sectors allocated to the file when it is defined. If this parameter is 0, then one sector is allocated. The return value indicates an error if nonzero.

```
 int err;
 err = xdfl("NEWFILE",4);
```

See Also:

XDLF - Delete file
XRNF - Rename file
XZFL - Zero file

XDLF

Delete File

Format:
```
int xdlf(filename);
char *filename;
```

Description:
XDLF deletes a file. The return value indicates an error if nonzero.

```
int err;
err = xdlf("OLDFILE");
```

See Also:
XDFL - Define file
XRNF - Rename file
XZFL - Zero file

UNLINK - STDLIB entry point

XDMP

Dump Memory

Format:

```
xdmp(size,ptr);
int size;
char *ptr;
```

Description:
The XDMP function allows you to dump a block of memory to the screen. This can be handy during development for debugging purposes, to provide a quick look at system tables or other program memory. The first parameter is how many bytes to dump; the second is the starting address of the memory.

The following code dumps the first 256 bytes of SYRAM to the screen:

```
        #include <SYRAM:H>
              .
              .
              .
        xdmp(256,_syram);
```

The following is displayed:

```
        0000A800: 0000 0B64 0007 FE00 0008 1000 FC10 0000 ...d..~.....|...
        0000A810: 0000 0002 8B04 CD1E 2B04 0017 0008 4D7C ......M.+.....M}
        0000A820: 0A08 5600 102C 2400 00FF 00FC FBFA 0000 ..V..,$....|{z..
        0000A830: 0000 0000 0000 00F9 0000 0000 0000 0000 .......y........
        0000A840: 0000 0000 0000 0000 0004 0000 0000 0000 ................
        0000A850: 0000 0000 0000 0000 0001 0100 0000 0000 ................
        0000A860: 0000 0000 0000 0000 0000 0100 0000 0000 ................
        0000A870: 0000 0000 0000 0000 0000 0000 0000 0000 ................
        0000A880: 0000 0000 2000 FE00 0400 0000 0000 0000 .... .~.........
        0000A890: 0000 0000 0000 0000 0000 0009 0000 0001 ................
        0000A8A0: 0000 000A 0000 0014 0000 0000 0000 B3EA ..............3j
        0000A8B0: 0004 F800 0000 0000 0002 4005 0000 0080 ..x.......@.....
        0000A8C0: 0000 0100 0000 0000 0000 0000 0000 0000 ................
        0000A8D0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
        0000A8E0: 0000 0000 0000 0007 B460 0007 B480 0000 ........4`..4...
        0000A8F0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

See Also:
XRDM -- Dump registers to console
XBUG -- Debug call

XERR

Error Exit

Format:
```
int xerr(err);
int err;
```

Description:
XERR returns to the PDOS monitor and gives an error message "PDOS ERR #."

This return can be trapped by setting the ERR$ vector in the Task Control Block (_tcbptr->_err -- see TCB:H) to point to your own trap function. See the example for XLDF to see how this might work. In such a case, the error code is passed in in register D1.

```
int err;
if ( err = xdlf("OLDFILE")) xerr(err);
```

Notes:
The exit(n) call performs this function in a more portable way.

See Also:
XCHF - Chain to file
XEXT - Exit to monitor
XEXZ - Exit to monitor w/ command
XLER - Load error register

XEXT

Exit to Monitor

Format:
asm("xext");

Description:
XEXT exits to the PDOS monitor. No status is returned.

This call may be trapped by setting the EXT$ vector in the Task Control Block (_tcbptr->_ext -- see TCB:H) to point to your own code. See the example for XLDF.

```
.
.
.
asm("xext")'
printf("\n this shouldn't print");
```

See Also:
XCHF - Chain to file
XERR - Return error D0 to monitor
XEXZ - Exit to monitor w/ command
XLER - Load error register

XEXZ

Exit to Monitor with Command

Format:

```
xexz(command);
char *command;
```

Description:
XEXZ exits to the PDOS monitor. The monitor then executes the command line passed as a parameter. This may be exploited as a form of the 'chain' command, only with command line parameters. No status is returned.

This call may be trapped by setting the EXT$ vector in the Task Control Block (_tcbptr->_ext -- see TCB:H) to point to your own code. See the example for XLDF.

```
xexz("LT");     /* exit program and list tasks */
```

See Also:
XCHF - Chain to file
XERR - Return error D0 to monitor
XEXT - Exit to monitor
XLER - Load error register

XFAC

File Altered Check

Format:

```
int xfac(filename);
char *filename;
```

Description:
XFAC tests and clears the file altered bit in the file status of a file. The function requires a pointer to the name of a file as input.

It returns a 255 if the file altered bit was set and a 0 if the bit was clear. The bit is cleared on the file status in either case.

```
        if (xfac("TEST:DAT"))
           printf("File was altered since last check");
```

XFBF

Flush Buffer to Disk

Format:
int xfbf();

Description:
XFBF flushes buffers to disk. It effectively causes a checkpoint on an open and altered file. The return value is an error #, or 0 if no error occurred.

Stream files that are open should first be flushed with 'fflush,' after which XFBF will flush the file slot.

```
.
.
.
/* make sure the following is written to disk */
fprintf(myfile,"\nlogging message");
fflush (myfile);
xfbf();
.
.
.
```

XFFN

Fix File Name

Format:

```
char *xffn(filename,disknoptr);
char *filename;
int *disknoptr;
```

Description:
XFFN parses a filename into its components. It parses a character string for filename, extension, directory level, and disk number. The results are converted into a standard format in the 32-character monitor work buffer (MWB$(a6)). The address of this buffer is returned, or 0 if an invalid filename.

The formatted character array has the following structure:

```
        [0-7] filename
        [8-10] extension
        [11] level
```

The disk number is returned in *disknoptr.

System defaults are used for disk number and directory level if they are not specified in the filename.

```
char *buffer,*xffn();
int diskno;
buffer = xffn("MYFILE:SR",&diskno);
```

Notes:
The record created is within the monitor work buffer, and overwrites or is overwritten by other primitives using the monitor work buffer.

See Also:
XBFL - Build file directory list
XLST - List file directory
XRDE - Read next directory entry

XFTD

Fix Time/Date

Format:
```
xftd(timeptr,dateptr);
int *timeptr,*dateptr;
```

Description:
XFTD reads the time and date in the following packed binary format:

```
time = hours*256 + minutes
date = (year * 16 + month) * 32 + day
```

The binary code can be used for sorting or comparisons and then unpacked to ASCII for display. See XUDT and XUTM to unpack both date and time.

```
int time,day;

xftd(&time,&day);
printf("\nHour=%d",time >> 8);
printf("\nMinutes=%d",time & 0xff);
printf("\nDate=%d/%d/%d", day & 0x1e0, day & 0x1f, day >> 9);
```

See Also:
XPAD - Pack ASCII date
XRDT - Read date
XRTM - Read time
XUAD - Unpack ASCII date
XUDT - Unpack date
XUTM - Unpack time
XWDT - Write date
XWTM - Write time

XFUM

Free User Memory

Format:

```
int xfum(kbytes,begadr);
int kbytes;
char *begadr;
```

Description:
XFUM frees user memory for use by the operating system. It allows a task to give 'kbytes' of memory back to the operating system. The second parameter gives the starting address of the memory. If the status return is nonzero, there is a memory error.

```
 char *begin;
 status = xfum(32,begin);
 if(status) xplc("Can't give back that much");
```

See Also:
XGML - Get memory limits
XGUM - Get user memory

XGCB

Get Conditional Character

Format:
int xgcb();

Description:
XGCB checks the current AC file, input message pointer, or input port for input. The status returned is:

```
 0...255 for character available
-1 if no character available
-2 if [ESC] entered
-3 if [CTRL-C] entered (buffer cleared)
```

Unlike XGCC, XGCB also checks the AC file and input message pointer for input. And, unlike XGCR, it returns immediately if no data is available.

```
int c;
asm("xpcl");
while ((c = xgcb()) > -2)
    switch(c){
        case -1: putchar('?'); break; /* NO INPUT */
        default: if (c > 31)
                    putchar(c);
                 else
                    printf("%d",c);
    }
printf("exiting on %d",c);
```

Notes:
The name is confusing.

If the control character disable bit is set on the port, XGCB will not return a status of -2 or -3.

See Also:
XGCC - Get character conditionally
XGCP - Get port character
XGCR - Get character

XGCC

Get Character Conditionally

Format:
int xgcc();

Description:
XGCC gets one character from the console keyboard. It returns as status:

```
 0...255 for character pressed (ASCII value of character)
-1 if no character available
-2 if [ESC] entered
-3 if [CTRL-C] entered (buffer cleared)
```

Unlike XGCB, XGCC only samples the input port. And, unlike XGCP, it returns immediately if no data is available.

```
int c;
asm("xpcl");
while ((c = xgcc()) > -2)
     switch(c){
          case -1: putchar('?'); break; /* NO INPUT */
          default: if (c > 31)
                         putchar(c);
                    else
                         printf("%d",c);
     }
printf("exiting on %d",c);
```

Notes:
If the control character disable bit is set on the port, XGCC will not return a status of -2 or -3. Instead, it will return a value of 27 or 3 -- the ASCII value of [ESC] and [CTRL-C].

See Also:
XGCB - Get conditional character
XGCP - Get port character
XGCR - Get character

XGCP

Get Character From Port

Format:
int xgcp();

Description:
XGCP gets one character from the console port. It waits for the character to come. The status returned is:

```
 0...255 for character available
-2 if [ESC] entered
-3 if [CTRL-C] entered (buffer cleared)
```

Unlike XGCR, XGCP only reads from the input port. And, unlike XGCC, it waits for a character if none is available.

```
main()
{
    int c;
    asm("xpcl");
    while ((c = xgcp()) > -2)
        printf("%x",c);
    printf("exiting on %d",c);
}
```

Notes:
If the control character disable bit is set on the port, XGCP will not return a status of -2 or -3. Instead, it will return a value of 27 or 3 -- the ASCII value of [ESC] and [CTRL-C].

See Also:
XGCB - Get conditional character
XGCC - Get character conditional
XGCR - Get character

XGCR

Get Character

Format:
int xgcr();

Description:
XGCR gets one character from the input stream (AC file, Internal Message Pointer, or input keyboard). It waits for the character to come. The status returned is:

```
 0...255 for character available
-2 if [ESC] entered
-3 if [CTRL-C] entered (buffer cleared)
```

Unlike XGCP, XGCR samples the AC filid and the Input Message Pointer as well as the keyboard. And, unlike XGCB, it suspends waiting for input if none is available.

```
main()
{
    int c;
    asm("xpcl");
    while ((c = xgcr()) > -2)
        printf("%x",c);
    printf("exiting on %d",c);
}
```

Notes:
If the control character disable bit is set on the port, XGCR will not return a status of -2 or -3. Instead, it will return a value of 27 or 3 -- the ASCII value of [ESC] and [CTRL-C].

See Also:
XGCB - Get conditional character
XGCC - Get character conditional
XGCP - Get port character

XGLB

Get Line in Buffer

Format:
int xglb(buffer);
char buffer[80];

Description:
XGLB gets a line from the keyboard and puts it into the user buffer. PDOS editing characters may be used during input of the line. The input buffer must be at least 80 characters long. The function returns a status indicating the type of terminating character:

```
 0 if the line ended with a carriage return
-2 if the line ended with an [ESC]
-3 if the line ended with a [CTRL-C] (buffer cleared)
```

```
main()
{
    char buffer[100];
    int terminator,i;
    for(i=0;i<10;i++){
        terminator = xglb(buffer);
        printf("\nterminator was %d\n",terminator);
        printf("%s\n",buffer);
    }
}
```

Notes:
If the control character disable bit is set, XGLB will beep on entry of [ESC] or [CTRL-C] and not allow it to be entered. The function will only be able to return a value of zero.

See Also:
XGLM - Get line in monitor buffer
XGLU - Get line in user buffer

XGLM

Get Line in Monitor Buffer

Format:

```
int xglm(bufptr);
char **bufptr;
```

Description:

XGLM gets a line into the monitor buffer. The monitor buffer is a buffer in the TCB area where the current command line resides, along with all the parameters (argv[0],argv[1],..., argv[argc-1]). A program could issue an XGLM call from within an AC file to look at the next line, exit, and the line would still be executed.

You must pass the address of a character pointer to XGLM. This character pointer is then set to point to the monitor buffer.

The function returns a status indicating the type of terminating character:

```
 0 if the line ended with a carriage return
-2 if the line ended with an [ESC]
-3 if the line ended with a [CTRL-C] (buffer cleared)
```

During line entry, the PDOS editing characters may be used. The PDOS Reference Manual explains these characters under the description of XGLM. Also, the &0,&1,&2 parameter passing convention is followed if the file is run as an AC file. For example, in the line " hello there &1", parameter 1 of the command line replaces the &1.

