CHAPTER 4

USING THE COMPILER

This chapter explains how to use the C cross compiler to compile programs on your host system. It explains how to invoke the compiler, and describes the command line options the compiler accepts. It also describes the functions which constitute the C library.

Invoking the Compiler

To invoke the cross compiler, type the command name **c**, followed by the name(s) of the file or files you want to compile and any compiler options you choose. All the valid compiler options are described in this chapter. Commands to compile source files must have the form:

c [-<options>] <file>.[c|s|o]

c is the name of the "compiler driver" program that runs the compiler and other programs. The compiler driver and the compilation process are discussed in Chapter 1 and in Appendix C. The option list is optional. You must include the name of at least one source file <file>. <file> must be a C source file with the suffix **'.c'**, an assembly language source file with the suffix **'.s'**, or an object file with the suffix **'.o'**. You may specify multiple source files with any combination of these suffixes in any order. You can change the default suffixes that **c** expects as described in Appendix C, "Modifying Compiler Operation." For information on default file names, see the section "File Naming Conventions" below.

If you do not specify any command line options, **c** will compile and link your <files> by default. It will also write the name of each file to STDOUT as it is processed. It writes any error messages to STDERR.

The following example compiles the C source file **acia.c**:

```
c acia.c
```

This command compiles, assembles, and links the **acia.c** C program, creating the executable program **xeq.80**.

If the compiler finds an error in your program, it halts compilation. When an error occurs, the compiler sends a error message to your terminal screen, or to your listing if you request one. Appendix A, "Compiler Error Messages," lists the error messages the compiler generates. If one or more command line arguments are invalid, **c** processes the next file name on the command line and begins the compilation process again. The only error that **c** tests for is invalid command line length.

The example command above does not request any compiler options. In this case, the compiler driver will use only default information to compile, assemble, and link your program. You can change the operation of the compiler driver by specifying the options you want when you run the compiler driver. You can also change the operation of the compiler driver by modifying the "prototype file" that specifies options directly to the passes of the compiler. Appendix C, "Modifying Compiler Operation," tells you how to modify the prototype file and create your own compiler driver options.

To specify options to the compiler driver, type the appropriate option or options on the command as shown in the first example above. Options should be separated with spaces. You must include the '-' or '+' that is part of the option name.

Compiler Command Line Options

The **c** compiler driver accepts the following command line options:

```
c -[delim* d*^ o* prefix* proto* sep* s v x +*^] <files>
```

The options are described below.

-delim* change delimiter characters between the file name and suffix on host systems where the period '.' is not an acceptable symbol.

-d*^ specify * as the name of a user-defined "programmable" compiler command line option. You may specify up to 20 such options. See Appendix B for information on creating user-defined compiler options.

-d64180 use the HD64180 extensions. By default code is produced for a Z80.

-ddl1 produce line number information (type 1). Type 1 line number information includes the function name and line number.

-ddl2 produce line number information (type 2). Type 2 line number information includes the file name and line number.

-dfloat Link your C program with C library functions from the floating point library. This option directs the linker to load the floating point library before the integer library. Therefore, modules common to both libraries are loaded from the floating point library. By default, the linker links your program with C library functions from the integer library only. See the section "C Library Support" in this chapter for more information.

-dlincl force a listing of all #include files to be included with the source listing, and cause diagnostics (when produced) to indicate the actual #include file name and line number. This option is meaningful only in combination with **-dlistcs**.

-dlistc Create a C source listing interspersed with error messages; the listing output defaults to <file>.lst.

-dlistcs merge C source listing with assembly language code; listing output defaults to <file>.ls.

-dnobss Generate an object image without a bss section. By default, the compiler automatically places all initialized static data objects in the bss section.

-dnostrict Direct the compiler to enforce weaker type checking. When you specify **-dnostrict**, the compiler will not generate an error message if:

1. you assign a pointer value to an integer,

2. you assign an integer value other than zero (NULL) to a pointer,

3. you check two different data objects of type pointer to ... for assignment compatibility, or 4. the type of the operand of a conditional expression that the compiler evaluates is not compatible with the type of the result. For information on how the compiler evaluates conditional expressions, see the section "Expressions" in

Chapter 2 of the C Language Specification for Microcontroller Environments. By default, any of the above actions causes the compiler to generate an error message.

-dprom enable automatic data initialization, and select the corresponding startup object file.

-dproto enable prototype checking.

-drev Reorder bits in bitfields, from most significant to least significant. By default, bits are numbered from least significant to most significant.

schar Take "plain" char as **signed char.** By default char is taken to bu **unsigned** char.

-dsp Enable single precision floating point support. By default, the compiler provides double precision floating point support.

-dsprec Force all floating point arithmetic to single precision. If this option is enabled, all floats, doubles, and long doubles are treated as floats, and calculations are made in single precision.

-dstd Force your C source input to conform to the April 1985 ANSI Draft Proposed Standard for C. By default, your C source need not conform to the ANSI Draft Proposed Standard to compile successfully.

-dstrict Direct the compiler to enforce stronger type checking. When you specify **-dstrict,** the compiler will generate an error message if:

1. an argument to a function definition does not have an explicit type associated with it at the time that the compiler encounters the opening '{' of the function body. By default, the compiler will assign the type **int** to a function argument that does not have an explicit type.

2. an assignment operation narrows the type of the result. By default, the compiler will truncate the result of a narrowing assignment without generating an error message. For more information on type conversion, see the section "Expressions" in Chapter 2 of the C Language Specification for Microcontroller Environments.

3. you use the return value of a function and do not declare the return type of the func-

tion. By default, the compiler will assign the type **int** to to the return value of a function.

-dverbose Display to STDOUT the name of the function currently being processed by the code generator, and the number of optimizations performed by the optimizer. By default, the compiler does not provide this specific information.

-dxdebug generate a Version 3.3 object module containing debugging information. This information can be interpreted by **cxdb** or by the **lines** or **prdbg** utilities. Only modules compiled with this option will contain debugging information. See Chapter 7 for more information.

-o* send output to the file *, retaining and appending to * the suffix for the bound binary image (if any) specified in the prototype file.

-prefix* prefix names of all permanent output files with the pathname *.

-proto* take prototype input from the file *.

-sep* separate filenames in the grouping line with the character *. The default is a space.

-s be "silent". Do not write filenames or arguments to STDOUT during compilation.

-v be "verbose". Before executing a command, print the command, along with its arguments, to STDOUT. The default is to output only the names of each file processed, and the root name when (and if) the grouping line is run. Each name is followed by a colon and newline. The root name consists of the source line without its suffix. If you specify both the **-v** and **-s** options, **-v** is performed.

-x do not execute the commands in the prototype file. Instead, write the commands which otherwise would have been performed to STDOUT.

+* save permanent copies of (otherwise temporary) intermediate files with suffix * Compilation halts at the end of the pass producing files with that suffix. For example, to save the intermediate output of the **cp180** compiler pass (the suffix for which is **2**), specify the +2 option.

The compiler accepts up to ten instances of +*. Valid values for * are any or all of the suffixes that the com-

piler appends to the output of each pass. The default suffixes generated are described in the section "File Naming Conventions" below. See Appendix D for information on compiler passes.

The compiler appends suffixes to output files produced by each pass by reading the prototype file and copying the labels preceding each step. You can change the default suffix that the compiler appends to its output files by modifying the labels in your prototype file. See Appendix B for more information.

File Naming Conventions

The programs making up the C cross compiler generate the following output file names by default when passed as input a file whose name (minus any suffix) is <file>. See the documentation on a specific program for information about how to change the default file names that it accepts as input or generates as output.

program	input file name	output file name
C Compiler Passes		
cpp80	<file>.c	<file>.1
cp180	<file>.1	<file>.2
cp280	<file>.2	<file>.3
opt80	<file>.3	<file>.s
assembler listing		<file>.[c|s] <file>.ls
C source and errors		<file>.c <file>.lst
C header files	<file>.h	
Assembler		
x80	<file>.s	<file>.o
source listing	<file>.s	<file>.ls
symbol table	<file>.s	<file>.sm
Linker		
lnk80	<file>.o	**xeq.80**
Programming Support Utilities		
hex80	<file>	STDOUT
lby	<file>	user must specify name
lm	<file>	STDOUT
lord80	<file>	**dir**
pr	<file>	STDOUT
rel80	<file>	STDOUT
toprom	<file>	<file>
unhex	<file>	<file>

Generating Listings

You can generate listings of the output of any (or all) the compiler passes by specifying one or both of the command line options **-dlincl** and **-dlistcs** to the **c** compiler driver. These and other command line options are

described in the preceding section.

When any command line options are specified, the compiler driver specifies the **+list** option to the appropriate compiler passes, forcing invocation of **lm**, the listing merger utility, which merges the listings output by the compiler, and **pr**, which paginates the listing and places a header at the top of each page.

All passes of the compiler accept the same line references and error message flags, which are generated by each pass along with the source code translation. When the compiler driver exits, all files created will have embedded in them relevant error listings from the compiler pass that generated them, as well as listings from all previous passes where diagnostics were output. For example, assembly language files will contain diagnostics generated by the compiler and/or the assembler, depending on if either or both programs found errors. However, any error message points to the line in the original source code where the error occurred. The listing will hold line number references from the originating language.

It is also possible to generate error listings that point to lines in the intermediate language file. One does so by stopping the compiler at the desired intermediate step using the **+*** compiler option, removing the original source line references, and re-compiling with the intermediate code as the source. For example, to generate a listing of assembly language code with error messages that refer to the assembly language code itself instead of the original C or Pascal source,

- stop compilation after the code generator pass, thus automatically saving the resulting assembly language code in a permanent file.

- Strip out the original **#line** sequences, instead of the original C or Pascal source, and begin compilation again from there. Error messages will now point to assembly language statements.

