

# Better Working™

From Spinnaker®

## POWER C

POWER C is a disk based true C language compiler for the Commodore computers.



COMMODORE™ 64/128  
Disk

# Better Working<sup>TM</sup>

From Spinnaker<sup>®</sup>

## POWER C

POWER C is a disk based true C language compiler for the Commodore computers.

The use of the C programming language is one of the more important developments in the micro-computer industry. C is a simple and elegant programming language with an absence of restrictions and a generality that makes it far more effective and convenient than other programming languages. It combines the productivity of high level languages with the control of low level machine languages. It was developed at Bell Labs where it was used to program the popular UNIX operating system. For these reasons, C has rapidly become a major computer programming language and most serious micro-computer software development today is being performed in C.

POWER C is a compact, fast and powerful language that is easier to use than BASIC. Not only can programs be written more quickly in POWER C than in BASIC but they run more than ten times faster.

Welcome to the future of programming.

### POWER C DELIVERS:

- Native machine language object code which is more than ten times faster than BASIC.
- Complete language components such as C shell, full screen syntax checking editor, compiler, linker etc.
- Complete library of over 95 functions.
- Complete library of C utilities including such utilities as sort, find, and a powerful text formatter.
- A tutorial to get you started.
- Full support of floating point.
- Conformance to Kernighan and Ritchie standards.
- Un-copy protected disks so back-up copies are hassle free.
- Two complete development environments-one for the C-64 and one for the C-128



Better Working is a trademark of Spinnaker Software Corp. © 1986 Spinnaker Software Corp., One Kendall Square, Cambridge, MA 02139. All rights reserved. Spinnaker is a registered trademark of Spinnaker Software Corp.

**BX-PWC**

From Spinnaker®

[illegible]



Copyright 1986 by Spinnaker Software Corporation. All rights reserved. The distribution and sale of this product are intended for the use of the original purchaser only and for use only on the computer system specified. Lawful users of this program are hereby licensed only to read the program from its medium into memory of a computer for the purpose of executing this program. Copying, duplicating, selling or otherwise distributing this product is hereby expressly forbidden.

Commodore 64 and 128 are trademarks of Commodore Business Machines, Inc.

## FOREWORD

The use of the C programming language is probably one of the more important developments in the micro computer field because C is not tied to any one particular manufacturer's hardware or Disk Operating System. Thus it is rapidly becoming a major program transportability factor between the multitude of different CPU's and operating systems, from mainframes to minis to micros. C was originated and placed in the public domain by Dennis M. Ritchie, of Bell Labs.

The C language has been described as a "simple and elegant" programming language with an absence of restrictions and a generality that makes it far more effective and convenient than "supposedly more powerful programming languages". The UNIX operating system was written in C. Most current terminal programs and many of the new word processors have been written in the C language. What an incredible luxury for a programmer, to be able to compile his C source for each of several different CPU's instead of rewriting new source code for each.

C is easy to learn, as it has only about a dozen commands and depends on function libraries to gain speed and efficiency.

## GETTING STARTED

Your POWER C System Disk contains the C Shell, Editors, Compiler, Translator, Linker, Stdio library, Math library and five sample C source code listings: test.c, format.c and format2.c, print.c, sort.c, find.c and wfreq.c. We strongly advise that a back-up system disk be made from the original system disk and used instead of the original, which should then be stored safely away until needed to create another back-up.

The reverse side of the system disk is the Library Disk. It contains the Stdlib.l and Syslib.l Function libraries. This Library Disk should also be backed-up immediately and we suggest that you use that work copy of the Library Disk rather than take a chance with the original. Instructions for making back-up disks are contained in your disk drive manual.

## TABLE OF CONTENTS

Compatibility	1
ML Interface	5
Coding Efficiently	7
C SHELL, Command Interpreter	8
Commodore 64	8
Commodore 128	11
FUNCTION KEYS	
Commodore 64	10
Commodore 128	12, 13
EDITOR	
Commodore 64	10
Commodore 128	13
COMPILER	15
LINKER	15
LIBRARY, Introduction	16
FUNCTION, Index	18
FUNCTION, Library	19
TUTORIAL	47
C SHELL UTILITIES	53
RECOMMENDED READING LIST	59
INDEX	60





## Compatibility

Spinnaker's POWER C compiler is generally quite compatible with those found on other micro computers and mainframes. However, there are a few potential compatibility problems in all C language compilers which should be kept in mind if you plan to port code from other machines to the C64 or vice versa. The main areas of concern are:

1. Standard C features not implemented or not implemented completely
2. Implementation dependent details (e.g. type sizes)
3. Non-standard library functions

These will be discussed in the following sections.

### 1. Features Not Implemented

#### 1.1 Bit Fields

To obtain the effect of bit fields the programmer must use shifts and masks. For example, to get the value of the third through sixth bits of an int variable y one might use the code

```
x = (y >> 2) & 0xf;
```

To assign a quantity to the same field one might use

```
y = (y & ~0x3c) | (x << 2);
```

... where ~ is the bitwise not operator. This is admittedly awkward, but in practice bit fields are not frequently used so their absence should not present a great problem.

#### 1.2 Pointer Initialization

Static pointer variables may not be initialized except for character pointers initialized with strings. For example, the declaration:

```
static char *s = "this is a string";
```

is allowed, while:

```
static int x;  
static int *y = &x;
```

... is not allowed. Static variables include variables explicitly declared static and variables declared outside of a function but given no explicit storage class declaration. Auto variable initialization is fully supported.

### 1.3 Conditional Operator Type Conversions

The second and third operands of the conditional operator are not brought to a common type if they are different. If the types are different the programmer must make the conversion explicit. For example the code:

```
int i;  
float f, x;  
  
x = a==b ? f : i;
```

... may generate incorrect machine code, while the statement:

```
x = a==b ? f : (float) i;
```

... will generate correct machine code.

### 1.4 Operators

Certain operators under certain conditions will not work unless the expressions containing them are parenthesized. Namely the logical or, conditional, assignment, and comma operators behave this way in subscripts and constant expressions. For example, the expression:

```
a[i > j ? i : j]
```

... will generate a syntax error, while the expression:

```
a[(i > j ? i : j)]
```

... will be accepted.

## 2. Implementation Details

### 2.1 Type Sizes

The following table lists the size in bytes of all data types supported by the compiler:

<u>Type</u>	<u>Size</u>
char	1
short	2
int	2
long	2
unsigned	2
float	5
double	5
pointer	2

Note that short, int, and long are synonyms, as are float and double.

Integers and pointers are stored low byte first, high byte last. Float quantities are stored in the same format as BASIC variables.

### 2.2 Identifiers

All identifiers (including external identifiers) are significant to eight characters, except for goto labels, which are significant to seven characters. Identifiers may contain up to 255 characters.

### 2.3 Sign Extension and Sign Fill

The compiler does not sign extend character quantities. When characters are converted to integers, the high byte is simply set to zero.

Similarly, integers are not sign filled when shifted right. Vacated bits are set to zero even if the quantity being shifted is negative.

### 2.4 Register Declarations

All register declarations are ignored. However, the first 32 bytes of integers, characters, and pointers declared in a function are placed in the zero page, while the remainder are put in a stack in main memory. Thus if certain variables are used frequently they should be declared before those used less frequently.

## 2.5 Character and String Constants

Multi-character constants are not supported. If more than one character appears in a character constant only the first is taken.

String constants may not contain more than 255 characters.

## 2.6 Header Files

The pre-processor lines:

```
#include <filename>
and
#include "filename"
```

... are synonymous. The header file filename must be on the same disk as the source file containing the pre-processor line.

NOTE: If the file is one of the standard header files such as `stdio.h` or `math.h`, it must be copied from the system disk to your work disk.

## 2.7 Program Size

Due to the limited memory available in the C-64, the compiler may not be able to handle some very large source files. It is unlikely that the Power C 128 compiler will run into insufficient memory problems and not be able to handle the large source files.

If an overflow message is printed when you are compiling don't despair; split the big source file into two or more smaller source files and compile them separately. They will join up again when you link them.

If the linker gives an overflow message you have two possible courses of action: use more efficient coding techniques or rewrite some of the program in assembler (see the section on interfacing C with assembler).

## 2.8 EOF Marker

End Of File may be signaled from the keyboard with a period (.) on an otherwise blank line.

## 3.0 The Library

Although a substantial attempt has been made to make the library provided with the compiler as standard as possible, the lack of a standard operating system on the Commodore 64 has required that some compromises be made. Before using a function listed in the UNIX (or other) standard library look up the description of the Spinnaker version in the library section.