```
char *c;
int delim;
delim = xglm(&c);
printf("entered string %s with delimiter %d\n",c,delim);
```

Notes:

Executing an XGLM call overwrites the 'argc,argv' parameters. If you need these parameters, you should copy them into a separate location before calling XGLM.

If the control character disable bit is set, XGLM will beep on entry of [ESC] or [CTRL-C] and not allow it to be entered. The function will only be able to return a value of zero.

See Also:

XGLB - Get line in buffer
XGLU - Get line in user buffer

XGLU

Get Line in User Buffer

Format:

```
int xglu(bufptr);
char **bufptr;
```

Description:
XGLU reads a line into the user buffer. The user buffer is a buffer in the TCB area. You must pass the address of a character pointer to XGLU. This pointer is assigned the address of the user buffer. The address pointed to by 'bufptr' has its contents altered to become the address of the new string just entered as shown in the example below.

The function returns a status indicating the type of terminating character:

```
 0 if the line ended with a carriage return
-2 if the line ended with an [ESC]
-3 if the line ended with a [CTRL-C] (buffer cleared)
```

During line entry, the PDOS editing characters may be used as explained in the PDOS Reference Manual under the description of XGLU. Also, the &0,&1,&2 parameter passing convention is followed if the file is run as an AC file. For example, in the line " hello there &1" parameter 1 of the command line replaces the &1.

```
    char *c;
    int delim;
    delim = xglu(&c);
    printf("entered string %s with delimiter %d\n",c,delim);
```

Notes:
If the control character disable bit is set, XGLU will beep on entry of [ESC] or [CTRL-C] and not allow it to be entered. The function will only be able to return a value of zero.

See Also:
XGLB - Get line in buffer
XGLM - Get line in monitor buffer

XGML

Get Memory Limits

Format:

```
xgml(endtcbptr,endmemptr,begmemptr,syramptr,tcbptr);
long *endtcbptr,*endmemptr,*begmemptr,*syramptr,*tcbptr;
```

Description:
XGML gets the memory limits and locates task sections. The task may use up to but not including the upper memory limit. No value is returned.

The passed parameters must be the addresses of long words or pointers that receive the following values:

endtcb:	end of the task control block
endmem:	end of task memory
begmem:	last loaded address
syram:	address of SYRAM system RAM variable
tcb:	task control block address

The global variables '_syram' and '_tcbptr' are already loaded with pointers to those two structures. See "TCB:H" and "SYRAM:H".

```
long endtcb,uppermem,lastload,syram,tcb;
xgml(&endtcb,&uppermem,&lastload,&syram,&tcb);
printf("\nendtcb   = %lx",endtcb);
printf("\nuppermem = %lx",uppermem);
printf("\nlastload = %lx",lastload);
printf("\nsyram    = %lx",syram);
printf("\ntcb      = %lx",tcb);
```

See Also:
XGUM - Get user memory
XFUM - Free user memory

XGMP

Get Message Pointer

Format:

```
int xgmp(number,mptr);
int number;
char **mptr;
```

Description:
XGMP checks one of the sixteen message slots for a waiting message. If it finds a message there, it returns a pointer to it. The message number is indicated in the first parameter and the pointer to the message is returned in the second parameter. The function returns a value of 255 to indicate no message is waiting, or the number of the task that sent the original message.

This type of message is also linked to the PDOS system events 64-79. When a message is sent via XSMP, the corresponding event (message number + 64) is set. When the message is received via XGMP, the corresponding event is cleared. A task may suspend on the message event and thus require no PDOS resources while it waits for a message.

```
char *ptr;
int status;

if ((status=xgmp(5,&ptr))==255)         /* check message 5*/
     printf("\nNo message");
else
     printf("\ntask = %d, message=%s",status,ptr);
```

See Also:
XSMP -- Send message pointer
XSTM -- Send task message
XGTM -- Get task message
XKTM -- Kill task message

XGNP

Get Next Parameter

Format:

```
int xgnp(paramptr);
char *paramptr[];
```

Description:
XGNP gets the next command line parameter. The address of the next command line word is placed in the location 'paramptr'. The status is returned 0 if the word is there; otherwise, it is nonzero.

Normally, this call is used to parse the command line and gather command line parameters. However, the command line is already gathered during the initialization of a C program.

This call could be useful if the programmer supplied a different version of the initialization code. You could also use the call as a general-purpose parsing routine. XGNP uses the CMD$ and CLP$ (_tcbptr->_cmd and _tcbptr->_clp) fields of the TCB to get the next parameter. If these are modified by the program, XGNP parses an arbitrary string in the same way that it parses a command line.

```
#include "TCB:H"
char *c;
_tcbptr->_cmd = ' ';
_tcbptr->_clp = "WHAT, ME WORRY?";
while (xgnp(&c))
    printf("\n->%s<-",c);

/* prints out the message below */

            >WHAT<
            ><
            >ME<
            >WORRY?<
```

XGTM

Get Task Message

Format:

```
int xgtm(buffer);
char buffer[64];
```

Description:
XGTM gets a task message. The return value is the sender's task ID. If there are any errors, they are given in the return value. Task messages are created by tasks executing the XSTM call, or by a user executing the 'SM' monitor command.

The message buffer is a fixed length, and can contain data in any format. The length is normally 64 bytes, but it may be altered by customizing PDOS via a new SYSGEN.

```
char buffer[64];
int sender;
sender = xgtm(buffer);
if (sender >= 0){
    printf("\nmessage=%s",buffer);
    return 0;
}
else
    return -1;
```

See Also:
XGMP - Get message pointer
XSMP - Send message pointer
XSTM - Send task message
XKTM - Kill task message

XGUM

Get User Memory

Format:

```
int xgum(kbytes,begadrptr,endadrptr);
int kbytes;
long *begadrptr,*endadrptr;
```

Description:
XGUM gets memory from the operating system. It allows a task to request the operating system for 'kbytes' more memory. The begin and end addresses are stored at the location pointed to by the second and third parameters. If the status return is nonzero, there was an error (no memory available).

```
char *dummy;
int i;
long j,dummy;
int error;
int freesize;
long freeptr;

error = 73;
printf("Find largest block of available memory\n");
for (i=1024;i>0 && error != 0;i -= 2){
    printf("\rSearching for %4.4dK ",i);
    error = xgum(i,&j,&dummy);
}
if (i==0)
    return 73;
freesize = i;
freeptr = j;
return 0;
```

See Also:
XGML - Get memory limits
XFUM - Free user memory

XKTB

Kill Task

Format:

```
int xktb(task);
int task;
```

Description:
XKTB kills the task with a given task number. If there are any errors, they are given in the return value.

A task does not die immediately. The operating system first closes all files open to that file and performs other clean up chores. You can get an error (and potentially even crash the system) by killing a task and immediately overlaying the memory it occupies. If you need to reclaim a task's memory for some other use, use XRTS interspersed with XSWP calls which are waiting for the task status to go to zero. Then the memory may be safely overlayed. If you are reclaiming the memory, be sure to negate the task number passed to XKTB.

```
int err;
err = xktb(3);  /* kill task #3 */


xktb(-sontask);  /* kill it without deallocating mem */
do{
    asm("xswp");  /* twiddle fingers while it dies */
} while (xrts(sontask));

/* now, overlay the memory where that task was. */
```

See Also:
XCTB - Create task
XRTS - Read task status
XSWP - Swap task

XKTM

Kill Task Message

Format:

```
int xktm(task,buffer);
int task;
char *buffer;
```

Description:

XKTM reads and kills a task message. It intercepts the message waiting for a task. The message is then returned in the buffer. If there are any errors, they are given in the return value.

```
 int err;
 char buffer[64];
 err = xktm(3,buffer);    /* read task 3's message */
```

See Also:

XGMP - Get message pointer
XGTM - Get task message
XSMP - Send message pointer
XSTM - Send task message

XLDF

Load File

Format:
```
int xldf(eflag,memlow,memhigh,filename,lowptr,highptr);
int eflag;
char *memlow,*memhigh;
char *filename;
char **lowpr,**highptr;
```

Description:
XLDF loads an object code file into memory. If 'eflag' is nonzero, the program is executed immediately. Otherwise, control returns to your calling program. 'Memlow' and 'memhigh' are memory bounds for the load. 'Lowptr' and 'highptr' are locations where the lowest loaded address and highest loaded address are returned. The value returned is an error if nonzero.

```
errtrap()
{
       longjmp(env,3);
}

extrap()
{
      asm("MOVE D1,DØ\nXLER");/* save error */
      longjmp(env,4);

int doxldf()   /*xldf--load file*/
{
   long unsigned low,hi;
   long unsigned lowlod,hilod;
   char *oldext = _tcbptr->_ext;
   char *olderr = _tcbptr->_err;
   char *oldead = _tcbptr->_ead;
   if (!setjmp(env)){
      _tcbptr->_ext = extrap;
      _tcbptr->_err = errtrap;
              /* round up to a 2K boundary */
      low = (long unsigned) _eomem + 2Ø48L;
      low &= 2Ø47L;
      hi = low + 16384;      /* 16K task */
      error = xldf(1,low,hi,"HELLO",&lowlod,&hilod);
      xler(error);
   }
   _tcbptr->_ext = oldext;
   _tcbptr->_err = olderr;
   _tcbptr->_ead = oldead;
   return _tcbptr->_len;
}
```

Notes:
If you want to execute a program as if it were a function and have it return values to the calling program, there are a few things to watch for. First, you must trap all exits so that the executed program will not dump you into the monitor when it finishes. You must modify and restore the _ext and _err fields of the TCB. Second, save and restore the _ead field, since XLDF will modify it. If this isn't done, a "GO" command from the monitor begins execution from the starting point of the executed program -- which might not be correct. Third, some PDOS system programs can not be executed via XLDF because they depend on being loaded immediately after the TCB. The XEQ function in STDLIB uses XLDF and can either serve as an example of its use, or can be used in its stead.

See Also:
XCTB - Create task block
XCHF - Chain file
XEXZ - Exit to monitor with command

XLER

Load Error Register

Format:
```
xler(err);
int err;
```

Description:
XLER loads up the error register. The error register is the Last Error Number in the Task Control Block (LEN$(A6) or _tcbptr->_len).

A PDOS file of type AC can use passed parameters. The pseudo variable &0 is reserved for executing programs to return errors from the executing program. This primitive loads &0 with an error number.

This example shows a program called "main" and a fragment of an AC file. The fragment executes the list command since 32 was returned.

```
main()
{
 xler(32);
}

.MAIN
.IF &0=32 .LS
```

See Also:
XCHF - Chain to file
XERR - Return error D0 to monitor
XEXT - Exit to monitor
XEXZ - Exit to monitor w/ command

XLKF

Lock File

Format:
```
int xlkf(filid);
int filid;
```

Description:
XLKF locks a file so that no other task can access it. In order to unlock the file, you must call XULF. The returned value is an error code, or 0 if no error occurred. The lock and unlock file commands are used when tasks open a file in non-exclusive mode. In this mode, they share the file slot, and both have read/write access to the file. To prevent conflicts, each task must lock the file, position to the desired record number, read or write, and unlock the file.

```
int filtype,filid,err;
char fname[] = "MYFILE:SR";
if (err=xnop(fname,&filtype,&filid))     /* shared access open */
    _error("\nerror %d on file %s",err,fname);

...
while ((err = xlkf(filid)) == 75){
    asm("xswp");  /* hang while locked */
}
if (err)
    _error("\nerror %n locking file %s",err,fname);

/* position, read/write, etc. */

if(err = xulf(filid))
   _error("\nerror %n unlocking file %s",err,fname);
```

See Also:
XNOP - Open non-exclusive random
XRFP - Read file position
XPSF - Position file
XULF - Unlock file

XLKT

Lock Task

Format:
int xlkt();

Description:
XLKT locks the current task. A locked task has the undivided attention of the CPU for executing instructions. No other task is given a time slice. The task must be unlocked with the XULT call for normal multi-tasking to resume. The returned value is the status of the swap lock variable before the call was made: 0 is unlocked, nonzero is locked.