The example on the following pages shows the listing produced by compiling the C source file **acia.c** with the **-dlistcs** command line option:

```
c -dlistcs acia.c
```

```
 1                      .psect  _text
 2                ;     1
 3                ;     2  /*
 4                ;     3   *
 5                ;     4   * Each received character is copied in a buffer
 6                ;     5   * by the interrupt routine. main() reads
 7                ;     6   * characters from the buffer and echoes them.
 8                ;     7   *
 9                ;     8   */
10                ;     9  #define SIZE    512     /* size of the buffer */
11                ;    10
12                ;    11  /* the input port is a char at port 2
13                ;    12   * the output port is a char at port 5
14                ;    13   */
15                ;    14  @port char input @0x02;
16                ;    15  @port char output @0x05;
17                ;    16
18                ;    17  /* Some variables
19                ;    18   */
20                ;    19  char buffer[SIZE];      /* reception buffer */
21                ;    20  char *ptlec;            /* read pointer */
22                ;    21  char *ptecr;            /* write pointer */
23                ;    22
24                ;    23  /* "builtin" function for enabling interrupts
25                ;    24   */
26                ;    25  @builtin ei() @0xfb;
27                ;    26
28                ;    27  /* character reception. loops until a character
29                ;    28   is received */
30                ;    29  char getch()
31                ;    30      {
32                _getch:
33  0000  CD0000      call    c.savs0
34  0003  21F9FF      ld  hl,-7
35  0006  39          add hl,sp
36  0007  F9          ld  sp,hl
37                L1:
38                ;    31      char c;   /* character to be returned */
39                ;    32
40                ;    33      while (ptlec == ptecr)
41  0008  210000      ld  hl,_ptecr
42  000B  3A0200      ld  a,(_ptlec)
43  000E  BE          cp  (hl)
44  000F  2005        jr  nz,L4
45  0011  3A0300      ld  a,(_ptlec+1)
46  0014  23          inc hl
47  0015  BE          cp  (hl)
48                L4:
49  0016  28F0        jr  z,L1
50                ;    34                  ;
51                ;    35      c = *ptlec++;   /* get the received char */
52  0018  2A0200      ld  hl,(_ptlec)
53  001B  E5          push    hl
54  001C  23          inc hl
```

```
 55  001D  220200     ld  (_ptlec),hl
 56  0020  E1         pop hl
 57  0021  7E         ld  a,(hl)
 58  0022  DD77F9     ld  (ix-7),a
 59                ;  36      if (ptlec > &buffer[SIZE]) /* put in in buffer */
 60  0025  210402     ld  hl,_buffer+512
 61  0028  ED4B0200   ld  bc,(_ptlec)
 62  002C  7D         ld  a,l
 63  002D  91         sub c
 64  002E  7C         ld  a,h
 65  002F  98         sbc a,b
 66  0030  3006       jr  nc,L12
 67                ;  37                ptlec = buffer;
 68  0032  210400     ld  hl,_buffer
 69  0035  220200     ld  (_ptlec),hl
 70                L12:
 71                ;  38      return (c);
 72  0038  DD6EF9     ld  l,(ix-7)
 73  003B  97         sub a
 74  003C  67         ld  h,a
 75  003D  4D         ld  c,l
 76  003E  44         ld  b,h
 77  003F  C30000     jp  c.retsO
 78                ;  39      }
 79                ;  40
 80                ;  41  /*  Send a char to the output port
 81                ;  42   */
 82                ;  43  outch(c)
 83                ;  44      char c;
 84                ;  45      {
 85                _outch:
 86  0042  CD0000     call    c.savs
 87                ;  46      output = c;        /* send it */
 88  0045  DD7E04     ld  a,(ix+4)
 89  0048  D305       out (5), a
 90                ;  47      }
 91  004A  C30000     jp  c.rets
 92                ;  48
 93                ;  49  /* character reception routine.
 94                ;  50   * This routine is called on interrupt
 95                ;  51   * It puts the received char in the buffer
 96                ;  52   */
 97                ;  53  @port recept()
 98                ;  54      {
 99                _recept:
100  004D  F5         push    af
101  004E  C5         push    bc
102  004F  D5         push    de
103  0050  E5         push    hl
104  0051  CD5A00     call    L21
105  0054  E1         pop hl
106  0055  D1         pop de
107  0056  C1         pop bc
108  0057  F1         pop af
```

```
109 0058  ED4D      reti
110             L21:
111             ;   55      *ptecr++ = input;   /* get the char */
112 005A  2A0000    ld  hl,(_ptecr)
113 005D  E5        push    hl
114 005E  23        inc hl
115 005F  220000    ld  (_ptecr),hl
116 0062  E1        pop hl
117 0063  DB02      in  a, (2)
118 0065  77        ld  (hl),a
119             ;   56      if (ptecr >= &buffer[SIZE]) /* put in buffer*/
120 0066  210402    ld  hl,_buffer+512
121 0069  3A0000    ld  a,(_ptecr)
122 006C  95        sub l
123 006D  3A0100    ld  a,(_ptecr+1)
124 0070  9C        sbc a,h
125 0071  3806      jr  c,L13
126             ;   57              ptecr = buffer;
127 0073  210400    ld  hl,_buffer
128 0076  220000    ld  (_ptecr),hl
129             L13:
130             ;   58      }
131 0079  C9        ret
132             ;   59
133             ;   60  /*  Main program : an infinite loop
134             ;   61   *  of receive transmit
135             ;   62   */
136             ;   63  main()
137             ;   64      {
138             _main:
139 007A  CD0000    call    c.savs0
140 007D  21F9FF    ld  hl,-7
141 0080  39        add hl,sp
142 0081  F9        ld  sp,hl
143             ;   65      char c;          /* received char */
144             ;   66
145             ;   67      ptecr = ptlec = buffer; /* init pointers */
146 0082  210400    ld  hl,_buffer
147 0085  220200    ld  (_ptlec),hl
148 0088  220000    ld  (_ptecr),hl
149             ;   68      ei();               /*enable interrupts */
150 008B  FB        .byte   251
151             L14:
152             ;   69      for (;;)            /* loop */
153             ;   70          {
154             ;   71          c = getch();    /* get a char */
155 008C  CD0000    call    _getch
156 008F  DD71F9    ld  (ix-7),c
157             ;   72          outch(c);       /* send it */
158 0092  DD6EF9    ld  l,(ix-7)
159 0095  97        sub a
160 0096  67        ld  h,a
161 0097  CD4200    call    _outch
162             ;   73          }
```

```
163 009A  18F0        jr  L14
164               ;   74        }
165                   .psect  _bss
166               _ptecr:
167                  .byte   [2]
168               _ptlec:
169                   .byte   [2]
170               _buffer:
171                   .byte   [512]
172                   .external   c.rets0
173                   .external   c.savs0
174                   .public _ptecr
175                   .public _ptlec
176                   .public _getch
177                   .public _recept
178                   .public _buffer
179                   .external   c.rets
180                   .external   c.savs
181                   .public _main
182                   .public _outch
183                   .end
```

Return Status

c returns success if it can open all files successfully. It prints a message to STDERR and returns failure if there are errors in the prototype file, or if any files cannot be opened.

Examples

To echo the names of each program that the compiler driver runs to your terminal screen:

```
c -v file.c
```

To generate information for use by the **cxdb,** the C source debugger:

```
c -dxdebug file.c
```

To save the intermediate files created by the code generator and the assembler, and to halt compilation after creating an object file:

```
c +s +o file.c
```

To give the name myprog to the executable file that the compiler generates:

```
c -o myprog file.c
```

C Library Support

This section describes the facilities provided by the C library. The C cross compiler for Z80/HD64180 includes all the standard C library functions, including those useful to programmers writing applications for ROM-based systems.

How C Library Functions are Packaged

The functions in the C library are packaged in three separate sub-libraries; one for machine-dependent routines (the machine library), one for functions that do not require floating point support (the integer library) and one for functions that do require floating point support (the floating point library). If your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by including only functions from the integer library.

Inserting Assembler Code Directly

Assembler instructions can be quoted directly into C source files, and entered unchanged into the output assembly stream, by use of the **_asm()** function. This function is not part of either library; it is recognized by the compiler itself.

Linking Libraries with Your Program

By default, the compiler links its input programs with appropriate modules from the integer library only. If your application requires floating point support, specify the -dfloat option when invoking the compiler, or at link time, if compilation and linking occur separately. This option directs the linker to load the floating point library before the integer library. Modules common to both libraries will therefore be loaded from the floating point library, followed by the appropriate modules from the floating point and integer libraries, in that order.

Integer Library Functions

The following table lists the C library functions in the integer library.

Integer Library Functions

abs	islower	printf	strchr
atoi	isprint	putchar	strcmp
atol	ispunct	puts	strcpy
calloc	isspace	rand	strcspn
free	isupper	realloc	strlen
getchar	isxdigit	sbreak	strncat
gets	longjmp	scanf	strncmp
isalnum	malloc	setjmp	strncpy
isalpha	memchr	sprintf	strpbrk
iscntrl	memcmp	srand	strrchr
isdigit	memcpy	sscanf	strspn
isgraph	memset	strcat	tolower
toupper			

Floating Point Library Functions

Floating Point Library Functions

acos	cos	log10	sprintf
asin	cosh	pow	sqrt
atan	exp	printf	sscanf
atan2	fabs	scanf	tan
atof	floor	sin	tanh
ceil	log	sinh	

Common Input/Output Functions

Four of the functions that perform stream input/output are included in both the integer and floating point libraries. The functionalities of the versions in the integer library are a subset of the functionalities of their floating point counterparts. The versions in the integer library cannot print or manipulate floating point numbers. These functions are: **printf**, **scanf**, **sprintf** and **sscanf**.

