PERKIN-ELMER

C PROGRAMMING

Manual

48-103 F00 R00

The Perkin-Elmer Corporation, Data Systems Group, 2 Crescent Place, Oceanport, New Jersey 07757

TABLE OF CONTENTS

CHAPTERS (Continued)

PREFACE

This manual describes the Perkin-Elmer C Language. Chapter 1 is an introduction to the C Language and the C Compiler. Chapter 2 introduces the EDITION VII compatible C run-time library (RTL). Chapter 3 presents the IDRIS compatible C RTL. The operating system interface library is described in Chapter 4, and Chapter 5 describes the machine interface library.

Appendix A introduces the input/output (I/O) interface to the OS/32 file system. Appendix B presents operating instructions for the C Compiler.

This manual is intended for use with software release R07.2 of OS/32 or EDITION VII for Perkin-Elmer Series 3200 processors, or IDRIS for Perkin-Elmer Series 7000 processors.

For information on the contents of all Perkin-Elmer 32-bit manuals, see the 32-Bit Systems User Documentation Summary.

CHAPTER 1
THE C LANGUAGE

TABLE OF CONTENTS

NAME:

Introduction - The C Compiler

FUNCTION:

The C compiler is a set of three programs that take as input, to the first of the programs, one or more files of C source code, and produce as output, from the last of the programs, assembler code that performs the semantic intent of the source code. Output from the files may be separately compiled, then combined at load time to form an executable program; or C subroutines can be compiled for later inclusion with other programs. One can also look on the compiler as a vehicle for implementing an instance of an abstract C machine, i.e., a machine that executes statements in the language defined by some standard. That standard is generally accepted to be Appendix A of Kernighan and Ritchie, "The C Programming Language", Prentice-Hall, Inc., 1978.

This section describes the current implementation as succinctly and as precisely as it is defined in the language standard. It is organized into loosely-coupled subsections, each covering a different aspect of the language. No serious attempt is made to be tutorial; the interested student is referred to Kernighan and Ritchie, then back to this section for a review of the differences.

The recommended order of reading is:

Introduction	This is it.
Syntax	How to spell the words and punctuate the statements.
Identifiers	The naming of things, the scope of names and their important attributes.
Declarations	How to introduce identifiers and associate attributes with them.
Initializers	How to specify the initial values of all sorts of data types.
Statements	How to specify the executable code that goes with a function name.
Expressions	The binding of operators, order of evaluation and coercion of types.
Constants	The kinds of expressions that can be evaluated at compile time.
Preprocessor	#define and #include expansion.

Style	Recommendations for what parts of the language to use, and what to avoid; how to format code.
Portability	Techniques for writing C that is maximally portable.
Differences	Comparative anatomy of this implementation, the language standard and other implementations.
Diagnostics	Things the compiler complains about.

SEE ALSO:

Documentation for proper compiler operation is found in Appendix B of this manual. The standard library of C callable functions is documented in the remaining sections of this manual. It is also recommended that the user be familiar with Appendix A, which describes the interface to the OS/32 file system.

NAME:

Syntax - Syntax Rules For C

FUNCTION:

At the lowest level, a C program is represented as a text file, consisting of lines each terminated by a newline character. Any characters between /* and */ inclusive, including newlines, are comments and are replaced by a single blank character. A newline preceded by '\' is discarded so that lines may be continued. The compiler cannot accommodate a text line larger than 512 characters, either before or after the processing of comments and continuations.

Text lines are broken into tokens, strings of characters possibly separated by white space. White space consists of one or more nonprintable characters, such as space, tab or newline; its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms:

Identifiers — consist of a letter or underscore, followed by zero or more letters, underscores or digits. Upper-case letters are distinct from lower-case letters; no more than eight characters are significant in comparing identifiers. More severe restrictions may be placed on external identifiers by the world outside the compiler (see Differences). There are also a number of identifiers reserved for use as keywords:

auto	extern	short
break	float	sizeof
case	for	static
char	goto	struct
continue	if	switch
default	int	typedef
do	long	union
double	register	unsigned
else	return	while

Numeric constants — consist of a decimal digit, followed by zero or more letters and digits. If the leading characters are "0x" or "0X", the constant is a hexadecimal literal and may contain the letters 'a' through 'f', in either case, to represent the digit values 10 through 15, respectively. Otherwise, a leading '0' implies an octal literal, which may contain the digits '8' and '9'. A nonzero leading digit implies a decimal literal.

Any of these forms may end in 'l' or 'L' to specify a long constant. The constant is also made long if:

a. a decimal literal cannot be properly represented as a signed integer, or

b. any other literal constant cannot be properly represented as an unsigned integer. Overflow is not diagnosed.

Floating literals

consist of a decimal integer part, a decimal point '.', a decimal fraction part, and an exponent, where an exponent consists of an 'e' or 'E' and an optionally signed ('+' or '-') decimal power of ten by which the integer part plus fraction part must be multiplied. Either the decimal point or the exponent may be omitted, but not both; either the integer part or the fraction part may be omitted, but not both. Any numeric constant that is not one of these literal forms is illegal, e.g.; "5ax3". A floating literal is of type double. Overflow is not diagnosed.

Character constants

consist of a single quote, followed by zero or more character literals, followed by a second single quote. A character literal consists of:

- any character except '\', newline or single quote, the value being the ASCII representation of that character,
- a \ followed by any character except newline, the value being the ASCII representation of that character, except that characters in the sequence <'b', 't', 'v', 'f', 'n', 'r', '(', '!', ')', '^'> have the ASCII values for the corresponding members of the sequence <backspace, horizontal tab, vertical tab, form feed, newline, carriage return, '{', '|', '}', '~'>, or,
- a '\' followed by one to three decimal digits, the value being the octal number represented by those digits.

A newline is never permitted inside quotes, except when escaped with a '\' for line continuation. The value of the constant is an integer base 256, whose digits are the character literal values. The constant is long if it cannot be properly represented as an unsigned integer. Overflow is not diagnosed.

String constants — just like character constants, except that double quotes '"' are used to delimit the string. The value is the (secret) name of a NUL-terminated array of characters, the elements initialized to the character literals in the string.

Punctuation — consists of predefined strings of one to three characters. The complete set of punctuation for C is:

```
!     *     :     =+    >)    \)
!=    +     ;     =-    >=    \^
%     ++    <     =/    >>    ^
&     ,     <<    =<<   ?     {
&&    -     <=    ==    [     |
(     --    =     =>>   ]     |)
(<    ->    =%    =^    \!    ||
(|    .     =&    =|    \!!   }
)     /     =*    >     \(    ~
```

Punctuation in the sequence <'(<', '(|', '>)', '\!', '\!!', '\(', '\)', '\^', '|)'> is entirely equivalent to the corresponding member of the sequence <'{', '[', '}', '|', '||', '{', '}', '~', ']'>.

The longest possible punctuation string is matched, so that "==+", for example, is recognized as "==", "+", and never as "=", "=+".

Other characters — such as '@' or '\' alone are illegal outside of character or string constants, and are diagnosed.

NOTATION:

Grammar, the rules by which the syntactic elements above are put together, permeates the remaining discussions. To avoid lengthy descriptions, some simple shorthand is used throughout this section to describe grammatical constructs.

A name enclosed in angle brackets, such as <statement>, is a "metanotion", i.e., some grammatical element defined elsewhere. Presumably, any sequence of tokens that meets the grammatical rules for that metanotion can be used in its place, subject to any semantic limitations explicitly stated. Just about any other symbol stands for itself, i.e., it must appear literally, like the semicolon in the following:

```
<expression> ;
```

Exceptions are the punctuation "[", "]", "]*", "{", "}", and "¦"; these have special meanings unless made literal by being enclosed in single quotes.

Brackets surround an element that may occur zero or one time. The optional occurrence of a label, for instance, is specified by:

```
[ <identifier> : ] <statement>
```

This means that the metanotion <identifier> may, but need not, appear before <statement>. If it does, it must be followed by a literal colon. To specify the optional, arbitrary repetition of an element, the notation "[]*" is used. A comma separated list of <id> metanotions, for example (i.e., one instance of <id> followed by zero or more repetitions), would be represented by:

```
<id> [, <id> ]*
```

Vertical bars are used to separate the elements in a list of alternatives, exactly one of which must be selected. The line:

```
char ¦ short ¦ long
```

requires the specification of any one of the three keywords listed. Such a list of alternatives is enclosed in braces in order to precisely delimit its scope. For example:

```
{ <decl> [ = ] <expr> ¦
  <decl> [ = ] <elist> }
```

emphasizes that a data initializer has the format given by the entirety of one of the two lines specified.