## Interfacing with Machine Language

Although C is well suited to a wide range of tasks, no high level language can match the speed and compactness of hand written assembly code, especially with a micro processor such as the 65xx or 85xx. Thus if speed is critical for a particular part of an application, or if a program is too large to be written entirely in C, it would be desirable to be able to call machine language subroutines from within a C program. This can be done with the library function `sys`, whose description can be found in the library section.

The recommended procedure is to first write those sections of the program which are to be eventually written in assembler as C functions. Once the program is debugged, the C functions may be replaced one at a time with assembler routines (Better Working's Power Assembler may be used). The advantage of this approach is that it maximizes portability since the very non-portable machine language calls are isolated in a set of functions. Also, if the computer the program is being ported to has a larger memory than the C-64, machine language routines may be unnecessary, in which case you can use the original C functions.

The following small example illustrates the procedure. The code which is to be written in assembler is to add three small positive integers. The code is first written as a C function `add3`, which is shown below along with a driver program:

```
#include <stdio.h>

main()
{
    int a, b, c;

    while ((printf("enter three numbers: "),
            scanf("%d %d %d", &a, &b, &c)) != EOF)

        printf("sum: %d", add3(a, b, c));
}

add3 (arg1, arg2, arg3)
{
    return (arg1 + arg2 + arg3);
}
```

The rewritten add3 with the associated machine language subroutine (located at c000 hex) is shown below:

<b>add3 (args)</b>	<b>c000 stx \$4b</b>
<b>float args;</b>	<b>sty \$4c</b>
<b>{</b>	<b>clc</b>
<b>char a, x, y;</b>	<b>ldy #0</b>
<b>x = (int) &amp;args;</b>	<b>lda (\$4b),y ; load arg 1</b>
<b>y = (int) &amp;args &gt;&gt; 8;adc (\$4b),y ; add</b>	<b>ldy #2</b>
<b>sys (0xc000, &amp;a, &amp;x, &amp;y);</b>	<b>ldy #4</b>
<b>return a;</b>	<b>adc (\$4b),y ; add arg 3</b>
<b>}</b>	<b>rts</b>

Note that only one argument is declared and that it is given type float. This is to ensure that the arguments remain in consecutive memory locations and in the correct order. Note also that the x and y registers are loaded with the low byte and high byte of the address of the argument list respectively and that the result is returned in the accumulator.

If the machine code is placed in higher memory than the C code, the library function highmem should be called to limit the memory which the C code can use. In the above example, the statement:

```
highmem (0xc000);
```

... should be included in the main function before the first call to add3.

If the machine code is to be placed below the C code remember to enter an appropriate starting address when linking the C program.

## Zero Page Usage (for Commodore 64)

The zero page locations 20-33, 43-74, and 253-254 decimal are reserved for permanent system use and should not be disturbed. The locations 34-42, 75-96, and 251-252 are used as temporary storage and may also be used as such in your machine language routines. The remainder of the zero page may be used as you wish, but caution should be observed when using locations used by kernel routine or BASIC floating point routines.

In addition, the tape buffer (\$033c-\$03fb) is used by the system and should not be disturbed.

## Memory Usage (for Commodore 128)

### Bank 0

\$0600 - \$06ff	System storage
\$0800 - \$09ff	Ram disk header information
\$1300 - \$13ff	Relocated zero page - used by all programs (including compiled programs) except the shell
\$1400 - \$37ff	Shell program
\$3800 - \$3fff	Character set
\$4000 - \$feff	Ram disk, or tables for linker and compiler, or user defined

### Bank 1

\$0400 - UP	Highmem-1 All programs (including compiled programs) except the shell, reside here
-------------	--

NOTE 1: "highmem" defaults to \$ff00 and may be changed with the "highmem()" library function.

NOTE 2: Zero page should be relocated when calling machine language subroutines from C, and restored when returning.

In addition, the buffer (\$033c-\$03ff) is used by the system and should not be disturbed.

### Coding Efficiently

Due to the way certain features have been implemented in the compiler and the nature of the 65xx/85xx instruction set there are a few non-obvious ways of improving the efficiency of some programs written with POWER C:

Declare variables unsigned rather than int if they never contain negative values. For example, variables which are used only as array indices should always be declared unsigned.

If a function contains many variables, declare those used most frequently first.

## C SHELL, COMMAND INTERPRETER

The first program on the System Disk, and the first one you should load and run, is the "Shell" mini command interpreter. The Shell is a program that supports command line arguments and I/O redirection along with the compiler and other programs designed to work under it. Load and run the program "shell" and you will soon see a dollar sign prompt on the screen waiting for one of the commands listed below.

The Shell lets you define a work disk and a system disk. The defaults for both are device 8 and drive 0, but they may be changed as described below if you are lucky enough to have two disk drives. There are a number of built-in commands which are also listed here. Items in square brackets are optional. Arguments may optionally be enclosed in double quotes, which is necessary if they contain spaces.

(for the Commodore 64)

**bye**

Exit to BASIC.

**l** [pattern]

List the directory of the work disk to the standard output (screen).

**ls** [pattern]

List the directory of the system disk to the standard output (screen).

**rm** [filename]

Remove (scratch) a file on the work disk.

**mv** [file1 file2]

Move (rename) file1 to file2 on the work disk.

**pr** [filename]

List the contents of a file on the work disk to the standard output (screen).

**pr >>** [filename]

Same as above but redirects the output to device 4, (usually the printer).

**disk** [command string]

Send a command string to the work disk device. ie: disk n0:[headername],[id#] would format the disk in work disk device.

**load** command

Load, but don't run, the specified command from the work disk or the system disk, wherever it is.

**work** [device# drive#]

With no arguments entered, displays the current work disk device and drive numbers. It may also be used to change the device and drive numbers.



There are a number of commands provided which are loaded from disk:

**ed** [filename]

**ced** [filename]

Load and run the editor (ed) or syntax checking editor (ced). If a file is specified it is loaded into the main editing buffer.

**cc** [-p] [filename.c]

Compile the C source code in the specified filename. If the -p option is specified the compiler will assume that you are using two disk drive units and will suppress the single drive prompts. The designated work drive will be assumed to hold the source/object disk and the system drive the the compiler disk.

**link** [-s [address]]

Run the linker. If no arguments are given programs are linked in such a way that they will run under the Shell. In this case the program names must end with ".sh". The -s option indicates that programs are to be linked so that they will run independently of the Shell. If no address is specified the programs will be linked at the start of BASIC memory so that they may be LOAded and RUN. If an address is specified in either decimal or hex (hex numbers must be identified by a preceding \$ symbol), the programs will be linked there. Object files will be taken from the designated work disk, and the libraries will be sought from the designated system disk.

Programs written in C may access command line arguments. Also, I/O may be redirected to and from disk files much like UNIX. the command `pr >>` will direct the standard output to device 4 (usually a printer).

The following examples were adapted from The C Programming Language by Kernighan and Ritchie and are provided to illustrate these features:

**sort** [-n]

Sorts the standard input and writes the result to the standard output. The -n option indicates that the input is to be sorted in numerical order. The default is alphabetical order (actually, lexicographic order).

**find** [-x] [-n] string

Finds all occurrences of the specified string in the standard input. If the -x option is specified, only lines NOT containing the string are output. If the -n option is specified output lines will be numbered.

**wfreq**

Counts the number of occurrences of each word in the standard input. A word is defined as a string of characters beginning with a letter and followed by up to 19 letters and digits.

As stated above, programs that work with the Shell mini command interpreter must have file names ending with ".sh". These programs may be invoked by merely typing the name without the ".sh" part along with any arguments and I/O redirection specifiers.

Once a program is loaded it may be run any number of times without re-loading using the built in run command by typing the name without the .sh extension and any arguments and/or redirection specifiers.

## EDITOR (ed.sh) (for the Commodore 64)

From the Shell program prompt (\$), type in `ed filename <RETURN>` or, if you want the syntax checking editor `ced filename <RETURN>` and the chosen editor will be loaded and run. If a unit number is specified in front of the filename, the specified unit will be searched for that filename.