Functions Implemented as Macros

Five of the functions in the C library are actually implemented as "macros." Unlike other functions, which (if they do not return int) are declared in header files and defined in a separate object module that is linked in with your program later, functions implemented as macros are defined using **#define** preprocessor directives in the header file that declares them. Macros can therefore be used independently of any library by including the header file that defines and declares them with your program, as explained below. The functions in the C library that are implemented as macros are: **max, min, va_arg, va_end,** and **va_start.**

Including Header Files

If your application calls a C library function that can return a value of a type other than int, you must include the header file that declares the function at compile time. You do this by writing a preprocessor directive of the form:

#include <header_name>
in your program, where **<header_name>** is the name of the appropriate header file enclosed in angle brackets. Include a required header file before you refer to any function that it declares.

C library functions returning int are not declared in header files, since int is the function return type that the compiler assumes by default. You can therefore call these functions without including any header file.

The names of the header files packaged with the C library and the functions declared in each header are listed below.

<math.h> Header file for mathematical functions: **abs, acos,** asin, atan, **atan2, ceil, cos, cosh, exp, fabs, floor,** log, log10, **pow, sin, sinh, sqrt, tan** and **tanh**

<setjmp.h> Header file for nonlocal jumps: **setjmp** and **longjmp**

<stdarg.h> Header file for walking argument lists: **va_arg, va_end** and **va_start.** Use these macros with any function you write that must accept a variable number of arguments.

<stdio.h> Header file for stream input/output: **getchar, gets, printf, putchar, puts, scanf, sprintf** and **sscanf**

<stdlib.h> Header file for general utilities: **atof, atoi, atol, calloc, free, malloc, max, min, rand, realloc** and **srand**

<string.h> Header file for string functions: **memchr, memcmp, memcpy, memset, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr** and **strspn**

<wslxa.h> This header file provides definitions for Whitesmiths' defined types and special storage classes, and for various system parameters. There are no C library functions defined in **<wslxa.h>.**

Functions returning int C library functions that return int, and can therefore be called without including any header file, are: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, sbreak, tolower** and **toupper**

Descriptions of C Library Functions

The following pages describe each of the functions in the C library in quick reference format. The descriptions are in alphabetical order by function name.

NAME

_asm() - generate inline assembly code

SYNOPSIS

```
/*      no header file need be included */
_asm(<string constant>);
```

FUNCTION

_asm() generates inline assembly code by copying <string constant> and quoting it into the output assembly code stream.

RETURNS

Nothing, unless **_asm()** is used in an expression. In that case, normal return conventions must be followed. See chapter 3, "Register Usage."

EXAMPLE

The sequence **ld hl, 1; call _func,** may be generated by the following call:

```
_asm("\tld hl, 1\n\tcall _func\n");
```

Note that the string-quoting syntax matches the familiar **printf()** function.

NOTES

_asm() is not packaged in any library. It is recognized (and its argument passed unchanged) by the compiler itself.

NAME

abs - find absolute value

SYNOPSIS

```
/*  no header file need be included */
int abs(i)
    int i;
```

FUNCTION

abs obtains the absolute value of i. No check is made to see that the result can be properly represented.

RETURNS

abs returns the absolute value of i, expressed as an int.

EXAMPLE

To print out a debit or credit balance:

```
printf("balance %d%s\n", abs(bal), (bal < 0)
    ? "CR" : "");
```

SEE ALSO

fabs

NOTES

abs is packaged in the integer library.

NAME

acos - arccosine

SYNOPSIS

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

FUNCTION

acos computes the angle in radians the cosine of which is x, to full double precision.

RETURNS

acos returns the closest internal representation to **acos(x)**, expressed as a double floating value in the range [0, pi]. If x is outside the range [-1, 1], **acos** returns zero.

EXAMPLE

To find the arccosine of **x:**

```
    theta = acos(x);
```

SEE ALSO

asin, atan, atan2

NOTES

acos is packaged in the floating point library.

NAME

asin - arcsine

SYNOPSIS

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

FUNCTION

asin computes the angle in radians the sine of which is x, to full double precision.

RETURNS

asin returns the nearest internal representation to **asin(x)**, expressed as a double floating value in the range [-pi/2, pi/2]. If x is outside the range [-1, 1], **asin** returns zero.

EXAMPLE

To compute the arcsine of **y**:

```
    theta = asin(y);
```

SEE ALSO

acos, atan, atan2

NOTES

asin is packaged in the floating point library.

NAME

atan - arctangent

SYNOPSIS

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

FUNCTION

atan computes the angle in radians; the tangent of which is x, to full double precision.

RETURNS

atan returns the nearest internal representation to **atan(x)**, expressed as a double floating value in the range [-pi/2, pi/2].

EXAMPLE

To find the phase angle of a vector in degrees:

```
theta = atan(y/x) * 180.0 / pi;
```

SEE ALSO

acos, asin, atan2

NOTES

atan is packaged in the floating point library.

NAME

atan2 - arctangent of y/x

SYNOPSIS

```
#include <math.h>
double atan2(y, x)
    double y, x;
```

FUNCTION

atan2 computes the angle in radians the tangent of which is y/x to full double precision. If y is negative, the result is negative. If x is negative, the magnitude of the result is greater than **pi/2**.

RETURNS

atan2 returns the closest internal representation to **atan(y/x)**, expressed as a double floating value in the range [-pi, pi]. If both input arguments are zero, **atan2** returns zero.

EXAMPLE

To find the phase angle of a vector in degrees:

```
theta = atan2(y/x) * 180.0/pi;
```

SEE ALSO

acos, asin, atan

NOTES

atan2 is packaged in the floating point library.

NAME

atof - convert buffer to double

SYNOPSIS

```
#include <stdlib.h>
double atof(nptr)
    const char *nptr;
```

FUNCTION

atof converts the string at nptr into a double. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable inputs match the pattern

[+|-]d*[.d*][e[+|-]dd*]

where d is any decimal digit and e is the character **'e'** or **'E'**. No checks are made against overflow, underflow, or invalid character strings.

RETURNS

atof returns the converted double value. If the string has no recognizable characters, it returns zero.

EXAMPLE

To read a string from STDIN and convert it to a double at **d:**

```
gets(buf);
d = atof(buf);
```

SEE ALSO

atoi, atol

NOTES

atof is packaged in the floating point library.

NAME

atoi - convert buffer to integer

SYNOPSIS

```
#include <stdlib.h>
int atoi(nptr)
    const char *nptr;
```

FUNCTION

atoi converts the string at nptr into an integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **l** or **L**, it is skipped over.

No checks are made against overflow or invalid character strings.

RETURNS

atoi returns the converted integer value. If the string has no recognizable characters, zero is returned.

EXAMPLE

To read a string from STDIN and convert it to an int at **i**:

```
gets(buf);
i = atoi(buf);
```

SEE ALSO

atof, atol

NOTES

atoi is packaged in the integer library.

NAME

atol - convert buffer to long

SYNOPSIS

```
#include <stdlib.h>
long atol(nptr)
    const char *nptr;
```

FUNCTION

atol converts the string at nptr into a long integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **l** or **L** it is skipped over.

No checks are made against overflow or invalid character strings.

RETURNS

atol returns the converted long integer. If the string has no recognizable characters, zero is returned.

EXAMPLE

To read a string from STDIN and convert it to a long **l**:

```
gets(buf);
l = atol(buf);
```

SEE ALSO

atof, atoi

NOTES

atol is packaged in the integer library.

NAME

calloc - allocate and clear space on the heap

SYNOPSIS

```
#include <stdlib.h>
void *calloc(nelem, elsize)
    unsigned int nelem;
    unsigned int elsize;
```

FUNCTION

calloc allocates space on the heap for an item of size nbytes, where nbytes = nelem * elsize. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, **calloc** returns a null pointer. The pointer returned may be assigned to an object of any type without casting. The allocated space is initialized to zero.

RETURNS

calloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL.

EXAMPLE

To allocate an array of ten doubles:

```
    double *pd;
    pd = calloc(10, sizeof (double));
```

SEE ALSO

free, malloc, realloc

NOTES

calloc is packaged in the integer library.

NAME

ceil - round to next higher integer

SYNOPSIS

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

FUNCTION

ceil computes the smallest integer greater than or equal to x.

RETURNS

ceil returns the smallest integer greater than or equal to x, expressed as a double floating value.

EXAMPLE

```
    x  ceil(x)

  5.1     6.0
  5.0     5.0
  0.0     0.0
 -5.0    -5.0
 -5.1    -5.0
```

SEE ALSO

floor

NOTES

ceil is packaged in the floating point library.

NAME

cos - cosine

SYNOPSIS

```
#include <math.h>
double cos(x)
    double x;
```

FUNCTION

cos computes the cosine of x; expressed in radians, to full double precision. If the magnitude of x is too large to contain a fractional quadrant part, the value of **cos** is 1.

RETURNS

cos returns the nearest internal representation to **cos(x)** in the range [0, pi], expressed as a double floating value. A large argument may return a meaningless value.

EXAMPLE

To rotate a vector through the angle **theta:**

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

sin, tan

NOTES

cos is packaged in the floating point library.

NAME

cosh - hyperbolic cosine

SYNOPSIS

```
#include <math.h>
double cosh(x)
    double x;
```

FUNCTION

cosh computes the hyperbolic cosine of x to full double precision.

RETURNS

cosh returns the nearest internal representation to **cosh(x)** expressed as a double floating value. If the result is too large to be properly represented, **cosh** returns zero.

EXAMPLE

To use de Moivre's theorem to compute **(cosh x + sinh x)** to the **nth** power:

```
    demoivre = cosh(n * x) + sinh(n * x);
```

SEE ALSO

exp, sinh, tanh

NOTES

cosh is packaged in the floating point library.

NAME

exp - exponential

SYNOPSIS

```
#include <math.h>
double exp(x)
    double x;
```

FUNCTION

exp computes the exponential of x to full double precision.

RETURNS

exp returns the nearest internal representation to **exp** x, expressed as a double floating value. If the result is too large to be properly represented, **exp** returns zero.

EXAMPLE

To compute the hyperbolic sine of x:

```
sinh = (exp(x) - exp(-x)) / 2.0;
```

SEE ALSO

log

NOTES

exp is packaged in the floating point library.