EXAMPLE:

Various ways of writing a ten are shown below:

```
integer   10, 012, 0Xa, '\n'
long      10L
double    10.0, 1e1, 1.0e+01
```

NAME:

Identifiers - Naming Things in C

FUNCTION:

Identifiers are used to give names to the objects created in a C program. Syntactically, an identifier is a sequence of letters, underscores and digits, starting with a nondigit. For the sake of comparisons, only the first eight characters are significant, so "summation" and "summations" are the same identifier. Externally published identifiers are typically even more restricted (see Differences).

There are several name spaces in a C program, so that the same identifier may have different meanings in the same extent of program text, depending on usage. Things such as struct or union tags, members of struct or union and labels are considered separately later on. The bulk of this discussion concerns the name space occupied by the names of objects that occupy storage at execution time.

Each such identifier acquires, from its usage in a C program, a precisely defined lexical scope, storage class, and type. The scope is the extent of program text over which the compiler knows that a given meaning holds for an identifier. The storage class determines both the lifetime of values assigned to an object and the extent of program text over which a given meaning holds for its identifier, whether the compiler knows it or not. The type determines what operations can be performed on an object and how its values are encoded. Needless to say, these important attributes often interact.

SCOPE:

There are two basic contexts in a C program: inside a program block and outside a program block. The program block can be the entire body of a defined function, including its argument declarations, or any contained region enclosed in braces "{}". In either case, the scope of an identifier depends strongly on what context it is first mentioned in.

If an identifier first appears outside a program block, its scope is from its first appearance to the end of the file, less any contained program blocks in which that identifier is explicitly declared to have a storage class other than extern; i.e., a local redeclaration. To legally appear outside a program block, an identifier must be:

- explicitly declared to be extern or static, or
- used in an initializer for an object of type pointer.

In the latter case, the identifier is implicitly declared to be extern, with type (tentatively) int.

If an identifier appears inside a program block and is explicitly declared to have a storage class other than extern, its scope is from that appearance to the end of that program block, less any local redeclarations.

The only other place an identifier may legally first appear is inside a program block, within an expression, where the name of a function is required. In this case, the identifier is implicitly declared to be extern, with type function returning (tentatively) int.

In short, locals remain local while externals are made known as globally as possible, without requiring the compiler to back up over the text.

STORAGE CLASSES:

A variety of storage classes are available. Depending on context, there may be subtle differences within storage classes.

extern — outside a program block, means that the name should be published for common use among any of the files composing the program that also publish the same name. The published name may be shortened to as little as six significant characters, and/or compressed to one case, depending on the target operating system; so while the compiler distinguishes "Counted" and "counter", subsequent processing of the compiled text may not. Inside a program block, extern merely emphasizes that an earlier definition in a containing block holds, if any was made. If no earlier definition was made, then the name is published as above. The lifetime of extern objects is the duration of program execution.

static — outside a program block, means that the name should not be published outside the file. Inside a program block, static means that the identifier names an object known only within the program block, less any local redeclarations. The lifetime of static objects is the duration of program execution, so the value of a local static is retained between invocations of the program block that knows about it.

auto — can only be declared inside a program block and means that the identifier names an object known only within the program block, less any local redeclarations. The lifetime of auto objects is the time between each entry and exit of the program block, so the value of an auto is lost between invocations of the program block that knows about it. Multiple instances of the same auto may exist simultaneously, one instance for each dynamic activation of its program block.

register — can only be declared inside a program block, and means much the same as auto, except that:

- efficient storage, such as the machine's fast registers, should be favored to hold the object,
- the address of the object cannot be taken.

It is not considered an error to declare more objects of class register than can be accommodated; excess ones are simply taken as auto. The lifetime of register objects is the same as auto objects. An argument declared to be of class register is copied into a fast register on entry to the function. Currently, all implementations support at least three simultaneous register declarations, none of which hold an object larger than int.

typedef — means that the name should be recognized as a type specifier, not associated with any object. Lifetime is, therefore, irrelevant. Redeclaring a typedef in a contained program block is permissible but mildly perilous.

TYPE:

All types in C must be built from a fixed set of basic types: the integer forms char unsigned char, short, unsigned short, long and unsigned long; and the floating types float and double. The type int is a synonym either for short or long integer, depending on the size of pointers on the target machine; char, short, int and long are signed unless explicitly declared to be unsigned. From these, the composite forms struct, union, bitfield, pointer to, array of and function returning are derived. Recursive application of the rules for deriving composite types lead to a large, if not truly infinite, assortment of types.

[unsigned] char

is a byte integer; it should be large enough to hold all of the characters in the machine's character set. Printable characters and common white space codes are small, positive integers or zero.

[unsigned] short

is typically a two-byte integer; it should be large enough to hold reasonable counts.

[unsigned] int

is either a short or a long, depending on the machine. It should be large enough to count all the bytes in the machine's address space.

[unsigned] long

is typically a four-byte integer.

float

is a floating number of short precision, typically four bytes.

double

is a floating number of longer precision than float, if possible, typically eight bytes. Also known as "long float".

struct

is a sequence of one or more member declarations, with holes as needed to keep everything on proper storage boundaries for the machine. There are contexts in which a struct may have unknown content. Members may be any type but function returning, array of unknown size and struct of unknown content.

union

is an alternation of one or more members, the union being large enough and aligned well to accept any of its member types. Members may be any type but function returning, array of unknown size and struct of unknown content. A union of unknown content is treated just like a struct of unknown content.

bitfield

is a contiguous subfield of an unsigned int, always declared as a member of struct. It participates in expressions much like an unsigned int, except that its address may not be taken.

pointer to

is an unsigned int that is used to hold the address of some object. No C object will ever have an address of zero.

array of

is a repetition of some type, whose size is either a compile-time constant or unknown. Any type but function returning, array of unknown size and struct of unknown content may be used in an array.

function returning

is a body of executable text whose invocation returns the value of some type. Only the basic types or pointer to may be returned by a function.

OTHER NAME SPACES:

struct or union tags have a scope that extends from first appearance through the end of the program file; they may not be redefined. struct tags are a separate name space and union tags are a separate name space.

Labels in a function body have a scope that extends from first appearance, in a goto or as a statement label, through the end of the function body; they may not be redefined within that scope. Labels are a separate name space.

Members of a struct or union have a scope that extends from first appearance in the content definition of the struct or union, through the end of the program file. They may not be redefined within any struct or union, unless the new definition calls for the same type and offset. (As a compile-time option, each struct or union may be given its own name space.)

NAME:

Declarations - Declaring Names in C

FUNCTION:

Declarations form the backbone of a C program. They are used to associate a scope, storage class and type with most identifiers, to specify the initial values of objects named by identifiers, and to introduce the body of executable text associated with each function name. There are four types of declarations, external, structure, argument and local. The cast operator uses an abbreviated form of declaration to specify type.

EXTERNAL DECLARATIONS:

Declarations have one of the forms shown below:

```
[ <sc> ] [ <ty> ] <decl> <fn-body>
[ <sc> ] [ <ty> ] [ <decl> [ <dinit> ] [, <decl> [ <dinit> ] ]* ] ;
```

i.e., a storage class and type specifier, optionally followed by either a function body or a comma separated list of declarators <decl>, each optionally initialized, the list ending in a semicolon.

The storage class <sc> may be extern, static or typedef; the default is extern. The type <ty> may be

- a basic type,
- a struct or union declaration, (described below), or
- an identifier earlier declared to be typedef; the default is int.

The basic types may be written as:

```
{ [ unsigned ] [ char | short | long ] [ int ] |
  [ long ] float |
  double }
```

where long float is the same as double.

A <decl> is recursively defined as, in order of decreasing binding:

ident

ident is of type <ty>.

<decl> ([<id> [, <id>]*]) - <decl>

is of type function returning <ty>. The comma separated list of identifiers is used only if the declaration is associated with a function body.

<decl> '[' <const> ']'

where '[' and ']' signify actual brackets. <decl> is of type array of <ty>. <const> is the unsigned repetition count.

<decl> '[' ']'

where '[' and ']' signify actual brackets. <decl> is of type array of <ty>, of unknown size.

*<decl>

<decl> is of type pointer to <ty>.

(<decl>)

<ty> is redefined, inside the parentheses, as that type obtained for X if the entire declaration were rewritten with (<decl>) replaced by X.

