

HI-TECH

C

User's Manual



## CONTENTS

1.	Introduction .....	1
1.1.	Features.....	1
1.2.	System Requirements .....	2
1.3.	Using this Manual.....	2
2.	Getting Started.....	3
3.	Compiler Structure.....	5
4.	Operating Details .....	7
5.	Specific Features.....	13
5.1.	ANSI C Standard Compatibility .....	13
5.2.	Type Checking.....	13
5.3.	Member Names.....	13
5.4.	Unsigned Types .....	14
5.5.	Arithmetic Operations .....	14
5.6.	Structure Operations .....	15
5.7.	Enumerated Types .....	16
5.8.	Initialization Syntax.....	16
5.9.	Function Prototypes .....	17
5.10.	Void and Pointer to Void.....	18
5.11.	Type qualifiers .....	19
5.12.	20	
5.13.	Pragma Directives.....	20
6.	Machine Dependencies .....	21
6.1.	Predefined Macros .....	21
7.	Error Checking and Reporting.....	23
8.	Standard Libraries.....	25
8.1.	Standard I/O.....	25
8.2.	Compatibility .....	25
8.3.	Libraries for Embedded Systems .....	25
8.4.	Binary I/O.....	26
8.5.	Floating Point Library.....	27
9.	Stylistic Considerations .....	29
9.1.	Member Names.....	29
9.2.	Use of Int .....	30
9.3.	Extern Declarations .....	30
10.	Memory Models.....	31
11.	What Went Wrong.....	33
12.	Z80 Assembler Reference Manual.....	35
12.1.	Introduction.....	35
12.2.	Usage .....	35
12.3.	The Assembly Language.....	36
12.3.1.	Symbols.....	36
12.3.1.1.	Temporary Labels .....	37
12.3.2.	Constants .....	37
12.3.2.1.	Character Constants .....	38
12.3.2.2.	Floating Constants .....	38
12.3.2.3.	Opcode Constants .....	38
12.3.3.	Expressions.....	38
12.3.3.1.	Operators.....	38
12.3.3.2.	Relocatability .....	39

12.3.4. Pseudo-ops .....	40
12.3.4.1. DEFB, DB.....	40
12.3.4.2. DEFF .....	41
12.3.4.3. DEFW .....	41
12.3.4.4. DEFS .....	41
12.3.4.5. EQU .....	41
12.3.4.6. DEFL .....	41
12.3.4.7. DEFM.....	42
12.3.4.8. END .....	42
12.3.4.9. COND, IF, ELSE, ENDC .....	42
12.3.4.10. ELSE.....	42
12.3.4.11. ENDC .....	42
12.3.4.12. ENDM .....	43
12.3.4.13. PSECT .....	43
12.3.4.14. GLOBAL .....	43
12.3.4.15. ORG.....	44
12.3.4.16. MACRO.....	44
12.3.4.17. LOCAL.....	45
12.3.4.18. REPT .....	46
12.3.5. IRP and IRPC .....	47
12.3.6. Extended Condition Codes .....	48
12.4. Assembler Directives .....	48
12.5. Diagnostics.....	49
12.6. Z80/Z180/64180 Instruction Set .....	50
13. Linker Reference Manual .....	69
13.1. Relocation and Psects .....	69
13.1.1. Program Sections.....	69
13.1.2. Local Psects and the Large Model.....	70
13.2. Global Symbols.....	70
13.3. Operation .....	71
13.4. Examples.....	74
13.5. Invoking the Linker.....	74
14. Librarian .....	75
14.1. The Library Format.....	75
14.2. Using .....	75
14.3. Examples.....	76
14.4. Supplying Arguments .....	77
14.5. Listing Format.....	77
14.6. Ordering of Libraries .....	77
14.7. Error Messages.....	78
15. Objtohex .....	79
16. Cref.....	83
APPENDIX 1 -- Error Messages.....	85
APPENDIX 2 -- Standard Library Functions .....	99
INDEX.....	165

## HI-TECH C COMPILER

User's Manual

March 1989

## 1. Introduction

The HI-TECH C Compiler is a set of software which translates programs written in the C language to executable machine code programs. Versions are available which compile programs for operation under the host operating system, or which produce programs for execution in embedded systems without an operating system.

## 1.1. Features

Some of HI-TECH C's features are:

A single command will compile, assemble and link entire programs.

The compiler performs strong type checking and issues warnings about various constructs which may represent programming errors.

The generated code is extremely small and fast in execution.

A full run-time library is provided implementing all standard C input/output and other functions.

The source code for all run-time routines is provided.

A powerful general purpose macro assembler is included.

Programs may be generated to execute under the host operating system, or customized for installation in ROM.

PC-DOS/MS-DOS  
CP/M-86  
Concurrent DOS  
Atari ST  
Xenix  
Unix  
CP/M-80

Table 1. Supported Hosts

## 1.2. System Requirements

The HI-TECH C Compilers operate under the operating systems listed in table 1. Ensure that the version of the compiler you have matches the system you have. Note that in general you must have a hard disk or two floppy disks on your system (it is possible to use one floppy disk of 800K or more). A hard disk is strongly recommended. Note that the CP/M-80 native compiler does not have all the features described in this manual, as it has not been upgraded past V3.09 due to memory limitations. The Z80 cross compiler does support all of the features described here and can be used to generate programs to execute under CP/M-80.

## 1.3. Using this Manual

The documentation supplied with the HI-TECH C compiler comprises two separate manuals within the one binder. The manual you are reading now covers all versions of the compiler (reflecting the portable nature of the compiler). A separate manual covers machine dependent aspects of your compiler, e.g. installation.

This manual assumes you are familiar with the C language already. If you are not, you should have at least one reference book covering C, of which a large number are available from most computer bookstores, e.g. "A Book on C" by Kelley and Pohl. Other suitable texts are "Programming in ANSI C" by S. Kochan and "The C Programming Language", by Kernighan and Ritchie. You should read the "Getting Started" chapter in this manual, and the "Installation" chapter in the machine-specific manual. This will provide you with sufficient information to work through the introductory examples in the C reference you are using.

Once you have a basic grasp of the C language, the remainder of this manual will provide you with information to enable you to explore the more advanced aspects of C.

Most of the manual covers all implementations of the HI-TECH C compiler. A separate manual is provided for the macro assembler for your particular machine, and other machine-dependent information.

## 2. Getting Started

If using the compiler on a hard disk system you will need to install the compiler before using it. See the "Installation" chapter for more details. If using a floppy disk based system, in general you should have a copy of the distribution disk #1 in drive A: and maintain your files on a disk in the B: drive. Again see the "Installation" chapter for more information.

```
main()
{
    printf("Hello, world\n");
}
```

Fig. 1. Sample C Program

Before compiling your program it must be contained in a file with an extension (or file type, i.e. that part of the name after the '.') of .C. For example you may type the program shown in fig. 1 into a file called HELLO.C. You will need a text editor to do this. Generally any text editor that can create a straight ASCII file (i.e. not a "word processor" type file) will be suitable. If using editors such as Wordstar, you should use the "non-document mode". Once you have the program in such a file all that is required to compile it is to issue the C command, e.g. to compile HELLO.C, simply type the command

```
C -V HELLO.C
```

Cross compilers (i.e. compilers that operate on one system but produce code for a separate target system) will have a compiler driver named slightly differently, e.g. the 68HC11 cross compiler driver is called C68.

If you are using a floppy disk based system (or a CP/M system) it may be necessary to specify where to find the C command, e.g. if the C command is on a disk in drive A: and you are working on B:, type the command

```
A:C -V HELLO.C
```

The compiler will issue a sign on message and then proceed to execute the various passes of the compiler in sequence to compile the program. If you are using a floppy disk based system where the compiler will not fit on a single disk you will be prompted to change disks whenever the compiler cannot find a pass. In this case you should insert a copy of the next distribution disk in drive A: and press RETURN.

As each pass of the compiler is about to be executed, a command line to that pass will be displayed on the screen. This is because the -V option has been used. This stands for Verbose, and had it not been given the compilation would have been silent except for the sign on message. Error messages can be redirected to a file by using the standard output redirection notation, e.g.

```
> somefile
```

After completion of compilation, the compiler will exit to command level. You will notice that several temporary files created during compilation will have been deleted, and all that will be left on the disk (apart from the original source file HELLO.C) will be an executable file. The name of this executable file will be HELLO.EXE for MS-DOS, HELLO.PRG for the Atari ST, HELLO.COM for CP/M-80 and HELLO.COM for CP/M-86. For cross compilers it will be called HELLO.HEX or HELLO.BIN depending on the default output format for the particular compiler. To execute this program, simply type

```
HELLO
```

and you should be rewarded with the message "Hello, world!" on your screen. If you are using a cross compiler you will need to put the program into EPROM or download to the target system to execute it. Cross compilers do not produce programs executable on the host system.

There are other options that may be used with the C command, but you will not need to use them unless you wish to do so. If you are new to the C language it will be advisable to enter and compile a few simple programs (e.g. drawn from one of the C reference texts mentioned above) before exploring other capabilities of the HI-TECH C compiler.

There is one exception to the above; if you compile a program which uses floating point arithmetic (i.e. real numbers) you MUST specify to the compiler that the floating point library should be searched. This is done with a -LF option at the END of the command line, e.g.

```
C -V FLOAT.C -LF
```

### 3. Compiler Structure

The compiler is made up of several passes; each pass is implemented as a separate program. Note that it is not necessary for the user to invoke each pass individually, as the C command runs each pass automatically. Note that the machine dependent passes are named differently for each processor, for example those with 86 in their name are for the 8086 and those with 68K in their name are for the 68000.

The passes are:

CPP	The preprocessor - handles macros and conditional compilation
P1	The syntax and semantic analysis pass. This writes intermediate code for the code generator to read.
CGEN, CG86 etc.	The code generator - produces assembler code.
OPTIM, OPT86 etc.	The code improver - may optionally be omitted, reducing compilation time at a cost of larger, slower code produced.
ZAS, AS86 etc.	The assembler - in fact a general purpose macro assembler.
LINK	The link editor - links object files with libraries.
OBJTOHEX	This utility converts the output of LINK into the appropriate executable file format (e.g. .EXE or .PRG or .HEX).

The passes are invoked in the order given. Each pass reads a file and writes a file for its successor to read. Each intermediate file has a particular format; CPP produces C code without the macro definitions and with uses of macros expanded; P1 writes a file containing a program in an intermediate code; CGEN translates this to assembly code; AS produces object code, a binary format containing code bytes along with relocation and symbol information. LINK accepts object files and libraries of object files and writes another object file; this may be in absolute form or it may preserve relocation information and be input to another LINK command.

There are also other utility programs:

LIBR Creates and maintains libraries of object modules

CREF Produces cross-reference listings of C or assembler programs.

This page is left blank for your notes.

#### 4. Operating Details

HI-TECH C was designed for ease of use; a single command will compile, assemble and link a C program. The syntax of the C command is as follows:

C [ options ] files [ libraries ]

The options are zero or more options, each consisting of a dash ('-'), a single key letter, and possibly an argument, following the key letter with no intervening space. The files are one or more C source files, assembler source files, or object files. Libraries may be zero or more library names, or the abbreviated form -lname which will be expanded to the library name libname.lib.

The C command will, as determined by the specified options, compile any C source files given, assemble them into object code unless requested otherwise, assemble any assembler source files specified, and then link the results of the assemblies with any object or library files specified.

If the C command is invoked without arguments, then it will prompt for a command line to be entered. This command line may be extended by typing a backslash ('\') on the end of the line. Another line will then be requested. If the standard input of the command is from a file (e.g. by typing C < afile) then the command lines will be read from that file. Within the file more than one line may be given if each line but the last ends with a backslash. Note that this mechanism does not work in MS-DOS batch file, i.e. the command file for the C command must be a separate file. MS-DOS has no mechanism for providing long command lines or standard input from inside a batch file.

The options recognized by the C command are as follows:

- S Leave the results of compilation of any C files as assembler output. C source code will be interspersed as comments in the assembler code.
- C Leave the results of all compiles and assemblies as object files; do not invoke the linker. This allows the linker to be invoked separately, or via the C command at a later stage.
- CR Produce a cross reference listing. -CR on its own will leave the raw cross-reference information in a temporary file, allowing the user to run CREF explicitly, while supplying a file name, e.g. -CR**FRED.CRF** will cause CREF to be invoked to process the raw information into the specified file, in this case **FRED.CRF**.
- CPM For the Z80 cross compiler only, produce CP/M-80 COM files. Unless the -CPM option is given, the Z80 cross compiler uses the ROM runtime startoff module and produces hex or binary images. If the -CPM option is given, CP/M-80 runtime startoff code is linked used and a CP/M-80 COM file is produced.
- O Invoke the optimizer on all compiled code; also requests the assembler to perform jump optimization.
- OUTFILE** Specify a name for the executable file to be created. By default the name for the executable file is derived from the name of the first source or object file specified to the compiler. This option allows the default to be overridden. If no dot ('.') appears in the given file name, an extension appropriate for the particular operating system will be

added, e.g. **-OFRED** will generate a file **FRED.EXE** on MS-DOS or **FRED.COM** on CP/M-86. For cross compilers this also provides a means for specifying the output format, e.g. specifying an output file **PROG.BIN** will make the compiler generate a binary file, while specifying **PROG.HEX** will make it generate a hexadecimal file.

- V Verbose: each step of the compilation will be echoed as it is executed.
- I Specify an additional filename prefix to use in searching for #include files. For CP/M the default prefix is A0:: (user number 0, disk drive A). For MS-DOS the default prefix is A:\HITECH\. Under Unix and Xenix the default prefix is /usr/hitech/include/. Note that on MS-DOS a trailing backslash must be appended to any directory name given as an argument to -I; e.g. -I\FRED\ not -I\FRED. Under Unix a trailing slash should be added.
- D Define a symbol to the preprocessor: e.g. -DCPM will define the symbol CPM as though via #define CPM 1.
- U Undefine a pre-defined symbol. The complement of -D.
- F Request the linker to produce a symbol file, for use with the debugger.
- R For the Z80 CP/M compiler only this option will link in code to perform command line I/O redirection and wild card expansion in file names. See the description of \_getargs() in appendix 5 for details of the syntax of the redirections.
- X Strip local symbols from any files compiled, assembled or linked. Only global symbols will remain.
- M Request the linker to produce a link map.
- A This option, for the Z80 only, will cause the compiler to produce an executable program that will, on execution, self-relocate itself to the top of the TPA (Transient

Program Area). This allows the writing of programs which may execute other programs under themselves. Note that a program compiled in such a manner will not automatically reset the bdos address at location 6 in order to protect itself. This must be done by the program itself.

For cross compilers this provides a way of specifying to the linker the addresses that the compiled program is to be linked at. The format of the option is **-AROMADR, RAMADR, RAMSIZE**. **ROMADR** is the address of the ROM in the system, and is where the executable code and initialized data will be placed. **RAMADR** is the start address of RAM, and is where the bss psect will be placed, i.e. uninitialized data. **RAMSIZE** is the size of RAM available to the program, and is used to set the top of the stack.

For the 6801/6301/68HC11 compiler, the -A option takes a fourth value which is the address of a four byte direct page area called ctemp which the compiled code uses as a scratch pad. If the ctemp address is omitted from the -A option, it defaults to address 0. Normally this will be acceptable, however some 6801 variants (like the 6303) have memory mapped I/O ports at address 0 and start their direct page RAM at address \$80.

For the large memory model of the 8051 compiler, the -A option takes the form **-AROMADR, INTRAM, EXTRAM, EXTSIZE**. **ROMADR** is the address of ROM in the system. **INTRAM** is the start address of internal RAM, and is where the rbss psect will be placed. The 8051 internal stack will start after the end of the rbss psect. **EXTRAM** is the start address of external RAM, and is where the bss psect will be placed. **EXTSIZE** is the size of external RAM available to the program, and is used to set the top of the external stack.

- B For compilers which support more than one "memory model", this option is used to select which memory model code is to be generated for. The format of this option is -Bx where x is one or more letters specifying which memory model to use. For the 8086, this option is used to select which one of five memory models (Tiny, Small, Medium, Compact or Large) is to be used.

For the 8051 compiler this option is used to select which one of the three memory models (Small, Medium or Large) is to be used. For the 8051 compiler only, this option can also be used to select static allocation of *auto* variables by appending an A to the end of the -B option. For example, -Bsa would select small model with static allocation of all variables, while -Bm would select medium model with *auto* variables dynamically allocated on the stack.

- E By default the 8086 compiler will initialize the executable file header to request a 64K data segment at run time. This may be overridden by the -E option. It takes an argument (usually in hexadecimal representation) which is the number of BYTES (not paragraphs) to be allocated to the program at run time. For example -E0ffff0h will request a megabyte. Since this much will not be available, the operating system will allocate as much as it can.
  
- W This options sets the warning level, i.e. it determines how picky the compiler is about legal but dubious type conversions etc. -W0 will allow all warning messages (default), -W1 will suppress the message "Func() declared implicit int". -W3 is recommended for compiling code originally written with other, less strict, compilers. -W9 will suppress all warning messages.
  
- H This option generates a symbol file for use with debuggers. The format of the symbol file is described elsewhere. The default name of the symbol file is *l.sym*. An alternate name may be specified with the option, e.g. *-Hsymfile.abc*.
  
- G Like -H, -G also generates a symbol file, but one that contains line and file number information for a source level debugger. Like -H a file name may be specified. When used in conjunction with -O, only partial optimization will be performed to avoid confusing the debugger.
  
- P Execution profiling is available on native compilers running under DOS, CP/M-86 and on the Atari ST. This option generates code to turn on execution profiling when the program is run. A -H option should also be specified to provide a symbol table for the profiler EPROF.
  
- Z For version 5.xx compilers only, the -Z option is used to select global optimization of the code generated. For the 8086 and 6801/6301/68HC11 compilers the only valid -Z option is -Zg. For the 8051 compiler, the valid -Z options are -Zg which invokes global optimization, -Zs which optimizes for space, and -Zf which optimizes for speed. The s and f options may be used with the g option, thus the options -Zgf and -Zgs are valid. Speed and space optimization are mutually exclusive, i.e. the s and f options cannot be used together.
  
- 1 For the 8086 compiler only, request the generation of code which takes advantage of the extra instructions of the 80186 processor. A program compiled with -1 will not execute on an 8086 or 8088 processor. For the 68000 compiler, generate instructions for the 68010 processor.

- 2 Like -1 but for the 80286 and 68020.
- 11 For the 6801/HC11 compiler, this option will request generation of instructions specific to the 68HC11 processor.
- 6301 For the 6801/HC11 compiler, this option will request generation of instructions specific to the 6301/6303 processors.

Some examples of the use of the C command are:

```
c prog.c  
  
c -mlink.map prog.c x.obj -lx  
  
c -S prog.c  
  
c -O -C -CRprog.crf prog.c prog2.c  
  
c -v -Oxfile.exe afile.obj anfile.c -lf
```

Upper and lower case has been used in the above examples for emphasis: the compiler does not distinguish between cases, although arguments specifying names, e.g. -D, are inherently case sensitive.

Taking the above examples in order; the first compiles and links the C source file prog.c with the standard C library. The second example will compile the file prog.c and link it with the object file x.obj and the library libx.lib; a link map will be written to the file link.map.

The third example compiles the file prog.c, leaving the assembler output in a file prog.as. It does not assemble this file or invoke the linker. The next example compiles both prog.c and prog2.c, invoking the optimizer on both files, but does not perform any linking. A cross reference listing will be left in the file *prog.crf*.

The last example pertains to the 8086 version of the compiler. It runs the compiler with the verbose option, and will cause anfile.c to be compiled without optimization to object code, yielding anfile.obj, then afile.obj and anfile.obj will be linked together with the floating point library (from the -LF option) and the standard library to yield the executable program xfile.exe (assuming this is performed on an MS-DOS system). One would expect this program to use floating point, if it did not then the -LF option would have been required.

If more than one C or assembler source file is given to the C command, the name of each file will be printed on the console as it is processed. If any fatal errors occur during the compilation or assembly of source files, further source files will be processed, but the linker will not be invoked.

Other commands which may be issued by the user, rather than automatically by the C command, are:

ZAS	The Z80 assembler.
AS86	The 8086 assembler.
LINK	The linker
LIBR	The library maintainer
OBJTOHEX	Object to hex converter
CREF	Cross reference generator.

In general, these commands accept the same type of command line as the C command, i.e. zero or more options (indicated by a leading '-') followed by one or more file arguments. If the linker or the librarian is invoked with no arguments, it will prompt for a command line. This allows command lines of more than 128 bytes to be entered. Input may also be taken from a file by using the redirection capabilities (see `_getargs()` in the library function listing). See the discussion above of the C command. These commands are described in further detail in their respective manuals.

## 5. Specific Features

The HI-TECH C compiler has a number of features, which while largely compatible with other C compilers, contribute to more reliable programming methods.

### 5.1. ANSI C Standard Compatibility

At the time of writing the Draft ANSI Standard for the C Language was at an advanced stage, though not yet an official standard. Accordingly it is not possible to claim compliance with that standard, however HI-TECH C includes the majority of the new and altered features in the draft ANSI standard. Thus it is in the sense that most people understand it "ANSI compatible".

### 5.2. Type Checking

Previous C compilers have adopted a lax approach to type checking. This is typified by the Unix C compiler, which allows almost arbitrary mixing of types in expressions. The HI-TECH C compiler performs much more strict type checking, although in most cases only warning messages are issued, allowing compilation to proceed if the user knows that the errors are harmless. This would occur, for example, when an integer value was assigned to a pointer variable. The generated code would almost certainly be what the user intended, however if in fact it represented an error in the source code, the user is prompted to check and correct it where necessary.

### 5.3. Member Names

In early C compilers member names in different structures were required to be distinct except under certain circumstances. HI-TECH C, like most recent implementations of C, allows member names in different structures and unions to overlap. A member name is recognized only in the context of an expression whose type is that of the structure in which the member is defined. In practice this means that a member name will be recognized only to the right of a '.' or '->' operator, where the expression to the left of the operator is of type structure or pointer to structure the same as that in which the member name was declared. This not only allows structure names to be re-used without conflict in more than one structure, it permits strict checking of the usage of members; a common error with other C compilers is the use of a member name with a structure pointer of the wrong type, or worse with a variable which is a pointer to a simple type.