## Edit Mode, Function Keys (for the Commodore 64)

<b>crsr keys</b>	- up, down, left and right
<b>f1, f2</b>	- page down, page up
<b>f3, f4</b>	- search down, search up
<b>f5</b>	- cut text (<SHIFT><RUN/STOP> starts, use cursor keys to set range)
<b>f6</b>	- paste text (f5 deletes, f6 inserts)
<b>f7</b>	- go to end of line
<b>f8</b>	- go to start of line
<b>&lt;clr/home&gt;</b>	- go to bottom of buffer
<b>&lt;shift&gt;&lt;clr/home&gt;</b>	- go to top of buffer
<b>&lt;shift&gt;&lt;RETURN&gt;</b>	- open a line
<b>&lt;shift&gt;&lt;run/stop&gt;</b>	- enter select mode for cut & paste
<b>&lt;run/stop&gt;</b>	- enter command mode

## Command Mode (for the Commodore 64)

In command mode the following commands may be entered:

<b>dir</b>	- list the specified disk directory
<b>disk [command string]</b>	- send the command string to the specified disk drive unit
<b>GEt [filename]</b>	- reads file from disk
<b>PUt [filename]</b>	- writes file to disk
<b>PRint</b>	- dumps contents of current buffer to device 4
<b>GOto buffername</b>	- change current edit buffer
<b>List</b>	- lists buffers currently in use
<b>CHeck</b>	- (ced.sh only) - check syntax
<b>QUIT</b>	- exit editor
<b>CLEAR</b>	- clear current buffer
<b>CLEAR buffername</b>	- clear specified buffer
<b>/searchstring&lt;f3&gt;</b>	- set search string and search down
<b>/searchstring&lt;f4&gt;</b>	- set search string and search up
<b>/dog&lt;f3&gt;</b>	- search for next occurrence of 'dog'
<b>/dog/cat&lt;f3&gt;</b>	- search for next 'dog' and replace it with 'cat'
<b>/dog/cat/&lt;f3&gt;</b>	- search and replace all 'dog's with 'cat's
<b>/dog/&lt;f3&gt;</b>	- delete next occurrence of 'dog'
<b>/dog//&lt;f3&gt;</b>	- delete all occurrences of 'dog'
<b>&lt;RETURN&gt;</b>	- return to edit mode

**bye**

Exit to BASIC.

**dir [unit:][pattern]**

List the directory of the designated disk to the standard output (screen).

**setu [unit device drive]**

Redefines any of the 4 available disk drive units. e.g. setu 1 9 0 would define Unit 1 as device 9, drive 0.

**rm [unit:]pattern**

Remove (scratch) a file on the designated disk drive unit.

**mv [unit:]oldfilename newfilename**

Move (rename) oldfilename to newfilename on the designated disk drive unit.

**pr [unit:][filename]**

List the contents of a file on the work disk to the standard output (screen).

**pr >> [unit:][filename]**

Same as above but redirects the output to device 4, (usually the printer).

**dos [unit:][command string]**

Send a command string to the designated disk drive unit. ie: dos 0:[header],[id] would format the disk in the designated disk unit, drive 0.

**load command**

Load, but don't run, the specified command from the designated disk unit.

**cp sourceunit:pattern destinationunit:[pattern]**

Copies all files matching the pattern from the source unit to the destination unit.

**rdon**

Enables RAMDISK.

**rdoff**

Disables RAMDISK.

**col foreground [background [status line [border]]]**

e.g. col 1 0 5 7 would set the foreground color to white, background to black, status line (in editor only) to green and the border (in 40 column mode only) to yellow.

## Function Keys

(for the Commodore 128)

The Function Keys have been defined as follows:

F1: "dir "	F5: "cc "
F2: "setu "	F6: "link "
F3: "ed "	F7: "pr "
F4: "ced "	F8: "cp "

NOTE: You may change between 40 and 80 column displays by putting the <DISPLAY> in the opposite position, then press the <RUN/STOP> and <RESTORE> keys at the same time. The monitor will, of course, also have to be switched.

## Commands

(for the Commodore 128)

There are a number of commands provided which are loaded from disk:

**ed [unit:][filename]**

**ced [unit:][filename]**

Run the editor (ed) or syntax checker (ced) from the designated unit. If a file is specified it is loaded into the main editing buffer.

**cc [-p] [unit:][filename.c]**

Compile the C source code in the specified filename, on the designated unit. If the -p option is specified the compiler will assume that both the source/object and compiler disks are present and will not tell you to change disks all the time

**link [-s] [unit:][address]**

Run the linker. If no arguments are given programs are linked in such a way that they will run under the Shell. In this case the program names must end with ".sh". The -s option indicates that programs are to be linked so that they will run independantly of the Shell. If no address is specified the programs will be linked at the start of BASIC memory with a "boot" program so that they may be LOAded and RUN. If an address is specified in either decimal or hex (hex numbers must be identified by a preceding \$ symbol), the programs will be linked there.

Programs written in C may access command line arguments. Also, I/O may be redirected to and from disk files much like UNIX. ">>" will direct the standard output to device 4 (usually a printer).

A few programs (adapted from examples in "The C Programming Language" book by Kernighan & Ritchie) are provided to illustrate these features:

**sort [-n]**

Sorts the standard input and writes the result to the standard output. The -n option indicates that the input is to be sorted in numerical order. The default is alphabetical order (actually, lexicographic order).

**find [-x] [-n] string**

Finds all occurrences of the specified string in the standard input. If the -x option is specified, only lines NOT containing the string are output. If the -n option is specified output lines will be numbered.

## wfreq

Counts the number of occurrences of each word in the standard input. A word is defined as a string of characters beginning with a letter and followed by up to 19 letters and digits.

As stated above, programs that work with the Shell mini command interpreter must have file names ending with ".sh". These programs may be invoked by merely typing the name without the ".sh" part along with any arguments and I/O redirection specifiers.

Once a program is loaded it may be run any number of times without re-loading by using the built in run command.

## EDITOR (ed.sh) (for the Commodore 64)

At the Shell program prompt (\$), type in ed filename <RETURN> or, if you want the syntax checking editor ced filename <RETURN> and the chosen editor will be loaded and run. If a unit number is specified in front of the filename, the specified unit will be searched for that filename.

## Edit Mode, Function Keys (for the Commodore 128)

crsr keys	- up, down, left and right
f1, f2	- page down, page up
f3, f4	- search down, search up
f5	- cut text (<SHIFT><RUN/STOP> starts, use cursor keys to set range
f6	- paste text (f5 deletes, f6 inserts)
f7	- go to end of line
f8	- go to start of line
<clr/home>	- go to bottom of buffer
<shift><clr/home>	- go to top of buffer
<shift><RETURN>	- open a line
<run/stop>	- enter select mode for cut
<esc>	- enter command mode

## Command Mode (for the Commodore 128)

In command mode the following commands may be entered:

dir [unit:]	- list the specified disk directory
dos [unit:][command string]	- send the command string to the specified disk drive unit
GEt [unit:][filename]	- reads file from disk
PUt [unit:][filename]	- writes file to disk
PRint	- dumps contents of current buffer to device 4
GOto buffername	- change current edit buffer
List	- lists buffers currently in use
CHeck	- (ced.sh only) - check syntax
QUIT	- exit editor

<b>CLEAR</b>	- clear current buffer
<b>CLEAR buffername</b>	- clear specified buffer
<b>/searchstring&lt;f3&gt;</b>	- set search string and search down
<b>/searchstring&lt;f4&gt;</b>	- set search string and search up
<b>/dog&lt;f3&gt;</b>	- search for next occurrence of 'dog'
<b>/dog/cat&lt;f3&gt;</b>	- search for next 'dog' and replace it with 'cat'
<b>/dog/cat/&lt;f3&gt;</b>	- search and replace all 'dog's with 'cat's (needs <f2> or <f3> to start search up, or down)
<b>/dog/&lt;f3&gt;</b>	- delete next occurrence of 'dog'
<b>/dog//&lt;f3&gt;</b>	- delete all occurrences of 'dog'
<b>&lt;RETURN&gt;</b>	- return to edit mode

### Special Symbols used in C

The special C characters used in creating C language source code may be obtained as follows:

<b>curly brackets ({ and }):</b>	<b>&lt;shift&gt; + and &lt;shift&gt; -</b>
<b>back slash (\):</b>	<b>&lt;english pound sign&gt;</b>
<b>tilde (~):</b>	<b>&lt;logo&gt; p</b>
<b>underscore (_):</b>	<b>&lt;logo&gt; @</b>
<b>vertical bar ( ):</b>	<b>&lt;logo&gt; *</b>

**NOTE 1:** Use the <logo> key in the above examples the same as you would a shift key ... hold it down while pressing the next character.

**NOTE 2:** To CUT and PASTE in EDIT mode, first enter SELECT mode by pressing the <SHIFT><RUN/STOP> key. Set desired range with the <CURSOR> up and down keys and then the <RETURN> key to accept the chosen (highlighted) range. Now press <F5> to CUT (delete) or <F6> to PASTE (insert).

## COMPILER (cc.sh)

From the Shell program prompt (\$) type `cc [-p] filename.c` <RETURN>, and the compiler will be loaded from the specified unit and run with the specified filename.c. If you type in the `-p` prior to the filename.c, the compiler will assume you are using two disk drives. The filename should always end with ".c". When finished, the compiler will leave an object file on disk with the same name as the source except the extension ".c" will be replaced with ".o".