The last rule has profound implications. It is intended, along with the rest of the <decl> notation, to permit declarators to be written much as they appear when used in expressions. Thus, "*" for "pointer to" corresponds to '*' for "indirect on", "()" for "function returning" corresponds to "()" for "called with", and "[]" for "array of" corresponds to "[]" for "subscripted with". Declarators must be read inside out, in the order in which the operators would be applied.

The two critical examples are:

```
int *fpi(); /* function returning pointer to int */
int (*pfi)();   /* pointer to function returning int */
```

Initializers come in two basic types: for objects of type function returning, and for everything else. The former is usually referred to as the definition of a function, i.e., the body of executable text associated with the function name. A typedef may not be initialized; a static must be initialized exactly once in the program file; an extern must be initialized exactly once among the entire set of files making up a C program. Function bodies ⟨fn-body⟩ are described in Statements; data initializers ⟨dinit⟩ are described in Initializers. For now, observe that each function body begins with an argument declaration list, and each program block within the function body begins with a local declaration list. Functions may only be declared to return a basic type or a pointer to some other type.

STRUCTURE DECLARATIONS:

If the type specifier in a declaration begins with struct or union, it must be followed by one of the following forms:

```
{ ⟨tag⟩ '{' ⟨dlist⟩ '}' |
  ⟨tag⟩ |
  '{' ⟨dlist⟩ '}' }
```

where '{' and '}' signify actual braces. The first form defines the content of the structure as ⟨dlist⟩ and associates the definition with the identifier ⟨tag⟩. The second form can be used to refer either to a structure of unknown content or as an abbreviation for an earlier instance of the first form. The last form is used to define content without defining a tag.

dlist is a sequence of one or more member declarations of the form:

```
[ ⟨ty⟩ ] ⟨sudecl⟩ [, ⟨sudecl⟩ ]* ;
```

where ⟨sudecl⟩ is one of the forms:

```
{ ⟨decl⟩ [ : ⟨width⟩ ] |
  : ⟨width⟩ }
```

⟨ty⟩ and ⟨decl⟩ are the same as for external declarations, except that the types function returning, array of unknown size and struct of unknown content may not be declared.

If the type is int or unsigned int, a bitfield specifier may follow <decl>, or stand alone. It consists of a colon ':' followed by a compile-time constant giving its width in bits. Adjacent <sudecl> declarators with bitfield specifiers are packed as tightly as possible into adjacent bitfields in an unsigned int; bitfield specifiers that stand alone call for unnamed padding. A new unsigned int is begun:

- for the first field specifier in a declaration,
- for the first bitfield specifier following a nonbitfield specifier,
- for a bitfield specifier that does not fit in the remaining space in the current unsigned int, or
- for a stand alone-field specifier whose width is zero, e.g.; ":0".

Bitfields are packed right-to-left, i.e., the least significant bit (LSB) is used first.

ARGUMENT DECLARATIONS:

A function initializer begins with an argument declaration list, which is a sequence of zero or more declarations of the form:

```
[ register ] [ <ty> ] <decl> [, <decl> ]* ;
```

<ty> and <decl> are the same as for external declarations, except that the types char (and possibly short), function returning, array of, struct and union are misleading, if used. On a function call, any integer type shorter than int is widened to int; function returning and array of become pointers; and struct or union cannot be sent, so any such declaration is a (possibly dangerous) reinterpretation of the actual arguments sent.

The only storage class that may be declared is register; default is normal argument. The default type for undeclared arguments is int.

LOCAL DECLARATIONS:

Each program block begins with a local declaration list, which is a sequence of zero or more declarations having one of the forms:

```
{ <lsc> [ <ty> ] [ <decls> ] ; |
  [ <lsc> ] <ty> [ <decls> ] ; }
```

where <decls> is:

<decl> [<linit>] [, <decl> [<linit>]]*

In other words, either the storage class <lsc> or type <ty> must be present.

The storage class <lsc> may be auto, register, extern, static or typedef; the default is auto. <ty> and <decl> are the same as for external declarations.

A static may be followed by a data initializer <linit>, just like the <dinit> of external declarations, described in Initializers. An auto or register may be followed by an linit of the form: optional '=', followed by any expression that may appear as the right operand of '=' in an expression in the same context. Such initializers for auto and register become code which is executed on each entry to the program block.

A register may hold only an object of size int (which includes unsigned and pointer). Anything larger than an unsigned int declared to be in a register is made an auto; anything declared smaller than int is taken as register int.

CASTS:

A cast is an operator that coerces a value to a specified type. It takes the form:

([<ty>] <a-decl>)

<ty> is the same as for external declarations, except that in conjuncion with <a-decl>, only the basic types and pointer to may be specified. <a-decl> is an abstract declarator, much like the <decl> used for external declarations, but with the identifier omitted. Therefore:

```
(int *) /* coerces to pointer to int */
(struct x *(*)())   /* coerces to pointer to function
                       returning pointer to struct x (!) */
```

"()" is always taken as function returning and never as (unnecessary) parentheses around the omitted identifier.

EXAMPLE:

Some simple declarations are:

```
char c;
int i, j;
long lo {37};
double df();
```

More elaborate declarations are:

```
int *fpi(); /* fpi is a function returning pointer to int */
typedef struct {
    double re, im;
    } COMPLEX;  /* COMPLEX is a synonym for the structure */
static COMPLEX *pc; /* pc is a static pointer to COMPLEX */
```

NAME:

Initializers - Giving Values to Data

FUNCTION:

As part of the declaration process, a data object can be given an initial value. This value is established at load time for objects with storage class extern or static, or on each entry to a program block, for objects with storage class auto or register. If no initializer is specified in a declaration, then an extern must be initialized in another declaration (not necessarily in the same file), a static outside a program block must be initialized in another declaration in the same file, a static inside a program block is set to all zeros, while the contents of auto or register storage classes are indeterminate.

The two basic formats for initializers are:

```
{ <decl> [ = ] <expr> |
  <decl> [ = ] <elist> }
```

where <elist> is

```
'{' [ <expr> | <elist> ] [, [ <expr> | <elist> ] ]* [, ] '}'
```

i.e.; a comma-separated list of expressions and lists, each list enclosed in braces. Note that a trailing comma is explicitly permissible in an elist.

Auto and register declarations with initializers behave much like assignment statements. Only scalar variables may be initialized, but the initializer may be any expression that can appear to the right of an assignment operator with that variable on the left. An <elist> is never acceptable in an auto or register initializer.

The remaining discussion concerns initializers for objects with storage class extern or static.

A scalar object is initialized with one <expr>. If it is an integer type (char, short, int, long, or bitfield), <expr> must be an expression reducible at compile-time to an integer literal, i.e.; a constant expression as described in Constants. If the object is a floating type (float or double), <expr> must be a floating literal or a constant expression; a constant expression is converted to a floating literal by the compiler. The compiler does not perform even obvious arithmetic involving floating literals, other than to apply unary '+' or '-' operators.

A pointer is initialized with a constant expression or with the address of an external object, plus or minus a constant expression. Any constant other than zero is extremely machine-dependent, hence, this freedom should be exploited only by hardware interface code. If the address of an object appears in a pointer initializer and the object has not yet been declared, it is implicitly declared to be (tentatively) an external int.

A union is initialized with one expression; the first member of the union is taken as the object to be initialized.

A struct is initialized with either an expression or a list. If an expression is used, the first member of the struct is taken as the object to be initialized. If a list is used, the elements of the list are used to initialize corresponding members of the struct. If there are more members than initializers, excess members are initialized with zeros. It is an error for there to be more initializers in a list than members in the struct.

An array of known size is initialized much like a struct: an expression initializes the first element only, while a list initializes elements starting with the first. If there are more elements than initializers, excess elements are initialized with zeros. It is an error for there to be more initializers in the list than there are elements in the array.

An array of unknown size, however, cannot have an excess of initializers, as its multiplicity is determined by the number of initializers provided. After initialization, therefore, an array always has a known size.

By special dispensation, an array of characters may be initialized by a string literal. For example:

```
char a[] {"help"};  /* is the same as */
char a[5] {'h', 'e', 'l', 'p', '\0'};
```

Elaborate composite types, such as arrays of structs, are naturally initialized with lists of sublists, whose structure reflects the structure of the composite types being initialized. It is often permissible, however, to write an initializer by omitting braces around one or more sublists. In this case, a struct or array (sub)element uses only as many elements as it needs, leaving the rest for subsequent subelements.

In general, it is recommended that complex initializers either have a structure that exactly matches the object to be initialized, or have no internal structure at all. It is difficult to get either of these extremes correct; intermediate forms frequently defy analysis.