There is however an escape from this, where the user desires to use as a structure pointer something which is not declared as such. This is via the use of a typecast. For example, suppose it is desired to access a memory-mapped i/o device, consisting of several registers. The declarations and use may look something like the code fragment in fig. 2.

```

struct io_dev
{
    Short  io_status;    /* status */
    Char   io_rxdata;   /* rx data */
    Char   io_txdata;   /* tx data */
};

#define RXRDY    01    /* rx ready */
#define TXRDY    02    /* tx ready */

/* define the (absolute) device address */

#define DEVICE    ((struct io_dev *)0xFF00)

send_byte(c)
char c;
{
    /* wait till transmitter ready */
    while(!(DEVICE->io_status & TXRDY)) continue;
    /* send the data byte */
    DEVICE->io_txdata = c;
}

```

Fig. 2. Use of Typecast on an Absolute Address

In this example, the device in question has a 16 bit status port, and two 8 bit data ports. The address of the device (i.e. the address of its status port) is given as (hex)0FF00. This address is typecast to the required structure pointer type to enable use of the structure member names. The code generated by this will use absolute memory references to access the device, as required.

Some examples of right and wrong usage of member names are shown in fig. 3.

#### 5.4. Unsigned Types

HI-TECH C implements unsigned versions of all integral types; i.e. unsigned char, short, int and long. If an unsigned quantity is shifted right, the shift will be performed as a logical shift, i.e. bringing zeros into the rightmost bits. Similarly right shifts of a signed quantity will sign extend the rightmost bits.

#### 5.5. Arithmetic Operations

On machines where arithmetic operations may be performed more efficiently in lengths shorter than int, operands shorter than int will not be extended to int length unless necessary.

For example, if two characters are added and the result stored into another character, it is only necessary to perform arithmetic in 8 bits, since any overflow into the top 8 bits will be lost.

```
struct fred
{
    Char  a;
    Int   b;
} s1, * s2;

struct bill
{
    float c;
    long  b;
} x1, * x2;

main()
{
    /* wrong - c is not a member of fred */
    s1.c = 2;

    /* correct */
    s1.a = 2;

    /* wrong - s2 is a pointer */
    s2.a = 2;

    /* correct */
    x2->b = 24L;

    /* right, but note type conversion from long to int */
    s2->b = x2->b;
}
```

Fig. 3. Examples of Member Usage

However, if the sum of two characters is stored into an int, the addition should be done in 16 bits to ensure the correct result.

In accordance with the draft ANSI standard, operations on float rather than double quantities will be performed in the shorter precision rather than being converted to double precision then back again.

## 5.6. Structure Operations

HI-TECH C implements structure assignments, structure arguments and structure-valued functions in their full generality. The example in fig. 4 is a function returning a structure. Some legal (and illegal) uses of the function are also shown.

```
struct bill
{
    char a;
    int b;
}
afunc()
{
    struct bill    x;

    return x;
}

main()
{
    struct bill    a;

    a = afunc();          /* ok */
    pf("%d", afunc().a); /* ok */

    /* illegal, afunc() cannot be assigned to,
       therefore neither can afunc().a */
    afunc().a = 1;

    /* illegal, same reason */
    afunc().a++;
}
```

Fig. 4. Example of a Function Returning a Structure

### 5.7. Enumerated Types

HI-TECH C supports enumerated types; these provide a structured way of defining named constants.

The uses of enumerated types are more restricted than that allowed by the Unix C compiler, yet more flexible than permitted by LINT. In particular, an expression of enumerated type may be used to dimension arrays, as an array index or as the operand of a switch statement. Arithmetic may be performed on enumerated types, and enumerated type expressions may be compared, both for equality and with the relation operators. An example of the use of an enumerated type is given in fig. 5.

### 5.8. Initialization Syntax

Kernighan and Ritchie in "The C Programming Language" state that pairs of braces may be omitted from an initializer in certain contexts; the draft ANSI standard provides that a conforming C program must either include all braces in an initializer, or leave

```

/* a represents 0, b -> 1 */
enum fred { a, b, c = 4 };

main()
{
    enum fred    x, y, z;

    x = z;
    if(x < z)
        func();
    x = (enum fred)3;
    switch(z) {
    case a:
    case b:
    default:
    }
}

```

Fig. 5. Use of an Enumerated Type

them all out. HI-TECH C allows any pairs of braces to be omitted providing that the front end of the compiler can determine the size of any arrays being initialized, and providing that there is no ambiguity as to which braces have been omitted. To avoid ambiguity, if any pairs of braces are present, then any braces which would enclose those braces must also be present. The compiler will complain ("initialization syntax") if any ambiguity is present.

### 5.9. Function Prototypes

A new feature of C included in the proposed ANSI for C, known as "function prototypes", provides C with an argument checking facility, i.e. it allows the compiler to check at compile time that actual arguments supplied to a function invocation are consistent with the formal parameters expected by the function. The feature allows the programmer to include in a function declaration (either an external declaration or an actual definition) the types of the parameters to that function. For example, the code fragment shown in fig. 6 shows two function prototypes.

```

void fred(int, long, char *);

char *
bill(int a, short b, ...)
{
    return a;
}

```

Fig. 6. Function Prototypes

The first prototype is an external declaration of the function *fred()*, which accepts one integer argument, one long argument, and one argument which is a pointer to char. Any usage of *fred()* while the prototype declaration is in scope will cause the actual parameters to be checked for number and type against the prototype, e.g. if only two arguments were supplied or an integral value was supplied for the third argument the compiler would report an error.

In the second example, the function *bill()* expects two or more arguments. The first and second will be converted to int and short respectively, while the remainder (if present) may be of any type. The ellipsis symbol (...) indicates to the compiler that zero or more arguments of any type may follow the other arguments. The ellipsis symbol must be last in the argument list, and may not appear as the only argument in a prototype.

All prototypes for a function must agree exactly, however it is legal for a definition of a function in the old style, i.e. with just the parameter names inside the parentheses, to follow a prototype declaration provided the number and type of the arguments agree. In this case it is essential that the function definition is in scope of the prototype declaration.

Access to unspecified arguments (i.e. arguments supplied where an ellipsis appeared in a prototype) must be via the macros defined in the header file <stdarg.h>. This defines the macros *va\_start*, *va\_arg* and *va\_end*. See *va\_start* in the library function listing for more information.

NOTE that it is a grave error to use a function which has an associated prototype unless that prototype is in scope, i.e. the prototype **MUST** be declared (possibly in a header file) before the function is invoked. Failure to comply with this rule may result in strange behaviour of the program. HI-TECH C will issue a warning message ("func() declared implicit int") whenever a function is called without an explicit declaration. It is good practice to declare all functions and global variables in one or more header files which are included wherever the functions are defined or referenced.

#### 5.10. Void and Pointer to Void

The *void* type may be used to indicate to the compiler that a function does not return a value. Any usage of the return value from a void function will be flagged as an error.

The type *void \**, i.e. pointer to void, may be used as a "universal" pointer type. This is intended to assist in the writing of general purpose storage allocators and the like, where a pointer is returned which may be assigned to another variable of some other pointer type. The compiler permits without typecasting and without reporting an error the conversion of *void \** to any other pointer type and vice versa. The programmer is advised to use this facility carefully and ensure that any *void \** value is usable as a pointer to any other type, e.g. the alignment of any such pointer should be suitable for storage of any object.

### 5.11. Type qualifiers

The ANSI C standard introduced the concept of *type qualifiers* to C. These are keywords that qualify the type to which they are applied. The type qualifiers defined by ANSI C are `const` and `volatile`. HI-TECH C also implements several other type qualifiers. The extra qualifiers include:

- Far
- Near
- Interrupt
- fast interrupt
- port

Not all versions of the compilers implement all of the extra qualifiers. See the machine dependent section for further information.

When constructing declarations using type qualifiers, it is very easy to be confused as to the exact semantics of the declaration. A couple of rules-of-thumb will make this easier. Firstly, where a type qualifier appears at the left of a declaration it may appear with any storage class specifier and the basic type in any order, e.g.

```
static void interrupt func();
```

is semantically the same as

```
interrupt static void func();
```

Where a qualifier appears in this context, it applies to the basic type of the declaration. Where a qualifier appears to the right of one or more '\*' (*star*) pointer modifiers, then you should read the declaration from right to left, e.g.

```
char * far fred;
```

should be read as "fred is a far pointer to char". This means that fred is qualified by *far*, not the char to which it points. On the other hand,

```
char far * bill;
```

should be read as "bill is a pointer to a far char", i.e. the char to which bill points is located in the far address space. In the context of the 8086 compiler this will mean that bill is a 32 bit pointer while fred is a 16 bit pointer. You will hear bill referred to as a "far pointer", however the terminology "pointer to far" is preferred.

## 5.12.

There are two methods provided for in-line assembler code in C programs. The first allows several lines of assembler anywhere in a program. This is via the `#asm` and `#endasm` preprocessor directives. Any lines between these two directives will be copied straight through to the assembler file produced by the compiler. Alternatively you can use the `asm("string");` construct anywhere a C statement is expected. The *string* will be copied through to the assembler file. Care should be taken with using in-line assembler since it may interact with compiler generated code.

## 5.13. Pragma Directives

The draft ANSI C standard provides for a `#pragma` preprocessor directive that allows compiler implementations to control various aspects of the compilation process. Currently HI-TECH C only supports one pragma, the *pack* directive. This allows control over the manner in which members are allocated inside a structure. By default, some of the compilers (especially the 8086 and 68000 compilers) will align structure members onto even boundaries to optimize machine accesses. It is sometimes desired to override this to achieve a particular layout inside a structure. The *pack* pragma allows specification of a maximum packing factor. For example, `#pragma pack(1)` will instruct the compiler that no additional padding be inserted between structure members, i.e. that all members should be aligned on boundaries divisible by 1. Similarly `#pragma pack(2)` will allow alignment on boundaries divisible by 2. In no case will use of the *pack* pragma force a greater alignment than would have been used for that data type anyway.

More than one *pack* pragma may be used in a program. Any use will remain in force until changed by another *pack* or until the end of the file. Do not use a *pack* pragma before include files such as `<stdio.h>` as this will cause incorrect declarations of run-time library data structures.

## 6. Machine Dependencies

HI-TECH C eliminates many of the machine dependent aspects of C, since it uniformly implements such features as unsigned char. There are however certain areas where machine dependencies are inherent in the C language; programmers should be aware of these and take them into account when writing portable code.

The most obvious of these machine dependencies is the varying size of C types; on some machines an int will be 16 bits, on others it may be 32 bits. HI-TECH C conforms to the following rules, which represent common practice in most C compilers.

char	is at least 8 bits
short	is at least 16 bits
long	is at least 32 bits
int	is the same as either short or long
float	is at least 32 bits
double	is at least as wide as float

Because of the variable width of an int, it is recommended that short or long be used wherever possible in preference to int. The exception to this is where a quantity is required to correspond to the natural word size of the machine.

Another area of machine differences is that of byte ordering; the ordering of bytes in a short or long can vary significantly between machines. There is no easy approach to this problem other than to avoid code which depends on a particular ordering. In particular you should avoid writing out whole structures to a file (via *fwrite()*) unless the file is only to be read by the same program then deleted. Different compilers use different amounts of padding between structure members, though this can be modified via the `#pragma pack(n)` construct.

### 6.1. Predefined Macros

One technique through which machine unavoidable machine dependencies may be managed is the predefined macros provided by each compiler to identify the target processor and operating system (if any). These are defined by the compiler driver and may be tested with conditional compilation preprocessor directives.

The macros defined by various compilers are listed in table 2. These can be used as shown in the example in Table 2.

Macro	Defined for
i8051	8051 processor family
i8086	8086 processor family
i8096	8096 processor family
z80	Z80 processor and derivatives
m68000	68000 processor family
m6800	6801, 68HC11 and 6301 processors
m6809	6809 processor
DOS	MS-DOS and PC-DOS
CPM	CP/M-80 and CP/M-86
TOS	Atari ST

Table 2. Predefined Macros

```
#if    DOS
char * filename = "c:file";
#endif    /* DOS */
#if CPM
char * filename = "0:B:afile";
#endif    /* CPM */
```

Use this page for notes:

## 7. Error Checking and Reporting

Errors may be reported by any pass of the compiler, however in practice the assembler and optimizer will not encounter any errors in the generated code. The types of errors produced by each pass are summarized below. In general any error will be identified by the name of the source file in which it was encountered, and the line number at which it was detected. P1 will also nominate the name of the function inside which the error was detected.

Errors may be redirected to a file with the usual syntax, i.e. a 'greater than' symbol (>) followed by the name of a file into which the errors should be written. The file name may of course be a device name, e.g. LST: for CP/M or PRN for MS-DOS.

CPP will report errors relating to macro definitions and expansions, as well as conditional compilation.

P1 is the pass which reports most errors; it performs syntax and semantic checking of the input, and will report both fatal and warning errors when encountered. Syntax errors will normally be expressed as "symbol expected" or "symbol unexpected". Semantic errors may relate to undeclared, redeclared or misdeclared variables. P1 will also report variable definitions which are unused or unreferenced. These errors are warnings, as are most type checking errors.

When P1 encounters errors it will list the source-line containing the error on the screen, and underneath display the error message and an up arrow pointing to the point at which the compiler detected the error. In some cases the actual cause of the error may be earlier in the line or even on previous line.

CGEN may occasionally report errors, usually warnings, and mostly related to unusual combinations of types with constants, for example testing if an unsigned quantity is less than zero. One fatal error produced by CGEN is "can't generate code for this expression" and means that the expression currently being compiled is in some way too complicated to produce code for. This can usually be overcome by rewriting the source code. Such errors are rare and will occur only for unusual constructs.

The linker will report undefined or multiply defined symbols. Note that variable declarations inside a header file which is included in more than one source file must be declared as extern, to avoid multiply defined symbol errors. These symbols must then be defined in one and one only source file.

A comprehensive list of error messages is included in an appendix.

Use this page for notes

## 8. Standard Libraries

### 8.1. Standard I/O

C is a language which does not specify the I/O facilities in the language itself; rather all I/O is performed via library routines. In practice this results in I/O handling which is no less convenient than any other language, with the added possibility of customising the I/O for a specific application. For example it is possible to provide a routine to replace the standard `getchar()` (which gets one character from the standard input) with a special `getchar()`. This is particularly useful when writing code which is to run on a special hardware configuration, while maintaining a high level of compatibility with "standard" C I/O.

There is in fact a Standard I/O library (STDIO) which defines a portable set of I/O routines. These are the routines which are normally used by any C application program. These routines, along with other non-I/O library routines, are listed in detail in a subsequent section of the manual.

### 8.2. Compatibility

The libraries supplied with HI-TECH C are highly compatible with the ANSI libraries, as well as the V7 UNIX libraries, both at the Standard I/O level and the UNIX system call level. The Standard I/O library is complete, and conforms in all respects with the UNIX Standard I/O library. The library routines implementing UNIX system call like functions are as close as possible to those system calls, however there are some UNIX system calls that cannot be simulated on other systems, e.g. the `link()` operation. However the basic low level I/O routines, i.e. `open`, `close`, `read`, `write` and `lseek` are identical to the UNIX equivalents. This means that many programs written to run on UNIX, even if they do not use Standard I/O, will run with little modification when compiled with HI-TECH C.

### 8.3. Libraries for Embedded Systems

The cross compilers, designed to produce code for target systems without operating systems, are supplied with libraries implementing a subset of the STDIO functions. Not provided are those functions dealing with files. Included are *`printf()`*, *`scanf()`* etc. These operate by calling two low level functions *`putch()`* and *`getch()`* which typically send and receive characters via a serial port. Where the compiler is aimed at a single-chip micro with on-board UART this is used. The source code for all these functions is supplied enabling the user to modify it to address a different serial port.

#### 8.4. Binary I/O

On some operating systems, notably CP/M, files are treated differently according to whether they contain ASCII (i.e. printable) or binary data. MD-DOS also suffers from this problem, not due to any lack in the operating system itself, but rather because of the hangover from CP/M which results in many programs putting a redundant ctrl-Z at the end of files. Unfortunately there is no way to determine which type of data a file contains (except perhaps by the name or type of the file, and this is not reliable). To overcome this difficulty, there is an extra character which may be included in the MODE string to a fopen() call. To open a file for ASCII I/O, the form of the fopen() call is:

```
fopen("filename.ext", "r")    /* for reading */
fopen("filename.ext", "w")    /* for writing */
```

To open a file for binary I/O, the character 'b' may be appended to the argument.

```
fopen("filename.ext", "rb")
fopen("filename.ext", "wb")
```

The additional character instructs the STDIO library that this file is to be handled in a strict binary fashion. On CP/M or MS-DOS, a file opened in ASCII mode will have the following special character handling performed by the STDIO routines:

newline	(\n) converted to carriage return/newline on output.
Return	(\r) ignored on input
Ctrl-Z	interpreted as End-Of-File on input and, for CP/M only, appended when closing the file.

The special actions performed on ASCII files ensure that the file is written in a format compatible with other programs handling text files, while eliminating the requirement for any special handling by the user program - the file appears to the user program as though it were a UNIX-like text file.

None of these special actions are performed on a file opened in binary mode. This is required when handling any kind of binary data, to ensure that spurious bytes are not inserted, and premature EOF's are not seen.

Since the binary mode character is additional to the normal mode character, this usage is quite compatible with UNIX C. When compiled on UNIX, the additional character will be ignored.

A mention here of the term 'stream' is appropriate; stream is used in relation to the `STDIO` library routines to mean the source or sink of bytes (characters) manipulated by those routines. Thus the `FILE` pointer supplied as an argument to the `STDIO` routines may be regarded as a handle on the corresponding stream. A stream may be viewed as a featureless sequence of bytes, originating from or being sent to a device or file or even some other indeterminate source. A `FILE` pointer should not be confused with the 'file descriptors' used with the low-level I/O functions *open()*, *close()*, *read()* and *write()*. These form an independent group of I/O functions which perform unbuffered reads and writes to files.

## 8.5. Floating Point Library

HI-TECH C supports floating point as part of the language, however the Z80 implementation provides single precision only; double floats are permitted but are no different to floats. In addition, the standard library, `LIBC.LIB`, does not contain any floating point routines. These have been separated out into another library, `LIBF.LIB`. This means that if this library is not searched, no floating point support routines will be linked in, thus avoiding any size penalty for the floating point support if it is not used. This is particularly important for `printf` and `scanf`, and thus `LIBF.LIB` contains versions of `printf` and `scanf` that do support floating point formats.

Thus, if floating point is used, a `-LF` option should be used AFTER the source and/or object files to the C command. E.g.:

```
C -V -O x.c y.c z.obj -LF
```

Use this page for notes

## 9. Stylistic Considerations

Although it is not the purpose of this manual to set out a coding standard for C, some comments regarding use of some of the features of HI-TECH C may be useful.

### 9.1. Member Names

Although HI-TECH C allows the same structure or union member name to be used in more than one structure or union, this may not be allowed by another C compiler. To help ensure portability of the code, it is recommended that member names all be distinct, and a useful way of ensuring this is to prefix each member name with one or two letters derived from the name of the structure itself. An example is given in fig. 7.

```
struct tree_node
{
    struct tree_node *    t_left;
    struct tree_node *    t_right;
    short t_operator;
};
```

Fig. 7. Member Naming

Because HI-TECH C insists on use of all intermediate names when referencing a member nested inside several structures, some simple macro definitions can serve as a shorthand. An example is given in fig. 8.

```
struct tree_node
{
    short t_operator; union
    {
        struct tree_node *    t_un_sub[2];
        char *                t_un_name; long t_un_val;
    } t_un;
};

#define t_left    t_un.t_un_sub[0]
#define t_right  t_un.t_un_sub[1]
#define t_name   t_un.t_un_name
#define t_val    t_un.t_un_val
```

Fig. 8. Member Name Shorthand

This enables the variant components of the structure to be referred to by short names, while guaranteeing portability and presenting a clean definition of the structure.

## 9.2. Use of Int

It is recommended that the type `int` be avoided wherever possible, in preference to the types `short` or `long`. This is because of the variable size of `int`, whereas `short` is commonly 16 bits and `long` 32 bits in most C implementations.

## 9.3. Extern Declarations

Some compilers permit a non-initialized global variable to be declared in more than one place, with the multiple definitions being resolved by the linker as all defining the same thing. HI-TECH C specifically disallows this, since it may lead to subtle bugs. Instead, global variables may be declared `extern` in as many places as you wish, and must be defined in one and one only place. Typically this will mean declaring global variables in a header file as `extern`, and defining each variable in the file most closely associated with that variable.

This usage will be portable to virtually all other C implementations.

## 10. Memory Models

With many of the processors supported by the HI-TECH C compilers there are more than one address space accessible to a program. Typically one address space is more economical to access than another, larger address space. Thus it is desirable to be able to tailor a program's use of memory to achieve the greatest economy in addressing (thus minimizing program size and maximizing speed) while allowing access to as much memory as the program requires.

This concept of different address spaces is not catered for by either K&R or ANSI C (except to recognize the possibility of separate address spaces for code and data). Without any extensions to the language itself it is possible to devise more than one memory model for a given processor, selected at compile time. This has the effect of selecting one addressing method for all data and/or code. This permits the model for a particular program to be chosen depending on that program's memory requirements.

In many programs, however, only one or two data structures are large enough to need to be placed in the larger address space. Selection of a "large" memory model for the whole of the program makes the whole program larger and slower just to allow a few large data structures. This can be overcome by allowing individual selection of the address space for each data structure. Unfortunately this entails extensions to the language, never a desirable approach. To minimize the effect of such extensions they should satisfy the following criteria:

1. As far as possible the extensions should be consistent with common practice.
2. The extensions should fit a machine-independent model to maximize portability across processors and operating systems.

These goals have been achieved within HI-TECH C by means of the following model:

Each memory model defines three address spaces each for code and data. These address spaces are known as the *near*, *far* and *default* spaces. Any object qualified by the *near* keyword will be placed in the *near* address space, any object qualified by the *far* keyword shall be placed in the *far* address space, and all other objects shall be placed in the *default* address space. The *near* address space shall be a (possibly improper) subspace of the *default* address space, while the *default* address space shall be a (possibly improper) subspace of the *far* address space. There shall be up to three kinds of pointers corresponding to the three address spaces, each capable of addressing an object in its own address space or a subspace of that address space.