```
int lockstate;
lockstate = xlkt();
printf("previous state was %d",lockstate);

/* in here do critical code */

asm("xult");
printf(" now normal task swapping resumes");
```

See Also:
XSWP - Swap to next task
XULT - Unlock task

XLST

List File Directory

Format:

```
int xlst(filespec);
char *filespec;
```

Description:
XLST displays a disk directory on the console. The directory displayed is based on the file specification string.

It returns an error if there was a problem accessing the disk.

```
xlst("@:C;12");          /* display all C source files in level 12 */
```

See Also:
XBFL - Build file directory list
XFFN - Fix file name
XRDE - Read next directory entry

XNOP

Non-exclusive Random Open File

Format:

```
int xnop(filename,filtype_ptr,filid_ptr);
char *filename;
int *filtype_ptr, *filid_ptr;
```

Description:

XNOP opens a file for non-exclusive random access. The first parameter is the filename. If the file successfully opens, the file attribute is stored at location 'type_ptr' and the 'filid' is stored at location 'filid_ptr'.

The returned value is zero or a PDOS error number.

When two tasks open a file in non-exclusive open mode, they both share the same file slot. That means that ANY file operation by one task on that file affects any file operation by the other task. Each task must lock the file and position to the desired record before reading or writing to avoid conflicts.

```
int filtype,filid,err;
char fname[] = "MYFILE:SR";
if(err=xnop(fname,&filtype,&filid))      /* shared access open */
    _error("\nerror %d on file %s",err,fname);

while ((err = xlkf(filid)) == 75)
    asm("xswp");  /* hang while locked */
 if (err)
    _error("\nerror %n locking file %s",err,fname);

  /* position, read/write, etc. */

 if(err=xulf(filid))
    _error("\nerror %n unlocking file %s",err,fname);
```

Notes:
PDOS sometimes gets confused about when a shared file should be closed, especially when a task aborts without closing a shared file.

See Also:
XSOP - Open sequential
XROO - Open random read only
XROP - Open random

XLKF - Lock file
XULF - Unlock file
XRFP - Read file position
XPSF - Position file

XCFA - Close file w/attribute
XCLF - Close file

XPAD

Pack ASCII Date

Format:

```
int xpad(datestr);
char *datestr;
```

Description:
XPAD converts an ASCII string in the form dd-mon-yy to the packed format used internally by PDOS. Thus, dates such as "8-OCT-86" and "7-JUL-76" are converted to binary encoded format. The date string does not require special delimiters between the different fields. The month may be lowercase, uppercase, or a combination of both. If PDOS cannot figure out the date, it returns a value of -1.

```
main()
{
        showdate("7-jul-76");
        showdate("7-JUL-76");
        showdate("7JUL76");
        showdate("Ø7-JUL-76");
        showdate("lkjlkjlkj");
}

showdate(datestr)
char *datestr;
{
        if((date=xpad(datestr))==-1)
           printf("\nBad date");
        else
           printf("\n%u",date);
}
```

The following is displayed:

```
39143
39143
39143
39143
Bad date
```

See Also:
XRDT -- Read Date
XWDT -- Write Date
XFTD -- Fix Time and Date
XUDT -- Unpack Date
XUAD -- Unpack ASCII Date

XPBC

Put User Buffer to Console

Format:
```
asm("xpbc");
```

Description:
XPBC outputs the user buffer to the console. It outputs the TCB user buffer to the console and/or spool unit. This buffer is a scratch buffer that the user can use for whatever purpose is necessary. The XGLU call fills the user buffer with data from the keyboard. This call dumps the user buffer to the screen.

```
 char *c;
 xglu(&c);
 asm("xpbc");
```

Notes:
Be aware that some primitives use the user buffer implicitly. It is convenient to use the user buffer at times, but it can be overwritten if you aren't careful.

See Also:
XGLU - Get line in user buffer

XPCC - Put character(s) to console
XPCL - Put CRLF to console
XPSP - Put space to console

XPCC

Put Console Character

Format:

```
int xpcc(c);
char c;
```

Description:
XPCC outputs a character to the console or the spool unit. It returns the value of the character sent. No error status.

```
while(xpcc(*str++));  /* output a string */
```

See Also:
XPBC - Put buffer to console
XPCL - Put CRLF to console
XPCR - Put character raw
XPDC - Put data to console
XPSP - Put space to console

XPCL

Put Carriage Return/Line Feed to Console

Format:
asm("xpcl");

Description:
XPCL outputs a carriage return/line feed to console or spool unit. No value is returned.

See Also:
XPBC - Put buffer to console
XPCC - Put character(s) to console
XPSP - Put space to console

XPCR

Put Character Raw

Format:
```
xpcr(c);
char c;
```

Description:
XPCR works much the same as XPCC because it outputs the indicated character to the output port. However, XPCR does not expand tabs. The current column counter is not updated in the TCB and the port row/column is not updated in SYRAM. XPCR is useful when a port is connected to another computer or to some other device where the data must be output without modification.

```
char *c;
c = "There is a tab here->\t<--";
while(*c)
    xpcr(*c++);
c = "There is a tab here->\t<--";
while(*c)
    xpcc(*c++);
```

The following is displayed:

```
There is a tab here-><--           tab not expanded
There is a tab here->   <--        tab expanded
```

See Also:
XPCC -- Put console character
XPDC -- Put data to console

XPDC

Put Data to Console

Format:
```
xpdc(number_of_bytes,strptr);
int number_of_bytes;
char *strptr;
```

Description:
XPDC outputs data to console or spool unit. This routine is different from other routines because instead of a null delimiter, the number of bytes is specified. Hence, you can output a buffer containing nulls or other binary characters.

```
 xpdc(39,"there is a null here->\0<--did it print?");
```

Notes:
XPDC goes outside any PDOS output formatting. Tabs are not expanded to blanks, the port row/column is not updated in SYRAM, and the output column counter is not updated in the TCB. This primitive is primarily intended for use on a serial port connected to another computer.

See Also:
XPCR - Put character raw
XPEL - Put encoded line to console
XPLC - Put line to console

XPEL

Put Encoded Line to Console

Format:
```
int xpel(buffer);
char *buffer;
```

Description:
XPEL outputs a line to the console or spool unit. It terminates on a null. No newline is added. The value returned is the address of the string.

An encoded line is like a regular ASCII string except that if a byte is negated, a space is printed after that byte; a null with the eighth bit set signals a carriage return/line feed; and up to 31 blanks may be printed by outputting the negative of the number of blanks. (i.e., -5 prints five blanks.) The intent is to save space by reducing the storage necessary for text strings containing lots of blanks.

```
/* \200 means carriage return/line feed, \373 means 5 spaces */
xpel("\200Testing\373Testing");
```

Notes:
Data compression probably does not make up for the loss of comprehension. Encoded data is really cryptic.

See Also:
XPDC - Put data to console
XPLC - Put line to console

XPLC

Put Line to Console

Format:

```
int xplc(buffer);
char *buffer;
```

Description:
XPLC outputs a line to console or spool unit. It terminates on a null. No newline is added. The value returned is the address of the string.

```
#define putstr xplc
putstr("hello there");
```

See Also:
XPDC - Put data to console
XPEL - Put encoded line to console

XPSC

Position Cursor

Format:

```
xpsc(row,col);
int row,col;
```

Description:
XPSC positions the cursor. The first parameter is the row, the second is the column. The 'row' and 'column' start at zero and range to 23 and 79 respectively. XPSC relies on the PSC$ field in the Task Control Block (_tcbptr->_psc) to tell what character sequence to output for the local terminal. This field is normally initialized by the MTERM utility, or it may be set under program control.

```
xpsc(0,0); xplc("This is the home position");
xpsc(12,40); xplc("hello from row 12,col 40");
```

See Also:
XCLS - Clear screen
XRCP - Read port cursor position
XTAB - Tab to column

XPSF

Position File

Format:

```
int xpsf(filid,position);
int filid;
long int position;
```

Description:
XPSF moves the file byte pointer to any byte position within a file. The index is a long word, so it can be a very large file. The file must have been opened via XROO, XROP, XNOP or XSOP. An error occurs if the 'position' points beyond the end of file. Errors are returned, and a 0 value is returned if there are no errors.

Positioning is much quicker in contiguous files. In non-contiguous files, the sector links must be followed to find the desired data.

```
err = xpsf(filid,10L);
```

Notes:
In PDOS versions 3.0 and later, you can position within a file regardless of the way that file was opened. In previous versions it was not possible to position within a file if it had been opened with XSOP.

See Also:
XRWF - Rewind file
XRFP - Read file position

LSEEK (STDLIB)

XPSP

Put Space to Console

Format:
```
asm("xpsp");
```

Description:
XPSP prints out a space to the console or spool unit.

Notes:
This function has limited utility in a high level language.

See Also:
XPBC - Put buffer to console
XPCC - Put character(s) to console
XPCL - Put CRLF

XRBF

Read Bytes From File

Format:
```
int xrbf(count,filid,buffer,bytesread);
long int count;
int filid;
char *buffer;
long int *bytesread;
```

Description:
XRBF reads a block of 'count' bytes from the file. On a read error, the number of bytes actually read is stored in the last parameter. If there is no error, the number of bytes read corresponds to the count given and the last parameter is not changed. The block is read into the buffer from the file specified by the 'filid'. The function returns zero or a PDOS error number.

```
 int filid
 long len;
 char buffer[14];
 .
 .
 .
 err = xrbf(14L,filid,buffer,&len);
```

See Also:
XRLF - Read line from file

XWBF - Write bytes to file
XWLF - Write line to file

XRCN

Reset Console Inputs

Format:
asm("xrcn");

Description:
XRCN resets console inputs. This routine is the same as the RC monitor command. It ends processing of an AC file.

When the program below is in an AC file, all the commands in the AC file that follow are ignored.

```
main()
{
 xplc("begin test");
 asm("xrcn");
 xplc("end test");
}
```

See Also:
XRST - Reset disk

XRCP

Read Port Cursor Position

Format:
```
xrcp(port,rowptr,colptr);
int port, *rowptr,*colptr;
```

Description:
XRCP reads the cursor position of a given port. There is no return value. The row value is stored at location specified by the second parameter; the column value is stored at the location specified by the third parameter. These values are maintained by PDOS, and are valid if the PDOS primitives (not user defined) have been used for console I/O. If the port is 0, the current task port is used.

```
 int row,col;
 xrcp(0,&row,&col);
```

Notes:
This call requires a port parameter but the output, input, and position calls do not.

See Also:
XCLS - Clear screen
XPSC - Position cursor
XTAB - Tab to column

XRDE

Read Next Directory Entry

Format:

```
int xrde(diskno,readflg,lastentptr,sectorptr);
int diskno,readflg;
long *lastentptr;
int *sectorptr;
```

Description:
XRDE reads the next directory entry. This routine is called to read consecutively through a disk directory. The first time it is called, 'readflg' should be 0. All other times it should be nonzero. 'Lastentptr' is the location where a pointer to the 32 byte directory entry is stored.

The include file "DIRENT:H" describes the different fields in a directory entry.

An error status is returned -- 0 for no error, or the error number.

Between calls do not alter 'lastentptr', the user I/O buffer, or TW1$,TW2$ of the TCB.

```
int i,err;
int diskno;
char *lastentry;
int sectornum;
diskno = getnum("Enter disk number ");
err = xrde(diskno,0,&lastentry,&sectornum);
for (i=0;err==0;i++){
    printf("\nEntry#%d: %s sector# %d",i,lastentry,sectornum);
    err = xrde(diskno,1,&lastentry,&sectornum);
}
if (err == 53)
    err = 0;
return err;
```

See Also:
XBFL - Build file directory list
XFFN - Fix file name
XLST - List file directory

XRDM

Dump Registers to Console

Format:
asm("xrdm");

Description:
XRDM dumps the address, data and status registers as well as the program counter to the console. This routine can be useful for debugging, especially for register variables.