EXAMPLE:

```
char *p = "help";    /* p points at the string */
char a[] {"help"};   /* a contains five chars */
struct complex {
    float real, imag;
    } xx[3][2]
    { { {0, 0}, {0, 1}, {0, 2} },
      { {1, 0}, {1, 1}, {1, 2} },
      { {2, 0}, {2, 1}, {2, 2} } };
```

NAME:

Statements - The Executable Code

FUNCTION:

A C function definition consists of the function declaration proper, followed by any argument declarations, followed by a <program-block> which describes the action to be performed when the function is called. A <program-block> begins with a '{', optionally followed by a sequence of local declarations, optionally followed by a sequence of statements and ends with a '}'. In addition to the <program-block> just described, which may be used recursively whenever a <statement> is permitted, the following are the legal <statement>s of a C program:

<expression>;

An <expression> terminated by a semicolon is a statement that causes the <expression> to be evaluated and the result discarded. Assignments and function calls are simply special cases of the expression statement. It is considered an error if <expression> produces no useful side effect; "a = b;" is useful but "a + b;" is not.

;

A semicolon standing alone is a null statement. It does nothing.

if (<expression>) <statement> [else <statement>]

If <expression> evaluates to a nonzero of any type, the <statement> following it is evaluated and the else part, if present, is skipped; otherwise the <statement> following <expression> is skipped and the else part, if present, has its <statement> evaluated. As in all languages, each else part in a nested if <statement> is associated with the inner-most "un-else'd" if.

switch (<expression>) <statement>

If <expression>, converted to an int, matches the value associated with any of the case labels in the statement following, execution resumes immediately following the matching label. Otherwise, if the label "default" is present in the statement following, execution resumes immediately following it. Otherwise, execution resumes with the statement following the switch statement. The statement controlled by a switch is typically a <program-block>, but does not have to be.

case <value>: <statement>

The case <statement> may only occur within a switch <statement>, as described above. The value must be an int computable at compile-time and must not match any other case <value>s in the same switch.

default: <statement>

The default <statement> may only occur within a switch <statement>, as described above. It may occur, at most, once in any switch.

while (<expression>) <statement>

As long as <expression> evaluates to a nonzero of any type, <statement> is executed. <expression> is evaluated prior to each execution of the <statement>, plus one more time if it ever evaluates to zero. The <statement> can be executed zero or more times.

do <statement> while (<expression>);

The <statement> is executed and, as long as <expression> evaluates to nonzero of any type, the <statement> is repeated. The <statement> may be executed one or more times; <expression> is evaluated following each execution of the <statement>.

for (<ex1>; <ex2>; <ex3>) <statement>

<ex1>, <ex2> and <ex3> are all <expression>s. <ex1> is evaluated exactly once, and as long as <ex2> evaluates to nonzero, the <statement> is executed and <ex3> is evaluated. Thus, the for behaves much like the sequence {<ex1>; while (<ex2>) {<statement> <ex3>; }}

break;

A break <statement> causes immediate exit from the inner-most containing switch, while, do or for <statement>, i.e.; execution resumes with the <statement> following. A break <statement> may only occur inside a switch, while, do or for.

continue;

A continue <statement> causes immediate exit from the <statement> part of the inner-most containing while, do or for <statement>, i.e.; execution resumes with the test part of a while or do, or with the <ex3> part of a for. A continue <statement> may only occur inside a while, do or for.

goto <identifier>;

A goto <statement> causes execution to resume immediately following the <statement> labelled with the matching identifier contained within a common <program-block>. Such a labelled <statement> must be present.

<identifier>: <statement>

A label <statement> serves as a potential target for a goto, as described above. All labels within a given <program-block> must have unique <identifier>s.

return [<expression>];

If the <expression> is present, it is evaluated and coerced to the type returned by the function, then the function returns with that value. If the <expression> is absent, the function returns with an undefined value. There is an implicit return statement (with no defined value) at the end of each <program-block> at the outer-most level of a function definition.

NAME:

Expressions - Computing Values in C

FUNCTION:

C offers a rich collection of operators to specify actions on integers, floats, pointers and, occasionally, composite types. Operators can be classified as addressing, unary or binary. Addressing operators bind most tightly, left-to-right from the basic term outward; then all unary operators are applied right-to-left, beginning with (at most one) postfix "++" or "--"; finally, all binary operators are applied, binding either left-to-right or right-to-left and on a multilevel scale of precedence.

Parentheses may be used to override the default order of binding, without fear that redundant parentheses will alter the meaning of an expression, i.e.; f(p) is the same as f((p)) or (f(p)). The language makes few promises about the order of evaluation, however, or even whether certain redundant computations occur at all. Expressions with multiple side effects can be fragile, e.g.; *p++ = *++p can legitimately be evaluated in a number of incompatible ways.

Some operands must be in the class of "lvalues", i.e.; things that make sense on the left side of an assignment operator. An identifier is the simplest lvalue, but any expression that evaluates to a recipe for locating declared objects can also be an lvalue. All scalar expressions also have an "rvalue", i.e.; a thing that makes sense on the right side of an assignment operator. All lvalues are also rvalues.

Nearly all C operators deal only with scalar types, i.e.; the basic types, bitfield or pointer to. Where a scalar type is required and a composite type is present, the following implicit coercions are applied: array of... is changed to pointer to... with the same address value; function returning... is changed to pointer to function returning... with the same address value; structure or union is illegal.

ADDRESSING OPERATORS:

func([expr [, expr]*])

func must evaluate to type "function returning..." and is the function to be called to obtain the rvalue of the expression, which is of type... Any arguments are evaluated, in unspecified order, and fresh copies of their values are made for each function call (thus, the function may freely alter its arguments with limited repercussions). char or short expressions are widened to int and float to double; all arguments must be scalor.

No checking is made for mismatched arguments or an incorrect number of arguments, but no harm is done providing the highest numbered argument actually used and all its predecessors do correspond properly. Note that a function declared as returning anything smaller than an int actually returns int, while a function returning float actually returns double.

a [i]

is entirely equivalent to "*(a + i)", so the unary '*' and binary '+' should be examined for subtle implications. If a is of type array of... and i is of integer type, however, the net effect is to deliver the ith element of a, having type...

x.m

x must be an lvalue, which should be of type struct or union containing a member named m (m can never be an expression). The value is that of the m member of x, with type specified by m.

p->m

p must be coercible to a pointer, which should be a pointer to a struct or union containing a member named m (m can never be an expression). The value is that of the m member of the struct or union pointed at by p, with type specified by m.

UNARY OPERATORS:

*p

p must be of type pointer to... The value is the value of the object currently pointed at by p, with type...

&x

x must be an lvalue. The result is a pointer that points at x; the type is pointer to... for x of type...

+x

x must be of type integer or float. The result is an rvalue of the same value and type as x.

-x

x must be of type integer or float. The result is an rvalue which is the negative of x and the same type as x.

++x

x must be a scalar lvalue. x is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as x.

--x

x must be a scalar lvalue. x is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as x.

x++

x must be a scalar lvalue other than floating. x is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as x.

x--

x must be a scalar lvalue other than floating. x is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as x.

~x

x must be an integer. The result is the ones complement of x, having the same type as x.

!x

x must be scalar. The result is an integer 1 if x is zero; otherwise it is an integer 0.

((<a-type>)) x

<a-type> is any scalar type declaration with the identifier omitted, e.g.; (char *). The result is an rvalue obtained by coercing x to <a-type>. This operator is called a "cast" (see Declarations). Note that a cast to any type smaller than int is taken as (int), while (float) is taken as (double).

sizeof x, sizeof (<a-type>)

The result is an integer rvalue equal to the size in bytes of x or the size in bytes of an object of type <a-type>.

BINARY OPERATORS:

There is an implicit "widening" order among the arithmetic types, i.e.; char, unsigned char, short, bitfield (if int is equivalent to short), unsigned short, long, bitfield (if int is equivalent to long), unsigned long, float and double; double is the widest type. In general, the type of a binary operator is the wider of the types of its two operands, narrower operand being implicitly coerced to match the wider. If arithmetic is not done in place, as in i =+ j, then integer arithmetic is always performed on operands coerced by widening to at least int, and floating arithmetic is always performed on operands widened to double.

Coercions are made up from a series of transformations: A char or short becomes an int of the same value. Sign extension occurs for all int types not declared as unsigned; the latter are widened by zero fill on the left. An int is simply redefined as an unsigned, on twos complement machines at least, with no change in representation. Bitfields are unpacked into unsigned integers. An integer is converted to a double of the same numerical value, while a float is reformatted as a double of the same value, often simply by right fill with zeros.