NAME

fabs - find double absolute value

SYNOPSIS

```
#include <math.h>
double fabs(x)
    double x;
```

FUNCTION

fabs obtains the absolute value of x.

RETURNS

fabs returns the absolute value of x, expressed as a double floating value.

EXAMPLE

```
   x  fabs(x)
 5.0    5.0
 0.0    0.0
-3.7    3.7
```

SEE ALSO

abs

NOTES

fabs is packaged in the floating point library.

NAME

floor - round to next lower integer

SYNOPSIS

```
#include <math.h>
double floor(x)
    double x;
```

FUNCTION

floor computes the largest integer less than or equal to x.

RETURNS

floor returns the largest integer less than or equal to x, expressed as a double floating value.

EXAMPLE

x	floor(x)
5.1	5.0
5.0	5.0
0.0	0.0
-5.0	-5.0
-5.1	-6.0

SEE ALSO

ceil

NOTES

floor is packaged in the floating point library.

NAME

free - free space on the heap

SYNOPSIS

```
#include <stdlib.h>
void free(ptr)
    void *ptr;
```

FUNCTION

free returns an allocated cell to the heap for subsequent reuse. The cell pointer ptr must have been obtained by an earlier **calloc**, **malloc**, or **realloc** call; otherwise the heap will become corrupted. **free** does its best to check for invalid values of ptr. A NULL value for ptr is explicitly allowed, however, and is ignored.

RETURNS

Nothing.

EXAMPLE

To give back an allocated area:

```
free(pd);
```

SEE ALSO

calloc, malloc, realloc

NOTES

No effort is made to lower the system break when storage is freed, so it is quite possible that earlier activity on the heap may cause problems later on the stack.

free is packaged in the integer library.

NAME

getchar - get character from input stream

SYNOPSIS

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

FUNCTION

getchar obtains the next input character, if any, from the user supplied input stream. This user must rewrite this function in C or in assembly language to provide an interface to the input mechanism of the C library.

RETURNS

getchar returns the next character from the input stream. If end of file (break) is encountered, or a read error occurs, **getchar** returns EOF.

EXAMPLE

To copy characters from the input stream to the output stream:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

SEE ALSO

getc, putchar

NOTES

getchar is packaged in the integer library.

NAME

gets - get a text line from input stream

SYNOPSIS

```
#include <stdio.h>
char *gets(s)
    char *s;
```

FUNCTION

gets copies characters from the input stream to the buffer starting at s. Characters are copied until a newline is reached or end of file is reached. If a newline is reached, it is discarded and a NUL is written immediately following the last character read into s.

gets uses **getchar** to read each character.

RETURNS

gets returns s if successful. If end of file is reached, **gets** returns NULL. If a read error occurs, the array contents are indeterminate and **gets** returns NULL.

EXAMPLE

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

SEE ALSO

puts

NOTES

There is no assured limit on the size of the line read by **gets**.

gets is packaged in the integer library.

NAME

isalnum - test for alphabetic or numeric character

SYNOPSIS

```
/*  no header file need be included */
int isalnum(c)
    int c;
```

FUNCTION

isalnum tests whether c is an alphabetic character (either upper or lower case), or a decimal digit.

RETURNS

isalnum returns nonzero if the argument is an alphabetic or numeric character; otherwise the value returned is zero.

EXAMPLE

To test for a valid C identifier:

```
if (isalpha(*s) || *s == '_')
    for (++s; isalnum(*s) || *s == '_'; ++s)
        ;
```

SEE ALSO

isalpha, isdigit, islower, isupper, isxdigit, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isalnum is packaged in the integer library.

NAME

isalpha - test for alphabetic character

SYNOPSIS

```
/*  no header file need be included */
int isalpha(c)
    int c;
```

FUNCTION

isalpha tests whether c is an alphabetic character, either upper or lower case.

RETURNS

isalpha returns nonzero if the argument is an alphabetic character. Otherwise the value returned is zero.

EXAMPLE

To find the end points of an alphabetic string:

```
while (*first && !isalpha(*first))
    ++first;
for (last = first; isalpha(*last); ++last)
    ;
```

SEE ALSO

isalnum, isdigit, islower, isupper, isxdigit, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isalpha is packaged in the integer library.

NAME

iscntrl - test for control character

SYNOPSIS

```
/*  no header file need be included */
int iscntrl(c)
    int c;
```

FUNCTION

iscntrl tests whether c is a delete character (0177 in ASCII), or an ordinary control character (less than 040 in ASCII).

RETURNS

iscntrl returns nonzero if c is a control character; otherwise the value is zero.

EXAMPLE

To map control characters to percent signs:

```
for (; *s; ++s)
    if (iscntrl(*s))
        *s = '%';
```

SEE ALSO

isgraph, isprint, ispunct, isspace

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

iscntrl is packaged in the integer library.

NAME

isdigit - test for digit

SYNOPSIS

```
/*  no header file need be included */
int isdigit(c)
    int c;
```

FUNCTION

isdigit tests whether c is a decimal digit.

RETURNS

isdigit returns nonzero if c is a decimal digit; otherwise the value returned is zero.

EXAMPLE

To convert a decimal digit string to a number:

```
for (sum = 0; isdigit(*s); ++s)
    sum = sum * 10 + *s - '0';
```

SEE ALSO

isalnum, isalpha, islower, isupper, isxdigit, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isdigit is packaged in the integer library.

NAME

isgraph - test for graphic character

SYNOPSIS

```
/*  no header file need be included */
int isgraph(c)
    int c;
```

FUNCTION

isgraph tests whether c is a graphic character; i.e. any printing character except a space (040 in ASCII).

RETURNS

isgraph returns nonzero if c is a graphic character. Otherwise the value returned is zero.

EXAMPLE

To output only graphic characters:

```
    for (; *s; ++s)
        if (isgraph(*s))
            putchar(*s);
```

SEE ALSO

iscntrl, isprint, ispunct, isspace

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isgraph is packaged in the integer library.

NAME

islower - test for lowercase character

SYNOPSIS

```
/*  no header file need be included */
int islower(c)
    int c;
```

FUNCTION

islower tests whether c is a lowercase alphabetic character.

RETURNS

islower returns nonzero if c is a lowercase character; otherwise the value returned is zero.

EXAMPLE

```
To convert to uppercase:
if (islower(c))
    c += 'A' - 'a'; /* also see toupper() */
```

SEE ALSO

isalnum, isalpha, isdigit, isupper, isxdigit, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

islower is packaged in the integer library.

NAME

isprint - test for printing character

SYNOPSIS

```
/*  no header file need be included */
int isprint(c)
    int c;
```

FUNCTION

isprint tests whether c is any printing character. Printing characters are all characters between a space (040 in ASCII) and a tilde '~' character (0176 in ASCII).

RETURNS

isprint returns nonzero if c is a printing character; otherwise the value returned is zero.

EXAMPLE

To output only printable characters:

```
for (; *s; ++s)
    if (isprint(*s))
        putchar(*s);
```

SEE ALSO

iscntrl, isgraph, ispunct, isspace

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isprint is packaged in the integer library.

NAME

ispunct - test for punctuation character

SYNOPSIS

```
/*  no header file need be included */
int ispunct(c)
    int c;
```

FUNCTION

ispunct tests whether c is a punctuation character. Punctuation characters include any printing character except space, a digit, or a letter.

RETURNS

ispunct returns nonzero if c is a punctuation character; otherwise the value returned is zero.

EXAMPLE

To collect all punctuation characters in a string into a buffer:

```
for (i = 0; *s; ++s)
    if (ispunct(*s))
        buf[i++] = *s;
```

SEE ALSO

iscntrl, isgraph, isprint, isspace

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

ispunct is packaged in the integer library.

NAME

isspace - test for whitespace character

SYNOPSIS

```
/*  no header file need be included */
int isspace(c)
    int c;
```

FUNCTION

isspace tests whether c is a whitespace character. Whitespace characters are horizontal tab ('\t'), newline ('\n'), vertical tab ('\v'), form feed ('\f'), carriage return ('\r'), and space (' ').

RETURNS

isspace returns nonzero if c is a whitespace character; otherwise the value returned is zero.

EXAMPLE

To skip leading whitespace:

```
while (isspace(*s))
    ++s;
```

SEE ALSO

iscntrl, isgraph, isprint, ispunct

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isspace is packaged in the integer library.

NAME

isupper - test for uppercase character

SYNOPSIS

```
/*  no header file need be included */
int isupper(c)
    int c;
```

FUNCTION

isupper tests whether c is an uppercase alphabetic character.

RETURNS

isupper returns nonzero if c is an uppercase character; otherwise the value returned is zero.

EXAMPLE

To convert to lowercase:

```
if (isupper(c))
    c += 'a' - 'A'; /* also see tolower() */
```

SEE ALSO

isalnum, isalpha, isdigit, islower, isxdigit, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isupper is packaged in the integer library.

NAME

isxdigit - test for hexadecimal digit

SYNOPSIS

```
/*  no header file need be included */
int isxdigit(c)
    int c;
```

FUNCTION

isxdigit tests whether c is a hexadecimal digit, i.e. in the set [0123456789abcdefABCDEF].

RETURNS

isxdigit returns nonzero if c is a hexadecimal digit; otherwise the value returned is zero.