This implies that the address of an object may be converted to a pointer into a larger address space, e.g. a *near* object may have its address converted to a pointer to *far*, but a *far* object may not be able to be addressed by a pointer to *near*.

In practice the *default* address space will usually correspond exactly to either the near or far address spaces. If all three address spaces correspond to the same memory then there is only one memory model possible. This occurs with the 68000 processor. Where the *default* code and data spaces may each correspond to either the *near* or *far* address spaces then there will be a total of four memory models. This is the case with the 8086 processor.

The keywords far and near are supported by all the HI-TECH C compilers, but the exact correspondence of address spaces is determined by the individual characteristics of each processor and the choice of memory model (if there is a choice). However code written using these keywords will be portable providing it obeys the constraints of the model described above.

This model also corresponds well with other implementations using the near and far keywords, although such implementations do not appear to have been designed around a formal, portable model.

## 11. What Went Wrong

There are numerous error messages that the compiler may produce. Most of these relate to errors in the source code (syntax errors of various kinds and so forth) but some represent limitations, particularly of memory. The two passes most likely to be affected by memory limitations are the code generator and the optimizer. The code generator will issue the message "No room" if it runs out of dynamic memory. This can usually be eliminated by simplifying the expression at the line nominated in the error message. The more complex the expression, the more memory required to store the tree representing it. Reducing the number of symbols used in the program will also help.

Note that this error is different from the message "Can't generate code for this expression" which indicates that the expression is in some way too difficult for the code generator to handle. This message will be encountered very infrequently, and can be eliminated by changing the expression in some way, e.g. computing an intermediate value into a temporary variable.

The optimizer reads the assembler code for a whole function into memory at one time. Very large functions will not fit, giving the error message "Optim: out of memory in \_func" where func is the name of the function responsible. In this case the function should be broken up into smaller functions. This will only occur with functions with several hundred lines of C source code. Good coding practice will normally limit functions to less than 50 lines each.

If a pass exits with the message "Error closing file", or "Write error on file", this usually indicates that there is insufficient room on the current disk.

If you use a wordprocessing editor such as Wordstar, ensure that you use the "non-document" mode or whatever the corresponding mode is. The edited file should not contain any characters with the high bit set, and the line feed at the end of the line must be present. Lines should be not more than 255 characters long.

When using floating point, ensure that you use a `-LF` flag at the END of the command line, to cause the floating point library to be searched. This will cause floating versions of `printf` and `scanf` to be linked in, as well as specific floating point routines.

If the non-floating version of `printf` is used with a floating format such as `%f` then it will simply print the letter `f`.

If the linker gives an "Undefined symbol" message for some symbol which you know nothing about, it is possible that it is a library routine which was not found during the library search due to incorrect library ordering. In this case you can search the library twice, e.g. for the standard library add a `-LC` to the end of the C command line, or `-LF` for the floating library. If you have specified the library by name simply repeat its name.

(Use this page for notes.)

## 12. Z80 Assembler Reference Manual

### 12.1. Introduction

The assembler incorporated in the HI-TECH C compiler system is a full-featured relocating macro assembler accepting Zilog mnemonics. These mnemonics and the syntax of the Z80 assembly language are described in the "Z80 Assembly Language Handbook" published by Zilog and are included at the end of this manual as a reference. The assembler implements certain extensions to the operands allowed, and certain additional pseudo-ops, which are described here. The assembler also accepts the additional opcodes for the Hitachi 64180 and Z180 processors.

### 12.2. Usage

The assembler is named `zas`, and is invoked as follows:

ZAS options files ...

The files are one or more assembler source files which will be assembled, but note that all the files are assembled as one, not as separate files. To assemble separate files, the assembler must be invoked on each file separately. The options are zero or more options from the following list:

- N Ignore arithmetic overflow in expressions. The -N option suppresses the normal check for arithmetic overflow. The assembler follows the "Z80 Assembly Language Handbook" in its treatment of overflow, and in certain instances this can lead to an error where in fact the expression does evaluate to what the user intended. This option may be used to override the overflow checking.
- J Attempt to optimize jumps to branches. The -J option will request the assembler to attempt to assemble jumps and conditional jumps as relative branches where possible. Only those conditional jumps with branch equivalents will be optimized, and jumps will only be optimized to branches where the target is in branch range. Note that the use of this option slows the assembly down, due to the necessity for the assembler to make an additional pass over the input code.
- U Treat undefined symbols as external. The -U option will suppress error messages relating to undefined symbols. Such symbols are treated as externals in any case. The use of this option will not alter the object code generated, but merely serves to suppress the error messages.

**-Ofile** Place the object code in **file**. The default object file name is constructed from the name of the first source file. Any suffix or file type (i.e. anything following the rightmost dot ('.') in the name is stripped, and the suffix **.obj** appended. Thus the command

```
ZAS file1.as file2.z80
```

will produce an object file called **file1.obj**. The use of the **-O** option will override this default convention, allowing the object file to be arbitrarily named. For example:

```
ZAS -ox.obj file1.obj
```

will place the object code in **x.obj**.

**-Llist** Place an assembly listing in the file **list**, or on standard output if **list** is null. A listfile may be produced with the **-L** option. If a file name is supplied to the option, the list file will be created with that name, otherwise the listing will be written to standard output (i.e. the console). List file names such as **CON:** and **LST:** are acceptable.

**-Wwidth** The listing is to be formatted for a printer of given **width**. The **-W** option specifies the width to which the listing is to be formatted. E.g.

```
ZAS -Llst: -W80 x.as
```

will output a listing formatted for an 80 column printer to the list device.

**-C** This options requests ZAS to produce cross reference information in a file. The file will be called **xxx.crf** where **xxx** is the base part of the first source file name. It will then be necessary to run the **CREF** utility to turn this information into a formatted listing.

### 12.3. The Assembly Language

As mentioned above, the assembly language accepted by **zas** is based on the Zilog mnemonics. You should have some reference book such as the "Z80 Assembly Language Handbook". Described below are those areas where **zas** differs, or has extensions, compared to the standard Zilog assembly language.

#### 12.3.1. Symbols

The symbols (labels) accepted by the assembler may be of any length, and all characters are significant. The characters used to form a symbol may be chosen from the upper and lower case alphabets, the digits 0-9, and the special symbols underscore ('\_'), dollar ('\$') and question mark

('?'). The first character may not be numeric. Upper and lower case are distinct. The following are all legal and distinct symbols.

```
An_identifier
an_identifier
an_identifier1
$$$
?$_123455
```

Note that the symbol \$ is special (representing the current location) and may not be used as a label. Nor may any opcode or pseudo-op mnemonic, register name or condition code name. You should note the additional condition code names described later.

### 12.3.1.1. Temporary Labels

The assembler implements a system of temporary labels, useful for use within a localized section of code. These help eliminate the need to generate names for labels which are referenced only in the immediate vicinity of their definition, for example where a loop is implemented.

A temporary label takes the form of a digit string. A reference to such a label requires the same digit string, plus an appended b or f to signify a backward or forward reference respectively. Here is an example of the use of such labels.

```
entry_point:      ;This is referenced from far away
    ld    b,10
1:    dec    c
    jr    nz,2f    ;if zero, branch forward to 2:
    ld    c,8
    djnz  1b      ;decrement and branch back to 1:
    jr    1f      ;this does not branch to the ;same label as the djnz
2:    call  fred   ;get here from the jr nz,2f
1:    ret                ;get here from the jr 1f
```

The digit string may be any positive decimal number 0 to 65535. A temporary label value may be re-used any number of times. Where a reference to e.g. 1b is made, this will reference the closest label 1: found by looking backwards from the current point in the file. Similarly 23f will reference the first label 23: found by looking forwards from the current point in the file.

### 12.3.2. Constants

Constants may be entered in one of the radices 2, 8, 10 or 16. The default is 10. Constants in the other radices may be denoted by a trailing character drawn from the following set:

Character	Radix	Name
B	2	binary
O	8	octal
Q	8	octal
O	8	octal
Q	8	octal
H	16	hexadecimal
H	16	hexadecimal

Hexadecimal constants may also be specified in C style, for example LD A,0x21. Note that a lower case b may not be used to indicate a binary number, since 1b is a backward reference to a temporary label 1:.

#### 12.3.2.1. Character Constants

A character constant is a single character enclosed in single quotes ('). Multi character constants may be used only as an operand to a DEFM pseudo-op.

#### 12.3.2.2. Floating Constants

A floating constant in the usual notation (e.g. 1.234 or 1234e-3) may be used as the operand to a DEFF pseudo-op.

#### 12.3.2.3. Opcode Constants

Any z80 opcode may be used as a constant in an expression. The value of the opcode in this context will be the byte that the opcode would have assembled to if used in the normal way. If the opcode is a 2-byte opcode (CB or ED prefix byte) only the second byte of the opcode will be used. This is particularly useful when setting up jump vectors. For example:

```
Ld    a,jp          ;a jump instruction
Ld    (0),a         ;0 is jump to warm boot
Ld    hl,boot       ;done here
ld (1),hl
```

### 12.3.3. Expressions

Expressions are constructed largely as described in the "Z80 Assembly Language Handbook".

#### 12.3.3.1. Operators

The following operators may be used in expressions:

<u>Operator</u>	<u>Meaning</u>
&	Bitwise AND
*	Multiplication
+	Addition
-	Subtraction
.and.	Bitwise AND
.eq.	Equality test
.gt.	Signed greater than
.high.	Hi byte of operand
.low.	Low byte of operand
.lt.	Signed less than
.mod.	Modulus
.not.	Bitwise complement
.or.	Bitwise or
.shl.	Shift left
.shr.	Shift right
.ult.	Unsigned less than
.ugt.	Unsigned greater than
.xor.	Exclusive or
/	Divison
<	Signed less than
=	Equality
>	Signed greater than
^	Bitwise or

Operators starting with a dot "." should be delimited by spaces, thus label .and. 1 is valid but label.and.1 is not.

#### 12.3.3.2. Relocatability

Zas produces object code which is relocatable. This means that it is not necessary to specify assembly time where the code is to be located in memory. It is possible to do so, by use of the ORG pseudo-op, however the preferred approach is to use program sections or psects. A psect is a named section of the program, in which code or data may be defined at assembly time. All parts of a psect will be loaded contiguously into memory, even if they were defined in separate files, or in the same file but separated by code for another psect. For example, the following code will load some executable instructions into the psect named text, and some data bytes into the data psect.

```

        psect text, global

alabel: ld    hl, astring
call   putit
ld     hl, anotherstring

        psect data, global
astring:
        defm 'A string of chars'
        defb 0
anotherstring:
        defm 'Another string'
        defb 0

        psect text

putit: ld    a,(hl)
      or    a
      ret   z
      call outchar
      inc  hl
      jr   putit

```

Note that even though the two blocks of code in the text psect are separated by a block in the data psect, the two text psect blocks will be contiguous when loaded by the linker. The instruction "ld hl,anotherstring" will fall through to the label "putit:" during execution. The actual location in memory of the two psects will be determined by the linker. See the linker manual for information on how psect addresses are determined.

A label defined in a psect is said to be relocatable, that is, its actual memory address is not determined at assembly time. Note that this does not apply if the label is in the default (unnamed) psect, or in a psect declared absolute (see the PSECT pseudo-op description below). Any labels declared in an absolute psect will be absolute, that is their address will be determined by the assembler.

With the version of ZAS supplied with version 7 or later of HI-TECH C, relocatable expressions may be combined freely in expressions. Older versions of ZAS allowed only limited arithmetic on relocatable expressions.

#### 12.3.4. Pseudo-ops

The pseudo-ops are based on those described in the "Z80 Assembly Language Handbook", with some additions.

##### 12.3.4.1. DEFB, DB

This pseudo-op should be followed by a comma-separated list of expressions, which will be

assembled into sequential byte locations. Each expression must have a value between -128 and 255 inclusive. DB can be used as a synonym for DEFB. Example:

```
DEFB    10, 20, 'a', 0FFH
DB      'hello world',13,10,0
```

#### 12.3.4.2. DEFF

This pseudo-op assembles floating point constants into 32 bit HI-TECH C format floating point constants. For example:

```
pi:     DEFF    3.14159
```

#### 12.3.4.3. DEFW

This operates in a similar fashion to DEFB, except that it assembles expressions into words, without the value restriction. Example:

```
DEFW    -1, 3664H, 'A', 3777Q
```

#### 12.3.4.4. DEFS

DeFS reserves memory locations without initializing them. Its operand is an absolute expression, representing the number of bytes to be reserved. This expression is added to the current location counter. Note however that locations reserved by DEFS may be initialized to zero by the linker if the reserved locations are in the middle of the program. Example:

```
DEFS    20h      ;reserve 32 bytes of memory
```

#### 12.3.4.5. EQU

Equ sets the value of a symbol on the left of EQU to the expression on the right. It is illegal to set the value of a symbol which is already defined. Example:

```
SIZE    equ      46
```

#### 12.3.4.6. DEFL

This is identical to EQU except that it may redefine existing symbols. Example:

```
SIZE    defl    48
```

#### 12.3.4.7. DEFM

Defm should be followed by a string of characters, enclosed in single quotes. The ASCII values of these characters are assembled into successive memory locations. Example:

```
DEFM    'A string of funny *@$ characters'
```

#### 12.3.4.8. END

The end of an assembly is signified by the end of the source file, or the END pseudo-op. The END pseudo-op may optionally be followed by an expression which will define the start address of the program. This is not actually useful for CP/M. Only one start address may be defined per program, and the linker will complain if there are more. Example:

```
END     somelabel
```

#### 12.3.4.9. COND, IF, ELSE, ENDC

Conditional assembly is introduced by the COND pseudoop. The operand to COND must be an absolute expression. If its value is false (zero) the code following the COND up to the corresponding ENDC pseudo-op will not be assembled. COND/ENDC pairs may be nested. IF may be used as a synonym for COND. The ELSE pseudo operation may be included within a COND/ENDC block, for example:

```
IF      CPM
Call    5
ELSE
call os_func
ENDC
```

#### 12.3.4.10. ELSE

See COND.

#### 12.3.4.11. ENDC

See COND.

## 12.3.4.12. ENDM

See MACRO.

## 12.3.4.13. PSECT

This pseudo-op allows specification of relocatable program sections. Its arguments are a psect name, optionally followed by a list of psect flags. The psect name is a symbol constructed according to the same rules as for labels, however a psect may have the same name as a label without conflict. Psect names are recognized only after a PSECT pseudo-op. The psect flags are as follows:

ABS	Psect is absolute
GLOBAL	Psect is global
LOCAL	Psect is not global
OVRLD	Psect is to be overlapped by linker
PURE	Psect is to be read-only

If a psect is global, the linker will merge it with any other global psects of the same name from other modules. Local psects will be treated as distinct from any other psect from another module. Psects are global by default.

By default the linker concatenates code within a psect from various modules. If a psect is specified as OVRLD, the linker will overlap each module's contribution to that psect. This is particularly useful when linking modules which initialize e.g. interrupt vectors.

The PURE flag instructs the linker that the psect is to be made read-only at run time. The usefulness of this flag depends on the ability of the linker to enforce the requirement. CP/M fails miserably in this regard.

The ABS flag makes a psect absolute. The psect will be loaded at zero. This is useful for statically initializing interrupt vectors and jump tables. Examples:

PSECT	text, global, pure
PSECT	data, global
PSECT	vectors, ovrlld

## 12.3.4.14. GLOBAL

Global should be followed by one more symbols (comma separated) which will be treated by the assembler as global symbols, either internal or external depending on whether they are defined within the current module or not. Example:

```
GLOBAL    label1, putchar, _printf
```

#### 12.3.4.15. ORG

An ORG pseudo-op sets the current psect to the default (absolute) psect, and the location counter to its operand, which must be an absolute expression. Example:

```
ORG      100H
```

#### 12.3.4.16. MACRO

This pseudo-op defines a macro. It should be either preceded or followed by the macro name, then optionally followed by a comma-separated list of formal parameters. The lines of code following the MACRO pseudo-op up to the next ENDM pseudo-op will be stored as the body of the macro. The macro name may subsequently be used in the opcode part of an assembler statement, followed by actual parameters. The text of the body of the macro will be substituted at that point, with any use of the formal parameters substituted with the corresponding actual parameter. For example:

```
Print    MACRO string
          psect data
999:     db      string,'$'
          psect text
          ld      de,999b
          ld      c,9
          call    5
          ENDM
```

When used, this macro will expand to the 3 instructions in the body of the macro, with the actual parameters substituted for func and arg. Thus

```
print   'hello world'
```

expands to

```
          psect data
999:     db      'hello world','$'
          psect text
          ld      de,999b
          ld      c,9
          call    5
```

Macro arguments can be enclosed in angle brackets ('<' and '>') to pass arbitrary text including delimiter characters like commas as a single argument. For example, suppose you wanted to use the print macro defined above to print a string which includes the carriage return and linefeed characters. The macro invocation:

```
print 'hello world',13,10
```

would fail because 13 and 10 are treated as extra arguments and ignored. In order to pass a string which includes commas as a single argument, you could write:

```
print <'hello world',13,10>
```

which would cause the text 'hello world',13,10 to be passed through as a single argument. This would expand to the following code:

```

          psect data
999:      db          'hello world',13,10,'$'
          psect text
          ld          de,999b
          ld          c,9
          call        5

```

ZAS supports two forms of macro declaration for compatibility with older versions of ZAS and other Z80 assemblers. The macro name may be declared either in the label field before the MACRO pseudo-op, or in the operand field after the MACRO pseudo-op. Thus these two MACRO declarations are equivalent:

```

Bdos     MACRO func,arg
          ld          de,arg
          ld          c,func
          call        5
          ENDM

```

and

```

MACRO    bdos,func,arg
          ld          de,arg
          ld          c,func
          call        5
          ENDM

```

#### 12.3.4.17. LOCAL

The LOCAL pseudo-op allows unique labels to be defined for each expansion of a macro. Any symbols listed after the LOCAL directive will have a unique assembler-generated symbol substituted for them when the macro is expanded. For example:

```

Copy  MACRO source,dest,count
      LOCAL nocopy
      push    af
      push    bc
      lld     bc,source
      ld      a,b
      or      c
      jr      z,nocopy
      push    de
      push    hl
      ld      de,dest
      ld      hl,source
      ldir
      pop     hl
      pop     de
nocopy: pop     bc
        pop     af
      ENDM

```

when expanded will include a unique assembler generated label in place of nocopy. For example, copy (recptr),buf,(recsize) will expand to:

```

      push    af
      push    bc
      ld      bc,(recsize)
      ld      a,b
      or      c
      jr      z,??0001
      push    de
      push    hl
      ld de,buf
      ld hl,(recptr)
      ldir
      pop hl
      pop de
??0001: pop bc pop af

```

if invoked a second time, the label nocopy would expand to ??0002.

#### 12.3.4.18. REPT

The REPT pseudo-op defines a temporary macro which is then expanded a number of times, as determined by its argument. For example:

```

REPT 3
ld (hl),0
inc hl
ENDM

```

will expand to

```

ld (hl),0
inc hl
ld (hl),0
inc hl
ld (hl),0
inc hl

```

### 12.3.5. IRP and IRPC

The IRP and IRPC directives are similar to REPT, however instead of repeating the block a fixed number of times it is repeated once for each member of an argument list. In the case of IRP the list is a conventional macro argument list, in the case of IRPC it is successive characters from a string. For example:

```

                IRP      string,<'hello world',13,10>,'arg2'
LOCAL str
psect data
str:           db      string,'$'
psect text
ld            c,9
ld            de,str
call         5
ENDM

```

would expand to

```

                psect data
??0001:       db      'hello world',13,10,'$'
psect text
ld            c,9
ld            de,??0001
call         5
                psect data
??0002:       db      'arg2','$'
psect text
ld            c,9
ld            de,??0002
call         5

```

Note the use of LOCAL labels and angle brackets in the same manner as with conventional macros.

IRPC is best demonstrated using the following example:

```

IRPC      char,ABC
ld        c,2
ld e,'char'
call 5
ENDM

```

will expand to:

```

ld      c,2
ld      e,'A'
call   5
ld      c,2
ld      e,'B'
call   5
ld      c,2
ld      e,'C'
call   5

```

### 12.3.6. Extended Condition Codes

The assembler recognizes several additional condition codes. These are:

Code	Equivalent	Meaning
alt	m	Arithmetic less than
llt	c	Logical less than
age	p	Arithmetic greater or equal
lge	nc	Logical greater or equal
di		Use after ld a,i for test-
ei		ing state of interrupt
		enable flag - enabled or
		enable flag - enabled or
		disabled respectively

### 12.4. Assembler Directives

An assembler directive is a line in the source file which produces no code, but rather which modifies the behaviour of the assembler. Each directive is recognized by the presence of an asterisk in the first column of the line, followed immediately by a word, only the first character of which is looked at. The line containing the directive itself is never listed. The directives are:

- \*Title      Use the text following the directive as a title for the listing.
- \*Heading    Use the text following the directive as a subtitle for the listing; also causes an \*Eject.
- \*List        May be followed by ON or OFF to turn listing on or off respectively. Note that this directive may be used inside a macro or include file to control listing of that macro or include file. The previous listing state will be restored on exit from the macro or include file.

\*Include The file named following the directive will be included in the assembly at that point.

\*Eject A new page will be started in the listing at that point. A form feed character in the source will have the same effect.

Some examples of the use of these directives:

\*Title Widget Control Program

\*Heading Initialization Phase

\*Include widget.i

## 12.5. Diagnostics

An error message will be written on the standard error stream for each error encountered in the assembly. This message identifies the file name and line number and describes the error. In addition the line in the listing where the error occurred will be flagged with a single character to indicate the error. The characters and the corresponding messages are:

- A: Absolute expression required
- B: Bad arg to \*L  
Bad arg to IM  
Bad bit number  
Bad character constant  
Bad jump condition
- D: Directive not recognized  
Digit out of range
- E: EOF inside conditional  
Expression error
- G: Garbage after operands  
Garbage on end of line
- I: Index offset too large
- J: Jump target out of range
- L: Lexical error
- M: Multiply defined symbol
- O: Operand error
- P: Phase error  
Psect may not be local and global
- R: Relocation error
- S: Size error  
Syntax error
- U: Undefined symbol  
Undefined temporary label  
Unterminated string

## 12.6. Z80/Z180/64180 Instruction Set

The remainder of this chapter is devoted to a complete instruction set listing for the Z80, Z180, 64180 and NSC800 processors. The Z180 and 64180 will execute all Z80 instructions, although the timing is different.





