Assignment may call for a narrowing coercion, which is performed by the following operations: A double is rounded to its nearest arithmetic equivalent in float format; conversion to integer involves discarding any fractional part, then truncating as need be on the left without regard to overflow. Similarly, integers are converted to narrower types by left truncation.

The binary operators are listed in descending order of binding, those with highest precedence first:

x*y

Both operands must be arithmetic (integer or float). The result is the product of x and y, with the type of the wider.

x/y

Both operands must be arithmetic. The result is the quotient of x divided by y, with the type of the wider. Precedence is the same as for *.

x%y

Both operands must be integer. The result is the remainder obtained by dividing x by y, with the type of the wider. Precedence is the same as for *.

x+y

If either operand is of type pointer to..., the other operand must be of type integer, which is first multiplied by the size in bytes of the type... then added to the pointer to produce a result of type pointer. Otherwise both operands must be arithmetic; the result is the sum of x and y, with the type of the wider.

x-y

If x is of type pointer to... and y is of type integer, y is first multiplied by the size in bytes of the type... then subtracted from x. If x is of type pointer to... and y is of type pointer to... and both point to types of the same size in bytes, then x is subtracted from y and the result divided by the size in bytes of the type... to produce an integer result. Otherwise both x and y must be arithmetic; the result is y subtracted from x, with the type of the wider. Precedence is the same as for +.

x<<y

Both operands must be integer. The result is x left shifted y places, with the type of x. No promises are made if y is large (compared to the number of bits in x) or negative.

x>>y

Both operands must be integer. The result is x right shifted y places, with the type of x. If the result type is unsigned, no sign extension occurs on the shift; if it is signed, sign extension does occur. No promises are made if y is large or negative. Precedence is the same as for <<.

x<y,x<=y, x>y, x>=y

If either operand is of type pointer to... and the other is of type integer, the integer is scaled as for addition fore the comparison is made. Otherwise if both operands are of type pointer to... the pointers are compared as unsigned integers. Otherwise both x and y must be arithmetic, and the narrower is widened to match the type of the wider before the comparison is made. The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x==y, x!=y

The operands are coerced as for <, then compared for equality (==) or inequality (!=). The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x&y

Both operands must be integer. The result is the bitwise and of x and y, with the type of the wider.

x^y

Both operands must be integer. The result is the bitwise exclusive or of x and y, with the type of the wider.

x|y

Both operands must be integer. The result is the bitwise inclusive or of x and y, with the type of the wider.

x&&y

Both operands must be scalar. If x is zero, the result is taken as integer 0 without evaluating y. Otherwise the result is integer 1 only if both x and y are nonzero.

x||y

Both operands must be scalar. If x is nonzero, the result is taken as integer 1 without evaluating y. Otherwise the result is integer 1 if either x or y is nonzero.

t?x:y

If t, which must be scalar, is nonzero the result is x coerced to the final type; otherwise the result is y coerced to the final type; exactly one of the two operands x and y is evaluated. The final type is pointer to... if either operand is pointer to... and the other is integer (the integer is not scaled). Otherwise if both operands are of type pointer to... the final type is the same as x. Otherwise both operands must be arithmetic and the final type is the wider of the two types.

x=y

Both operands must be scalar, and x must be an lvalue. y is coerced to the type of x and assigned to x. If x is a pointer to..., y may be a pointer to... or an integer (the integer is not scaled). Otherwise both operands must be of arithmetic type. The result is an rvalue equal to the value just assigned, having the type of x.

x*=y, x/=y, x%=y, x+=y, x-=y, x<<=y, x>>=y, x&=y, x^=y, x|=y

Each of the operations "x op= y" is equivalent to "x = x op y", except that x is evaluated only once and the type of "x op y" must be that of x, e.g., "x -= y" cannot be used if x and y are both pointers. The operators may also be written =op, for historical reasons, but in this form no whitespace may occur after the =. Precedence is the same as for =.

x,y

Both operands must be scalar. x is evaluated first, then y. The result is the value and type of y. Note that commas in an argument list to a function call are taken as argument separators, not comma operators. Thus, f(a,b,c) represents three arguments, while f(a,(b,c)) represents two, the second one being c (after b has been evaluated).

NAME:

Constants - Compile-Time Arithmetic

FUNCTION:

There are four contexts in a C program where expressions must be able to be evaluated at compile-time: the expression part of a #if preprocessor control line, the size of an array in a declaration, the width of a bitfield in a struct declaration and the label of a case statement. In many other contexts, the compiler endeavors to reduce expressions, but this is not mandatory except in the interest of efficiency.

The #if statement evaluates expressions using long integer arithmetic. No assigment operators, casts or sizeof operators may be used. The result is compared against zero. There is no guarantee that large numbers will be treated the same across systems, due to the variation in operand size, but this variation is expected to be minimal among longs.

Bitfield widths, array sizes and case labels are also computed using long integer arithmetic, but only the integer part (if smaller than long) is retained. Moreover, the sizeof operator is permitted in such expressions.

In all other expressions, the compiler applies a number of reduction rules to simplify expressions at compile-time. These include the following (assumed) identities:

```
x * 1 == x
x / 1 == x
x + 0 == x
x - 0 == x
x + y == y + x
(x + y) + z == x + (y + z)
x && true == x
x || false == x
false && x == <nothing>
true || x == <nothing>
(x + y) * z == x * z + y * z
```

plus a number of others. In other words, certain subexpressions may generate no code at all, if the operation is patently redundant. This is worth keeping in mind when writing input/ouput (I/O) drivers and other machine-dependent routines that are expected to produce useful side effects not obvious to the compiler.

The compiler does not perform common subexpression elimination, however, nor rearrange the order of computation between statements, so that a minimal determinism is assured.

To ensure that compile-time reductions occur, on the other hand, it is best to group constant terms within an expression so the compiler does not have to guess the proper rearrangement to bring constants together.

NAME:

Preprocessor - Lines That Begin With #

FUNCTION:

A preprocessor is used by the C compiler to perform #define, #include and other functions signalled by a control character, typically #, before actual compilation begins. A number of options can be specified at preprocess time by the use of flags whose effects are sometimes mentioned below, but which are more fully explained in the section for the pp command, described in Appendix B of/this manual.

Unless -c is specified, /* comments */ are replaced by a single space and any line that ends with a '\' is merged with its successor. If the first nonwhitespace character on the resultant line matches either the preprocessor control character or the secondary preprocessor control character, the line is taken as a command to the preprocessor; all other lines are either skipped or expanded as described below.

The following command lines are recognized by the preprocessor:

#define <ident> <defn>

defines the identifier <ident> to be the definition string <defn> that occupies the remainder of the line. Identifiers consist of one or more letters, digits and underscores '_', where the first character is a nondigit; only the first eight characters are used for comparing identifiers. A sequence of zero or more formal parameters, separated by commas and enclosed in parentheses, may be specified, provided that no whitespace occurs between the identifier and the opening parenthesis. The definition string begins with the first nonwhitespace character following the identifier or its parameter list, and ends with the last nonwhitespace character on the line. It may be empty. If an identifier is redefined, the new definition is pushed down on top of the older ones.

#undef <ident>

pops one level of definition for <ident>, if any. It is not considered an error to undef an undefined identifier.

#include <fname>

causes the contents of the file specified by <fname> to be lexically included in place of the command line. A filename can be a simple identifier, or an arbitrary string inside (literal) quotes "", or an arbitrary string inside (literal) angle brackets <>. In the last case, a series of standard prefixes is prepended to the filename, normally "" unless otherwise specified at invocation time, to locate the file in one of several places. Included files may contain further includes.

#ifdef <ident>

commences skipping lines if the identifier <ident> is not defined, otherwise, processing proceeds normally for the range of control of the #ifdef. The range of control is up to and including a balancing #endif command. An #else command encountered in the range of control of the #ifdef causes skipping to cease if it was in effect, or normal processing to change to skipping if skipping was not in effect. It is permissible to nest #ifdef and other #if groups; entire groups will be skipped if skipping is in effect at the start. Preprocessor commands such as #define and #include are not performed while skipping.

#ifndef <ident>

is the same as #ifdef, except that #ifndef commences skipping if the identifier is defined.