## LINKER (link.sh)

From the Shell program prompt (\$), type in `link [-s [address]]` <RETURN> to load and run the linker. If no arguments are given, programs are automatically linked in such a way that they will run under the control of the Shell program. In this case all program names must end in ".sh". The `-s` option indicates that you want the programs to run independently of the Shell program. If no address is specified, the programs will be linked beginning at the start of basic address so that they may be LOAded and RUN. If an address is specified, in either hex or decimal (identify hex numbers with a preceding \$ symbol), the programs will be linked from that address on. Object files will be taken from the work disk, and the library files will be looked for on the system disk.

You will then be greeted by a ">" linker prompt. Type in the names of the object files (i.e. ".o" files) that you wish to link together. When you have finished this, insert the library disk and press <↑> <RETURN>. This will automatically link in the necessary functions from the standard library and system library.

When the above has been completed, press <RETURN>. If you get an "unresolved external reference" message then you forgot to link in something. No problem ... just link in that something now.

Once everything is linked you are asked for the name of the program to be saved on disk. If you did not specify a starting address then this program may simply be LOAded and RUN. Otherwise you must LOAD with the ".l" option and SYS to the starting address.

You may exit the linker at any time by entering `x` <RETURN> at any linker prompt (>).

NOTE 1: The linker will only accept files whose names end with ".o", ".obj" or ".l".

NOTE 2: Typically, the linking commands would appear thusly:

```
>filename1.o      (link first object file)
>filename2.o      (link second object file, if you have one)
>↑                (link libraries) (<UP ARROW> not <CURSOR UP>)
><RETURN>          (all finished)
```

## LIBRARY INTRODUCTION

The following pages describe the functions available in the library provided with the compiler. Most of the functions are in the library files `syslib.l` and `stdlib.l`; the remainder are in the file `math.l`. See the section on the linker for information on how to link these libraries into your C program.

Each page is divided into three or four sections:

### NAME

The names of the functions described on the page are listed here with a very brief description.

### SYNOPSIS

The purpose of this section is to show the number, order, and types of arguments each function takes, as well as the type the function returns. If no types are specified, `int` is to be assumed. For example:

```
float atof(fptr)
char *fptr;
```

... means that the function `atof` takes one argument of type pointer to `char`, and returns a `float`.

```
abs(i)
```

This means `abs` takes one argument of type `int`, and returns an `int`.

If a function returns a type other than `int`, it should be declared in the file which calls the function. For example, before any calls to `malloc` the following declaration should appear:

```
char *malloc();
```

Header files which are useful when using certain functions are also listed in this section. If the line

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

... is listed, for example, the same line should be put into the source file which calls the function(s) described on that page.



## DESCRIPTION

This section describes what the functions do, what the arguments are, and what the functions return (if anything). If a function checks the arguments for validity or does any other kind of error checking it will be described here. Otherwise it should be assumed that no error checking is done.

## EXAMPLES

For some of the more complicated functions examples illustrating their use are listed. In the examples the symbols `/* ... */` mean the preceding and following code may be separated by some arbitrary amount of code.

# FUNCTION INDEX

<u>For</u>	<u>See</u>	<u>Page#</u>	<u>For</u>	<u>See</u>	
<u>Page#</u>					
abort	exit	20	isalnum	isalpha	28
abs		19	isalpha		28
acos	sin	42	isascii	isalpha	28
asin	sin	42	isctrl	isalpha	28
atan	sin	42	isdigit	isalpha	28
atan2	sin	42	islower	isalpha	28
atof	atoi	19	isprint	isalpha	28
atoi		19	ispunct	isalpha	28
bcmp		20	isspace	isalpha	28
bcopy	bcmp	20	isupper	isalpha	28
bzero	bcmp	20	ldexp	frexp	25
cabs	hypot	27	log	exp	21
calloc	malloc	29	log10	exp	21
ceil	floor	22	longjmp	setjmp	41
close	open	30	malloc		29
closedir	opendir	31	modf	floor	22
cos	sin	42	open		30
cosh	sinh	43	open2( )		32
exit		20	opendir		31
exp		21	peek( )		32
fabs	abs	19	poke( )		33
fclose	fopen	23	pow	exp	21
feof	ferror	22	printf		33
ferror		22	putc		35
ffs	bcmp	20	putchar	putc	35
fgetc	getc	25	puts		35
fgets	gets	26	putw	putc	35
floor		22	qsort		36
fopen		23	random		37
fprintf	printf	33	readdir	opendir	31
fputc	putc	35	realloc	malloc	29
fputs	puts	35	rewinddir	opendir	31
fread		24	rindex	strcat	44
free	malloc	29	scanf		38
freopen	fopen	23	setjmp		41
frexp		25	sin		42
fscanf	scanf	38	sinh		43
fwrite	fread	24	sprintf	printf	33
getc		25	sqrt	exp	21
getchar	getc	25	srandom	random	37
gets		26	sscanf	scanf	38
getw	getc	25	strcat		44
highmem		27	strcmp	strcat	44
hypot		27	strcpy	strcat	44
index	strcat	44	strlen	strcat	44
			strncat	strcat	44
			strncpy	strcat	44
			strncpy	strcat	44
			sys		46
			tan	sin	42
			tanh	sinh	43

## NAME

abs, fabs - absolute value

## SYNOPSIS

```
abs(i)
int i;

float fabs(f)
float f;
```

## DESCRIPTION

Abs and fabs return the absolute value of their arguments.

## NAME

atoi, atof - convert strings to numbers

## SYNOPSIS

```
atoi(iptr)
char *iptr;

float atof(fptr)
char *fptr;
```

## DESCRIPTION

Atoi converts the string pointed to by its argument into an integer.

Atof converts the string pointed to by its argument into a float quantity.

Both functions ignore leading spaces.

## EXAMPLES

```
char *s;
int i;
float pi, atof();

s = " 123";
i = atoi(s);

s = "3.14159";
pi = atof(s);
```

## NAME

bcmp, bcopy, bzero, ffs - bit and byte string functions

## SYNOPSIS

```
bcmp(p1, p2, len)
char *p1, *p2;
```

```
bcopy(p1, p2, len)
char *p1, *p2;
```

```
bzero(p, len)
char *p;
```

```
ffs(i)
```

## DESCRIPTION

Bcmp compares len bytes of the strings p1 and p2 and returns zero if they are same, non-zero otherwise.

Bcopy copies len bytes from string p1 to string p2.

Bzero fills string p with len zeros.

Ffs returns the position of the first set bit in its argument. Bits are numbered starting at one. If the argument is zero ffs returns -1.

## NAME

exit, abort - terminate execution

## SYNOPSIS

```
exit()
```

```
abort()
```

## DESCRIPTION

Exit and abort end program execution. All files opened by fopen are closed.

## NAME

exp, log, log10, pow, sqrt - assorted math functions

## SYNOPSIS

```
#include <math.h>

float exp(x)
float x;

float log(x)
float x;

float log10(x)
float x;

float pow(x, y)
float x, y;

float sqrt(x)
float x;
```

## DESCRIPTION

Exp returns  $e^{**}x$ .

Log returns the natural logarithm of x.

Log10 returns the base 10 logarithm of x.

Pow returns  $x^{**}y$ .

Sqrt returns the square root of x.

## NAME

ferror, feof - check for error or end of file

## SYNOPSIS

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

```
ferror()
```

```
feof(stream)
```

```
FILE stream;
```

## DESCRIPTION

Ferror returns non-zero if an error occurred during the last disk operation, zero otherwise.

Feof returns non-zero if the specified stream has reached end of file, zero otherwise.

## NAME

floor, ceil, modf - get integer part of float

## SYNOPSIS

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

```
float floor(x)
```

```
float x;
```

```
float ceil(x)
```

```
float x;
```

```
float modf(x, ptr)
```

```
float x, *ptr;
```

## DESCRIPTION

Floor returns the greatest integer not greater than x.

Ceil returns the least integer not less than x.

Modf returns the positive fractional part of x and stores the integer part indirectly through ptr.

## NAME

fopen, freopen, fclose - open disk file for I/O

## SYNOPSIS

```
#include <stdio.h>

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

FILE freopen(filename, mode, stream)
char *filename, *mode;
FILE stream;

fclose(stream)
FILE stream;
```

## DESCRIPTION

Fopen opens a disk file for reading or writing. The string filename contains the name of the file and the first character of the string mode specifies read or write ('r' or 'w'). The default file type is sequential, but program file types may be selected. Fopen returns a file number (hereafter referred to as a stream) which may be used in later I/O, or it returns zero if the file cannot be opened.