```
register int i;
for (i=0;i<20;i++){
    asm("xrdm");
    printf("\n %d",i);
}

REGISTER DUMP: PC=00100606  SR=.....0....N..C
D0: 00100B9E 00000000 0000FFFF 00100500  00000011 00000007 00000000 00000000
A0: 00100B9E 00004B97 00100100 0019BF00  000003C2 00009800  0019BFE8 0019BFE0
 0
REGISTER DUMP: PC=00100606  SR=.....0....N..C
D0: 00190000 00000000 0000FFFF 00100500  00000011 00000007 00000000 00000001
A0: 00000000  0019BFBE 00100100 0019BF00  000003C2 00009800 0019BFE8 0019BFE0
 1
 .
 .
 .
REGISTER DUMP: PC=00100606  SR=.....0....N..C
D0: 00190000 00000012 0000FFFF 00100500  00000011 00000007 00000000 00000013
A0: 00000012 0019BFBE 00100100 0019BF00  000003C2 00009800  0019BFE8 0019BFE0
 19
```

See Also:
XBUG - Debug call
XDMP - Dump memory

XRDT

Read Date

Format:

```
char *xrdt();
```

Description:

XRDT reads the date from PDOS and returns an address to the string in the format: MN/DY/YR(null)

```
char *xrdt();
printf("\ntest xrdt: date=%s\n",xrdt());
```

Notes:

The string created is within the monitor work buffer, and overwrites or is overwritten by other primitives using the monitor work buffer.

See Also:

XFTD - Fix time & date
XPAD - Pack ASCII date
XRTM - Read time
XUAD - Unpack ASCII date
XUDT - Unpack date
XUTM - Unpack time
XWDT - Write date
XWTM - Write time

XRFA

Read File Attributes

Format:

```
int xrfa(filename,dirptr,diskptr,sizeptr,typptr);
char *filename;
long *dirptr;
int *diskptr;
long int *sizeptr,*typptr;
```

Description:
XRFA reads file attributes of a file. Input is the file name. The other parameters are pointers for where to store the output. The output is as follows, in order

```
     address of 32 byte directory entry (see DIRENT:H)
     disk number
     file size (in bytes)
     level/attributes (see PDOS Reference Manual under
                          XRFA for attribute description)
```

Errors are the return value. A zero returned indicates no errors.

```
#include "DIRENT:H"
struct DIRENT *buffer;
int disk,error;
long size,type;
char list[80];
getstr("Filename: ",list);
error = xrfa(list,&buffer,&disk,&size,&type);
if (!error){
    printf("\ndisk=%d",disk);
    printf("\nsize=%ld",size);
    printf("\ntype=0x%lx",type);
    printf("\nname: %.8s, ext: %.3s, level: %d",
        buffer->_fname,buffer->_fext,buffer->_level);
    printf("\nend of file sector %d",buffer->_seof);
    printf("\nend of file byte %d",buffer->_beof);
    printf("\ncreation time  %d",buffer->_ctime);
    printf("\ncreation date %d",buffer->_cdate);
    printf("\nupdate time %d",buffer->_utime);
    printf("\nupdate date %d",buffer->_udate);
}
```

Notes:

The directory entry created is within the monitor work buffer, and overwrites or is overwritten by other primitives using the monitor work buffer.

See Also:
XCFA - Close file w/attribute
XWFA - Write file attributes
XWFP - Write file parameters

XRFP

Read File Position

Format:

```
#include "FILESLOT:H";
int xrfp(filid,fsptrptr,posptr,eofptr);
struct FILESLOT **fsptrptr;   /* address of a pointer */
long int *posptr;             /* address of a long int */
long int *eofptr;             /* address of a long int */
int filid;
```

Description:
Given the 'filid', XRFP returns the current file slot address, the current byte position within the file, and the byte position of the end of file mark. XRFP is analogous to the LSEEK call, except that XRFP also returns a pointer to the file slot structure. The file slot retains various pieces of information, including the name of the file and the disk it is on.

```
#include "FILESLOT:H";
struct FILESLOT *fsptr;
long position;
long eof;
int fid;

if(err=xrfp(fid,&fsptr,&position,&eof))
    printf( "xrfp err %d\n",err);
else
    printf("the file %s:%.3s is at byte %ld",
             fsptr->name,fsptr->ext,position);
```

See Also:
XRWF - Rewind file
XPSF - Position file

XRLF

Read Line From File

Format:
```
int xrlf(filid,buffer,bytesread);
int filid,*bytesread;
char buffer[132];
```

Description:
XRLF reads a line from a file. It reads a line, delimited by a carriage return, into the buffer from the file specified by 'filid'. If a carriage return is not encountered after 132 characters, then the line and primitive are terminated.

Line feeds are dropped from the data stream, and the terminating carriage return is replaced by a null. On error return, the buffer data is not null terminated.

Errors are the return value, or a 0 is returned if there is no error. On an EOF error return, the third parameter specifies where the number of bytes actually read is stored.

```
int filid,err,bytesread;
int lastlinecnt;
char buffer[132];
.
.
.
err = xrlf(filid,buffer,&bytesread);
if(err = 56) lastlinecnt = bytesread;
```

Notes:
Be sure to leave enough room for the longest buffer or memory may be overwritten.

See Also:
XRBF - Read bytes from file
XWBF - Write bytes to file
XWLF - Write line to file

XRNF

Rename File

Format:

```
int xrnf(oldname,newname);
char *oldname,*newname;
```

Description:
XRNF renames a file from 'oldname' to 'newname'. It returns a 0 if no error, otherwise it returns the error (such as invalid filename).

```
 int err;
 err = xrnf("OLDNAME","NEWNAME");
```

See Also:
XDFL - Define file
XDLF - Delete file
XZFL - Zero file

RENAME - Rename file (STDLIB)

XROO

Open File Read-Only Random Access

Format:
```
int xroo(filename,filtype_ptr,filid_ptr);
char *filename;
int *filtype_ptr, *filid_ptr;
```

Description:
XROO opens a file for random read-only access. The first parameter is the pointer to the file name. If the file successfully opens, the file attribute is stored at '*filtype_ptr'. The filid is stored at '*filid_ptr'.

The returned value is an error #, or 0 if no errors.

```
int err,attribute,filid;
err = xroo("MYFILE:SR",&attribute,&filid);
```

See Also:
XSOP - Open sequential
XROP - Open random
XNOP - Open non-exclusive random

XCFA - Close file w/attribute
XCLF - Close file

XROP

Open File Random Access

Format:
```
int xrop(filename,filtype_ptr,filid_ptr);
char *filename;
int *filtype_ptr, *filid_ptr;
```

Description:
XROP opens a file for random access. The first parameter is the pointer to the file name. If the file successfully opens, the file attribute is stored at '*filtyp_ptr'. The filid is stored at '*filid_ptr'.

The returned value is an error #, or 0 if no errors.

```
 int err,attribute,filid;
 err = xrop("MYFILE:SR",&attribute,&filid);
```

See Also:
XSOP - Open sequential
XROO - Open random read only
XNOP - Open non-exclusive random

XCFA - Close file w/attribute
XCLF - Close file

XRPS

Read Port Status

Format:

```
int xrps(port,statusptr);
int port;
long int *statusptr;
```

Description:
XRPS reports the AC filid open to the current task (if any), the port flags (fwpi 8dcs), and the UART status byte in a 32-bit word. The upper sixteen bits are the AC filid. The next byte contains the port flags, and the lower 8 bits contain the UART status byte.

```
               PORT FLAGS
fwpi 8dcs
 \\\\ \\\\_ 0 = Enable ^S^Q software handshake
  \\\\ \\\_ 1 = Control character disable
   \\\\ \\_ 2 = Enable DTR hardware handshake
    \\\\ \_ 3 = 8-bit character enable
     \\\\__ 4 = Receiver interrupt enable
      \\\__ 5 = Even parity enable
       \\__ 6 = *Reserved (High/low water)
        \__ 7 = **Reserved (^S^Q flag bit)

               *Used to clear all bits
              **Used to set U2P$
```

The flag bits are the same set by the XBCP command. The port status is stored at the location in the second parameter. Port status has the format ACIF.W / Flag / Status.

The return value indicates the PDOS error, if nonzero.

```
long int portstatus;
int err;

err = xrps(1,&portstatus);
```

See Also:
XBCP - Baud console port
XSPF - Set port flag

XRSE

Read Sector From Disk

Format:
```
int xrse(disk,sector,buffer);
int disk,sector;
char buffer[256];
```

Description:
XRSE reads the sector number specified by the second parameter from the disk specified. The sector is copied to the buffer. This is a very low-level primitive, since most applications use PDOS files and their linkage system. The return value is an error, or 0.

```
 char buffer[256];
 err = xrse(8,2,buffer);
```

See Also:
XWSE - Write sector

XRST

Reset File

Format:
```
xrst(disk);
int disk;
```

Description:
XRST resets files. It closes all files on the disk. If the disk number is -1, then all files associated with the current task are closed. No errors are allowed to happen.

```
xrst(-1); /* close the files of this task */
xrst(filid >> 8); /* close all files on the disk this file is on */
```

Notes:
Be careful closing all files on the disk, since it may have bad effects on other tasks in the system. Similarly, if a program resets all files for that task, it aborts any procedure file currently in process, since the procedure file is closed.

See Also:
XRCN - Reset console inputs

XRTM

Read Time

Function:
char *xrtm();

Description:
XRTM reads the time from PDOS and returns an address to a string in the format: HR:MN:SC(null). This string is always 10 bytes long, counting the null. XRTM also returns the tics per second (B.TPS) in the word following the string (bytes 11 and 12), and the current value of the TICS counter (TICS.) in the long word after that (bytes 13-16).

```
struct TIME {
    char timestr[10];
    int tps;
    unsigned long tics;
} *xrtm();

printf("\ntest xrtm: time=%s\n",xrtm());
```

Notes:
The string created is within the monitor work buffer, and overwrites or is overwritten by other primitives using the monitor work buffer.

See Also:
XFTD - Fix time & date
XRDT - Read date
XRTP - Read time parameters
XUDT - Unpack date
XUTM - Unpack time
XWDT - Write date
XWTM - Write time

XRTP

Read Time Parameters

Format:

```
long xrtp(dateptr,timeptr,tpsptr);
long *dateptr,*timeptr,*tpsptr;
```

Description:
XRTP returns the different elements of the current time/date in a binary format. The call itself returns the current value of the SYRAM TICS variable.

The first parameter returns the month, day, and year as a series of bytes packed into a long word, with the month as the most significant byte, the day as the second byte, the last two digits of the year as the third, and a zero byte as the fourth.

The second parameter returns the hour, minute, and second as a similar series of bytes. The hour is the most significant byte, the minute is the second, the second is the third, and a zero byte is the fourth.

The third parameter returns the number of PDOS clock tics per second. This number is usually 100, but depending on the specific hardware may be 128 or some other value. Accurate time measurements based on the difference between two values of the TICS counter need to take the tics/second into account.

```
long tics,mdy,hms,tps,xrtp();
tics=xrtp(&mdy,&hms,&tps);
printf("\nTics=%ld \nmonth/day/year = 0x%lx",tics,mdy);
printf("\nhour/minute/second = 0x%lx \n Tics/second = %ld",hms,tps);
```

The following is displayed:

```
Tics=7007855
month/day/year = 0xA095600
hour/minute/second = 0xA293500
Tics/second = 100
```

See Also:
XRTM -- Read time
XFTD -- Fix time and date
XUTM -- Unpack time
XWTM -- Write time

XRTS

Read Task Status

Format:

```
int xrts(task);
int task;
```

Description:
XRTS reads the status of a task. The input is the task number. The status is returned. Possible values are:

0	task not executing
+N	running in time slice N
-N	suspended pending event N

If the input parameter is -1, the returned value is the current task number.

```
int i;

printf("\nmy task number is %d",xrts(-1));
for (i=0;i<16;i++){
       printf("\ntask: %d; status:  ",i,xrts(i))
}
```

See Also:
XSTP - Set/read task priority

XRWF

Rewind File

Format:

```
int xrwf(filid);
int filid;
```

Description:
XRWF rewinds a file. It returns an error number or 0 if no error. The parameter given is the 'filid' obtained from the open. Rewinding is defined as setting the file marker at the beginning of the file. It is the same as positioning to 0.