## 13. Linker Reference Manual

HI-TECH C incorporates a relocating assembler and linker to permit separate compilation of C source files. This means that a program may be divided into several source files, each of which may be kept to a manageable size for ease of editing and compilation, then each object file compiled separately and finally all the object files linked together into a single executable program.

The assembler is described in the machine-specific manual. This appendix describes the theory behind and the usage of the linker.

### 13.1. Relocation and Psects

The fundamental task of the linker is to combine several relocatable object files into one. The object files are said to be relocatable since the files have sufficient information in them so that any references to program or data addresses (e.g. the address of a function) within the file may be adjusted according to where the file is ultimately located in memory after the linkage process. Thus the file is said to be relocatable. Relocation may take two basic forms; relocation by name, i.e. relocation by the ultimate value of a global symbol, or relocation by psect, i.e. relocation by the base address of a particular section of code, for example the section of code containing the actual executable instructions.

#### 13.1.1. Program Sections

Any object file may contain bytes to be stored in memory in one or more program sections, which will be referred to as psects. These psects represent logical groupings of certain types of code bytes in the program. The section of the program containing executable instructions is normally referred to as the text psect. Other sections are the initialized data psect, called simply the data psect, and the uninitialized data psect, called the bss psect.

In fact the linker will handle any number of psects, and in fact more may be used in special applications. However, the C compiler uses only the three mentioned, and the names text, data and bss are simply chosen for identification; the linker assigns no special significance to the name of a psect.

The difference between the data and bss psects may be exemplified by considering two external variables; one is initialized to the value 1, and the other is not initialized. The first will be placed into the data psect, and the second in the bss psect. The bss psect is always cleared to zeros on startup of the program, thus the second variable will be initialized at run time to zero. The first will however occupy space in the program file, and will maintain its initialized value of 1 at startup. It is quite possible to modify the value of a variable in the data psect during execution, however it is better practice not to do so, since this leads to more consistent use of variables, and allows for restartable and romable programs.

The text psect is the section into which all executable instructions are placed. On CP/M-80 the text psect will normally start at the base of the TPA, which is where execution commences. The data psect will normally follow the text psect, and the bss will be last. The bss does not occupy space in the program (.COM) file. This ordering of psects may be overridden by an option to the linker. This is especially useful when producing code for special hardware.

For MS-DOS and CP/M-86 the psects are ordered in the same way, but since the 8086 processor has segment registers providing relocation, both the text and data psects start at 0, even though they will be loaded one after the other in memory. This allows 64k code and 64k data and stack. Sufficient information is placed in the executable file (.EXE or .CMD) for the operating system to load the program in memory.

### 13.1.2. Local Psects and the Large Model

Since for practical purposes the psects are limited to 64K on the 8086, to allow more than 64K code the compiler makes use of local psects. A psect is considered local if the .psect directive has a LOCAL flag. Any number of local psects may be linked from different modules without being combined even if they have the same name. Note however that no local psect may have the same name as a global psect.

All references to a local psect within the same module (or within the same library) will be treated as references to the same psect. Between modules, however, two local psects of the same name are treated as distinct. In order to allow collective referencing of local psects via the -P option (described later) a local psect may have a class name associated with it. This is achieved with the **CLASS** flag on the .psect directive.

### 13.2. Global Symbols

The linker handles only symbols which have been declared as global to the assembler. From the C source level, this means all names which have storage class external and which are not declared as static. These symbols may be referred to by modules other than the one in which they are defined. It is the linker's job to match up the definition of a global symbol with the references to it.

### 13.3. Operation

A command to the linker takes the following form:

LINK options files ...

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. Files is one or more object files, and zero or more library names. The options will be recognized in upper or lower case. They are recognized by the linker are as follows:

- R        Leave the output relocatable.
- L        Retain absolute relocation info. -LM will retain only segment relocation information.
- I        Ignore undefined symbols.
- N        Sort symbols by address.
- Caddr   Produce a binary output file offset by addr.
- S        Strip symbol information from the output file.
- X        Suppress local symbols in the output file.
- Z        Suppress trivial (compiler-generated) symbols in the output file.
- Oname   Call the output file name.
- Pspec   Spec is a psect location specification.
- Mname   Write a link map to the file name.
- Usymbol Make symbol initially undefined.
- Dfile    Write a symbol file.
- Wwidth   Specify map width.

Taking each of these in turn:

The -R option will instruct the linker to leave the output file (as named by a -O option, or l.obj by default) relocatable. This is normally because there are further files to be linked in, and the output of this link will be used as input to the linker subsequently. Without this option, the linker will make the output file absolute, that is with all relocatable addresses made into absolute references. This option may not be used with the -L or -C options.

The -L option will cause the linker to output null relocation information even though the file will be absolute. This information allows self-relocating programs to know what addresses must be relocated at run time. This option is not usable with the -C option. In order to create an executable file (i.e. a .COM file) the program objtohex must be used. If a -LM option is used, only segment relocation information will be retained. This is used in conjunction with the large memory model. Objtohex will use the relocation information (when invoked with a -L flag) to insert segment relocation addresses into the executable file.

The -I option is used when it is desired to link code which contains symbols which are not defined in any module. This is normally only used during top-down program development, when routines are referenced in code written before the routines themselves have been coded.

When obtaining a link map via the -M option, the symbol table is by default sorted in order of symbol name. To sort in order of address, the -N option may be used.

The output of the linker is by default an object file. To create an executable program, this must be converted into an executable image. For CP/M this is a .COM file, which is simply an image of the executable program as it should appear in memory, starting at location 100H. The linker will produce such a file with the -C100H option. File formats for other applications requiring an image binary file may also be produced with the -C option. The address following the -C may be given in decimal (default), octal (by using o or O suffix) or hexadecimal (by using an h or H suffix).

Note that because of the complexity of the executable file formats for MS-DOS and CP/M-86, LINK will not produce these (.EXE and .CMD resp.) formats directly. The compiler automatically runs OBJTOHEX with appropriate options to generate the correct file format.

The -S, -X and -Z options, which are meaningless when the -C option is used, will strip respectively all symbols, all local symbols or all trivial local symbols from the output file. Trivial symbols are symbols produced by the compiler, and have the form of one of a set of alphabetic characters followed by a digit string.

The default output file name is *l.obj*, or *l.bin* when the -C option is used. This may be overridden by the -*Oname* option. The output file will be called *name* in this instance. Note that no suffix is appended to the name; the file will be called exactly the argument to the option.

For certain specialized applications, e.g. producing code for an embedded microprocessor,

it is necessary to specify to the linker at what address the various psects should be located. This is accomplished with the `-P` option. It is followed by a specification consisting of a commaseparated list of psect names, each with an optional address specification. In the absence of an address specification for a psect listed, it will be concatenated with the previous psect. For example

```
-Ptext=0c000h,data,bss=8000h
```

This will cause the text psect to be located at 0C000H, the data psect to start at the end of the text psect, and the bss psect to start at 8000H. This may be for a processor with ROM at 0C000H and RAM at 8000H.

Where the link address, that is the address at which the code will be addressed at execution time, and the load address, that is the address offset within the output file, are different (e.g for the 8086) it is possible to specify the load address separately from the link address. For example:

```
-Ptext=100h/0,data=0C000h/
```

This specification will cause the text segment to be linked for execution at 100h, but loaded in the output file at 0, while the data segment will be linked for 0C000h, but loaded contiguously with the text psect in the file. Note that if the slash (`/`) is omitted, the load address is the same as the link address, while if the slash is supplied, but not followed by an address, the psect will be loaded after the previous psect.

In order to specify link and load addresses for local psects, the group name to which the psects belong may be used in place of a global psect name. The local psects will then have a link address as specified in the `-P` option, and load addresses incrementing upwards from the specified load address.

The `-Mname` option requests a link map, containing symbol table and module load address information to be written onto the file name. If name is omitted, the map will be written to standard output. `-W` may be used to specify the desired width of the map.

The `-U` option allows the specification to the linker of a symbol which is to be initially entered into the symbol table as undefined. This is useful when loading entirely from libraries. More than one `-U` flag may be used.

If it is desired to use the debugger on the program being linked, it is useful to produce a symbol file. The `-Dfile` option will write such a symbol file onto the named *file*, or *l.sym* if no file is given. The symbol file consists of a list of addresses and symbols, one per line.

### 13.4. Examples

Here are some examples of using the linker. Note however that in the normal case it is not necessary to invoke the linker explicitly, since it is invoked automatically by the C command.

```
LINK -MMAP -C100H START.OBJ MAIN.OBJ A:LIBC.LIB
```

This command links the files start.obj and main.obj with the library a:libc.lib. Only those modules that are required from the library will be in fact linked in. The output is to be in .COM format, placed in the default file l.bin. A map is to be written to the file of the name map. Note that the file start.obj should contain startup code, and in fact the lowest address code in that file will be executed when the program is run, since it will be at 100H.

```
LINK -X -R -OX.OBJ FILE1.OBJ FILE2.OBJ A:LIBC.LIB
```

The files file1.obj and file2.obj will be linked with any necessary routines from a:libc.lib and left in the file x.obj. This file will remain relocatable. Undefined symbols will not cause an error. The file x.obj will probably later be the object of another link invocation. All local symbols will be stripped from the output file, thus saving space.

### 13.5. Invoking the Linker

The linker is called LINK, and normally resides on the A: drive, under CP/M, or in the directory A:\HITECH\ under MS-DOS. It may be invoked with no arguments, in which case it will prompt for input from standard input. If the standard input is a file, no prompts will be printed. The input supplied in this manner may contain lower case, whereas CP/M converts the entire command line to upper case by default. This is useful with the -U and -P options. This manner of invocation is generally useful if the number of arguments to LINK is large. Even if the list of files is too long to fit on one line, continuation lines may be included by leaving a backslash ("\") at the end of the preceding line. In this fashion, LINK commands of almost unlimited length may be issued.

## 14. Librarian

The librarian program, LIBR, has the function of combining several object files into a single file known as a library. The purposes of combining several such object modules are several.

- a. fewer files to link
- b. faster access
- c. uses less disk space

In order to make the library concept useful, it is necessary for the linker to treat modules in a library differently from object files. If an object file is specified to the linker, it will be linked into the final linked module. A module in a library, however, will only be linked in if it defines one or more symbols previously known, but not defined, to the linker. Thus modules in a library will be linked only if required. Since the choice of modules to link is made on the first pass of the linker, and the library is searched in a linear fashion, it is possible to order the modules in a library to produce special effects when linking. More will be said about this later.

### 14.1. The Library Format

The modules in a library are basically just concatenated, but at the beginning of a library is maintained a directory of the modules and symbols in the library. Since this directory is smaller than the sum of the modules, the linker is speeded up when searching a library since it need read only the directory and not all the modules on the first pass. On the second pass it need read only those modules which are required, seeking over the others. This all minimizes disk i/o when linking.

It should be noted that the library format is geared exclusively toward object modules, and is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

### 14.2. Using

The librarian program is called LIBR, and the format of commands to it is as follows:

```
LIBR k file.lib file.obj ...
```

Interpreting this, LIBR is the name of the program, k is a key letter denoting the function requested of the librarian (replacing, extracting or deleting modules, listing modules or symbols), file.lib is the name of the library file to be operated on, and file.obj is zero or more object file names.

The key letters are:

r	replace modules
d	delete modules
x	extract modules
m	list module names
s	list modules with symbols

When replacing or extracting modules, the file.obj arguments are the names of the modules to be replaced or extracted. If no such arguments are supplied, all the modules in the library will be replaced or extracted respectively. Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the r key is used and the library does not exist, it will be created.

Under the d key letter, the named object files will be deleted from the library. In this instance, it is an error not to give any object file names.

The m and s key letters will list the named modules and, in the case of the s key letter, the symbols defined or referenced within (global symbols only are handled by the librarian). As with the r and x key letters, an empty list of modules means all the modules in the library.

### 14.3. Examples

Here are some examples of usage of the librarian.

LIBR m file.lib	List all modules in the library file.lib.
LIBR s file.lib a.obj b.obj c.obj	List the global symbols in the modules a.obj, b.obj and c.obj
LIBR r file.lib 1.obj 2.obj	Replace the module 1.obj in the file file.lib with the contents of the object file 1.obj, and repeat for 2.obj. If the object module is not already present in the library, append it to the end.
LIBR x file.lib	Extract, without deletion, all the modules in file.lib and write them as object files on disk.
LIBR d file.lib a.obj b.obj 2.obj	Delete the object modules a.obj, b.obj and 2.obj from the library file.lib.

#### 14.4. Supplying Arguments

Since it is often necessary to supply many object file arguments to LIBR, and command lines are restricted to 127 characters by CP/M and MS-DOS, LIBR will accept commands from standard input if no command line arguments are given. If the standard input is attached to the console, LIBR will prompt. Multiple line input may be given by using a backslash as a continuation character on the end of a line. If standard input is redirected from a file, LIBR will take input from the file, without prompting. For example:

```
LIBR libr> r file.lib 1.obj 2.obj 3.obj \ libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the .obj files had been typed on the command line. The libr> prompts were printed by LIBR itself, the remainder of the text was typed as input.