#if <expression>

is the same as #ifdef, except that the rest of the line is first expanded, then evaluated as a constant expression; if the expression is zero then skipping commences. An expression may contain parentheses, the unary operators +, -, ! and ~, the binary operators +, -, *, /, %, &, |, ^, <<, >>, <, ==, >, <=, =>, !=, && and ||, and the ternary operator ? :. The definitions and bindings of the operators match those for the C language, subject to the straint that only integer constants may be used as operands.

#line <num> <fname>

causes the line number used for diagnostic printouts to be set to num and the corresponding filename to be set to fname, if present. If no filename is specified, the filename used for diagnostic printouts is left unchanged. num must be a decimal integer.

#

is taken as an innocuous line, if empty. Anything else not recognized as a command causes a diagnostic.

Expansion of noncommand lines causes each defined identifier to be replaced by its definition string, then rescanned for further expansion. If the definition has formal parameters, and the next token on the line is a left parenthesis, then a group of actual parameters, inside balanced parentheses, must occur on the line; formal parameters with no corresponding actual parameters are replaced by null strings.

Note that no attempt is made to add whitespace, before or after replacement text, to avoid blurring of token boundaries, just as no parentheses are added to avoid bizarre arithmetic binding in expressions. No expansion occurs within "" or '' strings.

BUGS:

Circular definitions such as

```
#define x x
```

cause the preprocessor to behave in an unpredictable manner.

NAME:

Style - Rules for Writing Good C Code

FUNCTION:

C is too expressive a language to be used without discipline; it can rival APL in opacity or PL/I in variety. The following practices and restraints are recommended for writing good C code:

ORGANIZATION:

If a C program totals more than 500 lines of code, it should be split into files each no larger than 500 lines of code. As much as possible, related declarations should be packaged together, with as many of these declared static (LOCAL) as possible. Common definitions and type declarations should be grouped into one or more header files to be #included as need be with each file.

If any use is made of the standard library, its definitions should be included as well, as in the following:

```
/* GENERAL HEADER FOR FILE
 * copyright (c) 1981 by Whitesmiths, Ltd.
 */
#include <std.h>
#include "defs.h"

#define MAXN   100 /* definitions local to this file */
.
.
.
```

For the contents of <std.h>, including the definitions of LOCAL, etc., see std.h in Chapter 3.

Header files should be used to contain #defines, typedefs and declarations that must be known to all source files making up a program. They should not include any initializers, as these would be repeated by multiple inclusion. A good convention is to use upper-case letters for #define'd identifiers, as a warning to the reader that the language is being extended.

It is also good practice to explicitly import all external references needed in each function body by the use of extern (IMPORT) declarations. This not only documents any pathological connections, but also permits functions to be moved freely among files without creating problems.

Within a file, a good discipline is to put all data declarations first, then all function bodies in alphabetical order by function name. Data declarations are typically stored into logical groups, e.g.; all flags, all file control, etc.; an explanatory comment should precede each group of data and each function body, with a single blank line preceding the comment. If the body of a function is sufficiently complex, a good explanatory comment is the six or so lines of pseudo code that best summarize the algorithm. There is usually no need for additional comments, but if they are used, they are best placed to the right of the line being explained, separated by one tab stop from the end of the statement.

If the types provided in std.h are not sufficient to describe all the objects used in a program, then all other types needed should be provided by #defines or typedefs. All declarations should be typed, preferably with these defined types, to improve readability.

RESTRICTIONS:

The goto statement should never be used. The only case that can be made for it is to implement a multilevel break, which is not provided in C; but this seldom proves to be prudent in the long run. If a function has no goto statements, it has no need for labels.

Other constructs to avoid are the do-while statement, which inevitably evolves into a safer while or for statement, and the continue statement, which is typically just a poor way of avoiding the proper use of else clauses inside loops.

More than five levels of indenting (see FORMATTING below) is a sure sign that a subfuction should be split out, as is the case with a function body that goes much over a page of listing or requires more than six local variables.

The use of the quasi-Boolean operators &&, ||, !, etc., to produce integer ones and zeros should not be used to perform arithmetic, as shown below.

```
sum[i] = a[i] + b[i] + (10 <= sum[i - 1]);
```

Such practices, if used, should be commented, as should most tricky bit manipulations using &, | and ^.

Elaborate expressions involving ? and :, particularly multiple instances thereof, are often hard to read. Parenthesizing helps, but excessive use of parentheses is just as bad.

If the relational operators > and >= are avoided, then compound tests can be made to read like intervals along the number axis, as shown below which is demonstrably true when c is a digit.

```
if ('0' <= c && c <= '9')
.
.
.
```

FORMATTING:

While it may seem a trivial matter, the formatting of a C program can make all the difference between correct comprehension and repeated error. To get maximum benefit from support tools such as editors and cross referencers, one should apply formatting rules rigorously. The following rules of thumb have proved to be valuable.

Each external declaration should begin (with optional storage class and mandatory type) at the beginning of a line, immediately following its explanatory comment. All defining material, data initializers or function definitions, should be indented at least one tab stop, plus additional tab stops to reflect substructure. Tabs should be set uniformly every four to eight columns.

A function body, for instance, always looks like:

```
TYPE name(arg1, arg2, arg3)
TYPE1 arg1;
TYPE2 arg2, arg3;
{
<local declarations>

<statements>
}
```

This example assumes that arg2 and arg3 have the same type. If no <local declarations> are present, there is no empty line before <statements>.

<local declarations> consists of: extern (IMPORT), register (FAST), auto (no storage class specifier), static (INTERN) declarations, respectively; these are sorted alphabetically by type within storage class and alphabetically by name within type. Comma-separated lists may be used, as long as there are no initializers; an initialized variable should stand alone with its initializer. FAST and auto storage should be initialized at declaration time only if the value is not to change.

<statements> are formatted to emphasize control structure, according to the following basic patterns:

```
if (test)
    <statement>
if (test)
    {
    <statement>
    <statement>
    .
    .
    .
    }
else
    <statement>
if (test1)
    <statement>
else if (test2)
    <statement>
else if (test3)
    .
    .
    .
else
    <default statement>
switch (value)
    {
case A:
case B:
    <statement>
    .
    .
    .
    break;
case C:
    <statement>
    .
    .
    .
    break;
default:
    <statement>
    .
    .
    .
    }
while (test)
    <statement>
for (init; test; incr)
    <statement>
for (; test; incr)
    <statement>
for (init; ; )
    <statement>
FOREVER
    <statement>
return (expr);
```

Note that, with the exception of the else-if chain, each subordinate <statement> is indented one tab stop further to the right than its controlling statement. Without this, an else-if chain would be written:

```
if (test 1)
    <statement>
else
    if (test 2)
        <statement>
    else
        if (test 3)
            <statement>
        .
        .
        .
        else
            <default statement>
```

Within statements, there should be no empty lines, nor any tabs or multiple spaces imbedded in a line. Each keyword should be followed by a single space, and each binary operator should have a single space on each side. No spaces should separate unary or addressing operators from their operands. A possible exception to the operator rule is a composite constant, such as (GREEN|BLUE).

Parentheses should be used whenever there is a hint of ambiguity. Note, in particular, that & and | mix poorly with the relational operators, that the assigning operators are weaker than && and ||, and that << and >> are impossible to guess right. The worst case is shown below.

```
if ((a & 030) != 030)
.
.
.
```

which does entirely the wrong thing if the parentheses are omitted.

If an expression is too long to fit on a line (no more than 80 characters) it should be continued on the next line, indented one tab stop further than its start. A good rule is to continue only inside parentheses, or with a trailing operator on the preceding line, so that displaced fragments are more certain to cause diagnostics.

EXAMPLE:

A typical library function looks like this:

```
#include <std.h>

/*  CONVERT LONG TO BUFFER
 *  copyright (c) 1978 by Whitesmiths, Ltd.
 */
BYTES ltob(is, ln, base)
    FAST TEXT *is;
    LONG ln;
    BYTES base;
    {
    FAST TEXT *s;
    ULONG lb;

    s = is;
    if (ln < 0 && base == 0)
        {
        ln = -ln;
        *s++ = '-';
        }
    if (base == 0)
        base = 10;
    else if (base < 0)
        base = -base;
    lb = base;
    if (ln < 0 || lb <= ln)
        s =+ ltob(s, ln / lb, base);
    *s = ln % lb + '0';
    if ('9' < *s)
        *s =+ ('a' - ('9' + 1));
    return (s - is + 1);
    }
```

NAME:

Portability - Writing Portable Code

FUNCTION:

Writing highly portable C code is not difficult. While the features provided by the machine interface library may be convenient, they should be avoided whenever possible and clearly marked when it is impossible to avoid using them. When problems arise with C code that is machine-dependent, they can be extremely difficult to identify; and trying to keep them out of new code can often become counter-productive. The following set of rules eliminate nearly all machine dependencies.