```
xsop("MYFILE:SR",&filtype, &filid);
for (i=0;i<4;i++){
    xrlf(filid,buffer,&len);
    printf("\ndata read =%s\n",buffer);
}
xrwf(filid);    /* rewind that file */
xrlf(filid,buffer,&len);
printf("\ndata read =%s\n",buffer);
```

Notes:
This call is shorthand for 'xpsf(filid,0L)'.

See Also:
XRFP - Read file position
XPSF - Position file

XSEF

Set/Clear Event Flag With Swap

Format:
```
int xsef(event);
int event;
```

Description:
XSEF sets an event flag and forces an immediate task swap. An event bit flag is set or reset depending on whether the 'event' is positive or negative. A swap is then executed.

The status of the event prior to the call is the returned value.

```
 printf("event 42 was %d, it is now 1",xsef(42));
 printf("event 41 was %d, now it is 0",xsef(-42));
```

See Also:
XDEV - Delay set/reset event
XSEV - Set event flag
XSUI - Suspend until interrupt
XTEF - Test event flag

XSEV

Set Event Flag

Format:
```
int xsev(event);
int event;
```

Description:
XSEV sets an event flag. An event bit flag is set or reset depending on whether the 'event' is positive or negative.

The status of the event prior to the call is the returned value.

```
printf("event 42 was %d, it is now 1",xsev(42));
printf("event 41 was %d, now it is 0",xsev(-42));
```

See Also:
XDEV - Delay set/reset event
XSEF - Set event flag w/swap
XSUI - Suspend until interrupt
XTEF - Test event flag

XSMP

Send Message Pointer

Format:

```
int xsmp(msgno,ptr);
int msgno;
char *ptr;
```

Description:
XSMP loads an empty message slot with a pointer to a message. If the slot is not empty, XSMP returns an error status of 83. The message may then be retrieved by any other task, (including the one originating it).

Only the address of the message is actually transmitted by this primitive. The text must be in a static or global buffer -- if it is in an automatic variable (on the stack), the data may be overwritten before the message is retrieved.

This type of message is also linked to the PDOS system events 64-79. When a message is sent via XSMP, the corresponding event (message number + 64) is set. When the message is received via XGMP, the corresponding event is cleared. A task may suspend on the message event and thus require no PDOS resources while it waits for a message.

```
static char list[]="This is a test message";
if (xsmp(5,list))
    printf("\nMessage not sent");
else
    printf("\nMessage sent");
```

See Also:
XGMP -- Get message pointer
XSTM -- Send task message
XGTM -- Get task message
XKTM -- Kill task message

XSOP

Open File Sequentially

Format:
```
int xsop(filename,filtype_ptr,filid_ptr);
char *filename;
int *filtype_ptr, *filid_ptr;
```

Description:
XSOP opens a file for sequential access. The first parameter is the pointer to the file name. If the file successfully opens, the '*filtype_ptr' indicates where the file attribute is stored. The 'filid' is stored at location '*filid_ptr'.

The returned value is an error number, or 0 if no errors.

```
int err,attribute,filid;
err = xsop("MYFILE:SR",&attribute,&filid);
```

See Also:
XROO - Open random read only
XROP - Open random
XNOP - Open non-exclusive random

XCFA - Close file w/attribute
XCLF - Close file

XSPF

Set Port Flag

Format:

```
int xspf(port,newflag,oldflgptr);
int port;
char newflag,*oldflgptr;
```

Description:

XSPF will set the flags for a port. The old port flag is stored at the location 'oldflgptr'. The return value indicates the PDOS error, if nonzero.

```
               PORT FLAGS
 fwpi 8dcs
  \\\\ \\\\_ 0 = Enable ^S^Q software handshake
   \\\\ \\\_ 1 = Control character disable
    \\\\ \\_ 2 = Enable DTR hardware handshake
     \\\\ \_ 3 = 8-bit character enable
      \\\\__ 4 = Receiver interrupt enable
       \\\__ 5 = Even parity enable
        \\__ 6 = *Reserved (High/low water)
         \__ 7 = **Reserved (^S^Q flag bit)

             *Used to clear all bits
            **Used to set U2P$


 char oldportflag;
 int err;

 err = xspf(1,0x02,&oldportflag); /* don't recognize escape or [CTRL-C]
                                           as break characters */
```

See Also:

XBCP - Baud console port
XRPS - Read port status

XSTM

Send Task Message

Format:
```
int xstm(task,buffer);
int task;
char *buffer;
```

Description:
XSTM sends a message to a task. If there are any errors, they are given in the return value. Normally, the buffer is 64 bytes long, and independent of format. Thus, it can be used to send binary data as well as ASCII. The 64 byte limitation can be changed by re-building PDOS.

```
int err;
err = xstm(3,"ready");
```

See Also:
XGMP - Get message pointer
XGTM - Get task message
XKTM - Kill task message
XSMP - Send message pointer

XSTP

Set Task Priority

Format:

```
int xstp(task,priority,retpriorityptr);
int task,priority;
int *retpriorityptr;
```

Description:

XSTP sets/reads the task priority. The first argument is the task. If it is -1, then the current task is specified. The second argument is priority, and should be 1-255, with 1 being lowest priority. If 'priority' is 0, then the current priority is returned in the location specified by the second parameter. If priority is nonzero, then the new priority is assigned to the task.

The upper byte of the priority determines the number of tics per time slice to be allocated to the task. Thus a task might run at a low priority, but execute for a long time when it gets in.

The return value is an error number, or 0 if no error.

```
 int priority;
 xstp(-1,0,&priority);
 printf("current task priority is %d",priority);
```

See Also:

XRTS - Read task status

XSUI

Suspend Until Interrupt or Event

Format:
```
int xsui(event);
int event;
```

Description:
XSUI suspends the current task until the specified event is set. The task is swapped out until the specified event occurs. No errors are possible, no value is returned. See the PDOS Reference Manual for further information on events and how they are set/reset. The XSUI description in that manual is very extensive.

```
xplc("waiting 10 seconds...");
err = xdev(1000L,128);
if(err)
  xerr(err);
xsui(128); /* sleep until event 128 */
xplc("time is up");
```

Notes:
The event parameter is actually a pair of bytes in the sixteen-bit field. To wait on two events, set one of the events in the high byte. To wait on an event clearing, AND the negative event with 0xff.

```
xsui(-50&0xff);          /* suspend until 50 clears */
```

See Also:
XDEV - Delay set/reset event
XSEF - Set event flag w/swap
XSEV - Set event flag
XTEF - Test event flag

XSUP

Enter Supervisor Mode

Format:
asm("xsup");

Description:
Some equipment requires that a program be in supervisor mode before direct access is allowed. If you have a program that must manipulate registers on such a board, you must drop into supervisor mode, move the data, and return to user mode. The only way supported under C to accomplish this is through an in-line assembly language system call to XSUP as shown.

Programs running in supervisor mode have a different A7 (stack pointer) than programs in user mode. Most variable accesses in C are made through the frame pointer (A6), so this shouldn't affect short programs. However, you may have problems if you enter supervisor mode and call other functions, since the supervisor stack is generally smaller than the user stack. You definitely will have problems if you enter supervisor mode and return from the current function without first going back to user mode -- the return address is not on the supervisor stack, it is on the user stack. Generally, programs should spend as little time in supervisor mode as possible.

```
/* modify a bit in an I/O register */
asm("xsup");
*(int *) 0xFFB102 |= 0x010;
asm("xusp");
```

See Also:
XUSP -- Return to user mode

XSWP

Swap To Next Task

Format:
```
asm("xswp");
```

Description:
XSWP relinquishes the time remaining in the current time slice to the next task ready to run. There is no input and no output.

See Also:
XLKT - Lock task
XULT - Unlock task

XSZF

Get Disk Parameters

Format:
```
int xszf(diskno,directory,used,free);
int diskno;
int used[2],free[2],directory[2];
```

Description:
XSZF returns the size of the disk. It returns important parameters about disk space. Input is the disk number in the first parameter. The second, third, and fourth parameters are addresses of double words that return with information about the directory, space used, and space free.

The first word of 'directory' returns with the total number of directory entries; the second word is the number of directory entries used (i.e., files on the disk). The first word of 'used' returns the number of blocks allocated to files; the second word returns the number of blocks used. The first word of 'free' returns the size of the largest free block; the second word returns the total amount of free space.

The return value is the error status (nonzero indicates the error number).

```
int diskno;
int used[2],free[2],directory[2];
diskno = getnum("enter disk number ");
error = xszf(diskno,directory,used,free);
printf("\nlargest free=%d, total free=%d",free[0],free[1]);
printf("\nallocated=%d, in use=%d",used[0],used[1]);
printf("\ndirectory size=%d, files=%d",directory[0],directory[1]);
```

XTAB

Tab to Column on Screen

Format:
```
xtab(col);
int col;
```

Description:
XTAB tabs the cursor right to the column specified. If PDOS thinks that the cursor is already past the indicated point, it does nothing.

```
int i;
printf("\nput tics from 1-79 at intervals of 5");
printf("\n01234567890123456789012345678901234567890");
printf("1234567890123456789012345678901234567 8\n");
for (i=0;i<79;i+=5){
    xtab(i);
    xpcc('|');
}
```

Notes:
Currently, only one tab column counter is maintained per task, regardless of how many ports are used for I/O. As such, I/O to separate ports should be performed one line at a time, or the output column counter (CNT$) in the TCB should be saved/restored for the output to each port.

The tab cursor primitive only tabs to the right. Giving a tab column that is less than the current column has no effect. Use XPSC if a more general application is needed.

See Also:
XCLS - Clear screen
XPSC - Position cursor
XRCP - Read port cursor position

XTEF

Test Event Flag

Format:
```
int xtef(event);
int event;
```

Description:
XTEF tests an event flag. The return value is 0 if the event is reset and 1 if it is set.

```
if(xtef(42)) xpcc("event 42 is set");
else xpcc("event 42 is reset");
```

See Also:
XDEV - Delay set/reset event
XSEF - Set event flag w/swap
XSEV - Set event flag
XSUI - Suspend until interrupt

XUAD

Unpack ASCII Date

Format:

```
char *xuad(date);
unsigned int date;
```

Description:
XUAD converts a binary encoded date to an ASCII string in the format dd-mon-yy. Invalid dates are indicated by non-numeric characters in the day and year fields, or by "???" in the month field.

```
char *xuad();
printf("\n%s",xuad(65535));
printf("\n%s",xuad(39143));
```

The following is displayed:

```
31-???-<7
Ø7-Jul-76
```

See Also:
XRDT -- Read date
XWDT -- Write date
XFTD -- Fix time and date
XUDT -- Unpack date
XPAD -- Pack ASCII date

XUDT

Unpack Date Into String

Format:
char *xudt(date);
int date;

Description:
XUDT converts a binary encoded date into an eight character null-delimited string in the form MN/DY/YR. The return value is the string address.

```
char *xudt();
int time,date;
xftd(&time,&date);
printf("\nDate= %s",xudt(date));
```

Notes:
The string created is within the monitor work buffer, and will overwrite or be overwritten by other primitives using the monitor work buffer.

See Also:
XFTD - Fix time & date
XPAD - Pack ASCII date
XRDT - Read date
XRTM - Read time
XUAD - Unpack ASCII date
XUTM - Unpack time
XWDT - Write date
XWTM - Write time

XULF

Unlock File

Format:
```
int xulf(filid);
int filid;
```

Description:
XULF unlocks a file locked by XLKF. The returned value is an error code, or 0 if no error occurred. The lock and unlock file commands are used when tasks open a file in non-exclusive mode. In this mode, they share the file slot, and both have read/write access to the file. To prevent conflicts, each task must lock the file, position to the desired record number, read or write, and unlock the file.

```
int filtype,filid,err;
char fname[] = "MYFILE:SR";
 if(err=xnop(fname,&filtype,&filid))    /* shared access open */
     _error("\nerror %d on file %s",err,fname);

  ...
while ((err = xlkf(filid)) == 75){
     asm("xswp");  /* hang while locked */
}
if (err)
    _error("\nerror %n locking file %s",err,fname);

/* position, read/write, etc. */

if(err = xulf(filid))
    _error("\nerror %n unlocking file %s",err,fname);
```

See Also:
XLKF - Lock file
XNOP - Open non-exclusive random
XRFP - Read file position
XPSF - Position file

XULT

Unlock Task

Format:
asm("xult");

Description:
XULT unlocks the current task. A locked task has the undivided attention of the CPU for executing instructions. No other task is given a time slice. The task must be unlocked for normal multi-tasking to resume. The returned value is the status of the swap lock variable before the call was made: 0 is unlocked, nonzero is locked.