```
LIBR <lib.cmd
```

Libr will read input from lib.cmd, and execute the command found therein. This allows a virtually unlimited length command to be given to LIBR.

#### 14.5. Listing Format

A request to LIBR to list module names will simply produce a list of names, one per line, on standard output. The s keyletter will produce the same, with a list of symbols after each module name. Each symbol will be preceded by the letter D or U, representing a definition or reference to the symbol respectively. The -W option may be used to determine the width of the paper for this operation. For example LIBR -w80 s file.lib will list all modules in file.lib with their global symbols, with the output formatted for an 80 column printer or display.

#### 14.6. Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

## 14.7. Error Messages

Libr issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the `-w` option was used. In this case all warning messages will be suppressed.

## 15. Objtohex

The HI-TECH linker is capable of producing simple binary files, or object files as output. Any other format required must be produced by running the utility program OBJTOHEX. This allows conversion of object files as produced by the linker into a variety of different formats, including various hex formats. The program is invoked thus:

OBJTOHEX options inputfile outputfile

All of the arguments are optional. If outputfile is omitted it defaults to l.hex or l.bin depending on whether the -b option is used. The inputfile defaults to l.obj.

The options are:

- Baddr**      Produce a binary image output. This is similar to the -C option of the linker. If **addr** is supplied, the start of the image file will be offset by **addr**. If **addr** is omitted, the first byte in the file will be the lowest byte initialized. **Addr** may be given in decimal, octal or hexadecimal. The default radix is decimal, and suffix letters of o or O indicate octal, and h or H indicate hex. Thus -B100H will produce a file in .COM format.
- I**            Include symbol records in the Intel format hex output. Each symbol record has a form similar to an object record, but with a different record type. The data bytes in the record are the symbol name, and the address is the value of the symbol. This is useful for downloading to ROM debuggers.
- C**            Read a checksum specification from the standard input. The checksum specification is described below. Typically the specification will be in a file.
- Estack**     This option produces an MS-DOS .EXE format file. The optional stack argument will determine the maximum stack size the program will be allocated on execution. By default the program will be allocated the maximum stack available, up to the limit of 64K data. If a stack argument is supplied, the stack size will not exceed the argument. This is useful to limit the amount of memory a program will use. The stack argument takes the same form as the argument to -B above.
- 8stack**     This option will produce a CP/M-86 .CMD file. The stack argument is the same as for the -E option.

- Astack* This is used when producing a.out format files for unix systems (specifically Venix-86). If the stack argument is zero, the size of the data segment will be 64k, otherwise the stack will be placed below the data segment, and its size set to stack. This must be co-ordinated with appropriate arguments to the -p option of the linker.
  
- M This flag will instruct objtohex to produce Motorola 'S' format hex output.
  
- L This option is used when producing large model programs; the linker will have been used with the -LM option to retain segment relocation information in the object file. Use of the -L option to objtohex will cause it to convert that segment relocation information into appropriate data in the executable file for use when the program is loaded. Either the operating system or the run-time startup code will use the relocation data to adjust segment references based on where in memory the program is actually loaded. If the -L option is followed by a symbol name, then the relocation information will be stored at the address represented by that symbol in the output file, e.g. -L\_\_*Bbss* will cause it to be stored at the base of the bss psect (\_\_*Bbss* is defined by the linker to be the load address of the bss psect). If the special symbol *Doshdr* is used then the relocation information will be stored in the .EXE file header. This is only valid in conjunction with the -E option.
  
- S The -S option instructs objtohex to write a symbol file. The symbol file name is given after the -S, e.g. -*Sxx.sym*.

Unless another format is specifically requested, objtohex will produce a file in Intel hex format. This is suitable for down-line loading, PROM programming etc. The HP format is useful for transferring code to an HP64000 for emulation or PROM programming.

The checksum specification allows automated checksum calculation. The checksum specification takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

addr1-addr2 where1-where2 +offset

All of addr1, addr2, where1, where2 and offset are hex numbers, without the usual H suffix. Such a specification says that the bytes at addr1 through to addr2 inclusive should be summed and the sum placed in the locations where1 through where2 inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, where1 should be less than where2, and vice versa. The +offset is optional, but if supplied, the value offset will be used to initialize the checksum. Otherwise it is initialized to zero. For example:

0005-1FFF 3-4 +1FFF

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialized with 1FFFH to provide protection against an all zero rom, or a rom misplaced in memory. A run time check of this checksum would add the last address of the rom being checksummed into the checksum. For the rom in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

Use this page for notes

## 16. Cref

The cross reference list utility CREF is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing. A raw cross-reference file is produced with the -CR option to the compiler. The assembler will generate a raw cross-reference file with a -C option (Z80 or 8086 assemblers) or by using an OPT CRE directive (6800 series assemblers) or a REF control line (8096 assembler).. The general form of the CREF command is:

**CREF *options files***

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files. CREF takes the following options:

- O*outfile*** Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively using the -O option an output file name may be specified, e.g. **-O*xxx.lst***.
  
- P*width*** This option allows the specification of the width to which the listing is to be formatted, e.g. **-P*132*** will format the listing for a 132 column printer. The default is 80 columns.
  
- L*length*** Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a **-L*55*** option. The default is 66 lines.
  
- X*prefix*** The -X option allows the exclusion of symbols from the listing, based on a prefix given as argument to -X. For example if it was desired to exclude all symbols starting with the character sequence xyz then the option **-X*xyz*** would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. **-X*X0*** would exclude any symbols starting with the letter X followed by a digit.
  
- F** -F will exclude from the listing any references from files with a full path name. A full path name means either: a file name starting with a slash (/) or backslash (\) or a file name starting with a CP/M user number/drive letter prefix, e.g. 0:A:. This is intended to force omission from the listing of any symbol references derived from standard header files, e.g. using -F would omit any references from the header file STDIO.H.

- Hstring**      The -H option takes a *string* as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.
- Sstoplist**      The -S option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Multiple stoplists may be supplied with multiple -S options.

Cref will accept wild card filenames and I/O redirection. Long command lines may be supplied by invoking CREF with no arguments and typing the command line in response to the *cref*> prompt. A backslash at the end of the line will be interpreted to mean that more command lines follow.

## APPENDIX 1

## Error Messages

Error Messages produced by the compiler are listed below. Each message is followed by the name of the program which produces it, and some further description of what causes the message or what to do about it.

'.' expected after '..'	P1	The ellipsis symbol must have three dots
actuals too long	CPP	Reduce length of macro arguments
argument list conflicts with prototype	P1	The argument list in a function definition must agree with a prototype if one exists
argument redeclared	P1	This argument has been declared twice
arithmetic overflow in constant expression	CGEN	Evaluation of this constant expression produced an arithmetic overflow. This may or may not represent a true error.
array index out of bounds	P1	An array index expression evaluates to a constant which is less than zero or greater than or equal to the dimension of the array
Assertion	CGEN	Internal error - contact HI-TECH
attempt to modify const object	P1	An attempt has been made to assign to or otherwise modify an object designated as 'const'
bad bitfield type	P1	Bitfields must be of type 'int'
Bad conval	CGEN	Internal error - contact HI-TECH
Bad dimensions	CGEN	An array has bad dimensions - probably zero
Bad element count expr	CGEN	Internal error - contact HI-TECH
bad formal	CPP	Check macro definition syntax

bad include syntax	CPP	Use only "" and <> for include files
Bad int. code	CGEN	The intermediate code file has been corrupted - can be caused by running out of disk space
Bad -M option	CGEN	A -M option passed to the code generator is unknown
Bad mod '+' for how = c	CGEN	Internal error - contact HI-TECH
bad object code format	LINK	This file is either corrupted or not a valid object file
Bad op d to swaplog	CGEN	Internal error - contact HI-TECH
Bad op n to revlog	CGEN	Internal error - contact HI-TECH
bad origin format in spec	LINK	An address in a -p option is invalid
bad '-p' format	LINK	The -p option provided is invalid
Bad pragma c	CGEN	The code generator has been passed a pragma it does not know about
Bad putwsize	CGEN	Internal error - contact HI-TECH
bad storage class	P1, CGEN	The specified storage class is illegal
Bad U usage	CGEN	Internal error - contact HI-TECH
Bit field too large (n bits)	CGEN	A bit field may not be larger than an int
Cannot get memory	LINK	The linker has run out of dynamic memory
Can't be both far and near	P1	The 'far' and 'near' keywords cannot appear in the same type specifier
can't be long	P1	Chars and shorts cannot be long
can't be register	P1	An extern or static variable may not be register

can't be short	P1	Float and char cannot be short
can't be unsigned	P1	Float cannot be unsigned
can't call an interrupt function	P1	A function qualified 'interrupt' can only be called by hardware, not by an ordinary function call
Can't create filename	CGEN	The file specified could not be created
Can't create xref file	P1	The cross reference file specified could not be created
Can't create	CPP	Output file could not be created
Can't create	LINK	The linker cannot create a file
Can't find include file	CPP	Check and correct the include file name - spaces are not allowed in file names
Can't find register for bits	CGEN	Internal error - contact HI-TECH
Can't generate code for this expression	CGEN	The code generator is unable to generate code for this expression - simplifying the expression (e.g. computing values into temporary variables) will usually correct it, otherwise contact HI-TECH
can't have array of functions	P1	You cannot have an array of functions - you can have an array of pointers to functions
Can't have 'port' variable	CGEN	You cannot declare a variable to be qualified 'port' - you can only use port to qualify pointers or typecast constant values
can't have storage class	P1	A storage class may not appear in a prototype argument
can't initialise auto aggregates	P1	You cannot initialise a structure or array inside a function unless it is static
can't initialize arg	P1	An argument cannot have an initializer
can't mix proto and non-proto args	P1	You cannot mix prototype and non-prototype arguments even in a function definition

Can't open filename	CGEN	The file specified could not be opened for reading
Can't open	LINK	The linker cannot open a file
Can't seek	LINK	The linker could not seek in file
can't take address of register variable	P1	You can't take the address of a variable in a register
can't take sizeof func	CGEN	You can't take the size of a function. You can take the size of a function call
can't take this address	P1	The expression does not have an address
'case' not in switch	P1	A 'case' label is permitted only inside a switch
char const too long	P1	A character constant may have only one character in it
close error (disk space?)	P1	Probably out of disk space
common symbol psect conflict	LINK	A common symbol is defined to be in more than one psect
constant conditional branch	CGEN	You have a program structure testing a constant expression, e.g. while(1). You should substitute for this the more efficient for(;;)
constant expression required	P1	A constant expression is required in e.g. an array dimension
constant operand to    or &&	CGEN	A logical operator has a constant operand which has been optimized out
declarator too complex	P1	This declaration is too complex for the compiler to handle
default case redefined	P1	Only one default case is permitted in a switch
'default' not in switch	P1	A 'default' label is permitted only inside a switch
digit out of range	P1	An octal constant may not contain 7 or 8, and a decimal constant may not contain A-F

dimension required	P1	A dimension is required for all except the most significant in an array declaration
Division by zero	CGEN	Attempt to divide by zero in this expression
Duplicate case label n	CGEN	There are two case labels in this switch that have the same value
Duplicate -d flag	LINK	Only one -d flag is allowed to the linker
duplicate label	P1	This label is defined twice
Duplicate -m flag	LINK	Only one -m flag is allowed to the linker
duplicate qualifier	P1	The same qualifier appears more than once in this type specifier
entry point multiply defined	LINK	A program can only have one entry point (start address)
EOF in #asm	P1	End of file was encountered after #asm and before a #endasm was seen
Error closing output file	CGEN, CPP	Probably means you have run out of disk space
excessive -I file ignored	CPP	Use fewer -I options
expand - bad how	CGEN	Internal error - contact HI-TECH
expand - bad which	CGEN	Internal error - contact HI-TECH
exponent expected	P1	An exponent is expected after the 'e' or 'E' in a floating point constant. The exponent must contain only +, - and digits 0-9
Expression error	CGEN	Internal error - contact HI-TECH
expression generates no code	CGEN	This expression has no side effects and thus generates no code. It has been optimized out
expression syntax	P1	The expression is badly formed

expression too complex	P1	The expression has too many nested parentheses or other nested constructs
Fixup overflow referencing	LINK	The linker has relocated a reference to a psect or symbol and the relocated address is too big to fit into the space, e.g. a relocated one byte address exceeds 256 or a relocated 16 bit address exceeds 65536
float param coerced to double	P1	This float parameter has been converted to double - a prototype will override this coercion
function() declared implicit int	P1	This function has been called without an explicit declaration. It is wise to explicitly declare all functions, preferably with a prototype. This will avoid many potential errors where your program comprises more than one source file
function does not take arguments	P1	The prototype for this function indicates it takes no arguments
function or function pointer required	P1	A function identifier or pointer to function is required for a function call.
functions can't return arrays	P1	A function cannot return an array - it can return a pointer
functions can't return functions	P1	A function cannot return a function - it can return a pointer to function
hex digit expected	P1	A hex digit is expected after '0x'
identifier is a structure tag	P1	A structure tag has been used in a context where another kind of tag is expected, e.g. saying struct fred where fred has previously been declared as union fred.
identifier is a union tag	P1	Similar to the above error
identifier is an enum tag	P1	Similar to the above error
identifier: large offset	CGEN	Z80 only: This identifier has a large offset from the stack frame and thus access to it is inefficient. In a function any arrays should be declared after any simple variables

identifier redeclared	P1	The identifier has been redeclared with different attributes
identifier redefined	P1	An identifier has been defined twice
If-less else	CPP	Check #if usage
If-less endif	CPP	Check #if usage
illegal '#' directive	P1	A # directive passed through to the first pass is unknown. If this occurs with a #include it may be caused by a previous include file not having a <CR><LF> or newline on the last line.
Illegal character in preprocessor if	CPP	Check for strange character
illegal character	P1	A character unknown to the compiler has been encountered. The value given is the octal value of the character
illegal conversion between pointer types	P1	The expression causes one pointer type to be converted to another incompatible type
illegal conversion of integer to pointer	P1	An integer is used where a pointer is expected
illegal conversion of pointer to integer	P1	A pointer is used where an integer is expected
illegal conversion	P1	The type conversion here is illegal
Illegal flag	LINK	This option is illegal
illegal function qualifier(s)	P1	A function cannot have 'const' qualification
illegal initialization	P1	The initialisation of this variable is illegal
Illegal number	CPP	Check number syntax
illegal type for array dimension	P1	An array dimension must be an integral quantity
illegal type for index expression	P1	An array index must be a simple integral expression

illegal type for switch expression	P1	The expression in a 'switch' must be integral
illegal use of void expression	P1	Void expressions may not be used in any way
implicit conversion of float to integer	P1	A floating point value has been converted to integer - truncation may occur
implicit return at end of non-void function	P1	A function with a non-void type has returned without a return statement
implicit signed to unsigned conversion	P1	Unwanted sign extension may occur here. Add an explicit typecast to force exactly the conversion you want
inappropriate break/continue	P1	
inappropriate 'else'	P1	An 'else' has appeared without a matching 'if'
inconsistent storage class	P1	Only one storage class may be specified in a declaration
inconsistent type	P1	Only one basic type may be specified in a declaration
initialisation illegal in arg list	P1	You cannot initialise a function parameter
initialisation syntax	P1	The syntax of this initialisation is illegal
initializer in 'extern' declaration	P1	A declaration with the 'extern' keyword has an initializer; this is not permitted as the extern declaration reserves no storage
integer constant expected	P1	An integer constant was expected here
integer expression required	P1	An integral expression is required here
integral type required	P1	An integral type is required here
large offset	CGEN Z80 only:	This identifier has a large offset from the stack frame and thus access to it is inefficient. In a function any arrays should be declared after any simple variables

Line too long	P1	The source line is too long, or does not have a <CR><LF> or newline at the end
local psect conflicts with global psect of same name	LINK	A local psect cannot have the same ..name as a global psect
logical type required	P1	A logical type (i.e. an integral type) is required as the subject of a conditional expression
lvalue required	P1	An lvalue, i.e. something which can be assigned to, is required after an '&' or on the left hand of an assignment
macro recursion	CPP	A preprocessor macro has attempted to expand itself. This would create infinite recursion
member is not a member of the struct/union	P1	This member is not in the structure or union with which it is used
members cannot be functions	P1	A member cannot be a function - it can be a pointer to function
Missing arg to -u	LINK	-u requires an argument
Missing arg to -w	LINK	-w requires an argument
missing )	CPP	Put correct ) in expression
Missing number after % in -p option	LINK	After % in a -p option there must be a number
Missing number after pragma 'pack'	P1	The correct syntax is #pragma pack(n) where n is 1, 2 or 4.
module has code below file base	LINK	A -C option was specified but the program has code below the address specified as the base of the binary file
multiply defined symbol	LINK	A symbol is defined more than once
name is a union, struct or enum	P1	A union, struct or enum tag has been re-used in a different context

No case labels	CGEN	This switch has no case labels
no identifier in declaration	P1	This declaration should have an identifier in it
No room	CGEN	The code generator has run out of dynamic memory. You will need to reduce the number of symbols and/or the complexity of expressions
No source file	CPP	Source file could not be found - check spelling, directory paths etc.
no space	CPP	Reduce number/size of macro definitions
no start record: entry point defaults to zero	LINK	No start address has been specified for the program; the linker has set the start address to 0
Non-constant case label	CGEN	This case label does not evaluate to an integral constant
non-void function returns no value	P1	A function which should return a value has a 'return' statement with no value
not a variable identifier	P1	The identifier is not a variable - it may be e.g. a label or structure tag
not an argument	P1	This identifier is not in the argument list for this function
only functions may be qualified interrupt	P1	The type qualifier 'interrupt' may be applied only to functions, not variables.
only functions may be void	P1	Only functions, not variables, may be declared void
only lvalues may be assigned to or modified	P1	You have attempted to modify an expression which does not identify a storage location
only register storage class allowed	P1	A parameter may only be auto or register
operands of operator not same pointer type	P1	The operands to the named operator in the expression are both pointers but are not the same pointer type

operands of operator not same type	P1	The operands to the named operator in the expression are incompatible types
pointer required	P1	A pointer is required after a '*' (indirection) operator
popreg - bad reg	CGEN	Internal error - contact HI-TECH
portion of expression has no effect	CGEN	A portion of this expression has no effect on its value and no side effects
probable missing '}' in previous block	P1	A declaration has been encountered where an expression was expected. The likely cause of this is that you have omitted a closing '}' in the function above this point.
psect cannot be in classes a and b	LINK	A psect can only be in one class
psect exceeds max size	LINK	This psect is larger than a specified maximum size
psect is absolute	LINK	This psect is absolute and cannot have a link address specified in a -p option
Psect not loaded on 0xhexnum boundary	LINK	This psect must be loaded on a specific boundary
Psect not relocated on 0xhexnum boundary	LINK	This psect must be linked on a specific boundary
psect origin multiply defined	LINK	This psect has its link address defined more than once
pushreg - bad reg	CGEN	Internal error - contact HI-TECH
redundant & applied to array	P1	An array type has an '&' operator applied to it. It has been ignored since use of an array implicitly gives its address
regused - bad arg to G	CGEN	Internal error - contact HI-TECH
signatures do not match	LINK	An extern function has been declared with an incorrect prototype. For example if an argument is declared as a long in an extern declaration, but is really an int, a signature mismatch will occur.

signed bitfields not supported	P1	Only unsigned bitfields are supported
simple type required	P1	An array or structure type cannot be used here
Sizeof yields 0	CGEN	The size of an object has evaluated to zero in a context where this is illegal, e.g. incrementing a pointer to a zero length object.
storage class illegal	P1	A storage class may not be specified here
struct/union member expected	P1	A structure or union member is required after a '.' or '->'
struct/union redefined	P1	This structure or union has been defined twice
struct/union required	P1	A structure or union identifier is required before a '.'
Switch on long!	CGEN	Switching on a long expression is not supported
symbol cannot be global	LINK	Stack, filename or line number symbols cannot be global
Syntax error in checksum list	LINK	The checksum list provided is invalid
token too long	CPP	Shorten token (e.g. identifier)
too few arguments	P1	The prototype for this function lists more arguments than have been supplied
too many arguments	P1	More arguments have been supplied than listed in the prototype for this function
Too many cases in switch	CGEN, P1	There are too many cases in this switch
too many -D options	CPP	Use fewer -D options
too many defines	CPP	Reduce number of macro definitions
Too many errors	CGEN	CGEN has given up because there were too many errors.

too many formals	CPP	Reduce number of parameters to this macro definition
Too many initializers	CGEN	There are too many initializers for this object
Too many psects	LINK	There are too many psects for the symbol table
Too many symbols	LINK	There are too many symbols for the linker symbol table
too many -U options	CPP	Use fewer -U options
too much defining	CPP	Reduce number/size of macros
too much indirection	P1	Too many '*'s in this declaration
too much pushback	CPP	Simplify macro usage
type conflict	P1	There is a conflict of types in this expression, e.g. attempting to assign a structure to a simple type
type specifier reqd. for proto arg	P1	A prototype argument must have a basic type
undefined control	CPP	Check use of #
undefined enum tag	P1	This enumerated type tag has not been defined
undefined identifier	P1	This identifier has not been defined before use
undefined struct/union	P1	The structure or union used has not been defined
undefined symbol	LINK	A list of undefined symbols follows. If some of the symbols should be in a library which was linked, it may be caused by a library ordering problem. In this case rebuild the library with the correct ordering or specify the library more than once in the link command
unexpected EOF	P1	End of file was encountered in the middle of a C construct. This is commonly caused by omission of a closing '}' earlier in the program.

Unknown predicate	CGEN	Internal error - contact HI-TECH
unknown psect	LINK	The psect specified in a -p option is not present in the program. Check the spelling and check the case - upper case does not match lower case
unreachable code	P1	This section of code can never be executed as there is no possible path to reach it
Unreasonable include nesting	CPP	Reduce number of include files
Unreasonable matching depth	CGEN	Internal error - contact HI-TECH
unterminated macro call	CPP	Probably missing )
void function cannot return value	P1	A function declared void cannot return a value
Write error (out of disk space?)	LINK	Probably means the disk is full

## APPENDIX 2

## Standard Library Functions

The functions accessible to user programs in the standard library `libc.lib` are listed below, by category with a short comment, then alphabetically with a longer description. In the detailed description of each function, the SYNOPSIS section describes the function in roughly the manner in which the function would be declared in the source file defining it. Where an include file is shown, this implies that that include file must be included in any source file using that function.

Where an include file is not provided, it will normally be necessary for an extern declaration of the function to be included in any source module using it, to ensure that the type of the function is correct. For example, if the function *lseek()* was to be used, a declaration of the form

```
extern long lseek();
```

should be in either the source file itself or an include file included in the source file. This ensures that the compiler knows that *lseek()* returns a long value and not the default int.

Where reference is made to `STDIO`, this means the group of functions under the heading `STANDARD I/O` below. These all have one thing in common; they operate on pointers to a defined data type called `FILE`. Such a pointer is often referred to as a *stream* pointer. The concept of a stream is central to these routines. Essentially a stream is a source or sink of data bytes. To the operating system and library routines this stream is featureless, i.e. no record structure is implied or assumed. Some routines do however recognize end of line characters.

## STANDARD I/O

<code>fopen(name, mode)</code>	Open file for I/O
<code>freopen(name, mode, stream)</code>	Re-open existing stream
<code>fdopen(fd, mode)</code>	Associate a stream with a file descriptor
<code>fclose(stream)</code>	Close open file
<code>fflush(stream)</code>	Flush buffered data
<code>getc(stream)</code>	Read byte from stream
<code>fgetc(stream)</code>	Same as <code>getc</code>
<code>ungetc(c, stream)</code>	Push char back onto stream
<code>putc(c, stream)</code>	Write byte to stream
<code>fputc(c, stream)</code>	Same as <code>putc()</code>
<code>getchar()</code>	Read byte from standard input
<code>putchar(c)</code>	Write byte to standard output
<code>getw(stream)</code>	Read word from stream

putw(w, stream)	Write word to stream
gets(s)	Read line from standard input
fgets(s, n, stream)	Read string from stream
puts(s)	Write string to standard output
fputs(s, stream)	Write string to stream
fread(buf, size, cnt, stream)	Binary read from stream
fwrite(buf, size, cnt, stream)	Binary write to stream
fseek(stream, offs, wh)	Random access positioning
ftell(stream)	Current file read/write position
rewind(stream)	Reposition file pointer to start
setvbuf(stream, buf, mode, size)	Enable/disable buffering of stream
fprintf(stream, fmt, args)	Formatted output on stream
printf(fmt, args)	Formatted standard output
sprintf(buf, fmt, args)	Formatted output to a string
vfprintf(stream, fmt, va_ptr)	Formatted output on stream
vprintf(fmt, va_ptr)	Formatted standard output
vsprintf(buf, fmt, va_ptr)	Formatted output to a string
fscanf(stream, fmt, args)	Formatted input from stream
scanf(fmt, args)	Formatted standard input
sscanf(buf, fmt, va_ptr)	Formatted input from a string
vfscanf(stream, fmt, va_ptr)	Formatted input from stream
vscanf(fmt, args)	Formatted standard input
vsscanf(buf, fmt, va_ptr)	Formatted input from a string
feof(stream)	True if stream at EOF
ferror(stream)	True if error on stream
clrerr(stream)	Reset error status on stream
fileno(stream)	Return fd from stream
remove(name)	Remove (delete) file

## STRING HANDLING

atoi(s)	Convert ASCII decimal to integer
atol(s)	Convert ASCII decimal to long integer
atof(s)	Convert ASCII decimal to float
atoi(s)	Convert ASCII hexadecimal to integer
memchr(s, c, n)	Find char in memory block
memcmp(s1, s2, n)	Compare n bytes of memory
memcpy(s1, s2, n)	Copy n bytes from s2 to s1
memmove(s1, s2, n)	Copy n bytes from s2 to s1
memset(s, c, n)	Set n bytes at s to c
strcat(s1, s2)	Append string 2 to string 1
strncat(s1, s2, n)	Append at most n chars to string 1
strcmp(s1, s2)	Compare strings
strncmp(s1, s2, n)	Compare n bytes of strings
strcpy(s1, s2)	Copy s2 to s1
strncpy(s1, s2, n)	Copy at most n bytes of s2
strerror(errno)	Map errno to an error message string
strlen(s)	Length of string

strchr(s, c)	Find char in string
strrchr(s, c)	Find rightmost char in string
strspn(s1, s2)	Length of s1 composed of chars from s2
strcspn(s1, s2)	Length of s2 composed of chars not from s2
strstr(s1, s2)	Locate the first occurrence of s2 in s1
open(name, mode)	Open a file
close(fd)	Close a file
creat(name)	Create a file
dup(fd)	Duplicate file descriptor
lseek(fd, offs, wh)	Random access positioning
read(fd, buf, cnt)	Read from file
rename(name1, name2)	Rename file
unlink(name)	Remove file from directory
write(fd, buf, cnt)	Write to file
isatty(fd)	True if fd refers to tty-like device
stat(name, buf)	Get information about a file
chmod(name, mode)	Set file attributes

## CHARACTER TESTING

isalpha(c)	True if c is a letter
isupper(c)	Upper case letter
islower(c)	Lower case letter
isdigit(c)	Digit
isalnum(c)	Alphanumeric character
isspace(c)	Space, tab, newline, return or formfeed
ispunct(c)	Punctuation character
isprint(c)	Printable character
isgraph(c)	Printable non-space character
isctrl(c)	Control character
isascii(c)	Ascii character (0-127)

## FLOATING POINT

cos(f)	Cosine function
sin(f)	Sine function
tan(f)	Tangent function
acos(f)	Arc cosine function
asin(f)	Arc sine function
atan(f)	Arc tangent function
exp(f)	Exponential of f
log(f)	Natural log of f
log10(f)	Base 10 log of f
pow(x,y)	X to the y'th power
sqrt(f)	Square root
fabs(f)	Floating absolute value
ceil(f)	Smallest integral value $\geq$ f
floor(f)	Largest integral value $\leq$ f

sinh(f)	Hyperbolic sine
cosh(f)	Hyperbolic cosine
tanh(f)	Hyperbolic tangent
frexp(y, p)	Split into mantissa and exponent
ldexp(y, I)	Load new exponent

### CONSOLE I/O

getch()	Get single character
getche()	Get single character with echo
putch(c)	Put single character
ungetch(c)	Push character back
kbhit()	Test for key pressed
cgets(s)	Get line from console
cputs(s)	Put string to console

### DATE AND TIME FUNCTIONS

time(p)	Get current date/time
gmtime(p)	Get broken down Universal time
localtime(p)	Get broken down local time
asctime(t)	Convert broken down time to ascii
ctime(p)	Convert time to ascii

### MISCELLANEOUS

execl(name, args)	Execute another program
execv(name, argp)	Execute another program
spawnl(name, arg, ...)	Execute a subprogram
spawnv(name, argp)	Execute a subprogram
system(s)	Execute system command
atexit(func)	Install func to be executed on termination
exit(status)	Terminate execution
_exit(status)	Terminate execution immediately
getuid()	Get user id (CP/M)
setuid(uid)	Set user id (CP/M)
chdir(s)	Change directory (MS-DOS)
mkdir(s)	Create directory (MS-DOS)
rmdir(s)	Remove directory (MS-DOS)
getcwd(drive)	Get current working directory (MS-DOS)
signal(sig, func)	Set trap for interrupt condition
brk(addr)	Set memory allocation
sbrk(incr)	Adjust memory allocation
malloc(cnt)	Dynamic memory allocation
free(ptr)	Dynamic memory release
realloc(ptr, cnt)	Dynamic memory reallocation
calloc(cnt, size)	Dynamic memory allocation zeroed

perror(s)	Print error message
qsort(base, nel, width, func)	Quick sort
srand(seed)	Initialize random number generator
rand()	Get next random number
setjmp(buf)	Setup for non-local goto
longjmp(buf, val)	Non-local goto
_getargs(buf, name)	Wild card expansion and i/o redirection
inp(port)	Read port
outp(port, data)	Write data to port
bdos(func, val)	Perform bdos call (CP/M)
msdos(func, val, val, ...)	Perform msdos call
msdoscx(func, val, val, ...)	Alternate msdos call
intdos(ip, op)	Execute DOS interrupt
intdosx(ip, op, sp)	Execute DOS interrupt
segread(sp)	Get segment register values
int86(int, ip, op)	Execute software interrupt
int86x(int, ip, op, sp)	Execute software interrupt
bios(n, c)	Call bios entry (CP/M)
ei()	Enable interrupts
di()	Disable interrupts
set_vector(vec, func)	Set an interrupt vector
assert(e)	Run time assertion
getenv(s)	Get environment string (MS-DOS)

## ACOS, ASIN, ATAN, ATAN2

### SYNOPSIS

```
#include <math.h>

double   acos(double f)
double   asin(double f)
double   atan(double f)
double   atan2(double x, double y)
```

**DESCRIPTION:** These functions are the converse of the trigonometric functions cos, sin and tan. Acos and asin are undefined for arguments whose absolute value is greater than 1.0. The returned value is in radians, and always in the range  $-\pi/2$  to  $+\pi/2$ , except for *cos()*, which returns a value in the range 0 to  $\pi$ . *Atan2()* returns the inverse tan of  $x/y$  but uses the signs of its arguments to return a value in the range  $-\pi$  to  $+\pi$ .

### SEE ALSO

sin, cos, tan

## ATEXIT

## SYNOPSIS