EXAMPLE

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s)
        sum = sum * 10 + *s - '0';
    else
        sum = sum * 10 + tolower(*s) + (10 - 'a');
```

SEE ALSO

isalnum, isalpha, isdigit, islower, isupper, tolower, toupper

NOTES

If the argument is outside the range [-1, 255], the result is undefined.

isxdigit is packaged in the integer library.

NAME

log - natural logarithm

SYNOPSIS

```
#include <math.h>
double log(x)
    double x;
```

FUNCTION

log computes the natural logarithm of x to full double precision.

RETURNS

log returns the closest internal representation to **log(x)**, expressed as a double floating value. If the input argument is less than zero, or is too large to be represented, **log** returns zero.

EXAMPLE

To compute the hyperbolic arccosine of **x**:

```
arccosh = log(x + sqrt(x * x - 1));
```

SEE ALSO

exp

NOTES

log is packaged in the floating point library.

NAME

log10 - common logarithm

SYNOPSIS

```
#include <math.h>
double log10(x)
    double x;
```

FUNCTION

log10 computes the common log of x to full double precision by computing the natural log of x divided by the natural log of 10. If the input argument is less than zero, a domain error will occur. If the input argument is zero, a range error will occur.

RETURNS

log10 returns the nearest internal representation to **log10** x, expressed as a double floating value. If the input argument is less than or equal to zero, **log10** returns zero.

EXAMPLE

To determine the number of digits in **x**,
where **x** is a positive integer expressed as a double:

```
    ndig = log10(x) + 1;
```

SEE ALSO

log

NOTES

log10 is packaged in the floating point library.

NAME

longjmp - restore calling environment

SYNOPSIS

```
#include <setjmp.h>
void longjmp(env, val)
    jmp_buf env;
    int val;
```

FUNCTION

longjmp restores the environment saved in env by **setjmp.** If env has not been set by a call to **setjmp,** or if the caller has returned in the meantime, the resulting behavior is unpredictable.

All accessible objects have their values restored when **longjmp** is called, except for objects of storage class register, the values of which have been changed between the **setjmp** and **longjmp** calls.

RETURNS

When **longjmp** returns, program execution continues as if the corresponding call to **setjmp** had returned the value val. **longjmp** cannot force **setjmp** to return the value zero. If val is zero, **setjmp** returns the value one.

EXAMPLE

You can write a generic error handler as:

```
void handle(err)
    int err;
    {
    extern jmp_buf env;

    longjmp(env, err);  /* return from setjmp */
    }
```

SEE ALSO

setjmp

NOTES

longjmp is packaged in the integer library.

NAME

malloc - allocate space on the heap

SYNOPSIS

```
#include <stdlib.h>
void *malloc(nybytes)
    unsigned int nbytes;
```

FUNCTION

malloc allocates space on the heap for an item of size nbytes. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, **malloc** returns a null pointer.

RETURNS

malloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL. The pointer returned may be assigned to an object of any type without casting.

EXAMPLE

To allocate an array of ten doubles:

```
double *pd;

pd = malloc(10 * sizeof *pd);
```

SEE ALSO

calloc, free, realloc

NOTES

malloc is packaged in the integer library.

NAME

max - test for maximum

SYNOPSIS

```
#include <stdlib.h>
max(a, b)
```

FUNCTION

max obtains the maximum of its two arguments, a and b. Since **max** is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

RETURNS

max is a numerical rvalue of the form **((a <** b**) ?** b **:** a**),** suitably parenthesized.

EXAMPLE

To set a new maximum level:

```
    hiwater = max(hiwater, level);
```

SEE ALSO

min

NOTES

max is an extension to the proposed ANSI C standard.

max is a macro declared in the **<stdlib.h>** header file. You can use it by including **<stdlib.h>** with your program. Because it is a macro, **max** cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

memchr - scan buffer for character

SYNOPSIS

```
#include <string.h>
void *memchr(s, c, n)
    const void *s;
    int c;
    unsigned int n;
```

FUNCTION

memchr looks for the first occurrence of a specific character c in an n character buffer starting at s.

RETURNS

memchr returns a pointer to the first character that matches c, or NULL if no character matches.

EXAMPLE

To map **keybuf[]** characters into **subst[]** characters:

```
if ((t = memchr(keybuf, *s, KEYSIZ)) != NULL)
    *s = subst[t - keybuf];
```

SEE ALSO

strchr, strcspn, strpbrk, strrchr, strspn

NOTES

memchr is packaged in the integer library.

NAME

memcmp - compare two buffers for lexical order

SYNOPSIS

```
#include <string.h>
int memcmp(s1, s2, n)
    const void *s1, *s2;
    unsigned int n;
```

FUNCTION

memcmp compares two text buffers, character by character, for lexical order in the character collating sequence. The first buffer starts at s1, the second at s2; both buffers are n characters long.

RETURNS

memcmp returns a short integer greater than, equal to, or less than zero, according to whether s1 is lexicographically greater than, equal to, or less than s2.

EXAMPLE

To look for the string "include" in **name:**

```
if (memcmp(name, "include", 7) == 0)
    doinclude();
```

SEE ALSO

strcmp, strncmp

NOTES

memcmp is packaged in the integer library.

NAME

memcpy - copy one buffer to another

SYNOPSIS

```
#include <string.h>
void *memcpy(s1, s2, n)
    void *s1;
    const void *s2;
    unsigned int n;
```

FUNCTION

memcpy copies the first n characters starting at location s2 into the buffer beginning at s1.

RETURNS

memcpy returns s1.

EXAMPLE

To place "first string, second string" in **buf[]**:

```
memcpy(buf, "first string", 12);
memcpy(buf + 13, ", second string", 15);
```

SEE ALSO

strcpy, strncpy

NOTES

memcpy is packaged in the integer library.

NAME

memset - propagate fill character throughout buffer

SYNOPSIS

```
#include <string.h>
void *memset(s, c, n)
    void *s;
    int c;
    unsigned int n;
```

FUNCTION

memset floods the n character buffer starting at s with fill character c.

RETURNS

memset returns s.

EXAMPLE

To flood a 512-byte buffer with NULs:

```
memset(buf,'\0', BUFSIZ);
```

NOTES

memset is packaged in the integer library.

NAME

min - test for minimum

SYNOPSIS

```
#include <stdlib.h>
min(a, b)
```

FUNCTION

min obtains the minimum of its two arguments, a and b. Since **min** is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

RETURNS

min is a numerical rvalue of the form **((a < b) ? a : b),** suitably parenthesized.

EXAMPLE

To set a new minimum level:

```
    nmove = min(space, size);
```

SEE ALSO

max

NOTES

min is an extension to the proposed ANSI C standard.

min is a macro declared in the **<stdlib.h>** header file. You can use it by including **<stdlib.h>** with your program. Because it is a macro, **min** cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated more than once.

NAME

pow - raise x to the y power

SYNOPSIS

```
#include <math.h>
double pow(x, y)
    double x, y;
```

FUNCTION

pow computes the value of x raised to the power of y.

RETURNS

pow returns the value of x raised to the power of y, expressed as a double floating value. If x is zero and y is less than or equal to zero, or if x is negative and y is not an integer, **pow** returns zero.

EXAMPLE

x	y	pow(x, y)
2.0	2.0	4.0
2.0	1.0	2.0
2.0	0.0	1.0
1.0	any	1.0
0.0	-2.0	0
-1.0	2.0	1.0
-1.0	2.1	0

SEE ALSO

exp

NOTES

pow is packaged in the floating point library.

NAME

printf - output formatted arguments to stdout

SYNOPSIS

```
#include <stdio.h>
int printf(fmt, ...)
    const char *fmt;
```

FUNCTION

printf writes formatted output to the output stream using the format string at fmt and the arguments specified by ..., as described below.

printf uses **putchar** to output each character.

Format Specifiers

The format string at fmt consists of literal text to be output, interspersed with conversion specifications that determine how the arguments are to be interpreted and how they are to be converted for output. If there are insufficient arguments for the format, the results are undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. **printf** returns when the end of the format string is encountered.

Each <conversion specification> is started by the character **'%'**. After the **'%'**, the following appear in sequence:

<flags> - zero or more which modify the meaning of the conversion specification.

<field width> - a decimal number which optionally specifies a minimum field width. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left adjustment flag has been given) to the field width. The padding is with spaces unless the field width digit string starts with zero, in which case the padding is with zeros.

<precision> - a decimal number which specifies the minimum number of digits to appear for **d, i, o, u, x,** and **X** conversions, the number of digits to appear after the decimal point for **e, E,** and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be printed from a string in an **s** conversion. The

precision takes the form of a period followed by a decimal digit string. A null digit string is treated as zero.

h - optionally specifies that the following **d, i, o, u, x,** or **X** conversion character applies to a short int or unsigned short int argument (the argument will have been widened according to the integral widening conversions, and its value must be cast to short or unsigned short before printing). It specifies a short pointer argument if associated with the **p** conversion character. If an **h** appears with any other conversion character, it is ignored.

l - optionally specifies that the **d, i, o, u, x,** and **X** conversion character applies to a long int or unsigned long int argument. It specifies a long or far pointer argument if used with the **p** conversion character. If the **l** appears with any other conversion character, it is ignored.

L - optionally specifies that the following **e, E, f, g,** and **G** conversion character applies to a long double argument. If the **L** appears with any other conversion character, it is ignored.

<conversion character> - character that indicates the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' instead of a digit string. In this case, an int argument supplies the field width or precision. The arguments supplying field width must appear before the optional argument to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if it were missing.

The <flags> field is zero or more of the following:

space - a space will be prepended if the first character of a signed conversion is not a sign. This flag will be ignored if space and + flags are both specified.

- result is to be converted to an "alternate form". For **c, d, i, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be zero. For **p, x** and **X** conversion, a non-zero result will have 0x or 0X prepended to it. For **e, E, f, g,** and **G** conversions, the result will contain a decimal point, even if no digits follow the point. For **g** and **G** conversions, trailing zeros will not be removed from the result, as they normally are. For **p** conversion, it

designates hexadecimal output.

+ - result of signed conversion will begin with a plus or minus sign.

- - result of conversion will be left justified within the field.

The <conversion character> is one of the following:

% - a **'%'** is printed. No argument is converted.

c - the least significant byte of the int argument is converted to a character and printed.

d, i, o, u, x, X - the int argument is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**); the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** are used for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with precision of zero is no characters.

e, E - the double argument is converted in the style **[-]d.ddde+dd,** where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be printed is greater than or equal to **1E+100,** additional exponent digits will be printed as necessary.

f - the double argument is converted to decimal notation in the style **[-]ddd.ddd,** where the number of digits following the decimal point is equal to the precision specification. If the precision is missing, it is taken as 6. If the precision is explicitly zero, no decimal point appears. If a decimal point appears, at least one digit appears before it.

g, G - the double argument is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style **e** will be used only if the exponent resulting from the conversion is less than -4 or

greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

n - the argument is taken to be an int * pointer to an integer into which is written the number of characters written to the output stream so far by this call to **printf**. No argument is converted.

p - the argument is taken to be a void * pointer to an object. The value of the pointer is converted to a sequence of printable characters, and printed as a hexadecimal number with the number of digits printed being determined by the field width.

s - the argument is taken to be a char * pointer to a string. Characters from the string are written up to, but not including, the terminating NUL, or until the number of characters indicated by the precision are written. If the precision is missing, it is taken to be arbitrarily large, so all characters before the first NUL are printed.

If the character after **'%'** is not a valid conversion character, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using **%s** conversion or any pointer using **%p** conversion), unpredictable results will occur.

A nonexistent or small field width does not cause truncation of a field; if the result is wider than the field width, the field is expanded to contain the conversion result.

RETURNS

printf returns the number of characters transmitted, or a negative number if a write error occurs.

NOTES

A call with more conversion specifiers than argument variables will cause unpredictable results.

EXAMPLE

To print **arg**, which is a double with the value 5100.53:

```
printf("%08.2f\n", arg);
printf("%*.*f\n", 08, 2, arg);
```

both forms will output:

SEE ALSO

sprintf

NOTES

printf is packaged in both the integer library and the floating point library. The functionality of the integer only version of **printf** is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of **printf**, the following conversion specifiers are invalid: **e, E, f, g** and **G**. The **L** modifier is also invalid.

If **printf** encounters an invalid conversion specifier, the invalid specifier is ignored and no special message is generated.

NAME

putchar - put a character to output stream

SYNOPSIS

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

FUNCTION

putchar copies c to the user specified output stream.

You must rewrite **putchar** in either C or assembly language to provide an interface to the output mechanism to the C library.

RETURNS

putchar returns c. If a write error occurs, **putchar** returns EOF.

EXAMPLE

To copy input to output:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

SEE ALSO

putc

NOTES

putchar is packaged in the integer library.

NAME

puts - put a text line to output stream

SYNOPSIS

```
#include <stdio.h>
int puts(s)
    const char *s;
```

FUNCTION

puts copies characters from the buffer starting at s to the output stream and appends a newline character to the output stream.

puts uses **putchar** to output each character. The terminating NUL is not copied.

RETURNS

puts returns zero if successful, or else nonzero if a write error occurs.

EXAMPLE

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

SEE ALSO

gets

NOTES

puts is packaged in the integer library.

NAME

rand - generate pseudo-random number

SYNOPSIS

```
#include <stdlib.h>
int rand()
```

FUNCTION

rand computes successive pseudo-random integers in the range [0, 32767], using a linear multiplicative algorithm which has a period of 2 raised to the power of 32.

EXAMPLE

```
int dice()
    {
    return (rand() % 6 + 1);
    }
```

RETURNS

rand returns a pseudo-random integer.

SEE ALSO

srand

NOTES

rand is packaged in the integer library.

NAME

realloc - reallocate space on the heap

SYNOPSIS

```
#include <stdlib.h>
void *realloc(ptr, nbytes)
    void *ptr;
    unsigned int nbytes;
```

FUNCTION

realloc grows or shrinks the size of the cell pointed to by ptr to the size specified by nbytes. The contents of the cell will be unchanged up to the lesser of the new and old sizes. The cell pointer ptr must have been obtained by an earlier **calloc**, **malloc**, or **realloc** call; otherwise the heap will become corrupted.

RETURNS

realloc returns a pointer to the start of the possibly moved cell if successful. Otherwise **realloc** returns NULL and the cell and ptr are unchanged. The pointer returned may be assigned to an object of any type without casting.

EXAMPLE

To adjust **p** to be **n** doubles in size:

```
p = realloc(p, n * sizeof(double));
```

SEE ALSO

calloc, free, malloc

NOTES

realloc is packaged in the integer library.

NAME

sbreak - allocate new memory

SYNOPSIS

```
/*  no header file need be included */
void *sbreak(size)
    unsigned int size;
```

FUNCTION

sbreak modifies the program memory allocation as necessary, to make available at least size contiguous bytes of new memory, on a storage boundary adequate for representing any type of data. There is no guarantee that successive calls to **sbreak** will deliver contiguous areas of memory.

RETURNS

sbreak returns a pointer to the start of the new memory if successful; otherwise the value returned is NULL.

EXAMPLE

To buy space for an array of symbols:

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
    remark("not enough memory!", NULL);
```

NOTES

sbreak is packaged in the integer library.

sbreak is an extension to the proposed ANSI C standard.

NAME

scanf - read formatted input

SYNOPSIS

```
#include <stdio.h>
int scanf(fmt, ...)
    const char *fmt;
```

FUNCTION

scanf reads formatted input from the output stream using the format string at fmt and the arguments specified by ..., as described below.

scanf uses **getchar** to read each character.

The behavior is unpredictable if there are insufficient argument pointers for the format. If the format string is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

Format Specifiers

The format string may contain:

- Any number of spaces, horizontal tabs, and newline characters which cause input to be read up to the next non-whitespace character, and
- Ordinary characters other than '**%**' which must match the next character of the input stream.

Each <conversion specification>, the definition of which follows, consists of the character '**%**', an optional assignment-suppressing character '*****', an optional maximum field width, an optional **h**, **l** or **L** indicating the size of the receiving object, and a <conversion character>, described below.

A conversion specification directs the conversion of the next input field. The result is placed in the object pointed to by the subsequent argument, unless assignment suppression was indicated by a '*****'. An input field is a string of non-space characters; it extends to the next conflicting character or until the field width, if specified, is exhausted.

The conversion specification indicates the interpretation of the input field; the corresponding pointer argument must be a restricted type. The <conversion character> is

one of the following:

% - a single **%** is expected in the input at this point; no assignment occurs.

If the character after '**%**' is not a valid conversion character, the behavior is undefined.

c - a character is expected; the subsequent argument must be of type pointer to char. The normal behavior (skip over space characters) is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is specified, the corresponding argument must refer to a character array; the indicated number of characters is read.

d - a decimal integer is expected; the subsequent argument must be a pointer to integer.

e, f, g - a float is expected; the subsequent argument must be a pointer to float. The input format for floating point numbers is an optionally signed sequence of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e**, followed by an optionally signed integer.

i - an integer is expected; the subsequent argument must be a pointer to integer. If the input field begins with the characters **0x** or **0X**, the field is taken as a hexadecimal integer. If the input field begins with the character **0**, the field is taken as an octal integer. Otherwise, the input field is taken as a decimal integer.

n - no input is consumed; the subsequent argument must be an int * pointer to an integer into which is written the number of characters read from the input stream so far by this call to **scanf**.

o - an octal integer is expected; the subsequent argument must be a pointer to integer.

p - a pointer is expected; the subsequent argument must be a void * pointer. The format of the input field should be the same as that produced by the **%p** conversion of **printf**. On any input other than a value printed earlier during the same program execution, the behavior of the **%p** conversion is undefined.

s - a character string is expected; the subsequent argument must be a char *pointer to an array large enough to hold the string and a terminating NUL, which will be added automatically. The input field

is terminated by a space, a horizontal tab, or a newline, which is not part of the field.

u - an unsigned decimal integer is expected; the subsequent argument must be a pointer to integer.

x - a hexadecimal integer is expected; a subsequent argument must be a pointer to integer.

[- a string that is not to be delimited by spaces is expected; the subsequent argument must be a char * just as for **%s**. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex '^', the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. A NUL character will be appended to the input.

The conversion characters **d, i, o, u** and **x** may be preceded by **l** to indicate that the subsequent argument is a pointer to long int rather than a pointer to int, or by **h** to indicate that it is a pointer to short int. Similarly, the conversion characters **e** and **f** may be preceded by **l** to indicate that the subsequent argument is a pointer to double rather than a pointer to float, or by **L** to indicate a pointer to long double.

The conversion characters **e, g** or **x** may be capitalized. However, the use of upper case has no effect on the conversion process and both upper and lower case input is accepted.

If conversion terminates on a conflicting input character, that character is left unread in the input stream. Trailing white space (including a newline) is left unread unless matched in the control string. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** conversion.

RETURNS

scanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if end of file is encountered before the first conflict or conversion.

EXAMPLE

To be certain of a dubious request:

```
printf("are you sure?");
if (scanf("%c", &ans) && (ans == 'Y' || ans == 'y'))
        scrog();
```

SEE ALSO

sscanf

NOTES

scanf is packaged in both the integer library and the floating point library. The functionality of the integer only version of **scanf** is a subset of the functionality of the floating point version. The integer only version cannot read or manipulate floating point numbers. If your programs call the integer only version of **scanf**, the following conversion specifiers are invalid: **e, f, g** and **p.** The **L** flag is also invalid.

If an invalid conversion specifier is encountered, it is ignored.

NAME

setjmp - save calling environment

SYNOPSIS

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

FUNCTION

setjmp saves the calling environment in env for later use by the **longjmp** function.

Since **setjmp** manipulates the stack, it should never be used except as the single operand in a switch statement.

RETURNS

setjmp returns zero on its initial call, or the argument to a **longjmp** call that uses the same env.

EXAMPLE

To call **anyevent** until it returns 0 or 1 and calls **longjmp**, which will then start execution at the function **event0** or **event1:**

```
static jmp_buf ev[2];
switch (setjmp(ev[0]))
    {
case 0:          /* registered */
        break;
default:         /* event 0 occurred */
        event0();
        next();
    }
switch (setjmp(ev[1])
    {
    case 0:          /* registered */
        break;
    default:         /* event 1 occurred */
        event1();
        next();
    }
next();
    ...
next()
    {
    int i;
    for (; ;)
        {
        i = anyevent();
```

```
        if (i == 0 || i == 1)
            longjmp(ev[i]);
        }
    }
```

SEE ALSO

longjmp

NOTES

setjmp is packaged in the integer library.