```
int lockstate;
lockstate = xlkt();
printf("previous state was %d",lockstate);

/* in here do critical code */

asm("xult");
printf(" now normal task swapping resumes");
```

See Also:
XLKT - Lock task
XSWP - Swap to next task

XUSP

Return to User Mode

Format:

```
asm("xusp");
```

Description:

It is occasionally necessary to drop into supervisor mode in order to access some types of hardware, or to execute privileged instructions. To return to user mode it is possible to clear the supervisor bit directly from the status register, but the XUSP primitive offers a more convenient method.

```
/* modify a bit in an I/O register */
asm("xsup");
*(int *) 0xFFB102 |= 0x010;
asm("xusp");
```

See Also:

XSUP -- Enter supervisor mode

XUTM

Unpack Time Into String

Format:

```
char *xutm(time);
int time;
```

Description:
XUTM converts a binary encoded time value into an eight character, null-delimited string in the form HR:MN:SC. The returned value is the string address.

```
char *xutm();
int time,date;
xftd(&time,&date);
printf("\ntime= %s",xutm(time));
```

Notes:
The string created is within the monitor work buffer, and will overwrite or be overwritten by other primitives using the monitor work buffer.

See Also:
XFTD - Fix time & date
XRDT - Read date
XRTM - Read time
XUDT - Unpack date
XWDT - Write date
XWTM - Write time

XWBF

Write Bytes to File

Format:

```
int xwbf(count,filid,buffer);
long int count;
int filid;
char *buffer;
```

Description:
XWBF writes 'count' bytes to a file. 'count' specifies the number of bytes to write. A 'count' of 0 results in no data being written. If necessary, a contiguous file is extended and converted to a non-contiguous file. Any errors are returned, and a return value of 0 means no errors.

```
err = xwbf(14L,filid,"hello, world!/n");
```

See Also:
XRBF - Read bytes from file
XRLF - Read line from file

XWLF - Write line to file

XWDT

Write Date to System Clock

Format:
```
xwdt(mn,day,year);
int mn,day,year;
```

Description:
XWDT resets the operating system clock with a new date.

```
 char *xrdt();
 printf("\nbefore date=%s\n",xrdt());
 xwdt(2,14,53);
 printf("\nafter  date=%s\n",xrdt());
```

Notes:
The clock is reset for all users on the system.

See Also:
XFTD - Fix time & date
XPAD - Pack ASCII date
XRDT - Read date
XRTM - Read time
XUAD - Unpack ASCII date
XUDT - Unpack date
XUTM - Unpack time
XWTM - Write time

XWFA

Write File Attributes

Format:

```
int xwfa(filename,attributes);
char *filename,*attributes;
```

Description:

XWFA sets the file attributes on a file. The ASCII string of file attributes is assigned to the file. Any errors are returned; 0 is returned if there are no errors.

```
AC - Procedure file
BN - Binary file
OB - Object file
SY - Memory Image of machine code
BX - BASIC token file
EX - BASIC ASCII file
TX - Text file
DR - System I/O driver

*  - Delete protect
** - Delete/write protect
```

```
int err;
err = xwfa("MYFILE","BN**");  /* make file binary and protected */
```

Notes:

XCFA, XRFA, and XWFA do not use the same format.

See Also:

XCFA - Close file w/attribute
XRFA - Read file attributes

XWFP

Write File Parameters

Format:

```
int xwfp(eofsec,create,update,attr,filename);
long eofsec;                /* sector / byte */
long create,update,attr;    /* time / date */
char *filename;
```

Description:

XWFP is an operating system internal call used by the TF monitor command to assign a copy of a file the same creation/update time and date as the original of the file. It could also be used to modify the end of file mark on a file.

The first three parameters are all pairs of data. 'eofsec' is a long word with the end of file sector in the upper word and the end of file byte in the lower word. 'create' is a long word with the creation time in the upper word and the creation date in the lower word. 'update' is in the same format as 'create'. 'filename' is a pointer to a string containing the file name. 'attr' is a long word, but only the second half is used. This word has the attributes in the upper byte and the delete-/write-protect flags in the lower byte. The "contiguous" flag and the "file altered bit" are not overwritten by this call. 'xwfp' returns zero or a PDOS error number.

```
union{
    long l;
    struct {
        int time;
        int date;
    };
} create,update;
union{
    long l;
    struct{
        int sector;
        int byte;
    };
} eof;
```

continued . . .

```
char line[80];
long attr;
getstr("enter file name",line);    /* get new values */
eof.sector = getnum("end of file sector");
eof.byte = getnum("end of file byte");
create.time = getnum("creation time");
create.date = getnum("creation date");
update.time = getnum("update time");
update.date = getnum("update date");
attr = getnum("attribute");
return(xwfp(eof.l,create.l,update.l,attr,line));    /* write it */
}
```

Notes:
This function has limited utility.

See Also:
XRFA - Read file attributes

XWLF

Write Line to File

Format:

```
int xwlf(filid,buffer);
int filid;
char *buffer;
```

Description:
XWLF writes a string to a file. It writes out until a null character is found. If necessary, a contiguous file is extended and converted to a non-contiguous file. Any errors are returned, and a return value of 0 means no errors.

```
 err = xwlf(filid,"hello, world!/n");
```

See Also:
XRBF - Read bytes from file
XRLF - Read line from file

XWBF - Write bytes to file

XWSE

Write Sector

Format:

```
int xwse(disk,sector,buffer);
int filid;
unsigned int sector;
char buffer[256];
```

Description:
XWSE writes a sector to a disk. This is a very low-level primitive, since most applications use PDOS files and their linkage system. The return value is an error, or else 0.

```
char buffer[256];
unsigned int  i;
for(i=0;i<256;i++) buffer[i]=0; /* zero the sector */
err = xwse(8,2,buffer);
```

Notes:
This call is potentially destructive. Be sure to have a scratch disk in the drive during experimentation!

See Also:
XRSE - Read sector

XWTM

Write Time to PDOS

Format:

```
xwtm(hr,min,sec);
int hr,min,sec;
```

Description:
XWTM resets the operating system clock with a new time.

```
 char *xrtm();
 printf("\nbefore time=%s\n",xrtm());
 xwtm(17,30,01);
 printf("\nafter  time=%s\n",xrtm());
```

Notes:
The clock is reset for all users on the system.

See Also:
XFTD - Fix time & date
XRDT - Read date
XRTM - Read time
XUDT - Unpack date
XUTM - Unpack time
XWDT - Write date

XZFL

Zero File

Format:

```
int xzfl(filename);
char *filename;
```

Description:
XZFL zeroes a file. It clears a file of any data. If the file is already defined, the end of file marker is set to the beginning. If the file is not defined, it is defined with no data. The return value indicates an error if nonzero.

```
err = xzfl("NEWFILE");
```

See Also:
XDFL - Define file
XDLF - Delete file
XRNF - Rename file

CHAPTER FOUR
FLOATING POINT LIBRARIES

The overhead for handling floating point is significant, so C programs which have no need of floating point should not use it. Currently there is little compatibility between the floating point formats supported under the different language systems in PDOS. Future versions of all PDOS languages will offer compatibility through the IEEE floating point format.

There are three different floating point libraries supported under PDOS C -- the Motorola Fast Floating Point (supported by the FFP:LIB library), the IEEE floating point (supported by the IEEE:LIB library), and the MC68881 hardware floating point (supported by the M68881:LIB library). The following discussion explains the differences between each library.

The supported functions for the three libraries are the same, with the same interface, so only one discussion is given for each function. The subroutines are listed alphabetically. For each function, the calling sequence and parameters are given together with a brief description of the function.

FAST FLOATING POINT LIBRARY -- FFP:LIB

The library FFP:LIB is referenced by CC when the 'F' option is used. It supplies those modules required by programs using floating point, and maintains the numbers in Motorola Fast Floating Point format.

IEEE FLOATING POINT FORMAT LIBRARY -- IEEE:LIB

The library IEEE:LIB is referenced by CC when the 'E' option is used. It supplies those modules required by programs using floating point, and maintains the numbers in IEEE 32-bit or 64-bit floating point format.

MC68881 FLOATING POINT LIBRARY -- M68881:LIB

The library M68881:LIB is referenced by CC when the 'H' option is used. It supplies those modules required by programs running on a 68020 and using a 68881 math co-processor. The numbers are maintained in IEEE format, the same used by the IEEE:LIB library.

FLOATING POINT FORMATS

PDOS C supports two formats of floating point numbers; the Motorola Fast Floating Point (FFP) and the IEEE standard. They are not compatible.

The FFP format is generated by the compiler when the 'F' option is specified. If the 'E' or 'H' option is specified, the IEEE format is specified.

FFP floating point numbers are only available in single-precision. An FFP number is a 32-bit quantity in the following format:

```
 3 3 2 2 2 2 2         1 1 1 1 1 1
 1 0 9 8 8 7 6         5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
[ | | | | | | | ---- | | | | | | | | |S| | | | | | | ]
<-------------Fraction-------------->|S|<---- Exp --->
```

Here, the bottom byte is given to a seven-bit exponent and a one-bit sign. The remaining 24 bits are the fraction. The exponent is stored in two's complement form. There are no implied bits -- an unnormalized number is possible.

IEEE floating point numbers are available in single-precision (32-bit) and double-precision (64-bit) format.

The single precision format is as follows:

```
 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1
 1 0 9 8 8 7 6 5 4 3 2 1 0 9 8 7        6 5 4 3 2 1 0
[ | | | | | | | | | | | | | | | | ---- | | | | | | | ]
|S|<----- Exp --->|<----------- Fraction ----------->
```

The most significant bit is the sign, followed by an eight-bit exponent and a 23-bit fraction.

The double precision format is as follows:

```
 6 6 6 6 5 5 5 5 5 5 5 5 5 5 4 4 4
 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7      6 5 4 3 2 1 0
[ | | | | | | | | | | | | | | | | ---- | | | | | | | ]
|S|<------- Exp ------->|<----- Fraction ------------>
```

The most significant bit is the sign, followed by an eleven-bit exponent and a 52-bit fraction.

In both IEEE formats the exponent is biased (excess 128 in single-precision and excess 1024 in double precision) and the fraction is normalized, with an implied leading one bit. Thus, the effective size of the fraction is 24 bits for single precision, and 53 bits for double precision.

(Floating Point Formats cont.)

The three formats offer the following range of representable numbers:

	significant digits	max	min
FFP single precision	7	10^{19}	10^{-19}
IEEE single precision	7	10^{38}	10^{-38}
IEEE double precision	15	10^{308}	10^{-308}

Note: The FFP library may be slightly faster and smaller for those programs that do not need double precision. The M68881 library is preferable for both single and double precision where there is a 68020 CPU with 68881 co-processor. The IEEE library offers the advantage of double precision and compatibility with a standard format.

Programs can be written to run with more than one floating point format by isolating format-sensitive code under the preprocessor directives "#ifdef/#endif". The 'F' option defines the symbol FFP; the 'E' option defines the symbol IEEE; and the 'H' option defines the symbol M68881.

DECLARATIONS OF MATH ROUTINES

'math.h' contains the declarations necessary for the floating point mathematical routines, as well as for the floating point and long conversion routines.

```
/* math.h -- Declarations of Math Routines */

long atol();
char *ltoa();

double atof();
char *ftoa();        /* float to %f format */
char *etoa();        /* float to scientific notation format */

double sin();
double cos();
double tan();
double atan();
```

(Declarations of Math Routines cont.)