```
#include <stdlib.h>
```

```
int      atexit(void (*func)(void));
```

DESCRIPTION The *atexit()* function registers the function pointed to by *func*, to be called without arguments at normal program termination. *Atexit()* returns zero if the registration succeeds, nonzero if it fails. On program termination, all functions registered by *atexit()* are called, in the reverse order of their registration.

## SEE ALSO

exit

## ASCTIME SYNOPSIS

```
#include <time.h>
```

```
char      * asctime(time_t t)
```

DESCRIPTION *Asctime()* takes the broken down time pointed to by its argument, and returns a 26 character string describing the current date and time in the format

```
Sun Sep 16 01:03:52 1973\n\0
```

Note the newline at the end of the string. The width of each field in the string is fixed.

## SEE ALSO

ctime, time, gmtime, localtime

## ASSERT

## SYNOPSIS

```
#include    <assert.h>

void        assert(int e)
```

DESCRIPTION: This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An *assert()* may be used to ensure at run time that that assumption holds. For example, the following statement asserts that the pointer *tp* is nonnull:

```
assert(tp);
```

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of *assert* is impossible in limited space, but it is closely linked to methods of proving program correctness.

## ATOF, ATOI, ATOL

## SYNOPSIS

```
#include    <math.h>

double     atof(char * s)
int        atoi(char * s)

#include    <stdlib.h>

long       atol(char * s)
```

DESCRIPTION: These routines convert a decimal number in the argument string *s* into a double float, integer or long integer respectively. Leading blanks are skipped over. In the case of *atof()*, the number may be in scientific notation.

## BDOS (CP/M only)

## SYNOPSIS

```
#include    <cpm.h>

char       bdos(int func, int arg)

short      bdoshl(int func, int arg)(CP/M-80 only)
```

DESCRIPTION: *Bdos()* calls the CP/M BDOS with *func* in register C (CL for CP/M-86) and *arg* in register DE (DX). The return value is the byte returned by the BDOS in register A (AX). *Bdoshl()* is the same, except that the return value is the value returned by the BDOS in HL. Constant values for the various BDOS function values are defined in *cpm.h*.

These functions should be avoided except in programs which are not intended to be used on an operating system other than CP/M. The standard I/O routines are to be preferred, since they are portable.

## SEE ALSO

*bios*, *msdos*

## BIOS (CP/M only)

## SYNOPSIS

```
#includ    <cpm.h>

char       bios(int n, int a1, int a2)
```

DESCRIPTION: This function will call the *n*'th bios entry point (cold boot = 0, warm boot = 1, etc.) with register BC (CX) set to the argument *a1* and DE (DX) set to the argument *a2*. The return value is the contents of register A (AX) after the bios call. On CP/M-86, *bdos* function 50 is used to perform the bios call. This function should not be used unless unavoidable, since it is highly non-portable. There is even no guarantee of portability of bios calls between differing CP/M systems.

## SEE ALSO

*bdos*

## CALLOC

## SYNOPSIS

```
#include <stdlib.h>
```

```
char * calloc(size_t cnt, size_t size)
```

DESCRIPTION: *Calloc()* attempts to obtain a contiguous block of dynamic memory which will hold *cnt* objects, each of length *size*. The block is filled with zeroes. A pointer to the block is returned, or 0 if the memory could not be allocated.

## SEE ALSO

brk, sbrk, malloc, free

## CGETS, CPUTS SYNOPSIS

```
#include <conio.h>
```

```
char * cgets(char * s)
```

```
void cputs(char * s)
```

DESCRIPTION: *Cputs()* will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to *getche()*. *Cputs()* writes its argument string to the console, outputting carriage returns before each newline in the string. It calls *putch()* repeatedly.

## SEE ALSO

getch, getche, putch

## CHDIR

## SYNOPSIS

```
#include <sys.h>

int chdir(char * s)
```

DESCRIPTION: This function is available only under MS-DOS. It changes the current working directory to the path name supplied as argument. This path name can be absolute, as in A:\FRED, or relative, as in ..\SOURCES. A return value of -1 indicates that the requested change could not be performed.

## SEE ALSO

mkdir, rmdir, getcwd

## CHMOD

## SYNOPSIS

```
#include <stat.h>

int chmod(char * name, int mode);
char * name;
```

DESCRIPTION: This function changes the file attributes (or modes) of the named file. The argument name may be any valid file name. The mode argument may include all bits defined in *stat.h* except those relating to the type of the file, e.g. S\_IFDIR. Note however that not all bits may be changed under all operating systems, e.g. neither DOS nor CP/M permit a file to be made unreadable, thus even if mode does not include S\_IREAD the file will still be readable (and *stat()* will still return S\_IREAD in flags).

## SEE ALSO

stat, creat

## CLOSE

## SYNOPSIS

```
#include <unistd.h>
```

```
int close(int fd)
```

DESCRIPTION: This routine closes the file associated with the file descriptor *fd*, which will have been previously obtained from a call to *open()*. *Close()* returns 0 for a successful close, or -1 otherwise.

## SEE ALSO

open, read, write, seek

## CLRERR, CLREOF

## SYNOPSIS

```
#include <stdio.h>
void clrerr(FILE * stream)
void clreof(FILE * stream)
```

DESCRIPTION: These are macros, defined in *stdio.h*, which reset the error and end of file flags respectively for the specified stream. They should be used with care; the major valid use is for clearing an EOF status on input from a terminal-like device, where it may be valid to continue to read after having seen an end-of-file indication.

## SEE ALSO

fopen, fclose

**COS SYNOPSIS**

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

```
double cos(double f)
```

**DESCRIPTION:** This function yields the cosine of its argument.

**SEE ALSO**

sin, tan, asin, acos, atan

**COSH, SINH, TANH****SYNOPSIS**

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

```
double cosh(double f)
```

```
double sinh(double f)
```

```
double tanh(double f)
```

**DESCRIPTION:** These functions implement the hyperbolic trig functions.

## CREAT

## SYNOPSIS

```
#include <stat.h>
```

```
int creat(char * name, int mode)
```

DESCRIPTION: This routine attempts to create the file named by name. If the file exists and is writeable, it will be removed and re-created. The return value is -1 if the create failed, or a small non-negative number if it succeeded. This number is a valuable token which must be used to write to or close the file subsequently. Mode is used to initialize the attributes of the created file. The allowable bits are the same as for *chmod()*, but for Unix compatibility it is recommended that a mode of 0666 or 0600 be used. Under CP/M the mode is ignored - the only way to set a files attributes is via the *chmod()* function.

## SEE ALSO

open, close, read, write, seek, stat, chmod

## CTIME

## SYNOPSIS

```
#include <time.h>
```

```
char * ctime(time_t t)
```

DESCRIPTION: *Ctime()* converts the time in seconds pointed to by its argument to a string of the same form as described for *asctime*. Thus the following program prints the current time and date:

```
#include <time.h>
```

```
main()
{
    time_t t;

    time(&t);
    printf("%s", ctime(&t));
}
```

## SEE ALSO

gmtime, localtime, asctime, time

## DIV, LDIV

## SYNOPSIS

```
#include <stdlib.h>
```

```
div_t      div(int numer, int denom)
```

```
ldiv_t     div(long numer, long denom)
```

DESCRIPTION: The *div()* function computes the quotient and remainder of the division of numer by denom. The *div()* function returns a structure of type `div_t`, containing both the quotient and remainder. *ldiv()* is similar to *div()* except it takes arguments of type long and returns a structure of type `ldiv_t`. The types `div_t` and `ldiv_t` are defined in `<stdlib.h>` as follows:

```
typedef struct {
    intquot, rem;
} div_t;
```

```
typedef struct {
    longquot, rem;
} ldiv_t;
```

## DI, EI

## SYNOPSIS

```
void      ei(void);  
void      di(void);
```

DESCRIPTION: *Ei()* and *di()* enable and disable interrupts respectively.

## DUP

## SYNOPSIS

```
#include  <unixio.h>  
  
int      dup(int fd)
```

DESCRIPTION: Given a file descriptor, such as returned by *open()*, this routine will return another file descriptor which will refer to the same open file. -1 is returned if the fd argument is a bad descriptor or does not refer to an open file.

## SEE ALSO

open, close, creat, read, write

## EXECL, EXECV

## SYNOPSIS

```
#include <sys.h>
```

```
int      execl(char * name, pname, ...)
```

```
int      execv(char * name, ppname)
```

DESCRIPTION: *Execl()* and *execv()* load and execute the program specified by the string name. *Execl()* takes the arguments for the program from the zero-terminated list of string arguments. *Execv()* is passed a pointer to an array of strings. The array must be zero-terminated. If the named program is found and can be read, the call does not return. Thus any return from these routines may be treated as an error.

## SEE ALSO

spawnl, spawnv, system

## EXIT

## SYNOPSIS

```
#include <stdlib.h>
```

```
void      exit(int status)
```

DESCRIPTION: This call will close all open files and exit from the program. On CP/M, this means a return to CCP level. Status will be stored in a known place for examination by other programs. This is only useful if the program executing was actually invoked by another program which is trapping warm boots. The status value will be stored on CP/M at 80H. This call will never return.

## SEE ALSO

atexit

\_EXIT

## SYNOPSIS

```
#include    <stdlib.h>
void       _exit(int status)
```

DESCRIPTION: This function will cause an immediate exit from the program, without the normal flushing of stdio buffers that is performed by *exit()*.

## SEE ALSO

exit

## EXP, LOG, LOG10, POW

## SYNOPSIS

```
#include    <math.h>

double     exp(double f)
double     log(double f)
double     log10(double f)
double     pow(double x, y)
```

DESCRIPTION: *Exp()* returns the exponential function of its argument, *log()* the natural logarithm of f, and *log10()* the logarithm to base 10. *Pow()* returns the value of x raised to the y'th power.

## FABS, CEIL, FLOOR

## SYNOPSIS

```
#include    <math.h>

double     fabs(double f)
double     ceil(double f)
double     floor(double f)
```

DESCRIPTION: These routines return respectively the absolute value of f, the smallest integral value not less than f, and the largest integral value not greater than f.

## FCLOSE

## SYNOPSIS

```
#include    <stdio.h>

int         fclose(FILE * stream)
```

DESCRIPTION: This routine closes the specified i/o stream. Stream should be a token returned by a previous call to *fopen()*. NULL is returned on a successful close, EOF otherwise.

## SEE ALSO

fopen, fread, fwrite

## FEOF, FERROR

## SYNOPSIS

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

```
feof(FILE * stream)
```

```
ferror(FILE * stream)
```

DESCRIPTION: These macros test the status of the EOF and ERROR bits respectively for the specified stream. Each will be true if the corresponding flag is set. The macros are defined in `stdio.h`. Stream must be a token returned by a previous *fopen()* call.

## SEE ALSO

fopen, fclose

## FFLUSH

## SYNOPSIS

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

```
int fflush(FILE * stream)
```

DESCRIPTION: *fflush()* will output to the disk file or other device currently open on the specified stream the contents of the associated buffer. This is typically used for flushing buffered standard output in interactive applications.

## SEE ALSO

fopen, fclose

## FGETC

## SYNOPSIS

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

```
int fgetc(FILE * stream)
```

DESCRIPTION: *Fgetc()* returns the next character from the input stream. If end-of-file is encountered EOF will be returned instead. It is for this reason that the function is declared as int. The integer EOF is not a valid byte, thus end-of-file is distinguishable from reading a byte of all 1 bits from the file. *Fgetc()* is the non-macro version of *getc()*.

## SEE ALSO

fopen, fclose, fputc, getc, putc

## FGETS

## SYNOPSIS

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

```
char * fgets(char * s, size_t n, char * stream)
```

DESCRIPTION: *Fgets()* places in the buffer *s* up to *n-1* characters from the input stream. If a newline is seen in the input before the correct number of characters is read, then *fgets()* will return immediately. The newline will be left in the buffer. The buffer will be null terminated in any case. A successful *fgets()* will return its first argument; NULL is returned on end-of-file or error.

## FILENO

## SYNOPSIS

fileno(FILE \* stream)

DESCRIPTION: *Fileno()* is a macro from `stdio.h` which yields the file descriptor associated with `stream`. It is mainly used when it is desired to perform some low-level operation on a file opened as a `stdio` stream.

## SEE ALSO

`fopen`, `fclose`, `open`, `close`

## FOPEN

## SYNOPSIS

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

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

DESCRIPTION: *Fopen()* attempts to open file for reading or writing (or both) according to the mode string supplied. The mode string is interpreted as follows:

- r The file is opened for reading if it exists. If the file does not exist the call fails.
- r+ If the file exists it is opened for reading and writing. If the file does not already exist the call fails.
- W The file is created if it does not exist, or truncated if it does. It is then opened for writing.
- w+ The file is created if it does not already exist, or truncated if it does. The file is opened for reading and writing.
- a The file is created if it does not already exist, and opened for writing. All writes will be dynamically forced to the end of file, thus this mode is known as *append* mode.
- a+ The file is created if it does not already exist, and opened for reading and writing. All writes to the file will be dynamically forced to the end of the file, i.e. while any portion of the file may be read, all writes will take place at the end of the file and will not overwrite any existing data. Calling *fseek()* in an attempt to write at any other place in the file will not be effective.

The "b" modifier may be appended to any of the above modes, e.g. "r+b" or "rb+" are equivalent. Adding the "b" modifier will cause the file to be opened in binary rather than ASCII mode. Opening in ASCII mode ensures that text files are read in a manner compatible with the Unix-derived conventions for C programs, i.e. that text files contain lines delimited by newline characters. The special treatment of read or written characters varies with the operating system, but includes some or all of the following:

NEWLINE (LINE FEED)	Converted to carriage return, line feed on output.
RETURN	Ignored on input, inserted before NEWLINE on output.
CTRL-Z	Signals EOF on input, appended on fclose on output if necessary on CP/M.

Opening a file in binary mode will allow each character to be read just as written, but because the exact size of a file is not known to CP/M, the file may contain more bytes than were written to it. See *open()* for a description of what constitutes a file name.

When using one of the read/write modes (with a '+' character in the string), although they permits reading and writing on the same stream, it is not possible to arbitrarily mix input and output calls to the same stream. At any given time a stream opened with a "+" mode will be in either an input or output state. The state may only be changed when the associated buffer is empty, which is only guaranteed immediately after a call to *fflush()* or one of the file positioning functions *fseek()* or *rewind()*. The buffer will also be empty after encountering EOF while reading a binary stream, but it is recommended that an explicit call to *fflush()* be used to ensure this situation. Thus after reading from a stream you should call *fflush()* or *fseek()* before attempting to write on that stream, and vice versa.

SEE ALSO

fclose, fgetc, fputc, freopen

## FPRINTF

## SYNOPSIS

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

```
fprintf(FILE * stream, char * fmt, ...);  
vfprintf(FILE * stream, va_list va_arg);
```

DESCRIPTION: *Fprintf()* performs formatted printing on the specified stream. Refer to *printf()* for the details of the available formats. *Vfprintf()* is similar to *fprintf()* but takes a variable argument list pointer rather than a list of arguments. See the description of *vstart()* for more information on variable argument lists.

## SEE ALSO

printf, fscanf, sscanf

## FPUTC

## SYNOPSIS

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

```
int fputc(int c, FILE * stream)
```

DESCRIPTION: The character *c* is written to the supplied stream. This is the non-macro version of *putc()*. The character is returned if it was successfully written, EOF is returned otherwise. Note that "written to the stream" may mean only placing the character in the buffer associated with the stream.

## SEE ALSO

putc, fgetc, fopen, fflush

## FPUTS

## SYNOPSIS

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

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

DESCRIPTION: The null-terminated string *s* is written to the stream. No newline is appended (cf. *puts()*). The error return is EOF.

## SEE ALSO

puts, fgets, fopen, fclose

## FREAD

## SYNOPSIS

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

```
int fread(void * buf, size_t size, size_t cnt, FILE * stream)
```

DESCRIPTION: Up to *cnt* objects, each of length *size*, are read into memory at *buf* from the stream. The return value is the number of objects read. If none is read, 0 will be returned. Note that a return value less than *cnt*, but greater than 0, may not represent an error (cf. *fwrite()*). No word alignment in the stream is assumed or necessary. The read is done via successive *getc()*'s.

## SEE ALSO

fwrite, fopen, fclose, getc

## FREE

## SYNOPSIS

```
#include <stdlib.h>
```

```
void free(void * ptr)
```

DESCRIPTION: *Free()* deallocates the block of memory at *ptr*, which must have been obtained from a call to *malloc()* or *calloc()*.

## SEE ALSO

*malloc*, *calloc*

## FREOPEN

## SYNOPSIS

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

```
FILE * freopen(char * name, char * mode, FILE * stream)
```

DESCRIPTION: *Freopen()* closes the given stream (if open) then reopens the stream attached to the file described by *name*. The mode of opening is given by *mode*. It either returns the stream argument, if successful, or NULL if not. See *fopen()* for more information.

## SEE ALSO

*fopen*, *fclose*

## FREXP, LDEXP

## SYNOPSIS

```
#include    <math.h>

double     frexp(double f, int * p)
double     ldexp(double f, int i)
```

DESCRIPTION: *Frexp()* breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the int object pointed to by p. Its return value x is in the interval [0.5, 1.0) or zero, and f equals x times 2 raised to the power stored in \*p. If f is zero, both parts of the result are zero. *Ldexp()* performs the reverse operation; the integer i is added to the exponent of the floating point f and the resultant value returned.

## FSCANF

## SYNOPSIS

```
#include    <stdio.h>

int        fscanf(FILE * stream, char * fmt, ...)
```

DESCRIPTION: This routine performs formatted input from the specified stream. See *scanf()* for a full description of the behaviour of the routine. *Vfscanf()* is similar to *fscanf()* but takes a variable argument list pointer rather than a list of arguments. See the description of *vastart()* for more information on variable argument lists.

## SEE ALSO

scanf, sscanf, fopen, fclose

## FSEEK

## SYNOPSIS

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

```
int fseek(FILE * stream, long offs, int wh)
```

DESCRIPTION: *fseek()* positions the "file pointer" (i.e. a pointer to the next character to be read or written) of the specified stream as follows:

	resultant location
0	offs
1	offs+previous location
2	offs+length of file

It should be noted that *offs* is a signed value. Thus the 3 allowed modes give positioning relative to the beginning of the file, the current file pointer and the end of the file respectively. EOF is returned if the positioning request could not be satisfied. Note however that positioning beyond the end of the file is legal, but will result in an EOF indication if an attempt is made to read data there. It is quite in order to write data beyond the previous end of file. *fseek()* correctly accounts for any buffered data.

## SEE ALSO

*lseek*, *fopen*, *fclose*

## FTELL

## SYNOPSIS

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

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

DESCRIPTION: This function returns the current position of the conceptual read/write pointer associated with stream. This is the position relative to the beginning of the file of the next byte to be read from or written to the file.

## SEE ALSO

fseek

## FWRITE

## SYNOPSIS

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

```
int fwrite(void * buf, size_t size, size_t cnt, FILE * stream)
```

DESCRIPTION: Cnt objects of length size bytes will be written from memory at buf, to the specified stream. The number of whole objects written will be returned, or 0 if none could be written. Any return value not equal to cnt should be treated as an error (cf. *fread()* ).

## SEE ALSO

fread, fopen, fclose

\_GETARGS

## SYNOPSIS

```
#include    <sys.h>

char        ** _getargs(char * buf, char * name)
extern      int _argc_;
```

DESCRIPTION: This routine performs I/O redirection (CP/M only) and wild card expansion. Under MS-DOS I/O redirection is performed by the operating system. It is called from startup code to operate on the command line if the -R option is used to the C command, but may also be called by user-written code. If the buf argument is null, it will read lines of text from standard input. If the standard input is a terminal (usually the console) the name argument will be written to the standard error stream as a prompt. If the buf argument is not null, it will be used as the source of the string to be processed. The returned value is a pointer to an array of strings, exactly as would be pointed to by the argv argument to the main() function. The number of strings in the array may be obtained from the global \_argc\_. For example, a typical use of this function would be:

```
#include    <sys.h>

main(argc, argv)
char ** argv;
{
    extern char ** _getargs();
    extern int _argc_;

    if(argc == 1) {                /* no arguments */
        argv = _getargs(0, "myname");
        argc = _argc_;
    }
}
```

There will be one string in the array for each word in the buffer processed. Quotes, either single (') or double (") may be used to include white space in "words". If any wild card characters (? or \*) appear in a non-quoted word, it will be expanded into a string of words, one for each file matching the word. The usual CP/M conventions are followed for this expansion. On CP/M any occurrence of the redirection characters > and < outside quotes will be handled in the following manner:

```
> name      will cause standard output to be redirected to the file name.
< name      will cause standard input to be redirected from the file name.

>> name     will cause standard output to append to file name.
```

White space is optional between the > or < character and the file name, however it is an error for a redirection character not to be followed by a file name. It is also an error if a file cannot be opened for input or created for output. An append redirection (>>) will create the file if it does not exist. If the source of text to be processed is standard input, several lines may be supplied by ending each line (except the last) with a backslash (\). This serves as a continuation character. Note that the newline following the backslash is ignored, and not treated as white space.

## GETC

### SYNOPSIS

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

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

DESCRIPTION: One character is read from the specified stream and returned. EOF will be returned on end-of-file or error. This is the macro version of *fgetc()*, and is defined in *stdio.h*.

## GETCH, GETCHE, UNGETCH, PUTC

## SYNOPSIS

```
#include <conio.h>
```

```
char getch(void)
```

```
char getche(void)
```

```
void putch(int c)
```

DESCRIPTION: *Getch()* reads a single character from the console keyboard and returns it without echoing. *Getche()* is similar but does echo the character typed. *Ungetch()* will push back one character such that the next call to *getch()* or *getche()* will return that character. *Putch()* outputs the character *c* to the console screen, prepending a carriage return if the character is a newline.

## SEE ALSO

cgets, cputs

## GETCHAR

## SYNOPSIS

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

```
int getchar(void)
```

DESCRIPTION: *Getchar()* is a *getc(stdin)* operation. It is a macro defined in *stdio.h*. Note that under normal circumstances *getchar()* will NOT return unless a carriage return has been typed on the console. To get a single character immediately from the console, use the routine *getch()*.

## SEE ALSO

getc, fgetc, freopen, fclose

## GETCWD (MS-DOS only)

## SYNOPSIS

```
#include    <sys.h>

char * getcwd(int drive)
```

DESCRIPTION: *Getcwd()* returns the path name of the current working directory on the specified drive, where drive == 0 represents the current drive, drive == 1 represents A:, drive == 2 represents B: etc. The return value is a pointer to a static area of memory which will be overwritten on the next call to *getcwd()*.

## SEE ALSO

chdir

## GETENV

## SYNOPSIS

```
#include    <stdlib.h>

char * getenv(char * s)
extern char ** environ;
```

DESCRIPTION: *Getenv()* will search the vector of environment strings for one matching the argument supplied, and return the value part of that environment string. For example, if the environment contains the string

```
COMSPEC=A:\COMMAND.COM
```

then *getenv("COMSPEC")* will return A:\COMMAND.COM. The global variable *environ* is a pointer to an array of pointers to environment strings, terminated by a null pointer. This array is initialized at startup time under MS-DOS from the environment pointer supplied when the program was executed. Under CP/M no such environment is supplied, so the first call to *getenv()* will attempt to open a file in the current user number on the current drive called ENVIRON. This file should contain definitions for any environment variables desired to be accessible to the program, e.g.