Use the portable library. If at all possible, avoid using the C interface library.

If your program processes text files, assume that carriage returns, NULs, and other characters that do not print may disappear if written out and read back later. Assume that lseek will fail, even when it obviously should work on your system. Text lines may be as long as 512 characters, counting the terminating newline; but then they should never be longer than that. It is usually best not to depend on the presence of that trailing newline, if at all possible, in case the program is fed an unusually long line or a truncated last line.

If your program tries to process STDIN, STDOUT or STDERR as a binary file, assume that the data will be corrupted; binary files must be opened by name on most systems. The third argument to open and create should always be present and for binary files should always be nonzero. A third argument of 1 is always acceptable and will not lead to storage inefficiencies in the target file. The set of functions fopen, fcreate, getfiles, etc., were frozen before the text/binary dichotomy became apparent, so they work smoothly only on text files; getbfiles is a later addition. Performing a binary open, followed by a finit with third argument READ or BWRITE, does make the buffered I/O mechanism safely available for sequential binary I/O, however.

Many operating systems will pad a binary file with NULs, which are hard to detect in the interface code. Consequently, any program that does binary reads must be prepared to deal with trailing NULs. Treating NUL as end of file is best, whenever possible. Another aspect of this problem is that the length of a binary file is poorly determined on many systems; consequently, the ability to lseek relative to the end of a file is no longer supported in the portable specification.

The order in which bytes are stored for encoded arithmetic types varies. A long integer, '3210' for instance, can read out as '3210', '0123', or '2301' on three popular processors, where '0' is the least significant byte. The best rule is never to write a multibyte datum, unless it is an array of chars or unless it is

clearly understood that the resultant file will always be read into an identically declared datum on the same machine. Look for sizeof operators not connected with alloc calls; they are a sign of potential trouble. The library functions lstoi and itols are provided to ease transmission of two-byte integers among various implementations, while lstol and ltols provide a similar mechanism for four-byte quantities.

The portable specification states filenames should be no more elaborate than "xxxxxx.yy". Use uname to build temp file names and avoid wiring any other file names into code. In the same vein, external identifiers should be chosen for the worst case, i.e.; a system where only six characters in one case are retained. It is possible to have the compiler check the identifiers for conformance to this or similar constraints.

Declarations are designed to ease portability among machines with different data formats, but they must be used properly to do so. C is very tolerant to declarations for such things as: arguments to a function, values returned by a function, data held in registers and casts. Since all of these items are essentially rvalues, they are never smaller than an int, or a double if floating point, no matter how they are declared. The difference matters only when some lvalue enter, as when assigning to a register or argument, or taking the address of an argument.

Never assume that assigning to one of these items, or applying a cast such as (char), will perform any sort of truncation; it will not, at least not below integer. Storing a char in part of an int may work on some computers, but it will eventually cause trouble. Look for address of (unary &) operators applied to arguments and expect problems.

The standard header provides a constellation of defined types to stylize proper usage. Use the aliases for {char, short, long}: i.e.; {TINY, COUNT, LONG} or the unsigned versions {UTINY, UCOUNT, ULONG} regardless of what processor the program is run on, when you know the size of the datum being entered. If something must hold a pointer, declare it as such; if it must span most or all of the address space of the target machine, as a subscript for example, declare it as unsigned int, or BYTES. Note that case switch values are ints; hence, only short values are portable for case labels.

There are two other important flexible types besides BYTES: ARGINT and TEXT. ARGINT is used when talking about an argument that is known to have been widened to an integer; it should serve as a red flag that something special is happening. TEXT, on the other hand, is used heavily to ensure efficient code; it is declared in the header as either char or unsigned char, depending on which is more easily handled by the target machine. When using TEXT variables, the programmer must be careful to mask possible sign extensions, using BYTMASK, should other than ASCII characters or small positive integers be stored in them. This usage parallels the standard uncertainty of char variables in other implementations of C.

To ensure maximum portability between 16 and 32-bit pointer machines, the best mind set is that an int is not equal to a short and it is not equal to a long, but it can and will be equal to either some time or other. Avoid writing constants of fixed size, such as 0177777; instead write stretchable forms such as ~0 for the above. If a constant must be long on any machine, be sure to force it long; ~0 can be different from ~0L.

There will always be at least three registers available. These can hold anything up to an unsigned int and almost invariably offer substantial code space and execution time benefits if used for the most important variables. Pointers benefit particularly from being placed in registers. The assigning operators, such as =+ or +=, frequently lead to better code production. This is particularly true for the smaller data types such as char, since C is obliged to compute (c1 + c2) to int precision, but may do (c1 =+ c2) as a char operation. It is also possible to make code more efficient by writing expressions such as:

```
if (sizeof (int) == sizeof (short) && <short test> ||
    sizeof (int) == sizeof (long) && <long test>)
```

Only <short test> or <long test> will actually be generated as run-time code; the remainder is optimized out by the compiler.

NAME:

Differences - Comparative Anatomy

FUNCTION:

The definitive standard for C is Appendix A of Kernighan & Ritchie, as explained in the Introduction to this section. This implementation adheres closely to that standard, except for minor changes in emphasis. There are also several available implementations of C that differ in more important respects. A summary of standards is listed below.

THE STANDARD:

- This implementation includes the types unsigned [char short long], which are not yet in the standard.
- Backslash is used to continue strings in the standard; its use is generalized here.
- Character constants with more than one character are defined here, but not in the standard.
- All struct and union tags share the name space of all members of struct and union, in the standard; each kind of tag has its own name space here.
- This implementation permits, as an option, separate name spaces for each struct or union and much more rigorous checking of . and -> operators.
- A union may be initialized in this implementation.
- A preprocessor macro invocation, e.g., swap(a,b), must be written all on one line in this implementation.
- The sizeof operator is explicitly disallowed in #if expressions, in this implementation.

UNIX/V6:

- Not implemented in the UNIX/V6 compiler are: bitfields, short integers, unsigned integers, long integers, casts, unions, #if, #line, operators of the form op=, static external declaractions (local to a file) or register arguments.
- UNIX/V6 initializes a structure as if it were an array of integers.

UNIX/V7:

- Bitfields may not be initialized, in at least one of the UNIX/V7 compilers.
- Casts of the form (char) or (short) may actually truncate a value; they have no effect on ints in this implementation.
- The address of an array cannot be taken.
- Enumerated types, structure assignment and functions returning structs have been added in UNIX/V7 C.

NAME:

Diagnostics - Compiler Complaints

FUNCTION:

The first two passes of the compiler produce all user diagnostics, the initial (preprocessor) pass dealing with # control lines and lexical analysis, the next with everything else. If a pass produces diagnostics, later passes should not be run. Any compiler message containing an exclamation mark '!' or the word "panic" indicates problems with the compiler and they should be reported. A summary of the diagnostics that can be produced by erroneous C programs is listed below.

PREPROCESSOR DIAGNOSTICS:

- Bad #define - illegal define
- Bad #define arguments - cannot parse #define line
- Bad #include - illegal include
- Bad #line - illegal #line
- Bad #undef - illegal undef
- Bad #xxx - unrecognizable # control line
- Bad flag - see manual page for pp
- Bad macro arguments - cannot parse macro definitions
- Bad output file - cannot create output file
- Can't #include xxx - cannot open file specified in #include
- Can't open xxx - cannot open file specified as pp argument
- Illegal #if expression
- Illegal #if syntax
- Illegal ? : in #if
- Illegal character: x - not a recognizable token in C
- Illegal constant xxx - not a recognizable numeric form
- Illegal float constant
- Illegal number in #if

- Illegal operator in #if
- Illegal unary op in #if
- Misplaced #xxx - preprocessor control line out of place
- Missing) in #if
- Missing #endif - unbalanced #if, #ifdef, or #ifndef
- Missing */ - unbalanced /* comment
- String too long - more than 128 characters
- Too many -d arguments - more than 10 (see manual page for pp)
- Truncated line - more than 512 characters
- Unbalanced x - x is a delimiter: ', ", (, < or {

PASS 1 DIAGNOSTICS:

- Arithmetic type required - integer or floating
- Array size unknown
- Bad (declaration) - arguments inside () unrecognizable
- Bad field width - negative or larger than word size
- Bad flag - see manual page for pl
- Bad output file - cannot create output file
- Cannot initialize
- Constant required
- Declaration too complex - more than 5 modifiers.
- External name conflict - when truncated for output
- Function required - arguments declared, but no function body
- Function size undefined
- Illegal &
- Illegal =+
- Illegal assignment
- Illegal bitfield