```
double fabs();
double floor();
double ceil();
double fmod();
double log10();
double log();
double pow();
double sqrt();
double exp();
double modf();
double ldexp();
double frexp();
double atan2();

#ifdef FFP
#define asin(x) atan2((x),sqrt(1-(x)*(x)))
#define acos(x) (1.570796 - atan2((x),sqrt(1-(x)*(x))))
#else
double asin();
double acos();
#endif
```

ACOS

Arc Cosine

Format:
```
double acos(farg)
double farg;
```

Description:
'acos' returns the arc cosine of its argument. The input parameter is a double precision number of radians.

Notes:
'acos' is implemented as a macro in 'math.h' under the 'F' option. It is a function in IEEE:LIB under the 'E' option and is an in-line 68881 instruction under the 'H' option.

See Also:
ASIN -- arc sine

ASIN

Arc Sine

Format:

```
double asin(farg)
double farg;
```

Description:
'asin' returns the arc sine of its argument. The input parameter is a double precision number of radians.

Notes:
'asin' is implemented as a macro in 'math.h' under the 'F' option. It is a function in IEEE:LIB under the 'E' option and is an in-line 68881 instruction under the 'H' option.

See Also:
ACOS -- arc cosine

ATAN

Floating Point Arc Tangent

Format:
```
double atan(farg)
double farg;
```

Description:
'atan' returns the arctangent of its argument. The input parameter is a double precision number of radians.

```
#include <math.h>

main()
{
   double d;

   d = atan(2.2);          /* returns 1.1442 */
}
```

See Also:
ATAN2 -- arc tangent of x/y

ATAN2

Arc Tangent of X/Y

Format:
```
double atan2(x,y)
double x, y;
```

Description:
'atan2' returns the arc tangent of x/y.

```
#include <math.h>

main()
{
   double d;

   d = atan2(1.7,2.9);            /* returns 0.5302 */
}
```

See Also:
ATAN -- floating point arc tangent

ATOF

ASCII to Floating Point

Format:
```
double atof(buf)
char *buf;
```

Description:
'atof' converts an ASCII string into its floating point representation where the string is of the following format:

```
{sign}{digits}{'.'}{digits}{E}{sign}{digits}
```

Both signs and the exponent string are optional. The decimal point is optional, but may appear at any point in the digit string.

```
#include <math.h>

main()
{
   double d;
   char *s;

   d = atof("-23.1E+4");
   d = atof("+0.6245E-1");
   d = atof("23456789");
   d = atof("2.33");
   s = "2.3145";
   d = atof(s);
}
```

See Also:
ATOI -- ASCII to integer

ATOI

ASCII to Integer

Format:
```
int atoi(s)
char *s;
```

Description:
'atoi' converts an ASCII string to a sixteen-bit signed integer.

See Also:
ATOF -- ASCII to floating point

CEIL

Smallest Integer Not Less Than X

Format:
```
double ceil(x)
double x;
```

Description:
'ceil' returns the smallest integer (as a double precision floating point number) not less than x.

```
#include <math.h>

main()
{
   double x;

   x = ceil(4.2);          returns 5.0
   x = ceil(9.0);          returns 9.0
   x = ceil(3.9);          returns 4.0
   x = ceil(-2.9);         returns -2.0
   x = ceil(5.0001);       returns 6.0
```

See Also:
FLOOR -- largest integer not greater than x
FABS -- floating point absolute value
FPNEG -- floating point negation

COS

Cosine

Format:
double cos(farg)
double farg;

Description:
'cos' returns the cosine of the argument.

The argument is a double precision floating point number, interpreted as the angle in radians.

```
#include <math.h>

main()
{
   double d;

   d = cos(2.2);            /* returns -0.5885 */
}
```

See Also:
SIN -- sine function
TAN -- tangent function

COSH

Hyperbolic Cosine

Format:
```
double cosh(farg)
double farg;
```

Description:
'cosh' returns the hyperbolic cosine of the argument.

The argument is a double-precision floating point number representing the angle in radians.

See Also:
SINH -- hyperbolic sine
TANH -- hyperbolic tangent

ETOA

Floating Point to Scientific Notation
ASCII 'e' Format

Format:

```
char *etoa(f,buf,prec)
double f;
char *buf;
int prec;
```

Description:
'etoa' converts a float into its ASCII exponential representation. Where 'fp' is a double precision floating point number, 'buf' is the buffer in which to return the string, and 'prec' is the precision of the decimal places. A pointer to the beginning of 'buf' is returned.

If the precision is specified to be zero or negative, then the default precision of six decimal places will be used.

```
#include <math.h>
main()
{
   char buffer[25], *p;

   p = etoa(4.23,buffer,1);      /* "4.2e00" */
   p = etoa(54.9,buffer,3);      /* "5.490e01" */
   p = etoa(-.003,buffer,3);     /* "-3.000e-03" */
```

See Also:
FTOA -- floating point to ASCII 'f' format

EXP

Exponent

Format:

```
double exp(x)
double x;
```

Description:

'exp' returns : e ^ x (where e = 2.718...).

```
#include <math.h>

main()
{
   double x;

   x = exp(2.1245);          /* returns 8.3687 */
```

See Also:

POW --	floating point power
LOG --	natural logarithm
LOG10 --	common logarithm
SQRT --	square root

FABS

Floating Point Absolute Value

Format:
double fabs(x)
double x;

Description:
'fabs' returns the absolute value (as a double precision floating point number) of x.

```
#include <math.h>

main()
{
   double x;

   x = fabs(33.1);          /* returns 33.1 */
   x = fabs(-45.2);         /* returns 45.2 */
}
```

See Also:

CEIL --	smallest integer not less than x
FLOOR --	largest integer not greater than x
FPNEG --	floating point negation

FLOOR

Largest Integer Not Greater Than X

Format:

```
double floor(x)
double x;
```

Description:
'floor' returns the largest integer (as a double precision floating point number) not greater than x.

```
#include <math.h>

main()
{
   double x;

   x = floor(32.3);          /* returns 32.0 */
   x = floor(23.99);         /* returns 23.0 */
   x = floor(15.01);         /* returns 15.0 */
   x = floor(-33.3);         /* returns -34.0 */
}
```

See Also:
CEIL -- smallest integer not less than x
FABS -- floating point absolute value
FPNEG -- floating point negation

FMOD

Floating Point Modulus

Format:

```
double fmod(x,y)
double x,y;
```

Description:
'fmod' returns the number f such that

```
        ( x - iy + f ) and ( 0 <= f <= y ).
```

```
#include <math.h>

main()
{
   double x;

   x = fmod(3.0,39.0);          /* returns 3.0 */
}
```

See Also:
FPADD -- floating point add
FPCMP -- floating point compare
FPDIV -- floating point divide
FPMUL -- floating point multiply
FPSUB -- floating point subtraction

FPADD

Floating Point Add

Format:

```
double fpadd(addend,adder)
double addend, adder;
```

Description:
'fpadd' adds the two operands and returns the sum.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:
FPCMP -- floating point compare
FPDIV -- floating point divide
FMOD -- floating point modulus
FPMUL -- floating point multiply
FPSUB -- floating point subtraction

FPCMP

Floating Point Compare

Format:

```
int fpcmp(source,dest)
double source, dest;
```

Description:
'fpcmp' compares the two operands and returns zero if they are equal. It returns a positive integer if the source is greater than destination and a negative integer if the source is less than the destination.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:
FPADD -- floating point add
FPDIV -- floating point divide
FMOD -- floating point modulus
FPMUL -- floating point multiply
FPSUB -- floating point subtraction

FPDIV

Floating Point Divide

Format:

```
double fpdiv(divisor,dividend)
double divisor, dividend;
```

Description:
'fpdiv' divides the dividend by the divisor and returns the quotient.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:
FPADD -- floating point add
FPCMP -- floating point compare
FMOD -- floating point modulus
FPMUL -- floating point multiply
FPSUB -- floating point subtraction

FPFTOL

Floating Point Float to Long

Format:
```
long fpftol(fparg)
double fparg;
```

Description:
'fpftol' returns the long integer representation of a floating point number.

Notes:
This function is internal. It is normally used only by the support routines of the floating-point library. It is available for use at your discretion.

See Also:
FPLTOF -- long to floating point conversion

FPLTOF

Long to Floating Point

Format:

```
double fpltof(larg)
long larg;
```

Description:
'fpltof' returns the floating point representation of a long integer.

Notes:
This function is internal. It is normally used only by the support routines of the floating-point library. It is available for use at your discretion.

See Also:
FPFTOL -- floating point float to long

FPMUL

Floating Point Multiply

Format:

```
double fpmul(multiplier,multiplicand)
double multiplier, multiplicand;
```

Description:
'fpmul' returns the product of the two operands.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:
FPADD -- floating point add
FPCMP -- floating point compare
FPDIV -- floating point divide
FMOD -- floating point modulus
FPSUB -- floating point subtraction

FPNEG

Floating Point Negation

Format:

```
double fpneg(x)
double x;
```

Description:
'fpneg' returns the negative (as a double precision floating point number) of x.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:

CEIL --	smallest integer not less than x
FLOOR --	largest integer not greater than x
FABS --	floating point absolute value

FPSUB

Floating Point Subtraction

Format:
```
double fpsub(subtrahend,minuend)
double subtrahend, minuend;
```

Description:
'fpsub' returns the difference of the two operands.

Notes:
This function is internal. The compiler normally generates calls to it automatically when the code contains such operations on floating-point variables. It is included here for reference.

See Also:
```
FPADD -- floating point add
FPCMP -- floating point compare
FPDIV -- floating point divide
FMOD --  floating point modulus
FPMUL -- floating point multiply
```

FREXP

Miscellaneous Exponential

Format:

```
double frexp(fp,ptr)
double fp
int *ptr;
```

Description:
'frexp' returns the signed value of 'fp' reduced to the range '0.5 - 1.0' and stores the integral value which 2 raised to this power times the returned value will result in the original value (e.g. fp = return * (2 ^ *ptr)).

```
#include <math.h>

main()
{
   double d,d2;
   int i;

   d = modf(27.3421,&d2);        /* d <= 0.3421, d2 <= 27.0 */
   d = ldexp(10.0,4);            /* d <= 160.0 */
   i = 4;
   d = frexp(160.0,&i);          /* d <= 0.625 */
}
```

Notes:
This function is internal. It is normally used only by the support routines of the floating-point library. It is available for use at your discretion.

See Also:
LDEXP -- miscellaneous exponential function
MODF -- miscellaneous exponential function

FTOA

Floating Point to ASCII 'f' Format

Format:

```
char *ftoa(f,buf,prec)
double f;
char *buf;
int prec;
```

Description:
'ftoa' converts a float into its ASCII representation where 'fp' is a double precision floating point number, 'buf' is the buffer in which to return the string, and 'prec' is the precision of the decimal places. If the specified precision is zero then no decimal point is printed. If the precision is negative, then the default precision (6) will be used. A pointer to the beginning of 'buf' is returned.

```
#include <math.h>

main()
{
   char *p, buf[25];

   p = ftoa(3.2,buf,3);          /* "3.200" */
   p = ftoa(-1.54,buf,4);        /* "-1.5400" */
}
```

See Also:
ETOA -- floating point to scientific notation ASCII 'e' format

LDEXP

Miscellaneous Exponential

Format:
```
double ldexp(fp,exp)
double fp;
int exp;
```

Description:
'ldexp' returns the computation of (fp * (2 ^ exp)). If the exponent is larger than 31 it is set to 31.

```
#include <math.h>

main()
{
   double d,d2;
   int i;

   d = modf(27.3421,&d2);          /* d <= 0.3421, d2 <= 27.0 */
   d = ldexp(10.0,4);              /* d <= 160.0 */
   i = 4;
   d = frexp(160.0,&i);            /* d <= 0.625 */
}
```

Notes:
This function is internal. It is normally used only by the support routines of the floating-point library. It is available for use at your discretion.

See Also:
FREXP -- miscellaneous exponential functions
MODF -- miscellaneous exponential functions

LOG

Natural Logarithm

Format:
double log(x)
double x;

Description:
'log' returns the computed logarithm base 'e' of its argument.

```
#include <math.h>

main()
{

   double x;

   x = log(25.0);      /* returns 3.2188 */
}
```

See Also:

EXP --	exponent function
POW --	floating point power function
LOG10 --	common logarithm
SQRT --	square root function

LOG1Ø

Common Logarithm

Format:

```
double log10(d)
double d;
```

Description:
'log10' returns the computed logarithm base ten of its argument.