Freopen opens a file much the same as fopen does. The file stream is first closed, then if the open is successful the old stream is assigned to the new file. This is useful to assign the constant streams stdin and stdout to disk files.

Error should be checked after every fopen. Fclose closes the specified file.

## EXAMPLES

```
#include <stdio.h>

FILE f;

/* open sequential file for reading */
f = fopen("abc", "r");

/* open and replace program file */
f = fopen("@0:xyz,p", "w");

/* assign standard output to a disk file */
f = freopen("outfile", "w", stdout);
```

## NAME

fread, fwrite - array input/output

## SYNOPSIS

```
#include <stdio.h>

fread(ptr, elsize, nelem, stream)
char *ptr;
FILE stream;

fwrite(ptr, elsize, nelem, stream)
char *ptr;
FILE stream;
```

## DESCRIPTION

Fread/fwrite reads/writes an array containing nelem elements each of size elsize bytes beginning at ptr from/to the specified stream.

Fread returns zero upon end of file.

## EXAMPLE

```
#include <stdio.h>

#define N 500

float x[N];
FILE f;

f = fopen("datafile", "r");
fread(x, sizeof(float), N, f);
```



## NAME

frexp, ldexp - split float into mantissa and exponent

## SYNOPSIS

```
float frexp(value, ptr)
float value;
int *ptr;

float ldexp(value, exp)
float value;
```

## DESCRIPTION

Frexp splits value into a mantissa m of magnitude less than 1 (which is returned) and an exponent exp (which is stored indirectly through ptr) such that  $value = m * 2^{exp}$ .

Ldexp returns  $value * 2^{exp}$ .

## NAME

getc, getchar, fgetc, getw - input character or integer

## SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE stream;

int getchar()

int fgetc(stream)
FILE stream;

int getw(stream)
FILE stream;
```

## DESCRIPTION

Getc and fgetc read a character from the specified stream. Getchar reads a character from the standard input. Getw reads an integer (two bytes) from the specified stream. All of these functions return EOF upon end of file. However, since EOF is a valid integer, feof should be used to check for end of file after getw.

## NAME

gets, fgets - input a string

## SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE stream;
```

## DESCRIPTION

Gets inputs a string from the standard input. It reads characters into s until a newline character is encountered. The newline is replaced with a zero.

Fgets inputs a string from the specified stream. It reads n-1 characters or until a newline is encountered, whichever comes first. The newline is not replaced, but a zero is placed after the last character read.

Both functions return s upon normal completion, or NULL upon end of file.

## NAME

highmem - memory configuration

## SYNOPSIS

```
highmem(address)
unsigned address;
```

## DESCRIPTION

Highmem sets the highest address that a C program can use. The run time stack will not go past this address, and the memory allocation functions (malloc, calloc, realloc) will not allocate memory higher than this address. The value of the argument must be one greater than the desired address. If highmem is not called, address defaults to 0xd000, which means the highest address which can be used is 0xcfff.

## EXAMPLE

```
/* let program use all available memory */
highmem(0xffff);
```

## NAME

hypot, cabs - calculate hypotenuse

## SYNOPSIS

```
#include <math.h>

float hypot(x, y)
float x, y;

float cabs(c)
struct (* float x, y; *) *c;
```

## DESCRIPTION

Hypot and cabs return  $\sqrt{x^2 + y^2}$ .

## NAME

isalpha, ... - classify characters

## SYNOPSIS

isalpha(c)

## DESCRIPTION

The following functions return non-zero integers if the stated condition is true, zero otherwise.

isalpha	c is a letter
isupper	c is an upper case letter
islower	c is a lower case letter
isdigit	c is a digit
isalnum	c is a letter or digit
isspace	c is a space or newline
ispunct	c is a punctuation character
isprint	c is a printable character
isctrl	c is a control character
isascii	c has value less than 0200

## NAME

malloc, calloc, realloc, free - memory allocation

## SYNOPSIS

```
char *malloc(size)
unsigned size;

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

char *realloc(ptr, size)
char *ptr;
unsigned size;

free(ptr)
char *ptr;
```

## DESCRIPTION

Malloc returns a pointer to a block of memory containing at least size bytes.

Calloc returns a pointer to a block of zero-filled memory containing at least nelem \* elsize bytes.

Realloc copies the block pointed to by ptr into a new block containing at least size bytes. Ptr must point to a block allocated by malloc, calloc, or realloc.

Free releases the block pointed to by ptr into the free memory list.

Malloc, calloc, and realloc all return the null pointer (0) if there is not enough free memory to satisfy the request.

## EXAMPLE

```
/* Run time array allocation */

#define NELEM 100

char *malloc;
int *t;

t = (int *) malloc(NELEM * sizeof(int));

/* ... */

free(t); /* done with array */
```

## NAME

open, close - BASIC style open

## SYNOPSIS

```
open(fileno, device, secaddr, name)
char *name;

close(fileno);
```

## DESCRIPTION

The arguments of open correspond exactly to the file number, device number, secondary address, and file name arguments of the BASIC OPEN command. Consult a Commodore 64 manual for the meanings of the arguments. Similarly, close corresponds to the BASIC CLOSE command. Open returns zero if the file can't be opened, non-zero otherwise. As with fopen, ferror should be checked after opening a write file.

The file number argument may be used any place a stream (i.e. a value returned by fopen) is used (see example). File numbers 1 through 4 are reserved for system use. If open and fopen are to be used at the same time, file numbers passed to open should be limited to the range 5 through 9.

## EXAMPLE

```
/* display disk file on screen */

#include <stdio.h>

char c;

open(5, 8, 5, "filename,s,r");

for ((c = getc(5)) != EOF)
    putchar(c);

close(5);
```

## NAME

opendir, readdir, rewinddir, closedir - directory functions

## SYNOPSIS

```
#include <dir.h>

opendir(unit:)

struct direct *readdir()

rewinddir()

closedir()
```

## DESCRIPTION

Opendir opens a disk directory for reading. The unit from which the directory is to be read may be specified. If the directory can't be opened NULL is returned. NOTE: The directory functions do not apply to the RAMDISK.

Readdir reads the next directory entry and returns a pointer to it. If there are no more entries NULL is returned. See the header file dir.h and the VIC-1541 User's Manual page 56 for the format of a directory entry.

Rewinddir causes readdir to read the first entry upon the next call.

Closedir closes the directory for.

## EXAMPLE

```
/* Display contents of disk directory */

#include <dir.h>
#include <stdio.h>

struct direct *dp;

opendir(0:);
int unit;
for (dp = readdir(); dp != NULL; dp = readdir())
    puts (dp->name);
closedir();
```

## NAME (for Commodore 128)

open2() - BASIC style open to unit number

## SYNOPSIS

```
open2(name, filename, unit, channel)
char *name;
int filenum;
char unit;
int channel;
```

## DESCRIPTION

open2() is the same as open(), except that the disk drive is specified by a unit number rather than a device number. If "filename" is not empty, then the unit number is taken from there, otherwise it is taken from "unit".

## EXAMPLE

```
open2("3:filename", 5, don't care, 5);
open2("", 15, '0', 15)
```

## NAME (for Commodore 128)

peek() - BASIC style peek() command

## SYNOPSIS

```
peek( bank, address);
unsigned bank, address;

peek() returns
unsigned bank, address;
```

## DESCRIPTION

peek() returns the contents of the memory location in bank "bank" at address "address".



**NAME** (for Commodore 128)

`poke()` - BASIC style poke command

## SYNOPSIS

```
poke(bank, address, value)
unsigned bank, address, value;
```

## DESCRIPTION

`poke()` puts the byte "value" into the memory location specified by "bank" and "address".

## NAME

`printf`, `fprintf`, `sprintf` - formatted output

## SYNOPSIS

```
#include <stdio.h>

printf(control [, arg] ...)
char *control;

fprintf(stream, control [, arg] ...)
FILE stream;
char *control;

sprintf(s, control [, arg] ...)
char *s, *control;
```

## DESCRIPTION

These functions output optional lists of arguments according to a format specified in the null terminated control string. `Printf` sends output to the standard output. `Fprintf` sends output to the specified stream. `Sprintf` places output in the string `s`. `Sprintf` also places a null character in `s` after the last output character.

The control string may contain ordinary characters, which are output, and conversion specifiers, which specify how an argument is to be formatted. Each conversion specifier begins with a percent character (%) and is followed by:

An optional dash (-) which indicates left adjustment of the argument in the output field. Right adjustment is the default.

An optional number indicating the minimum field width. A converted argument will not be truncated even if it won't fit in the specified field. If the first digit of the field width is zero, the field will be padded with zeros; otherwise it will be padded with spaces. The maximum field width is 64 characters.