NAME

sin - sine

SYNOPSIS

```
#include <math.h>
double sin(x)
    double x;
```

FUNCTION

sin computes the sine of x, expressed in radians, to full double precision. If the magnitude of x is too large to contain a fractional quadrant part, the value of **sin** is 0.

RETURNS

sin returns the closest internal representation to **sin(x)** in the range [-pi/2, pi/2], expressed as a double floating value. A large argument may return a meaningless result.

EXAMPLE

To rotate a vector through the angle **theta:**

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

cos, tan

NOTES

sin is packaged in the floating point library.

NAME

sinh - hyperbolic sine

SYNOPSIS

```
#include <math.h>
double sinh(x)
    double x;
```

FUNCTION

sinh computes the hyperbolic sine of x to full double precision.

RETURNS

sinh returns the closest internal representation to **sinh(x)**, expressed as a double floating value. If the result is too large to be properly represented, **sinh** returns zero.

EXAMPLE

To obtain the hyperbolic sine of complex **z**:

```
typedef struct
    {
    double x, iy;
    }complex;
complex z;

z.x = sinh(z.x) * cos(z.iy);
z.iy = cosh(z.x) * sin(z.iy);
```

SEE ALSO

cosh, exp, tanh

NOTES

sinh is packaged in the floating point library.

NAME

sprintf - output arguments formatted to buffer

SYNOPSIS

```
#include <stdio.h>
int sprintf(s, fmt, ...)
    char *s;
    const char *fmt;
```

FUNCTION

sprintf writes formatted to the buffer pointed at by s using the format string at fmt and the arguments specified by ..., in exactly the same way as **printf**. See the description of the **printf** function for information on the format conversion specifiers. A NUL character is written after the last character in the buffer.

RETURNS

sprintf returns the numbers of characters written, not including the terminating NUL character.

EXAMPLE

To format a double at **d** into **buf**:

```
sprintf(buf, "%10f\n", d);
```

SEE ALSO

printf

NOTES

sprintf is packaged in both the integer library and the floating point library. The functionality of the integer only version of **sprintf** is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of sprintf, the following conversion specifiers are invalid: e, E, f, g and G. The L flag is also invalid.

NAME

sqrt - real square root

SYNOPSIS

```
#include <math.h>
double sqrt(x)
    double x;
```

FUNCTION

sqrt computes the square root of x to full double precision.

RETURNS

sqrt returns the nearest internal representation to **sqrt(x)**, expressed as a double floating value. If x is negative, **sqrt** returns zero.

EXAMPLE

To use **sqrt** to check whether **n > 2** is a prime number:

```
if (!(n & 01))
    return (NOTPRIME);
sq = sqrt((double)n);
for (div = 3; div <= sq; div += 2)
    if (!(n % div))
        return (NOTPRIME);
return (PRIME);
```

NOTES

sqrt is packaged in the floating point library.

NAME

srand - seed pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>
void srand(nseed)
    unsigned char nseed;
```

FUNCTION

srand uses nseed as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand.** If **srand** is called with the same seed value, the sequence of pseudo-random numbers will be repeated. The initial seed value used by **rand** and **srand** is 0.

RETURNS

Nothing.

EXAMPLE

To set up a new sequence of random numbers:

```
srand(103);
```

SEE ALSO

rand

NOTES

srand is packaged in the integer library.

NAME

sscanf - read formatted input from a string

SYNOPSIS

```
#include <stdio.h>
int sscanf(s, fmt, ...)
    char *s;
    const char *fmt;
```

FUNCTION

sscanf reads formatted input from the NUL-terminated string pointed at by s using the format string at fmt and the arguments specified by ..., in exactly the same way as **scanf**. See the description of the **scanf** function for information on the format conversion specifiers.

RETURNS

sscanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if the end of the string is encountered before the first conflict or conversion.

SEE ALSO

scanf

NOTES

sscanf is packaged in both the integer library and the floating point library. The functionality of the integer only version of **sscanf** is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of **sscanf**, the following conversion specifiers are invalid: **e**, **f**, **g** and **p**. The **L** flag is also invalid.

NAME

strcat - concatenate strings

SYNOPSIS

```
#include <string.h>
char *strcat(s1, s2)
    char *s1;
    const char *s2;
```

FUNCTION

strcat appends a copy of the NUL terminated string at s2 to the end of the NUL terminated string at s1. The first character of s2 overlaps the NUL at the end of s1. A terminating NUL is always appended to s1.

RETURNS

strcat returns **s1**.

EXAMPLE

To place the strings "first string, second string" in **buf[]**:

```
buf[0] = '\0';
strcpy(buf, "first string");
strcat(buf, ", second string");
```

SEE ALSO

strncat

NOTES

There is no way to specify the size of the destination area to prevent storage overwrites.

strcat is packaged in the integer library.

NAME

strchr - scan string for first occurrence of character

SYNOPSIS

```
#include <string.h>
char *strchr(s, c)
    const char *s;
    int c;
```

FUNCTION

strchr looks for the first occurrence of a specific character c in a NUL terminated target string s.

RETURNS

strchr returns a pointer to the first character that matches c, or NULL if none does.

EXAMPLE

To map **keystr[]** characters into **subst[]** characters:

```
    if (t = strchr(keystr, *s))
        *s = subst[t - keystr];
```

SEE ALSO

memchr, strcspn, strpbrk, strrchr, strspn

NOTES

strchr is packaged in the integer library.

NAME

strcmp - compare two strings for lexical order

SYNOPSIS

```
#include <string.h>
int strcmp(s1, s2)
    const char *s1, *s2;
```

FUNCTION

strcmp compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at s1, the second at s2. The strings must match, including their terminating NUL characters, in order for them to be equal.

RETURNS

strcmp returns an integer greater than, equal to, or less than zero, according to whether s1 is lexicographically greater than, equal to, or less than s2.

EXAMPLE

To look for the string "include":

```
if (strcmp(buf, "include") == 0)
    doinclude();
```

SEE ALSO

memcmp, strncmp

NOTES

strcmp is packaged in the integer library.

NAME

strcpy - copy one string to another

SYNOPSIS

```
#include <string.h>
char *strcpy(s1, s2)
    char *s1;
    const char *s2;
```

FUNCTION

strcpy copies the NUL terminated string at s2 to the buffer pointed at by s1. The terminating NUL is also copied.

RETURNS

strcpy returns s1.

EXAMPLE

To make a copy of the string **s2** in **dest:**

```
strcpy(dest, s2);
```

SEE ALSO

memcpy, strncpy

NOTES

There is no way to specify the size of the destination area, to prevent storage overwrites.

strcpy is packaged in the integer library.

NAME

strcspn - find the end of a span of characters in a set

SYNOPSIS

```
#include <string.h>
unsigned int strcspn(s1, s2)
    const char *s1, *s2;
```

FUNCTION

strcspn scans the string starting at s1 for the first occurrence of a character in the string starting at s2. It computes a subscript i such that

- **s1[i]** is a character in the string starting at **s1**
- **s1[i]** compares equal to some character in the string starting at **s2**, which may be its terminating null character.

RETURNS

strcspn returns the lowest possible value of **i**. s1[i] designates the terminating null character if none of the characters in s1 are in s2.

EXAMPLE

To find the start of a decimal constant in a text string:

```
if (!str[i = strcspn(str, "0123456789+-")])
    printf("can't find number\n");
```

SEE ALSO

memchr, strchr, strpbrk, strrchr, strpn

NOTES

strcspn is packaged in the integer library.

NAME

strlen - find length of a string

SYNOPSIS

```
#include <string.h>
unsigned int strlen(s)
    const char *s;
```

FUNCTION

strlen scans the text string starting at s to determine the number of characters before the terminating NUL.

RETURNS

The value returned is the number of characters in the string before the NUL.

NOTES

strlen is packaged in the integer library.

NAME

strncat - concatenate strings of length n

SYNOPSIS

```
#include <string.h>
char *strncat(s1, s2, n)
    char *s1;
    const char *s2;
    unsigned int n;
```

FUNCTION

strncat appends a copy of the NUL terminated string at s2 to the end of the NUL terminated string at s1. The first character of s2 overlaps the NUL at the end of s1. n specifies the maximum number of characters to be copied, unless the terminating NUL in s2 is encountered first. A terminating NUL is always appended to s1.

RETURNS

strncat returns s1.

EXAMPLE

To concatenate the strings "day" and "light":

```
strcpy(s, "day");
strncat(s + 3, "light", 5);
```

SEE ALSO

strcat

NOTES

strncat is packaged in the integer library.

NAME

strncmp - compare two n length strings for lexical order

SYNOPSIS

```
#include <string.h>
int strncmp(s1, s2, n)
    const char *s1, *s2;
    unsigned int n;
```

FUNCTION

strncmp compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at s1, the second at s2. n specifies the maximum number of characters to be compared, unless the terminating NUL in s1 or s2 is encountered first. The strings must match, including their terminating NUL character, in order for them to be equal.

RETURNS

strncmp returns an integer greater than, equal to, or less than zero, according to whether s1 is lexicographically greater than, equal to, or less than s2.

EXAMPLE

To check for a particular error message:

```
if (strncmp(errmsg, "can't write output file", 23) == 0)
    cleanup(errmsg);
```

SEE ALSO

memcmp, strcmp

NOTES

strncmp is packaged in the integer library.

NAME

strncpy - copy n length string

SYNOPSIS