```
#include <math.h>

main()
{

   double x;

   x = log1Ø(25.Ø);         /* returns 1.3979 */
}
```

See Also:

EXP --	exponent function
POW --	floating point power function
LOG --	natural logarithm
SQRT --	square root function

MODF

Miscellaneous Exponential

Format:
```
double modf(fp,ptr)
double fp, *ptr;
```

Description:
'modf' stores the fixed point portion of 'fp' in 'ptr' and returns the mantissa portion.

```
#include <math.h>

main()
{
   double d,d2;
   int i;

   d = modf(27.3421,&d2);        /* d <= 0.3421, d2 <= 27.0 */
   d = ldexp(10.0,4);            /* d <= 160.0 */
   i = 4;
   d = frexp(160.0,&i);          /* d <= 0.625 */
}
```

Notes:
This function is internal. It is normally used only by the support routines of the floating-point library. It is available for use at your discretion.

See Also:
FREXP -- miscellaneous exponential function
LDEXP -- miscellaneous exponential function

POW

Floating Point Power

Format:

```
double pow(x,y)
double x, y;
```

Description:

'pow' returns : x ^ y.

```
#include <math.h>

main()
{
   double x;

   x = pow(2.0,6.0);        /* returns 64.0 */
}
```

See Also:

```
EXP --   exponent function
LOG --   natural logarithm
LOG10 -- common logarithm
SQRT --  square root function
```

SIN

Sine

Format:
double sin(farg)
double farg;

Description:
'sin' returns the sine of the argument.

The argument is a double precision floating point number, interpreted as the angle in radians.

```
#include <math.h>

main()
{
   double d;

   d = sin(2.2);              /* returns Ø.8Ø85 */
}
```

See Also:
COS -- cosine
TAN -- tangent function

SINH

Hyperbolic Sine

Format:

```
double sinh(farg)
double farg;
```

Description:
'sinh' returns the hyperbolic sine of the argument.

The argument is a double-precision floating point number representing the angle in radians.

See Also:
COSH -- hyperbolic cosine
TANH -- hyperbolic tangent

SQRT

Square Root

Format:
double sqrt(farg)
double farg;

Description:
'sqrt' returns the square root of its argument.

```
#include <math.h>
{
   double d;

   d = sqrt(39.0);          /* returns 6.244 */
   d = sqrt(25.0);          /* returns 5.0 */
}
```

See Also:

EXP --	exponent function
POW --	floating point power function
LOT --	natural logarithm
LOG10 --	common logarithm

TAN

Tangent

Format:
```
double tan(farg)
double farg;
```

DESCRIPTION
'tan' returns the tangent of the argument.

The argument is a double precision floating point number, interpreted as the angle in radians.

```
#include <math.h>

main()
{
   double d;

   d = tan(2.2);            /* returns -1.3738 */
}
```

See Also:
COS -- cosine
SIN -- sine function

TANH

Hyperbolic Tangent

Format:

```
double tanh(farg)
double farg;
```

Description:
'tanh' returns the hyperbolic tangent of the argument.

The argument is a double-precision floating point number representing the angle in radians.

See Also:
COSH -- hyperbolic cosine
SINH -- hyperbolic sine

PDOS C
ERROR MESSAGES

Error messages are generated by all passes of the compiler. They are preceded by a symbol distinguishing the pass of the compiler in which the error exists and the line number on which the error occurred. Pre-processor error messages are preceded by a pound sign '#.' Parser error messages begin with a single asterisk '*.' Code generator error messages begin with a double asterisk '**.'

Pre-Processor Error Messages

Argument Buffer Overflow
: The total length of the arguments in a macro reference exceeds the compiler's capacity to store the arguments prior to substitution.

Bad Argument (NAME)
: The name enclosed in parentheses is invalid.

Bad Character (CHARACTER)
: The character enclosed in parentheses is not a valid token.

Bad Define Name (NAME)
: The indicated control line is missing a required name or the name is invalid.

Bad Include File (FILENAME)
: The INCLUDE filename must be in double quotes ("").

Bad Include Filename (FILENAME)
: The #include line is either missing a filename or specifies one that is syntactically invalid. The line is ignored by the compiler.

Can't Create File (FILENAME)
: The compiler is unable to open the specified output file, possibly because of directory protection attributes.

Can't Open Include File (FILENAME)
: The INCLUDE filename is not present, or is already open.

Can't Open Source File (FILENAME)
: The compiler cannot find or open the source file. You should check to see if the file exists or change the file specifications in the program to that of an existing file.

Condition Stack Overflow
: The nesting level in conditional compilation statements has exceeded the maximum number.

(Pre-Processor Error Messages cont.)

Define Recursion
    Macro subsitution cannot be performed during the scan of a macro reference.

Define Table Overflow
    The number of DEFINE statements has exceeded the maximum (1024) symbols.

Expressions Operator Stack Overflow
Expressions Stack Overflow
    The specified expressions is too complex.

Expression Syntax
    The specified expression is invalid.

Includes Nested Too Deeply
    Include files may be nested to a maximum of ten levels.

Invalid #else
    The specified #else control line is invalid, possibly because of a missing #if.

Invalid #endif
    The compiler did not encounter an #if, #ifdef, or #ifndef to match with the current #endif.

Invalid Pre-processor Command
    The pre-processor control line contains a missing or invalid keyword. The line is ignored by the compiler.

Line Overflow
    Pre-processor control lines must not exceed a single line.

Macro Argument Too Long
    Macro arguments must not exceed eight characters in length.

No */ Before EOF
    The compiler has encountered an EOF before the end of a comment.

String Cannot Cross Line
    Strings may not extend to multiple lines.

String Too Long
    A character string exceeds the maximum of 1024 characters.

Symbol Table Overflow
    The objects defined in this compilation have used up all of the address space allocated for the symbol table. You must reduce the number of objects.

(Pre-Processor Error Messages cont.)

Too Many Arguments
The number of arguments in a C source line exceeds the maximum of 64.

Too Many Files
The number of files specified in #include control lines has exceeded the maximum.

Unexpected EOF
An unexpected end-of-file is encountered in a pre-processor control line. The control line is ignored.

Unmatched Conditional
No #endif or #else for an #ifdef or #ifndef.

Parser Error Messages

Operator Illegal
The & ("address of") operator is used with an invalid operand. The operand must be an lvalue, that is, an expression that could appear as the left operand in an assignment statement.

+ OP Assumed
An ambiguous assignment statement has been interpreted as a + op assignment.

Address of Register
Register variables cannot be used with the '&' operator.

Assignable Operand Required
The specified operand must be an lvalue.

Bad Character Constant
A character constant is limited to a single character. The constant is truncated to this value.

Bad Indirection
The index operation ('[]') has been used with a non-pointer variable.

Bad Symbol Table
The number of symbols in a module has exceeded the maximum. The module should be divided into two modules.

Can't Open (FILENAME)
The compiler cannot find or open the specified file. You should check to see if the file exists or change the file specification in the program to that of an existing file.

(Parser Error Messages cont.)

Case Not Inside Switch Block
A case label has been used outside the body of a switch statement.

Dimension Table Overflow
An array contains more than five dimensions.

Duplicate Case Value
More than one case has been specified for the indicated value in a switch statement. The case label should be changed or the cases combined.

Expression Too Complex
The indicated expression is too complex to initialize and should be simplified.

Field Overflows Byte
The specified field is larger than one byte.

Field Overflows Word
The specified field is larger than one word.

Illegal Function Declaration
The function has been improperly declared and the compiler will ignore the function attribute.

Illegal Register Specification
The register storage class is restricted to function parameters and automatic variables.

Illegal Structure Operation
The only operators that are valid with structures are simple assignment (=), sizeof and passing as arguments or as return values from functions.

Illegal Type Conversions
The case operators may not force the conversion of any expression to an array, function, structure or union.

Indirection of Function Invalid
The case operator has been used to pass arguments to parameters of different types.

Initializer Alignment
The number of initialization values does not match the dimensions of the declared variable.

Initializer List Too Long
The initializer list exceeds the size of the array to which the variables are to be assigned.

(Parser Error Messages cont.)

Invalid ?: Operator Syntax
    Use of the conditional operator has resulted in an invalid expression.

Invalid Break Statement
    The break statement is valid only in 'for,' `while,' 'do-while,' and 'switch' statements.

Invalid Character (CHARACTER)
    The character enclosed in parentheses is not a valid C token.

Invalid Continue Statement
    The continue statement is valid only in 'for,' 'while,' and 'do' statements.

Invalid Conversion
    One of the operands in the indicated line cannot be converted as specified.

Invalid Data Type
    The data type keyword is specified in a declaration or definition that already has one. All but the first data type is ignored by the compiler.

Invalid Expression
    The expression contains a syntax error.

Invalid Field Size
    The indicated field declaration specifies a size greater than 32 bits in length.

Invalid Field Type Description
    Fields must be declared as integers (signed or unsigned, long or short).

Invalid Initializer
    An object cannot be initialized as specified. The initializer should be eliminated or corrected, or the storage class of the target object should be changed.

Invalid Long Declaration
    The initialization value is greater than 32 bits.

Invalid Operand Type
    The operand type is incompatible with the specified operator.

(Parser Error Messages cont.)

Invalid Register Specification
The register storage class is restricted to function parameters and automatic variables.

Invalid Short Declaration
The initialization value is greater than 16 bits.

Invalid Storage Class
The indicated storage class keyword is specified in a declaration that already has one. Only the first storage class is used.

Invalid Structure Assignment
Structure asignments are valid only if the source and the target are of equal value.

Invalid Structure Declaration (NAME)
The specified structure declaration has a syntax error.

Invalid Structure Member Name
The member name is not included in the declaration of the structure.

Invalid Structure Prototype (NAME)
The specified structure has a syntax error.

Invalid Type Declaration
The type declaration has a syntax error.

Invalid TYPEDEF
The type definition has a syntax error.

Invalid Unsigned Declaration
The declaration has a syntax error.

Label Redeclaration (LABEL NAME)
The label has been declared more than once.

Missing Colon
Syntax error, 'while' expected.

Missing { In Initialization
Curly braces ({}) expected to surround initializers.

No Structure Name
A structure name is missing.

Not In Parameter List (NAME)
The variable or function name has been declared more than once.

(Parser Error Messages cont.)

Short Assigned to Pointer
All pointers are 32 bits in length.

String Cannot Cross Line
Strings may not extend to multiple lines.

String Too Long
A character string may not exceed 1024 characters.

Structure Table Overflow
The number of structures exceeds the maximum.

Symbol Table Overflow
The number of symbols exceeds the maximum.

Temp Creation Error
The compiler is unable to create a temporary file.

Too Many Cases In Switch
The number of cases in a SWITCH statement may not exceed 256.

Too Many Initializers
The initializer list exceeds the size of the array or structure to which the variables are to be assigned.

Undefined label (NAME)
The compiler has encountered a "goto label-name" for an undefined label. The scope of a label is restricted to the function in which it is used as a label, and goto statements cannot branch to labels inside other functions.

Undefine Symbol (NAME)
The compiler has encountered an undefined symbol.

Unexpected EOF
An unexpected end-of-file is encountered.

Code Generator Error Messages

Can't Create (FILENAME)
The compier is unable to create a temporary or output file.

Can't Open (FILENAME)
The compiler cannot find or open the specified file. You should check to see if the file exists or change the file specification in the program to that of an existing file.

CDSIZE: Invalid Type (NUMBER)
An invalid object is being passed as a function argument.

(Code Generator Error Messages cont.)

Code Skeleton Error: (NUMBER)
    Developer error message; should not occur.

Divide By Zero
    An expression involving division by a constant zero has been encountered.

Expression Too Complex
    The source statement must be simplified.

Intermediate Code Error (NUMBER,NUMBER)
    Developer error message; should not occur. Try removing temporary files and recompile.

Invalid Floating Point Op
    The specified operation is not allowed on floating point variables.

Invalid Initialization
    A syntax error or undefined variable has been encountered in an initialization.

Invalid Operator (NUMBER)
    Developer error message; should not occur.

Invalid Register Expression
    The specified operation cannot be computed on a register variable. In particular, the address of a register cannot be taken.

Invalid Storage Class
    Developer error message; should not occur.

Modulus By Zero
    An expression involving modulo by a constant zero has been encountered.

No Code Table For (NUMBER)
    Developer error message; should not occur.

OPCALL Bad Op (NUMBER)
    Developer error message; should not occur.

PDOS C REFERENCE MANUAL
INDEX