```
HITECH=0:C:
```

Each variable definition should be on a separate line, consisting of the variable name (conventionally all in upper case) followed without intervening white space by an equal sign ('=') then the value to be assigned to that variable.

## GETS

## SYNOPSIS

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

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

DESCRIPTION: *Gets()* reads a line from standard input into the buffer at *s*, deleting the newline (cf. *fgets()*). The buffer is null terminated. It returns its argument, or NULL on end-of-file.

## SEE ALSO

*fgets*, *freopen*

## GETUID (CP/M only)

## SYNOPSIS

```
#include <sys.h>
```

```
int  getuid(void)
```

DESCRIPTION: *Getuid()* returns the current user number. On CP/M, the current user number determines the user number associated with an opened or created file, unless overridden by an explicit user number prefix in the file name.

## SEE ALSO

*setuid*, *open*

## GETW

## SYNOPSIS

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

```
int getw(FILE * stream)
```

DESCRIPTION: *Getw()* returns one word (16 bits for the Z80 and 8086) from the nominated stream. EOF is returned on end-of- file, but since this is a perfectly good word, the *feof()* macro should be used for testing for end-of- file. When reading the word, no special alignment in the file is necessary, as the read is done by two consecutive *getc()*'s. The byte ordering is however undefined. The word read should in general have been written by *putw()*.

## SEE ALSO

putw, getc, fopen, fclose

## GMTIME, LOCALTIME

## SYNOPSIS

```
#include <time.h>
```

```
struct tm * gmtime(time_t * t)  
struct tm * localtime(time_t * t)
```

DESCRIPTION: These functions convert the time pointed to by t which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in *timeh*. *Gmtime()* performs a straight conversion, while *localtime()* takes into account the contents of the global integer *time\_zone*. This should contain the number of minutes that the local time zone is WESTWARD of Greenwich. Since there is no way under MS-DOS of actually pre-determining this value, by default *localtime()* will return the same result as *gmtime()*.

## SEE ALSO

ctime, asctime, time

## INP, OUTP

## SYNOPSIS

```
char inp(unsigned port)
```

```
void outp(unsigned, unsigned data)
```

DESCRIPTION: These routines read and write bytes to and from I/O ports. *inp()* returns the data byte read from the specified port, and *outp()* outputs the data byte to the specified port.

## INT86, INT86X, INTDOS, INTDOSX

## SYNOPSIS

```
#include <dos.h>
```

```
int int86(int intno, union REGS * inregs, union REGS * outregs)
```

```
int int86x(int intno, union REGS inregs, union REGS outregs, struct SREGS * segregs)
```

```
int intdos(union REGS * inregs, union REGS * outregs)
```

```
int intdosx(union REGS * inregs, union REGS * outregs, struct SREGS * segregs)
```

DESCRIPTION: These functions allow calling of software interrupts from C programs. *Int86()* and *int86x()* execute the software interrupt specified by *intno* while *intdos()* and *intdosx()* execute interrupt 21(hex), which is the MS-DOS system call interrupt. The *inregs* pointer should point to a union containing values for each of the general purpose registers to be set when executing the interrupt, and the values of the registers on return are copied into the union pointed to by *outregs*. The *x* versions of the calls also take a pointer to a union defining the segment register values to be set on execution of the interrupt, though only ES and DS are actually set from this structure.

## SEE ALSO

segread

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

## SYNOPSIS

```
#include <ctype.h>
```

```
isalnum(char c)
isalpha(char c)
isascii(char c)
isctrl(char c)
isdigit(char c)
islower(char c)
isprint(char c)
isgraph(char c)
ispunct(char c)
isspace(char c)
isupper(char c)
char c;
```

**DESCRIPTION:** These macros, defined in `ctype.h`, test the supplied character for membership in one of several overlapping groups of characters. Note that all except `isascii` are defined for `c` iff *`isascii(c)`* is true.

<code>isalnum(c)</code>	<code>c</code> is alphanumeric
<code>isalpha(c)</code>	<code>c</code> is in A-Z or a-z
<code>isascii(c)</code>	<code>c</code> is a 7 bit ascii character
<code>isctrl(c)</code>	<code>c</code> is a control character
<code>isdigit(c)</code>	<code>c</code> is a decimal digit
<code>islower(c)</code>	<code>c</code> is in a-z
<code>isprint(c)</code>	<code>c</code> is a printing char
<code>isgraph(c)</code>	<code>c</code> is a non-space printable character
<code>ispunct(c)</code>	<code>c</code> is not alphanumeric
<code>isspace(c)</code>	<code>c</code> is a space, tab or newline
<code>isupper(c)</code>	<code>c</code> is in A-Z
<code>isxdigit(c)</code>	<code>c</code> is in 0-9 or a-f or A-F

## SEE ALSO

`toupper`, `tolower`, `toascii`

## ISATTY

## SYNOPSIS

```
#include <unixio.h>
```

```
int isatty(int fd)
```

DESCRIPTION: This tests the type of the file associated with `fd`. It returns true if the file is attached to a tty-like device. This would normally be used for testing if standard input is coming from a file or the console. For testing `STDIO` streams, use *isatty(fileno(stream))*.

## KBHIT

## SYNOPSIS

```
#include <conio.h>
```

```
int kbhit(void)
```

DESCRIPTION: This function returns 1 if a character has been pressed on the console keyboard, 0 otherwise. Normally the character would then be read via *getch()*.

## SEE ALSO

getch, getche

## LONGJMP

## SYNOPSIS

```
#include <setjmp.h>
```

```
void longjmp(jmp_buf buf, int val)
```

DESCRIPTION: *Longjmp()*, in conjunction with *setjmp()*, provides a mechanism for non-local gotos. To use this facility, *setjmp()* should be called with a *jmp\_buf* argument in some outer level function. The call from *setjmp()* will return 0. To return to this level of execution, may be called with the same *jmp\_buf* argument from an inner level of execution. Note however that the function which called *setjmp()* must still be active when *longjmp()* is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The *val* argument to *longjmp()* will be the value apparently returned from the *setjmp()*. This should normally be non-zero, to distinguish it from the genuine *setjmp()* call. For example:

```
#include <setjmp.h>
```

```
Static jmp_buf jb_err;
```

```
main()
{
    if(setjmp(jb_err)) {
        printf("An error occurred");
        exit(1);
    }
    a_func();
}
```

```
a_func()
{
    if(do_whatever() != 0)
        longjmp(jb_err, 1);
    if(do_something_else() != 0)
        longjmp(jb_err, 2);
}
```

The calls to *longjmp()* above will never return; rather the call to *setjmp()* will appear to return, but with a return value equal to the argument supplied to *longjmp()*.

## SEE ALSO

setjmp

## LSEEK SYNOPSIS

```
#include <unistd.h>
```

```
long lseek(int fd, long offs, int wh)
```

DESCRIPTION: This function operates in an analogous manner to *fseek()*, however it does so on unbuffered low-level i/o file descriptors, rather than on STDIO streams. It also returns the resulting pointer location. Thus *lseek(fd, 0L, 1)* returns the current pointer location without moving it. -1 is returned on error.

## SEE ALSO

open, close, read, write

## MALLOC

## SYNOPSIS

```
#include <stdlib.h>
```

```
void * malloc(size_t cnt)
```

DESCRIPTION: *Malloc()* attempts to allocate cnt bytes of memory from the "heap", the dynamic memory allocation area. If successful, it returns a pointer to the block, otherwise 0 is returned. The memory so allocated may be freed with *free()*, or changed in size via *realloc()*. *Malloc()* calls *sbrk()* to obtain memory, and is in turn called by *calloc()*. *Malloc()* does not clear the memory it obtains.

## SEE ALSO

calloc, free, realloc

## MEMSET, MEMCPY, MEMCMP, MEMMOVE

## SYNOPSIS

```
#include <string.h>
```

```
void  memset(void s, char c, size_t n)
void * memcpy(void * d, void * s, size_t n)
int   memcmp(void * s1, void * s2, size_t n)
void * memmove(void * s1, void * s2, size_t n)
void * memchr(void * s, int c, size_t n)
```

DESCRIPTION: *Memset()* initializes n bytes of memory starting at the location pointed to by s with the character c. *Memcpy()* copies n bytes of memory starting from the location pointed to by s to the block of memory pointed to by d. The result of copying overlapping blocks is undefined. *Memcmp()* compares two blocks of memory, of length n, and returns a signed value similar to *strncmp()*. Unlike *strncmp()* the comparison does not stop on a null character. The ascii collating sequence is used for the comparison, but the effect of including non-ascii characters in the memory blocks on the sense of the return value is indeterminate. *Memmove()* is similar to *memcpy()* except copying of overlapping blocks is handled correctly. The *memchr()* function locates the first occurrence of c (converted to unsigned char) in the initial n characters of the object pointed to by s.

## SEE ALSO

strncpy, strncmp, strchr

## MKDIR, RMDIR SYNOPSIS

```
#include <sys.h>
```

```
int mkdir(char * s)
```

```
int rmdir(char * s)
```

DESCRIPTION: These functions allow the creation (*mkdir()*) and deletion (*rmdir()*) of sub-directories under the MS-DOS operating system. The argument *s* may be an arbitrary pathname, and the return value will be -1 if the creation or removal was unsuccessful.

## SEE ALSO

chdir

## MSDOS, MSDOSCX

## SYNOPSIS

```
#include <dos.h>
```

```
long msdos(int ax, int dx, int cx, int bx, int si, int di)
```

```
long msdoscx(int ax, int dx, int cx, int bx, int si, int di)
```

DESCRIPTION: These functions allow direct access to MS-DOS system calls. The arguments will be placed in the registers implied by their names, while the return value will be the contents of AX and DX (for *msdos()*) or the contents of DX and CX (for *msdoscx()*). Only as many arguments as necessary need be supplied, e.g. if only AH and DX need have specified valued, then only 2 argument would be required. The following piece of code outputs a form-feed to the printer.

```
msdos(0x500, '\f');
```

Note that the system call number (in this case 5) must be multiplied by 0x100 since MS-DOS expects the call number in AH, the high byte of AX.

## SEE ALSO

intdos, intdosx, int86, int86x

## OPEN

## SYNOPSIS

```
#include <unixio.h>
```

```
int open(char * name, int mode)
```

DESCRIPTION: *Open()* is the fundamental means of opening files for reading and writing. The file specified by name is sought, and if found is opened for reading, writing or both. Mode is encoded as follows:

Mode	Meaning
0	Open for reading only
1	Open for writing only
2	Open for both reading and writing

The file must already exist - if it does not, *creat()* should be used to create it. On a successful open, a file descriptor is returned. This is a non-negative integer which may be used to refer to the open file subsequently. If the open fails, -1 is returned. The syntax of a CP/M filename is:

```
[uid:][drive:]name.type
```

where uid is a decimal number 0 to 15, drive is a letter A to P or a to p, name is 1 to 8 characters and type is 0 to 3 characters. Though there are few inherent restrictions on the characters in the name and type, it is recommended that they be restricted to the alphanumerics and standard printing characters. Use of strange characters may cause problems in accessing and/or deleting the file.

One or both of uid: and drive: may be omitted; if both are supplied, the uid: must come first. Note that the [ and ] are meta-symbols only. Some examples are:

```
fred.dat
file.c
0:xyz.com
0:a:file1.p
a:file2.
```

If the uid: is omitted, the file will be sought with uid equal to the current user number, as returned by *getuid()*. If drive: is omitted, the file will be sought on the currently selected drive. The following special file names are recognized:

```
lst:   Accesses the list device - write only
pun:   Accesses the punch device - write only
rdr:   Accesses the reader device - read only
con:   Accesses the system console - read/write
```

File names may be in any case - they are converted to upper case during processing of the name.

MS-DOS filenames may be any valid MS-DOS 2.xx filename, e.g.

```
fred.nrk  
A:\HITECH\STDIO.H
```

The special device names (e.g. CON, LST) are also recognized. These do not require (and should not have) a trailing colon.

#### SEE ALSO

close, fopen, fclose, read, write, creat

## PERROR

#### SYNOPSIS

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

```
void perror(char * s)
```

**DESCRIPTION:** This routine will print on the stderr stream the argument *s*, followed by a descriptive message detailing the last error returned from an open, close read or write call. Unfortunately CP/M does not provide definitive information relating to the error, except in the case of a random read or write. Thus this routine is of limited usefulness under CP/M. MS-DOS provides much more information however, and use of *perror()* after MS-DOS file handling calls will certainly give useful diagnostics.

#### SEE ALSO

open, close, read, write

## PRINTF, VPRINTF

## SYNOPSIS

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

```
int printf(char * fmt, ...)
```

```
int vprintf(char * fmt, va_list va_arg)
```

DESCRIPTION: *Printf()* is a formatted output routine, operating on stdout. There are corresponding routines operating on a given stream (*fprintf()*) or into a string buffer (*sprintf()*). *Printf()* is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values. Each conversion specification is of the form %m.nc where the percent symbol % introduces a conversion, followed by an optional width specification m. n is an optional precision specification (introduced by the dot) and c is a letter specifying the type of the conversion. A minus sign ('-') preceding m indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of m is 0, then padding will be performed with zeroes rather than blanks.

If the character \* is used in place of a decimal constant, e.g. in the format %\*d, then one integer argument will be taken from the list to provide that value. The types of conversion are:

- F Floating point - m is the total width and n is the number of digits after the decimal point. If n is omitted it defaults to 6.
- E Print the corresponding argument in scientific notation. Otherwise similar to f.
- G Use e or f format, whichever gives maximum precision in minimum width.
- o x X u d Integer conversion - in radices 8, 16, 10 and 10 respectively. The conversion is signed in the case of d, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. %8.4x will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an l indicates that the value argument is a long integer or unsigned value. The letter X prints out hexadecimal numbers using the upper case letters **A-F** rather than **a-f** as would be printed when using x.
- s Print a string - the value argument is assumed to be a character pointer. At most n characters from the string will be printed, in a field m characters wide.

- c The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus %% will produce a single percent sign. Some examples:

```
printf("Total = %4d%%", 23)
```

```
    yields 'Total =23%'
```

```
printf("Size is %lx", size)
```

```
    where size is a long, prints size as hexadecimal.
```

```
printf("Name = %.8s", "a1234567890")
```

```
    yields 'Name = a1234567' printf("xx%*d", 3, 4) yields 'xx 4'
```

Printf returns EOF on error, 0 otherwise. *Vprintf()* is similar to *printf()* but takes a variable argument list pointer rather than a list of arguments. See the description of *vastart()* for more information on variable argument lists.

SEE ALSO

fprintf, sprintf

## PUTC

## SYNOPSIS

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

```
int    putc(int c, FILE * stream)
```

DESCRIPTION: *Putc()* is the macro version of *fputc()* and is defined in `stdio.h`. See *fputc()* for a description of its behaviour.

## SEE ALSO

`fputc`, `getc`, `fopen`, `fclose`

## PUTCHAR

## SYNOPSIS

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

```
int    putchar(int c)
```

DESCRIPTION: *Putchar()* is a *putc()* operation on `stdout`, defined in `stdio.h`.

## SEE ALSO

`putc`, `getc`, `freopen`, `fclose`

## PUTS

## SYNOPSIS

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

```
int puts(char * s)
```

DESCRIPTION: *Puts()* writes the string *s* to the stdout stream, appending a newline. The null terminating the string is not copied. EOF is returned on error.

## SEE ALSO

fputs, gets, freopen, fclose

## PUTW

## SYNOPSIS

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

```
int putw(int w, FILE * stream)
```

DESCRIPTION: *Putw()* copies the word *w* to the given stream. It returns *w*, except on error, in which case EOF is returned. Since this is a good integer, *ferror()* should be used to check for errors.

## SEE ALSO

getw, fopen, fclose

## QSORT

## SYNOPSIS

```
#include <stdlib.h>
```

```
void qsort(void * base, size_t nel, size_t width, int (*func)())
```

DESCRIPTION: *Qsort()* is an implementation of the quicksort algorithm. It sorts an array of *nel* items, each of length *width* bytes, located contiguously in memory at *base*. *Func* is a pointer to a function used by *qsort()* to compare items. It calls *func* with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then *func()* should return a value greater than zero, equal to zero or less than zero respectively.

```
Static short array[100];
```

```
#define SIZE sizeof array/sizeof array[0]
a_func(p1, p2)
short * p1, * p2;
{
    return *p1 - *p2;
}

sort_em()
{
    qsort(array, SIZE, sizeof array[0], a_func);
}
```

This will sort the array into ascending values. Note the use of `sizeof` to make the code independent of the size of a short, or the number of elements in the array.

## RAND

## SYNOPSIS

```
#include <stdlib.h>
```

```
int rand(void)
```

DESCRIPTION: *rand()* is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call.

## SEE ALSO

srand

## READ

## SYNOPSIS

```
#include <unistd.h>
```

```
int read(int fd, void * buf, size_t cnt)
```

DESCRIPTION: *read()* will read from the file associated with *fd* up to *cnt* bytes into a buffer located at *buf*. It returns the number of bytes actually read. A zero return indicates end-of-file. A negative return indicates error. *Fd* should have been obtained from a previous call to *open()*. It is possible for *read()* to return less bytes than requested, e.g. when reading from the console, in which case *read()* will read one line of input.

## SEE ALSO

open, close, write

## REALLOC

## SYNOPSIS

```
void * realloc(void * ptr, size_t cnt)
```

DESCRIPTION: *Realloc()* frees the block of memory at *ptr*, which should have been obtained by a previous call to *malloc()*, *calloc()* or *realloc()*, then attempts to allocate *cnt* bytes of dynamic memory, and if successful copies the contents of the block of memory located at *ptr* into the new block. At most, *realloc()* will copy the number of bytes which were in the old block, but if the new block is smaller, will only copy *cnt* bytes. If the block could not be allocated, 0 is returned.

## SEE ALSO

*malloc*, *calloc*, *realloc*

## REMOVE

## SYNOPSIS

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

```
int remove(char * s)
```

DESCRIPTION: *Remove()* will attempt to remove the file named by the argument *s* from the directory. A return value of -1 indicates that the attempt failed.

## SEE ALSO

*unlink*

## RENAME

## SYNOPSIS

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

```
int rename(char * name1, char * name2)
```

DESCRIPTION: The file named by name1 will be renamed to name2. -1 will be returned if the rename was not successful. Note that renames across user numbers or drives are not permitted.

## SEE ALSO

open, close, unlink

## REWIND

## SYNOPSIS

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

```
int rewind(FILE * stream)
```

DESCRIPTION: This function will attempt to re-position the read/write pointer of the nominated stream to the beginning of the file. A return value of -1 indicates that the attempt was not successful, perhaps because the stream is associated with a non-random access file such as a character device.

## SEE ALSO

fseek, ftell

## SBRK

## SYNOPSIS

```
char * sbrk(int incr)
```

DESCRIPTION: *Sbrk()* increments the current highest memory location allocated to the program by *incr* bytes. It returns a pointer to the previous highest location. Thus *sbrk(0)* returns a pointer to the current highest location, without altering its value. If there is insufficient memory to satisfy the request, -1 is returned.

## SEE ALSO

brk, malloc, calloc, realloc, free

## SCANF

## SYNOPSIS

```
#include <stdio.h>

int scanf(char * fmt, ...)
int vscanf(char *, va_list ap);
```

DESCRIPTION: *Scanf()* performs formatted input ("de-editing") from the stdin stream. Similar functions are available for streams in general, and for strings. The function *vscanf()* is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialized with *vastart()*. The input conversions are performed according to the *fmt* string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. spaces, tabs or newlines. A conversion specification takes the form of the character %, optionally followed by an assignment suppression character (\*), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the *fmt* string, there should be two additional pointer arguments. The conversion characters are as follows:

- o x d Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

- f Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.
- s Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null-terminated.
- c Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field width is specified, then copy that many characters. This differs from the s format in that white space does not terminate the character sequence.

The conversion characters o, x, u, d and f may be preceded by an l to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding h will indicate that the pointer argument is a pointer to short rather than int.

*Scanf()* returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed. Some examples are:

```
scanf("%d %s", &a, &s)
    with input "12s", will assign 12 to a, and "s" to s.
```

```
scanf("%4cd %lf", &c, &f)
    with input " abcd -3.5", will assign " abc" to c, and -3.5 to f.
```

#### SEE ALSO

fscanf, sscanf, printf, va\_arg

## SEGREAD SYNOPSIS

```
#include    <dos.h>
int        segread(struct SREGS * segregs)
```

DESCRIPTION: *Segread()* copies the values of the segment registers into the structure pointed to by *segregs*.

## SEE ALSO

*int86*, *int86x*, *intdos*, *intdosx*

## SETJMP SYNOPSIS

```
#include    <setjmp.h>
int        setjmp(jmp_buf buf)
```

DESCRIPTION: *Setjmp()* is used with *longjmp()* for non-local gotos. See *longjmp()* for further information.

## SEE ALSO

*longjmp*

## SETUID (CP/M only)

## SYNOPSIS

```
#include <sys.h>
```

```
void setuid(int uid)
```

DESCRIPTION: *Setuid()* will set the current user number to uid. Uid should be a number in the range 0-15.

## SEE ALSO

getuid

## SETVBUF, SETBUF SYNOPSIS

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