An optional period (.) followed by a number indicating the precision for a float or string argument. For floats the precision indicates the number of digits to be printed after the decimal point (default is six). If the precision is explicitly zero no decimal point is printed. For strings the precision indicates the maximum number of characters from the string to be printed (default is the whole string).

A letter indicating the type of conversion to be performed. The following letters are recognized:

d - an integer argument is printed as a possibly signed decimal number

u - an integer argument is printed as an unsigned decimal number

o - an integer argument is printed as an octal number

x - an integer argument is printed as a hexadecimal number

f - a float argument is printed

s - a character pointer argument is assumed to point to a null terminated string which is printed

c - an integer argument is assumed to be a character and is printed as such

For each conversion specifier a corresponding argument of an appropriate type must be provided.

To output a percent character use %%.

A star (\*) may be used in place of the field width or precision. The value will be taken from an integer argument.

## EXAMPLES

```
printf("%d %f", 123, 3.14);
```

```
/* output: 123 3.140000 */
```

```
printf("%05x", 0x2a3);
```

```
/* output: 002a3 */
```

```
printf ("abc%-.*fxyz", 9, 2, 12.3456);
```

```
/* output: abc12.34 xyz */
```

## NAME

putc, putchar, fputc, putw - output a character or integer

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

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

```
putchar(c)
```

```
fputc(c, stream)
FILE stream;
```

```
putw(i, stream)
FILE stream;
```

## DESCRIPTION

Putc and fputc write the character *c* to the specified stream.

Putchar writes the character *c* to the standard output.

Puts writes the integer *i* (two bytes) to the specified stream.

## NAME

puts, fputs - output a string

## SYNOPSIS

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

```
puts(s)
    char *s;
```

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

## DESCRIPTION

Puts writes the null terminated string *s* to the standard output. Puts also writes a newline character after the string.

Fputs writes the null terminated string *s* to the specified stream. Fputs does not write an additional newline character.

## NAME

qsort - general purpose sort

## SYNOPSIS

```
qsort(base, nel, elsize, comp)
char *base;
int (*comp());
```

## DESCRIPTION

Qsort sorts an array beginning at base containing nel elements each of size elsize. Comp points to a function which compares elements. The function must take two pointers to elements and return an integer less than, equal to, or greater than zero as the first element is less than, equal to, or greater than the second.

## EXAMPLE

```
/* Sort array of floats */

#define NELEM 100

float t[NELEM];
int fcomp();

qsort(t, NELEM, sizeof(float), fcomp);

/* ... */

fcomp(p1, p2)
float *p1, *p2;
{
    if (*p1 < *p2)
        return (-1);
    else if (*p1 == *p2)
        return (0);
    else
        return (1);
}
```

## NAME

random, srandom - random number generator

## SYNOPSIS

```
random()
```

```
srandom(seed);
```

## DESCRIPTION

Random returns a pseudo-random integer.

Srandom sets the state of the random number generator. If srandom is called twice with the same seed the same sequence of random integers will be generated.

## NAME

scanf, fscanf, sscanf - formatted input

## SYNOPSIS

```
#include <stdio.h>

scanf(control [, arg] ...)
char *control;

fscanf(stream, control [, arg] ...)
FILE stream;
char *control;

sscanf(s, control [, arg] ...)
char *s, *control;
```

## DESCRIPTION

These functions read sequences of characters, perform conversions specified by the control string on them, and store the converted values indirectly through pointer arguments. Scanf reads from the standard input, fscanf reads from the specified stream, and sscanf reads from the string s.

The control string may contain blanks and newlines, which may match optional blanks and newlines from the input, other ordinary characters, which must match corresponding characters from the input, and conversion specifiers. Each conversion specifier begins with a percent character (%) and is followed by:

An optional star (\*) which suppresses assignment of the converted value.

An optional number which specifies the maximum field width. Characters are read up to the first unrecognized character for the type of conversion being performed or until the number of characters read equals the field width, whichever comes first. If no field width is specified characters are read up to the first unrecognized character.

...(MORE)

## **scanf - continued**

A letter indicating the type of conversion to be performed on the field. The following letters are recognized:

d- the field is expected to contain a possibly signed decimal number which is converted into an integer

x- the field is expected to contain a hexadecimal number which is converted into an integer

o- the field is expected to contain an octal number which is converted into an integer

f- the field is expected to contain a possibly signed decimal number with an optional decimal point and exponent which is converted into a float

s- no conversion is performed - the field is copied into a string argument with a null character appended

c- the field contains a single character which is copied into a character argument

For each conversion specifier (except for those which suppress assignment) there must be a corresponding argument which is a pointer to an appropriate type. For example, d conversion requires that there be a pointer to an int or an unsigned.

To match a percent character from the input use %%.

These functions return EOF upon end of file; otherwise they returned the number of conversions successfully performed. This number may be less than the number of conversion specifiers if, for example, characters in the control string do not match corresponding characters from the input.

To input strings with embedded spaces use gets or fgets.

...(MORE)

## scanf - continued

### EXAMPLES

```
#include <stdio.h>

int i;
float f;
char s[50], c;

scanf("%d %f", &i, &f);

/*
input: 123 456
result: i = 123, f = 456.0
*/

scanf("%3d %5f", &i, &f);

/*
input: 436504.3683
result: i = 436, f = 504.3
*/

scanf("%11s Spain %c", s, &c);

/*
input: The rain in Spain falls mainly ...
result: s = "The rain in", c = 'f'
*/
```



## NAME

setjmp, longjmp - long range goto

## SYNOPSIS

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

```
setjmp(env)  
jmp buf env;
```

```
longjmp(env, val)  
jmp buf env;
```

## DESCRIPTION

Setjmp stores its stack environment in env and returns zero.

Longjmp restores the stack environment saved by setjmp and returns in such a way that it appears that the original call to setjmp has returned with the value val.

The calls to setjmp and longjmp may be in different functions, but the function containing the setjmp call must not have returned before the call to longjmp.

## EXAMPLE

```
#include <setjmp.h>  
  
int errno, error;  
jmp buf env;  
  
errno = setjmp(env);  
if (errno != 0) {  
    printf ("error #%d", errno);  
    exit();  
}  
  
/* ... */  
  
if (error)  
    longjmp(env, 1);
```

## NAME

sin, cos, tan, asin, acos, atan, atan2 - trig functions

## SYNOPSIS

```
#include <math.h>

float sin(x)
float x;

float cos(x)
float x;

float tan(x)
float x;

float asin(x)
float x;

float acos(x)
float x;

float atan(x)
float x;

float atan2(x, y)
float x, y;
```

## DESCRIPTION

Sin, cos, and tan return the sine, cosine, and tangent of x respectively. X is measured in radians.

Asin, acos, and atan return the arcsine, arccos, and arctangent of x respectively.

Atan2 returns the arctangent of x/y.

## NAME

sinh, cosh, tanh - hyperbolic functions

## SYNOPSIS

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

```
float sinh(x)  
      float x;
```

```
float cosh(x)  
      float x;
```

```
float tanh(x)  
      float x;
```

## DESCRIPTION

Sinh, cosh, and tanh return the hyperbolic sine, cosine, and tangent of x respectively.

## NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex -  
string functions

## SYNOPSIS

```
#include <strings.h>

char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s2, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

## DESCRIPTION

All of the following functions operate on character strings terminated with zero.

Strcat and strncat concatenate the strings s1 and s2 and leave the result in s1. Strncat copies at most n characters from s2. Both functions return s1.

(MORE)

**strcat** - continued

**Strcmp** and **strncmp** compare the strings **s1** and **s2** and return an integer less than, equal to, or greater than zero as **s1** is lexically less than, equal to, or greater than **s2**. **Strncmp** compares at most **n** characters.

**Strcpy** and **strncpy** copy **s2** into **s1**. **Strncpy** copies at most **n** characters. Both functions return **s1**.

**Strlen** returns the number of non-zero characters in the string **s**.

**Index/rindex** returns a pointer to the leftmost/rightmost occurrence of the character **c** in the string **s**. If the character is not in the string a null pointer (**0**) is returned.

## NAME

sys - call a machine language subroutine

## SYNOPSIS

```
sys(bank, address, aptr, xptr, yptr)
int bank;
unsigned address;
char *aptr, *xptr, *yptr;
```

## DESCRIPTION

Sys loads the accumulator, x, and y registers of the 6510 processor with the values pointed to by a, x, and y respectively then jumps to the subroutine located at the specified address. Upon completion of the subroutine the (possibly) new values contained in the registers are stored indirectly through the pointer arguments. Sys returns zero if the carry flag is clear after the subroutine call; otherwise it returns one.

This function allows the programmer to combine assembler and C code in one program without having to use a special assembler. Another use of sys is to access kernal routines not otherwise supported by the standard library.

The bank in which the machine code resides must be specified by a number from 0 to 15 (SEE Basic "BANK" command).

## EXAMPLE