```
#include <string.h>
char *strncpy(s1, s2, n)
    char *s1;
    const char *s2;
    unsigned int n;
```

FUNCTION

strncpy copies the first n characters starting at location s2 into the buffer beginning at s1. n specifies the maximum number of characters to be copied, unless the terminating NUL in s2 is encountered first. In that case, additional NUL padding is appended to s2 to copy a total of n characters.

RETURNS

strncpy returns s1.

EXAMPLE

To make a copy of the string s2 in **dest:**

```
strncpy(dest, s2, n);
```

SEE ALSO

memcpy, strcpy

NOTES

If the string s2 points at is longer than n characters, the result may not be NUL-terminated.

strncpy is packaged in the integer library.

NAME

strpbrk - find occurrence in string of character in set

SYNOPSIS

```
#include <string.h>
char *strpbrk(s1, s2)
    const char *s1, *s2;
```

FUNCTION

strpbrk scans the NUL terminated string starting at s1 for the first occurrence of a character in the NUL terminated set s2.

RETURNS

strpbrk returns a pointer to the first character in s1 that is also contained in the set s2, or a NULL if none does.

EXAMPLE

To replace unprintable characters (as for a 64 character terminal):

```
while (string = strpbrk(string, "`{|}~"))
    *string = '@';
```

SEE ALSO

memchr, strchr, strcspn, strrchr, strspn

NOTES

strpbrk is packaged in the integer library.

NAME

strrchr - scan string for last occurrence of character

SYNOPSIS

```
#include <string.h>
char *strrchr(s, c)
    const char *s;
    int c;
```

FUNCTION

strrchr looks for the last occurrence of a specific character c in a NUL terminated string starting at s.

RETURNS

strrchr returns a pointer to the last character that matches c, or NULL if none does.

EXAMPLE

To find a filename within a directory pathname:

```
if (s = strrchr("/usr/lib/libc.user", '/')
    ++s;
```

SEE ALSO

memchr, strchr, strpbrk, strcspn, strspn

NOTES

strrchr is packaged in the integer library.

NAME

strspn - find the end of a span of characters not in set

SYNOPSIS

```
#include <string.h>
unsigned int strspn(s1, s2)
    const char *s1, *s2;
```

FUNCTION

strspn scans the string starting at s1 for the first occurrence of a character not in the string starting at s2. It computes a subscript i such that

- s1[i] is a character in the string starting at s1
- s1[i] compares equal to no character in the string starting at s2, except possibly its terminating null character.

RETURNS

strspn returns the lowest possible value of i. s1[i] designates the terminating null character if all of the characters in s1 are in s2.

EXAMPLE

To check a string for characters other than decimal digits:

```
if (str[strspn(str, "0123456789")])
    printf("invalid number\n");
```

SEE ALSO

memchr, strcspn, strchr, strpbrk, strrchr

NOTES

strspn is packaged in the integer library.

NAME

tan - tangent

SYNOPSIS

```
#include <math.h>
double tan(x)
    double x;
```

FUNCTION

tan computes the tangent of x, expressed in radians, to full double precision.

RETURNS

tan returns the nearest internal representation to **tan(x)**, in the range [-pi/2, pi/2], expressed as a double floating value. If the number in x is too large to be represented, **tan** returns zero. An argument with a large size may return a meaningless value, i.e. when **x/(2 * pi)** has no fraction bits.

EXAMPLE

To compute the tangent of **theta:**

```
y = tan(theta);
```

SEE ALSO

cos, sin

NOTES

tan is packaged in the floating point library.

NAME

tanh - hyperbolic tangent

SYNOPSIS

```
#include <math.h>
double tanh(x)
    double x;
```

FUNCTION

tanh computes the value of the hyperbolic tangent of x to double precision.

RETURNS

tanh returns the nearest internal representation to **tanh(x)**, expressed as a double floating value. If the result is too large to be properly represented, **tanh** returns zero.

EXAMPLE

To compute the hyperbolic tangent of **x**:

```
y = tanh(x);
```

SEE ALSO

cosh, exp, sinh

NOTES

tanh is packaged in the floating point library.

NAME

tolower - convert character to lowercase if necessary

SYNOPSIS

```
/*  no header file need be included */
int tolower(c)
    int c;
```

FUNCTION

tolower converts an uppercase letter to its lowercase equivalent, leaving all other characters unmodified.

RETURNS

tolower returns the corresponding lowercase letter, or the unchanged character.

EXAMPLE

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s)
        sum = sum * 16 + *s - '0';
    else
        sum = sum * 16 + tolower(*s) + (10 - 'a');
```

SEE ALSO

toupper

NOTES

tolower is packaged in the integer library.

NAME

toupper - convert character to uppercase if necessary

SYNOPSIS

```
/*  no header file need be included */
int toupper(c)
    int c;
```

FUNCTION

toupper converts a lowercase letter to its uppercase equivalent, leaving all other characters unmodified.

RETURNS

toupper returns the corresponding uppercase letter, or the unchanged character.

EXAMPLE

To convert a character string to uppercase letters:

```
for (i = 0; i < size; ++i)
    buf[i] = toupper(buf[i]);
```

SEE ALSO

tolower

NOTES

toupper is packaged in the integer library.

NAME

va_arg - get pointer to next argument in list

SYNOPSIS

```
#include <stdarg.h>
type va_arg(ap, type)
    va_list ap;
    type;
```

FUNCTION

The macro **va_arg** is an rvalue that computes the value of the next argument in a variable length argument list. Information on the argument list is stored in the array data object ap. You must first initialize ap with the macro **va_start**, and compute all earlier arguments in the list by expanding **va_arg** for each argument.

The type of the next argument is given by the type name type. The type name must be the same as the type of the next argument. Remember that the compiler widens an arithmetic argument to int, and converts an argument of type float to double. You write the type after conversion. Write int instead of char and double instead of float.

Do not write a type name that contains any parentheses. Use a type definition, if necessary, as in

```
typedef int (*Pfi)();
    /* pointer to function returning int */
    ...
fun_ptr = va_arg(ap, Pfi);
    /* get function pointer argument */
```

RETURNS

va_arg expands to an rvalue of type type. Its value is the value of the next argument. It alters the information stored in ap so that the next expansion of va_arg accesses the argument following.

EXAMPLE

To write multiple strings to a file:

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

main()
    {
    void strput();
```

```
    strput(pf, "This is one string\n",      \
        "and this is another...\n", (char *)0);
    }
void strput(FILE *pf, ...);
void strput(char *ptr, ....)
void strput(ptr)
    char *ptr;
    {
    char ptr;
    va_list va;

    if (!ptr)
        return;
    else
        {
        puts(ptr);
        va_start(va, ptr);
        while (ptr = va_arg(va, char *);
            puts(ptr);
        va_end(va);
        }
    }
```

SEE ALSO

va_end, va_start

NOTES

va_arg is a macro declared in the **<stdarg.h>** header file. You can use it with any function that accepts a variable number of arguments, by including **<stdarg.h>** with your program.

NAME

va_end - stop accessing values in an argument list

SYNOPSIS

```
#include <stdarg.h>
void va_end(ap)
    va_list ap;
```

FUNCTION

va_end is a macro which you must expand if you expand the macro **va_start** within a function that contains a variable length argument list. Information on the argument list is stored in the data object designated by ap. Designate the same data object in both **va_start** and **va_end.**

You expand **va_end** after you have accessed all argument values with the macro **va_arg,** before your program returns from the function that contains the variable length argument list. After you expand **va_end,** do not expand **va_arg** with the same ap. You need not expand **va_arg** within the function that contains the variable length argument list.

You must write an expansion of **va_end** as an expression statement containing a function call. The call must be followed by a semicolon.

RETURNS

Nothing. **va_end** expands to a statement, not an expression.

EXAMPLE

To write multiple strings to a file:

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

main()
    {
    void strput();
    strput(pf, "This is one string\n",      \
        "and this is another...\n", (char *)0);
    }
void strput(FILE *pf, ...);
void strput(char *ptr, ....)
void strput(ptr)
    char *ptr;
    {
    char ptr;
    va_list va;
```

```
if (!ptr)
    return;
else
    {
    puts(ptr);
    va_start(va, ptr);
    while (ptr = va_arg(va, char *);
        puts(ptr);
    va_end(va);
    }
}
```

SEE ALSO

va_arg, va_start

NOTES

va_end is a macro declared in the **<stdarg.h>** header file. You can use it with any function that accepts a variable number of arguments, by including **<stdarg.h>** with your program.

NAME

va_start - start accessing values in an argument list

SYNOPSIS

```
#include <stdarg.h>
void va_start(va_list ap, parmN)
    va_list ap;
    parmN;
```

FUNCTION

va_start is a macro which you must expand before you expand the macro **va_arg.** It initializes the information stored in the data object designated by ap. The argument parmN must be the identifier you declare as the name of the last specified argument in the variable length argument list for the function. In the function prototype for the function, parmN is the argument name you write just before the , ...

The type of parmN must be one of the types assumed by an argument passed in the absence of a prototype. Its type must not be float or char. Also, parmN cannot have storage class **register.**

If you expand **va_start,** you must expand the macro **va_end** before your program returns from the function containing the variable length argument list.

You must write an expansion of **va_start** as an expression statement containing a function call. The call must be followed by a semicolon.

RETURNS

Nothing. **va_start** expands to a statement, not an expression.

EXAMPLE

To write multiple strings to a file:

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

main()
    {
    void strput();
    strput(pf, "This is one string\n",      \
        "and this is another...\n", (char *)0);
    }
void strput(FILE *pf, ...);
void strput(char *ptr, ....)
```

```
void strput(ptr)
    char *ptr;
    {
    char ptr;
    va_list va;

    if (!ptr)
        return;
    else
        {
        puts(ptr);
        va_start(va, ptr);
        while (ptr = va_arg(va, char *);
            puts(ptr);
        va_end(va);
        }
    }
```

SEE ALSO

va_arg, va_end

NOTES

va_start is a macro declared in the **<stdarg.h>** header file. You can use it with any function that accepts a variable number of arguments, by including **<stdarg.h>** with your program.