```
int setvbuf(FILE * stream, char * buf, int mode, size_t size);
```

```
void setbuf(FILE * stream, char * buf)
```

DESCRIPTION: The *setvbuf()* function allows the buffering behaviour of a STDIO stream to be altered. It supersedes the function *setbuf()* which is retained for backwards compatibility. The arguments to *setvbuf()* are as follows: stream designates the STDIO stream to be affected; buf is a pointer to a buffer which will be used for all subsequent I/O operations on this stream. If buf is null, then the routine will allocate a buffer from the heap if necessary, of size BUFSIZ as defined in <stdio.h>. mode may take the values \_IONBF, to turn buffering off completely, \_IOFBF, for full buffering, or \_IOLBF for line buffering. Full buffering means that the associated buffer will only be flushed when full, while line buffering means that the buffer will be flushed at the end of each line or when input is requested from another STDIO stream. size is the size of the buffer supplied. For example:

```
setvbuf(stdout, my_buf, _IOLBF, sizeof my_buf);
```

If a buffer is supplied by the caller, that buffer will remain associated with that stream even over *fclose()*, *fopen()* calls until another *setvbuf()* changes it.

## SEE ALSO

fopen, freopen, fclose

## SET\_VECTOR SYNOPSIS

```
#include    <intrpt.h>
typedef interrupt void (*isr)();
isr set_vector(isr * vector, isr func);
```

**DESCRIPTION:** This routine allows an interrupt vector to be initialized. The first argument should be the address of the interrupt vector (not the vector number but the actual address) cast to a pointer to *isr*, which is a typedef'd pointer to an interrupt function. The second argument should be the function which you want the interrupt vector to point to. This must be declared using the *interrupt* type qualifier. The return value of *setvector()* is the previous contents of the vector.

## SEE ALSO

di(), ei()

## SIGNAL

## SYNOPSIS

```
#include <signal.h>
void (* signal)(int sig, void (*func)());
```

DESCRIPTION: *Signal()* provides a mechanism for catching control-C's (ctrl-BREAK for MS-DOS) typed on the console during I/O. Under CP/M the console is polled whenever an I/O call is performed, while for MS-DOS the polling depends on the setting of the BREAK command. If a control-C is detected certain action will be performed. The default action is to exit summarily; this may be modified with *signal()*. The sig argument to signal may at the present time be only SIGINT, signifying an interrupt condition. The func argument may be one of SIG\_DFL, representing the default action, SIG\_IGN, to ignore control-C's completely, or the address of a function which will be called with one argument, the number of the signal caught, when a control-C is seen. As the only signal supported is SIGINT, this will always be the value of the argument to the called function.

## SEE ALSO

exit

## SIN

## SYNOPSIS

```
#include <math.h>

double sin(double f);
```

DESCRIPTION: This function returns the sine function of its argument.

## SEE ALSO

cos, tan, asin, acos, atan

## SPAWN, SPAWNV, SPAWNVE

## SYNOPSIS

```
int    spawn(char * n, char * argv0, ...);
int    spawnv(char * n, char ** v)
int    spawnve(char * n, char ** v, char ** e)
```

DESCRIPTION: These functions will load and execute a sub-program, named by the argument *n*. The calling conventions are similar to the functions *execl()* and *execv()*, the difference being that the spawn functions return to the calling program after termination of the sub-program, while the exec functions return only if the program could not be executed. *Spawnve()* takes an environment list in the same format as the argument list which will be supplied to the executed program as its environment.

## SEE ALSO

execl, execv

## SPRINTF

## SYNOPSIS

```
#include    <stdio.h>

int    sprintf(char * buf, char * fmt, ...);
int    vsprintf(char * buf, char * fmt, va_list ap);
```

DESCRIPTION: *Sprintf()* operates in a similar fashion to *printf()*, except that instead of placing the converted output on the stdout stream, the characters are placed in the buffer at *buf*. The resultant string will be nullterminated, and the number of characters in the buffer will be returned. *Vsprintf()* takes an argument pointer rather than a list of arguments.

## SEE ALSO

printf, fprintf, scanf

## SQRT

## SYNOPSIS

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

```
double sqrt(double f)
```

DESCRIPTION: *Sqrt()* implements a square root function using Newton's approximation.

## SEE ALSO

exp

## SSCANF

## SYNOPSIS

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

```
int sscanf(char * buf, char * fmt, ...);
```

```
int vsscanf(char * buf, char * fmt, va_list ap);
```

DESCRIPTION: *Sscanf()* operates in a similar manner to *scanf()*, except that instead of the conversions being taken from stdin, they are taken from the string at buf.

## SEE ALSO

scanf, fscanf, sprintf

## SRAND

## SYNOPSIS

```
#include <stdlib.h>
```

```
void srand(int seed)
```

DESCRIPTION: *Srand()* initializes the random number generator accessed by *rand()* with the given seed. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by *rand()*. On the z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do.

## SEE ALSO

rand

## STAT

## SYNOPSIS

```
#include <stat.h>
```

```
int stat(char * name, struct stat * statbuf)
```

DESCRIPTION: This routine returns information about the file by name. The information returned is operating system dependent, but may include file attributes (e.g. read only), file size in bytes, and file modification and/or access times. The argument name should be the name of the file, and may include path names under DOS, user numbers under CP/M, etc. The argument statbuf should be the address of a structure as defined in *stat.h* which will be filled in with the information about the file. The structure of *struct stat* is as follows:

```
struct stat
{
  Short  st_mode;      /* flags */
  Long   st_atime;     /* access time */
  Long   st_mtime;     /* modification time */
  Long   st_size;      /* file size */
};
```

The access and modification times (under DOS these are both set to the modification time) are in seconds since 00:00:00 Jan 1 1970. The function *ctime()* may be used to convert this to a readable value. The file size is self explanatory. The flag bits are as follows:

Flag	Meaning
S_IFMT	mask for file type
S_IFDIR	file is a directory
S_IFREG	file is a regular file
S_IREAD	file is readable
S_IWRITE	file is writeable
S_IEXEC	file is executable
S_HIDDEN	file is hidden
S_SYSTEM	file is marked system
S_ARCHIVE	file has been written to

*Stat* returns 0 on success, -1 on failure, e.g. if the file could not be found.

SEE ALSO

ctime, creat, chmod

STRCAT, STRCMP, STRCPY, STRLEN et. al.

SYNOPSIS

```
#include <string.h>

char * strcat(char * s1,
char * s2); int strcmp(char * s1, char * s2);
char * strcpy(char * s1, char * s2);
int strlen(char * s);
char * strncat(char * s1, char * s2, size_t n);
int strncmp(char * s1, char * s2, size_t n);
char * strncpy(char * s1, char * s2, size_t n);
```

DESCRIPTION: These functions provide operations on null-terminated strings. *Strcat()* appends the string *s2* to the end of the string *s1*. The string at *s1* will be null terminated. Needless to say the buffer at *s1* must be big enough. *Strcmp()* compares the two strings and returns a number greater than 0, 0 or a number less than 0 according to whether *s1* is greater than, equal to or less than *s2*. The comparison is via the ascii collating order, with the first character the most significant. *Strcpy()* copies *s2* into the buffer at *s1*, null terminating it. *Strlen()* returns the length of *s1*, not including the terminating null. *Strncat()*, *strncmp()* and *strncpy()* will concatenate, compare and copy *s2* and *s1* in the same manner as their similarly named counterparts above, but involving at most *n* characters. For *strncpy()*, the resulting string may not be null terminated.

## STRCHR, STRRCHR

## SYNOPSIS

```
#include <string.h>
```

```
char * strchr(char * s, int c)  
char * strrchr(char * s, int c)
```

DESCRIPTION: These functions locate an instance of the character *c* in the string *s*. In the case of *strchr()* a pointer will be returned to the first instance of the character found by searching from the beginning of the string, while *strrchr()* searches backwards from the end of the string. A null pointer is returned if the character does not exist in the string.

## SEE ALSO

## SYSTEM

## SYNOPSIS

```
#include <sys.h>
```

```
int system(char * s)
```

DESCRIPTION: When executed under MS-DOS *system()* will pass the argument string to the command processor, located via the environment string COMSPEC, for execution. The exit status of the command processor will be returned from the call to *system()*. For example, to set the baud rate on the serial port on an MS-DOS machine:

```
system("MODE COM1:96,N,8,1,P");
```

This function will not work on CP/M-86 since it does not have an invocable command interpreter. Under Concurrent CP/M the CLI system call is used.

## SEE ALSO

spawnl, spawnv

## TAN

## SYNOPSIS

```
#include <math.h>

double tan(double f);
```

DESCRIPTION: This is the tangent function.

## SEE ALSO

sin, cos, asin, acos, atan

## TIME

## SYNOPSIS

```
#include <time.h>

time_t time(time_t * t)
```

DESCRIPTION: This function returns the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument *t* is nonnull, the same value is stored into the object pointed to by *t*. The accuracy of this function is naturally dependent on the operating system having the correct time. This function does not work under CP/M-86 or CP/M 2.2 but does work under Concurrent-CP/M and CP/M+.

## SEE ALSO

ctime, gmtime, localtime, asctime

## TOUPPER, TOLOWER, TOASCII

## SYNOPSIS

```
#include <ctype.h>
char toupper(int c);
char tolower(int c);
char toascii(int c);
char c;
```

DESCRIPTION: *Toupper()* converts its lower case alphabetic argument to upper case, *tolower()* performs the reverse conversion, and *toascii()* returns a result that is guaranteed in the range 0-0177. *Toupper()* and *tolower()* return their arguments if it is not an alphabetic character.

## SEE ALSO

islower, isupper, isascii et. al.

## UNGETC

## SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE * stream)
```

DESCRIPTION: *Ungetc()* will attempt to push back the character *c* onto the named stream, such that a subsequent *getc()* operation will return the character. At most one level of pushback will be allowed, and if the stream is not buffered, even this may not be possible. EOF is returned if the *ungetc()* could not be performed.

## SEE ALSO

getc

## UNLINK

## SYNOPSIS

```
int    unlink(char * name)
```

DESCRIPTION: *Unlink()* will remove (delete) the named file, that is erase the file from its directory. See *open()* for a description of the file name construction. Zero will be returned if successful, -1 if the file did not exist or it could not be removed.

## SEE ALSO

open, close, rename, remove

## VA\_START, VA\_ARG, VA\_END

## SYNOPSIS

```
#include    <stdarg.h>
```

```
void    va_start(va_list ap, parmN);
type    va_arg(ap, type);
void    va_end(va_list ap);
```

DESCRIPTION: These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time. The rightmost parameter to the function (shown as parmN) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type va\_list should be declared, then the macro va\_start invoked with that variable and the name of parmN. This will initialize the variable to allow subsequent calls of the macro va\_arg to access successive parameters. Each call to va\_arg requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int*, *unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by va\_arg(ap, int) since the char will have been widened to int. An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

```
#include    <stdarg.h>
prf(int n, ...)
{
    va_list    ap;

    va_start(ap, n);
    while(n--)
        puts(va_arg(ap, char *));
    va_end(ap);
}
```

## WRITE

### SYNOPSIS

```
#include    <unistd.h>
```

```
int    write(int fd, void * buf, size_t cnt)
```

**DESCRIPTION:** *Write()* will write from the buffer at buf up to cnt bytes to the file associated with the file descriptor fd. The number of bytes actually written will be returned. EOF or a value less than cnt will be returned on error. In any case, any return value not equal to cnt should be treated as an error (cf. *read()* ).

### SEE ALSO

open, close, read

## INDEX

8086	70	CP/M	26,106,131,155
absolute value	116	CP/M-80 compiler	2
acos()	103	cputs()	107
address		creat()	111
link	73	creating a library	76
load	73	CREF	83
ANSI Standard	13,17	cross compilers	3
Array		cross reference	36,83
dimension	16	ctime() 111	
index	16	ctrl-Z	26,120
size	16	date and time	104
ASCII	26	debugging 10,105	
asctime()	104	device name as file	23
asin()	103	device	
assembler		as file	141
in C programs	20	di()	113
assert()	105	directory	108,139
atan()	103	current	130
atan2()	103	disk changing on floppy disk	
Atari ST	3	systems 3	
atexit()	104	disk space	33
atof()	105	div()	12
atoi()	105	div_t	112
atol()	105	dup()	113
bdos()	106	editor	3,33
binary output file	8	ei()	113
bios()	106	entering long commands	7
braces		Enumerated Types	16
omission of	16	environment	130
byte ordering	21	error message:	
C command	3,7	Can't generate code	33
C reference books	2	Error closing file	33
calloc()	107	No room	33
carriage return	26,120	out of memory	33
ceil()	116	Undefined symbol	33
cgets()	107	Write error	33
change file attributes	108	error messages	23,33
character testing	134	format of	23
chdir()	108	LIBR	78
checksum	79,80	list of	23,85
chmod()	108	redirection of	23
class name	70	execl()	114
close()	109	executing a sub-program	156
cleof()	109	executing system commands	
clrerr()	109	160	
command line	7,12,77,160	execution profiling	10
Compatibility	25	execv()	114
compiler driver	3	exit()	114
Compiler Structure	5	exp()	115
console I/O	107,129,135	exponential functions	115
control-C handling	155	Extern Declarations	30
converting ascii to binary	105	fabs()	116
cos()	110	fclose()	116
cosh()	110	feof()	117
		ferror()	117

fflush()	117	getw()	132
fgetc()	118	global variables	30
fgets()	118	gmtime()	132
FILE	99	hex output file	8
file descriptor	27,111,119	I/O redirection	127
duplicating	113	I/O	
FILE pointer	27	ASCII	26
File		Binary	26
access time	158	standard	25,99
attributes	108,158	in-line assembler	20
closing a	109,116	Initialization Syntax	16
creating a	111	initializer	16
deleting a	148,163	inp()	133
flushing buffer to a	117	int86()	133
information about a	158	int86x()	133
modification time	158	intdos()	133
opening a	119,123,140	intdosx()	133
random access to a Intel		hex format	79
125,137		interrupt	133
reading from a	122,147	interrupt handling	155
renaming a	149	interrupt vector	
size	158	setting	154
type of a	135	isalnul()	134
writing to a	126,164	isalnum()	134
fileno()	119	isalpha()	134
floating point		isascii()	134
library	27	satty()	135
options required to use	4	iscntrl()	134
		isdigit()	134
floor()	116	isgraph()	134
fopen()	26,119	islower()	134
formatted input	124,150	ispunct()	134
formatted printing	121,142	isspace()	134
fprintf()	121	isupper()	134
fputc()	121	kbhit()	135
fputs()	122	Kernighan and Ritchie	2,16
fread()	122	large model	70,80
free()	123	ldexp()	124
freopen()	123	ldiv()	112
frexp()	124	LIBR	75
fscanf()	124	librarian	75
fseek()	125	libraries	7
ftell()	126	library manager	75
function parameter	163	libraryordering	77
function		library	
declaration	17	creating	76
large	33	floating point	33
parameters	17	ordering	33
prototype	17,163	line number symbols	10
fwrite()	126	linker	69
getc()	128	LINT	16
getch()	129	localtime	132
getchar()	25,129	log()	115
getche()	129	log10()	115
getcwd()	130	logarithmic functions	115
getenv()	130	long	30
gets()	131	longjmp()	136
getuid()	131	lseek()	137

Machine Dependencies	21	pack pragma	20
Macros		parameter	
predefined	21	type	17
malloc()	137	perror()	141
Member Names	13,29	portable code	21
memcmp()	138	pow()	115
memcpy()	138	pragma	20,21
memory allocation	107,148,150	printf	33
Releasing	123	printf()	142
		program execution	156
memory model	31	psect	
memory requirements	2	linking	69
memset()	138	local	70
mkdir()	139	putc()	144
Motorola hex format	80	putch()	129
MS-DOS	108,133,139,155	putchar()	144
msdos()	139	puts()	145
msdoscx()	139	putw()	145
multiple definitions	30	qsort()	146
newline	26	rand()	147
Newton's approximation	157	random numbers	147,158
non-local goto	136,152	read()	147
objtohex	72,79	realloc()	148
open()	140	relocation	69
Operating Details	7	remove()	148
Options	7	rename()	149
-1	10	return	26,120
-11	11	rewind()	149
-2	11	rmdir()	139
-6301	11	S format hex	80
-A	8	sbrk()	150
-B	9	scanf	33
-C	7	scanf()	150
-CPM	7	segment registers	152
-CR	7	segread()	152
-D	8	self relocation	72
-E	10	setbuf()	153
-F	8	setjmp()	152
-G	10	setuid()	153
-H	10	setvbuf()	153
-I	8	set_vector()	154
-LF	4,27,33	shift	
-M	8	signed	14
-O	8	unsigned	14
-Ooutfile	8	short	21,30
-P	10	signal()	155
-R	8	sin()	155
-S	7	sinh()	110
-U	8	size of data types	21
-V	3,8	sorting	146
-W	10	source file naming	3
-X	8	source level debugging	10
-Z	10	spawnl()	156
Linker	71	spawnv()	156
outp()	133	spawnve()	156
pack	21	Specific Features	13
		sprintf()	156
		sqrt()	157

square root function	157	Unsigned Types	14
srand()	158	user ID	131,153
sscanf()	157	V3.09	2
Standard Libraries	25	va_arg	18,163
Standard Library Functions	99	va_end	163
stat()	158	va_start	163
stdarg.h	18,163	void	18
STDIO	25,27,99	Pointer to	18
storage allocator	18	warning level	10
strcat()	159	warning messages	10
strchr()	160	suppressing	10
strcmp()	159	wild card expansion	127
strcpy()	159	Wordstar	33
stream	27,99	write()	164
manipulation	159	Z180 MMU	68
strlen()	159	Bank Base Register	68
strncat()	159	BBR	68
strncmp()	159	CBAR	68
strncpy()	159	CBR	68
strrchr()	160	Common Base Register	68
Structure Operations	15	Common/Bank Area Register	68
structure packing	20	Z180 Registers	68
structures		ASCII	68
as function argument	15	Bank Base Register	68
assignment	15	BBR	68
functions returning	15	BCR0H	68
Stylistic Considerations	29	BCR0L	68
symbol file	10,80	BCR1H	68
symbols		BCR1L	68
cross reference	83	CBAR	68
global	70,76	CBR	68
System Requirements	2	CNTLA0	68
system()	160	CNTLA1	68
tan()	161	CNTLB0	68
tanh()	110	CNTLB1	68
temporary files	4	CNTR	68
text editor	3	Common Base Register	68
time and date	104	Common/Bank Area Register	68
time()	161	CSI/0	68
toascii()	162	DAR0B	68
tolower()	162	DAR0H	68
toupper()	162	DAR0L	68
trigonometric functions	103,110	DCNTL	68
Hyperbolic	110	DMA	68
Type Checking	13	DMA Byte Count	68
type qualifiers	19	DMA Desitination Address	68
typecast	13	DMA Mode Register	68
typecasting	18	DMA Source Address	68
ungetc()	162	DMA Status Register	68
ungetch()	129	DMODE	68
Unix		DSTAT	68
C compiler	16	FRC	68
V7	25	Free Running Counter	68
unlink()	163	IAR1H	68
unsigned	23	IAR1L	68
unsigned char	21		

ICR	68	INC	55
IL	68	IND	55
Interrupt Vector Register		INDR	56
ITC	68	INI	56
MAR1B	68	INIR	56
MAR1H	68	JP	56
MAR1L	68	JR	56
MMU	68	LD	56,57,58,59
OMCR	68	LDD	59
RCR	68	LDDR	59
Refresh Control Register	68	LDI	59
		LDIR	59
RLDR0H	68	MLT	59
RLDR0L	68	NEG	59
RLDR1H	68	NOP	59
RLDR1L	68	OR	59,60
SAR0B	68	OTDM	61
SAR0H	68	OTDMR	60
SAR0L	68	OTDR	60
STAT0	68	OTIM	61
STAT1	68	OTIMR	60
TCR	68	OTIR	60
TDR0	68	OUT	60
TDR1	68	OUT0	60
TIMER 0	68	OUTD	60
TIMER 1	68	OUTI	61
TMDR0H	68	POP	61
TMDR0L	68	PUSH	61
TMDR1H	68	RES	61,62,63
TMDR1L	68	RET	63
TRDR	68	RETI	63
TSR0	68	RETN	63
TSR1	68	RL	63
Z80 Instructions	50	RLA	63
ADC	51	RLC	63
ADD	51	RLCA	63
AND	51	RLD	63
BIT	52,53	RR	63,64
CALL	53	RRA	64
CCF	53	RRC	64
CP	53,54	RRCA	64
CPD	54	RRD	64
CPDR	54	RST	64
CPI	54	SBC	64
CPIR	54	SCF	64
CPL	54	SET	64,65,66
DAA	54	SLA	66
DEC	54	SLP	66
DI	54	SRA	66
DJNZ	54	SRL	67
EI	54	SUB	67
EX	55	TST	67
EXX	55	TSTIO	67
HALT	55	XOR	67
IM	55	Z80 pseudo ops	
IN	55	IRP	47
IN0	55	IRPC	47
		ZAS directives	8

Title	48	operators	38
ZAS options		program sections	39
-J	35	PSECT	43
-L	36	psects	39,43
-N	35	pseudo ops	40
-O	36	relocatable object code	39
-U	35		
-W	36	temporary labels	37
ZAS psect flags		Z180	35
ABS	43	Zilog	35
GLOBAL	43	_exit()	115
LOCAL	43	_getargs()	127
OVRLD	43		
PURE	43		
ZAS pseudo ops	40		
COND	42		
DB	40		
DEFB	40		
DEFF 41			
DEFL 41			
DEFM 42			
DEFS 41			
DEFW 41			
ELSE 42			
END 42			
ENDC 42			
ENDM 44			
EQU 41			
GLOBAL 43			
IF 42			
LOCAL 45			
MACRO 44			
ORG 44			
PSECT 43			
REPT 46			
ZAS			
64180	35		
arithmetic overflow	35		
binary constants	37		
character constants	38		
condition codes	48		
conditional assembly	42		
constants	37		
directives	48		
extended condition codes	48		
external symbols	35		
floating point constants	38		
hexadecimal constants	37		
jump optimization	35		
jumps	35		
labels	36		
listing	36		
macros	44		
mnemonics	35		
octal constants	37		
opcode constants	38		