```
char *s, x, y;

s = "this string will be written to the screen";

while (*s++)
    sys(15, 0xffd2, s, &x, &y);

/* ffd2 is the kernal routine for printing a character */
```

## POWER C TUTORIAL

### System Configuration (for Commodore 64)

POWER C has been designed to work with a Commodore 64 computer equipped with up to four Commodore 1571 or Commodore 1541 disk drive units. A Commodore 1525, MPS 801, MPS 803 or equivalent printer is optional. Please be aware that some printers do not have the special C language characters in their built-in fonts, and so cannot print them. POWER C will output 40 column video to a Commodore 1701 or 1702 color monitor. We cannot guarantee that any other system configuration will function properly.

### (for Commodore 128)

POWER C 128 has been designed to work with a Commodore 128 computer equipped with up to four Commodore 1571 or Commodore 1541 disk drive units. A Commodore 1525, MPS 801, MPS 803 or equivalent printer is optional. POWER C 128 will output either 128/40 column or 128/80 column video. We cannot guarantee that any other system configuration will function properly.

### Startup (for Commodore 64)

You will need a properly formatted blank diskette to use as a work disk, as well as your POWER C System and Library disks.

POWER C was designed to be compatible with the Spinnaker Turbo Load and Save cartridge. Turbo Load and Save, used in conjunction with a Commodore 64 computer equipped with a Commodore 1541 Disk Drive unit, speeds all POWER C loading and saving operations by up to 500%.

Load POWER C by typing:

```
load"*",8 <RETURN>
```

... then, when the READY prompt returns, type:

```
run <RETURN>
```

... when the shell program is run the screen will turn grey and a white \$ sign with a flashing cursor next to it will appear at the upper left corner of the screen.

### (for Commodore 128)

You will need a properly formatted blank diskette to use as a work disk, as well as your POWER C 128 System and Library disks.

The POWER C 128 System Disk may be auto-booted. If the POWER C 128 System Disk is in the disk drive unit when the computer is turned on or is reset, the Shell program will load and run automatically.

Otherwise:

```
dload"shell <RETURN>
```

... will do the loading job nicely, and:

```
run <RETURN>
```

... will RUN it.

## Shell

The Shell program creates the operating environment that all the rest of the POWER C compiler package programs work within. When the Shell program is active, it's unique \$ sign prompt will appear on the screen. We will compile, trim, and link a small source program called test.c contained on the System disk. All source code programs are identified by a .c extension or ending to their names.

When the \$ prompt appears on your screen, and with the POWER C System disk in your disk drive unit, call the compiler program and tell it what source file you wish to compile by typing:

```
cc test.c <RETURN>
```

## Compiler

When the compiler asks you to insert your source disk, just press <RETURN> because the source program test.c is located on the system disk already in your disk drive unit. Normally, your source code would be located on a work disk which would be inserted at this time. When the compiler is finished loading test.c, it will prompt you to insert your compiler disk. Again, just press <RETURN> because your compiler disk is already in the drive. When the compilation of the test.c source file is complete, POWER C will ask you to insert your object disk.

Now is the time to use the formatted, blank disk that you have ready. Remove the system disk from the drive, insert the blank work disk and close the drive door. Now press <RETURN> and the compiler will write an object file called test.o to the work disk. Note that compiled object code filenames always end with the extension .o (lower case letter o, not numeral 0). Now that the compiler has completed it's job, it will return you to the Shell environment and the unique Shell prompt \$.



## Optimizing

The next step, to optimize the object code and make it as small as possible, is optional. Normally we would not bother to optimize such a small object file, but we will trim the object code here just for practice. Remove the work disk with the test.o file on it and place the System disk into the drive. Now call the Trim program from the disk by typing after the Shell \$ prompt:

```
trim <RETURN>
```

When Trim is loaded, the program will remind you of the syntax it expects you to use. Remove the System disk and replace it with the work disk that contains the test.o program. Type:

```
trim test.o <RETURN>
```

Trim will proceed to optimize the test.o object file, and when it is through, will show the results of it's efforts on the screen. Wait for the trimmed file to be written back to your work disk. Now your test.o file is ready for linking, and the program is back in the shell environment and is displaying the \$ prompt.

## Linking

From the \$ prompt in the shell program, place the system disk back in the disk drive unit and call the Linker program by typing:

```
link <RETURN>
```

When the Linker program is loaded, the screen will display the Linker's unique prompt sign, >. Any time this > sign appears on the screen, you will know that the Linker is ready for your commands. Place your work disk that contains the test.o file in the disk drive unit. Tell the Linker that you want it to link in the object code file named test.o by typing:

```
test.o <RETURN>
```

The Linker will now load test.o from the work disk and then, once again display the > Linker prompt.

If the test.c source file had consisted of two files called test.c and test2.c both would have to have been compiled separately and would have produced two object files called test.o and test2.o. This would be the time to link in the second file test2.o. However, since our test program is a simple one, that is not the case and we will now link in the appropriate library functions.

## Linking Library Functions

At the next > linker prompt, remove your work disk from the disk drive unit and replace it with your C Power Library diskette located on the reverse side of your system diskette. Link the appropriate function libraries automatically by typing:

<↑> <RETURN>

(Arrow Up, not Cursor Up)

When the > Linker program prompt returns to the screen, indicate that you are through linking by pressing <RETURN>. The Linker will now request a filename for the compiled C program. Place your work disk in drive 0 and type in:

**test.sh** <RETURN>

An .sh extension or ending to the filename is necessary because it is required when the compiled C program is to run from within the shell environment.

## Test.sh

Once the test.sh program has been written to the work disk by the linker, the program will return to the Shell environment and the \$ prompt will return to the screen. When this occurs, you may test your newly compiled C language program by typing:

**test** <RETURN>

If all has gone well, Pro-Line Software Ltd.'s corporate name and address will appear on the screen.

The test program could have been compiled to run directly from BASIC if, back at the step where you called the Link program, you had typed in:

**link -s** <RETURN>

... informing the linker that it should not compile the test program to run under the shell, but from the start of BASIC address. The compilation procedure is exactly the same up until the end of the trim operation.

You may wish to re-link the test.o files to run from BASIC, in which case use a slightly different name for the compiled program like "test1" to distinguish it from the "test.sh" program you have already compiled. To run this test1 program from BASIC, reset the computer back to BASIC from the Shell environment by typing:

**bye** <RETURN>

... then, when the computer resets, type <SHIFT><LOGO> to put the C-64 in upper/lower case mode, then type:

**load"test1",8** <RETURN>

When loading is complete, type <RUN> <RETURN> and the results should be the same as when you ran it from the shell environment.

(for the Commodore 128)

... then, when the computer resets, type <SHIFT><LOGO> to put the C128 in upper/lower case mode, then type:

**dload"test <RETURN>**

When loading is complete, type <RUN> and the results should be the same as when you ran it from the shell environment.

## **Ramdisk**

(for the Commodore 128)

C Power 128 gives you the option of using the Commodore 128's extra memory as either an extra large workspace or as a pair of small software ramdisks. When C Power 128 is freshly loaded, the internal ramdisks are not enabled. From the Shell environment, use the commands **rdon** to enable the ramdisks, and **rdoff** to shut them off.

There are two built-in, internal, software ramdisks available for use. They are set up as device 7, drive 0, and device 7, drive 1. Their default unit numbers are Unit 2 and Unit 3. Each ramdisk has 191 blocks free when first enabled, and will retain any files saved to them until the computer is powered down. Obviously, you must copy any important files saved on the ramdisks to a regular floppy diskette before you turn off your computer.

Why bother with ramdisks, then? Because they are lightning fast compared to regular disk drives, and if you are concerned with the length of time it takes to load the compiler and translator programs, try copying them into ramdisk unit 2 and accessing them from there. You can use a ramdisk unit as a work disk if you wish, and enjoy the freedom from disk swapping that a second disk drive brings. Just don't forget to copy your work down to a floppy before you shut down.

In an external dual-drive system, C Power 128 assumes that unit 0 will contain the work disk, and unit 1 will contain the system disk.

## Example

(for the Commodore 128)

From the Shell environment try this:

Insert the System Disk in your regular disk drive

```
$ rdon                                /* turn on ramdisks
$ cp 0:test.c 2                       /* copy test.c from unit 0 to unit 2
$ cp 0:stdio.h 2                     /* copy stdio.h from unit 0 to unit 2
$ cc 2:test.c                         /* compile test.c found on unit 2
$ trim 2:test.o                      /* trim test.o found on unit 2
$ link                               /* load linker
> 2:test.o                          /* link test.o file
```

Insert Library Disk

```
> <↑>                                /* link libraries
> <RETURN>                          /* to signal finished with linker
Program file name: 2:test           /* write test.sh to unit 2
$ dir 2                             /* get a directory of unit 2
$ test                              /* run test program
```

Insert work disk in unit 0

```
$ cp 2:test.sh 0                    /* copy test.sh from unit 2 to unit 0
$ rm 2:*                            /* erase all files from unit 2
```

Try copying both the test.c and stdio.h files from your system disk onto your work disk, and the three files, cc.sh, compiler, and translator from your system disk to ramdisk unit 2. Then insert the work disk in the regular drive (unit 0). Next, compile using the -n command parameter to indicate you are using two drives:

```
$ cc -n test.c
$ dir
```

Compilation will have taken place without any disk swapping required, and will have resulted in the object file test.o being written to the work disk in unit 0, ready for linking.

## C SHELL UTILITIES - Adapted from "SOFTWARE TOOLS" by K & R

### Name

`find` - pattern matcher

### Synopsis

`find [-x] [-n] [-f] pattern [filename] [filename] ...`

### Description

Find searches the input and writes all lines matching the pattern to the standard output. Input is taken from the named files, if any, otherwise it is taken from the standard input.

The options are:

- x : Only write lines which don't match the pattern
- n : Write the line number of each matched line.
- f : Write the file name before each matched line.

Patterns may consist of ordinary characters which match corresponding characters in the input, and special characters or meta characters which match special patterns.

These characters are:

- ? Match any character.
- \* Match zero or more occurrences of the previous element of the pattern.
- % Match the start of a line.
- \$ Match the end of a line.

[class] Match any character belonging to the specified character class. A character class is simply a list of characters. For example, [aeiou] matches any lower case vowel. Character classes may be abbreviated if they contain sequences of consecutive letters or digits. For example, [A-Z] matches any upper case letter, and [aeiou0-9] matches any vowel or digit.

[!class] Match any character NOT belonging to the character class.

Preceding a meta-character with an "at" symbol (@) will cause it to be treated as an ordinary character. Thus @? matches a single question mark, and @@ matches a single "at" symbol.

## Examples

`abc`

Match lines containing the string "abc".

`%abc`

Match lines starting with the string "abc".

`x?y`

Match lines containing an x, followed by any character, followed by a y.

`x?*y$`

Match lines containing and x and ending with a y.

`%[a-z]*[A-Z][A-Z]*$`

Match lines starting with a possibly empty string of lower case letters and ending with a non-empty string of upper case letters.

`[_a-zA-Z][_a-zA-Z0-9]*[ ]*=[ ]*[0-9][0-9 ]*`

Match lines containing an assignment of an integer constant to a C identifier. Note that this pattern would have to be enclosed in double quotes since it contains spaces.

## Name

`format - text formatter`

## Synopsis

`$ format [filename] [filename] ...`

## Description

Format reads text from the standard input if no arguments are given, otherwise it reads from the specified files. Output is written to standard output.

Input may consist of ordinary text, which is filled and justified by default, and formatting commands. Formatting commands consist of a period (.) in the first character position of a line, a two character code, and for most commands an optional argument. The commands recognized by the formatter are:

`.bp n`

Begin page numbered `n`. This forces the start of a new page with page number `n`. The default for `n` is the current page number plus 1.

`.br`

Cause a break. This forces any accumulated text not yet written to be written immediately. Several commands implicitly cause a break before they perform their function. These are: `.bp`, `.ce`, `.fi`, `.ne`, `.nf`, `.sp`, and `.tl`.

`.ce n`

Center the next `n` lines. Default: `n=1`

`.fi`

Fill text. Text will be filled (output lines will contain as many words as possible) and right justified (right margins will be lined up). Format fills by default.

`.fo /left footer/center footer/right footer/`

Set footer (bottom of page titles). The strings "left footer", "center footer", "right footer" will be written at the bottom of each page left justified, centered and right justified respectively. All occurrences of the character '#' in the strings will be replaced with the current page number.

`.he /left header/center header/right header/`

Set header (top of page titles).

`.in n`

Set indentation. N spaces will be placed at the start of each output line. Default: n=0

`.ls n`

Set line spacing. N-1 blank lines will be inserted between each line of text. Default: n=1

`.m1 n`

Set margin above and including the header to n, Default: n=3

`.m2 n`

Set margin below header to n. Default: n=3

`.m3 n`

Set margin above the footer to n. Default: n=3

`.m4 n`

Set margin below and including the footer to n. Default: n=3

`.ne n`

Need n lines. If there are fewer than n lines remaining on the current page, then skip to a new page.



`.nf`

Stop filling text. Lines will be copied from input to output without change except for indentation and line spacing.

`.pl n`

Set page length (number of lines per page). Default: `n=66`

`.rm n`

Set right margin (the right-most character position to be written to). Default: `n=60`

`.sp n`

Write `n` blank lines.

`.ti n`

Temporary indent. The next output line (and only that line) will be given an indentation of `n` rather than the value set by `.in`. Default: `n=0`

Numeric arguments may be specified in two ways: as absolute (unsigned) integers, or as signed integers. Absolute arguments are assigned to parameters in the obvious way:

`.ls 2`

sets the line spacing to 2. Signed arguments indicate a change in the current value of the parameter being set; the value of the argument is added to or subtracted from the current value. For example, the commands

`.pl 66`  
`.pl -10`

will set the page length to 56, and

`.in 10`  
`.in +5`

will cause a temporary indent of 15.

Blank lines and lines starting with spaces occurring in the input are special cases. Blank lines cause a break and a number of blank lines equal to the current line spacing to be written. Lines starting with spaces cause a break and a temporary indent of `+n` where `n` is the number of spaces before the first non-space character on the line.

## **Name**

`print` - page files

## **Synopsis**

`$ print [filename] [filename] ...`

## **Description**

`Print` writes the named files (or the standard input if none are specified) to the standard output with margins at the top and bottom of each page, and a header at the top of each page.

## **Name**

`trim` - code optimizer

## **Synopsis**

`$ trim [filename.o] ...`

## **Description**

`Trim` optimizes the `.o` (or `.obj`) files produced by the compiler prior to `LINKing`. You can expect to reduce compiled file sizes in the order of 3 to 8 percent.

## RECOMMENDED READING LIST

<u>REFERENCE</u>	<u>AUTHOR</u>	<u>PUBLISHER</u>
Commodore 64 System Guide		Commodore
Commodore 128 System Guide		Commodore
Commodore 64 Programmer's Reference Guide		SAMS BOOKS
INNER SPACE ANTHOLOGY	Karl Hildon	Transactor
THE C PROGRAMMING LANGUAGE	Kernighan & Ritchie	Prentice-Hall
C PRIMER PLUS	The Waite Group	SAMS BOOKS
THE C PRIMER	Hancock & Krieger	McGraw-Hill
THE C PROGRAMMING TUTOR	Wortman & Sidebottom	Prentice-Hall
PROGRAMMING IN C	Robert J. Traister	Prentice-Hall
C PROGRAMMING GUIDELINES	Thomas Plum	Prentice-Hall
C, A REFERENCE MANUAL	Tarton Labs	Prentice-Hall
THE C PROGRAMMER'S HANDBOOK	Thom Hogan	Prentice-Hall

## POWER C INDEX

bit fields	1	quit	12
bye command	8,11	register declaration	3
ced command	9,12	remove command	8,11
character constants	4	screen colors	10,11
check command	10,13	search and replace	10,14
clear command	10,14	setu command	11
coding efficiently	7	shell	8,11
color command	11	sign extension	3
command interpreter	8,11	sign fill	3
command mode	10,13	sort command	9,12
commands	9,12	special symbols	14
compatibility	1	string constants	4
compiler	15	system configuration	47
conditional operator	2	trim program	58
copy command	11	tutorial type sizes	47
cp command	11	type sizes	3
cut and paste	14	wfreq command	9,13
dir command	11	zero page usage	6
dos command	11		
ed command	9,12		
editor	10,13		
efficient coding	7		
end-of-file marker	4		
exit command	8,11		
find command	9,12		
find program	53		
format program	55		
function index	18		
function keys	10,12		
get command	10,13		
goto	10,13		
header files	4		
identifiers	3		
interfacing with ML	5		
library	4		
library intro	16		
link command	9,12		
linker	15		
liner prompt	15		
linking functions	49		
list command	10,13		
load command	8,11		
machine language	5		
move command	8,11		
operators	2		
optimizing	49		
pointer init	1		
print command	8,11		
print program	33,55		
program size	4		
put command	10,13		