- Illegal break
- Illegal case
- Illegal cast
- Illegal comparison
- Illegal continue
- Illegal default
- Illegal double initializer
- Illegal field
- Illegal field initializer
- Illegal indirection - unary "*" operator
- Illegal integer initializer
- Illegal member
- Illegal operand type
- Illegal pointer initializer
- Illegal return type
- Illegal selection
- Illegal storage class
- Illegal structure reference
- Illegal type moifier
- Illegal unsigned compare
- Incomplete declaration
- Integer type required
- Lvalue required - see Expressions
- Missing argument
- Missing expression
- Missing goto label
- Missing member name - identifier must follow . or ->
- No structure definition

- String initializer too long
- Structure size unknown
- Unexpected EOF
- Union size unknown
- Useless expression - result unused, no side effect

CHAPTER 2
EDITION VII COMPATIBLE
C RUN-TIME LIBRARY (RTL)

TABLE OF CONTENTS

NAME:

assert - Program Verification

SYNOPSIS:

```
#include <assert.h>

assert (expression)
```

DESCRIPTION:

Assert is a macro that indicates expression is expected to be true at this point in the program. It causes an exit (2) with a diagnostic comment on the standard output when expression is false (0). Compiling with the pp option, DNDEBUG effectively deletes assert from the program.

DIAGNOSTICS:

'Assertion failed: file f line n.' F is the source file and n the source line number of the assert statement.

NAME:

abs - Integer Absolute Value

SYNOPSIS:

```
abs(i)
```

DESCRIPTION:

abs returns the absolute value of its integer operand.

SEE ALSO:

floor for fabs

BUGS:

The magnitude of the largest negative integer is hardware dependent.

NAME:

atof, atoi, atol - Convert ASCII to Numbers

SYNOPSIS:

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION:

These functions convert a string pointed to by nptr to floating, integer and long integer representation, respectively. The first unrecognized character ends the string.

atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and atol recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO:

scanf

BUGS:

There are no provisions for overflow.

NAME:

crypt, setkey, encrypt - DES Encryption

SYNOPSIS:

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

DESCRIPTION:

crypt is the password encryption routine. It is based on the National Bureau of Standards (NBS) Data Encryption Standard, with variations intended to frustrate use of hardware implementations of the DES for key search.

The first argument to crypt is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The salt string is used to perturb the DES algorithm in one of 4,096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two charactrs are the salt itself.

The other entries provide access to the actual DES algorithm. The argument of setkey is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the encrypt entry is likewise a character array of length 64 containing 0's and 1's. The argumnet array is modified in place to a similar array representing the bits of the argument after having been sujected to the DES algorithm using the key set by setkey. If edflag is 0, the argument is encrypted; if nonzero, it is decrypted.

BUGS:

The return value points to static data whose content is overwritten by each call.

NAME:

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - Character Classification

SYNOPSIS:

```
#include <ctype.h>

isalpha(c)
.
.
.
```

DESCRIPTION:

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true and zero for false. isacii is defined on all integer values; the rest are defined only where isascii is true and on the single non-ASCII value end of file (EOF) (see stdio(2)).

isalpha	c is a letter
isupper	c is an upper-case letter
islower	c is a lower-case letter
isdigit	c is a digit
isalnum	c is an alphanumeric character
isspace	c is a space, tab, carriage return, newline or form feed
ispunct	c is a punctuation character (neither control nor alphanumeric)
isprint	c is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl	c is a delete character (0177) or ordinary control character (less than 040).
isascii	c is an ASCII character, code less than 0200.

NAME:

ecvt,fcvt,gcvt - Output Conversion

SYNOPSIS:

```
char*ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION:

ecvt converts the value to a NUL-terminated string of ndigit ASCII digits and returns a pointer there. The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by sign is nonzero, otherwise it is zero. The low-order digit is rounded.

fcvt is identical to evct, except fcvt has been rounded for FORTRAN F format output of the number of digits specified by ndigits.

gcvt converts the value to a NUL-terminated ASCII string in buf and returns a pointer to buf. It attempts to produce ndigit significant digits in FORTRAN F format, if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO:

printf

BUGS:

The return values point to static data whose content is overwritten by each call.

NAME:

exp,log,log10,pow,sqrt - exponential,logarithm,power,square root - Exponential Functions

SYNOPSIS:

```
#include<math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x,y)
double x,y;

double sqrt(x)
double x;
```

DESCRIPTION:

exp returns the exponential function of x.

log returns the natural logarithm of x; log 10 returns the base 10 logarithm.

pow returns x^y.

sqrt returns the square root of x.

SEE ALSO:

hypot, sinh,

DIAGNOSTICS:

exp and pow return a large value when the correct value would overflow; pow returns 0 and sets errno to EDOM when the second argument is negative and nonintegral and when both arguments are 0.

log returns 0 when x is zero or negative.

sqrt returns 0 when x is negative; errno is set to EDOM.

NAME:

fclose, fflush - Close or Flush a Stream

SYNOPSIS:

```
#include<stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION:

fclose causes any buffers for the named stream to be emptied, and the file to be closed. Buffers allocated by the standard input/output (I/O) system are freed.

fclose is performed automatically upon calling exit(5).

fflush causes any buffered data for the named output stream to be written to that file. The stream remains open.

SEE ALSO:

close(4), fopen, setbuf

DIAGNOSTICS:

These routines return EOF if stream is not associated with an output file or if buffered data cannot be transferred to that file.

NAME:

feof, ferror, clearerr, fileno - Stream Status Inquiries

SYNOPSIS:

```
#include<stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;
```

DESCRIPTION:

feof returns nonzero when EOF is read on the named input stream, otherwise zero.

ferror returns nonzero when an error has occurred reading or writing the named stream, otherwise zero. Unless cleared by clearerr, the error indication lasts until the stream is closed.

clrerr resets the error indication on the named stream.

fileno returns the integer file descriptor (fd) associated with the stream, see open(4).

These functions are implemented as macros; they cannot be redeclared.

SEE ALSO:

fopen, open(4)

NAME:

fabs, floor, ceil - Absolute Value, Floor, Ceiling Functions

SYNOPSIS:

```
#include<math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double(x);
```

DESCRIPTION:

fabs returns the absolute value x.

floor returns the largest integer not greater than x.

ceil returns the smallest integer not less than x.

SEE ALSO:

abs

NAME:

fopen, freopen, fdopen - Open a Stream

SYNOPSIS:

```
#include<stdio.h>

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

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

FILE *fdopen(fildes,type)
char *type;
```

DESCRIPTION:

fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

- "r" open for reading ASCII files
- "w" create for writing ASCII files
- "a" append: open for writing at end of file, or create for writing ASCII files
- "rb" open for reading binary files
- "wb" create for writing binary files
- "ab" append: open for writing at end, or create for writing binary files

freopen substitutes the named file in place of the open stream. It returns the original value of stream. The original stream is closed.

freopen is typically used to attach the preopened constant names, stdin, stdout, stderr, to specified files.

fdopen associates a stream with an fd obtained from open or creat(4). The type of the stream must agree with the mode of the open file.

SEE ALSO:

open(4), fclose, Appendix A

DIAGNOSTICS:

fopen and freopen return the pointer NULL if filename cannot be accessed.

BUGS:

fdopen is not portable to OS/32.

NAME:

frexp, ldexp, modf - Split into Mantissa and Exponent

SYNOPSIS:

```
double frexp(value,eptr)
double value;
int *eptr;

double ldexp(value,exp)
double value;

double modf(value,iptr)
double value, *iptr;
```

DESCRIPTION:

frexp returns the mantissa of a double value as a double quantity, x, of magnitude less than 1 and stores an integer n such that value = x*2**n indirectly through eptr.

ldexp returns the quantity value *2**exp.

modf returns the positive fractional part of value and stores the integer part indirectly through iptr.

NAME:

fseek, ftell, rewind - Reposition a Stream

SYNOPSIS:

```
#include <stdio.h>
fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)
```

DESCRIPTION:

fseek sets the position of the next input or output on the stream. The new position is at the signed distance offset bytes from the beginning, the current position, or the end of the file, according as ptrname has the value 0, 1 or 2.

fseek undoes any effects of ungetc(2).

ftell returns the current value of the offset relative to the beginning of the file associated with the named stream. It is measured in bytes on UNIX; on some other systems it is the only foolproof way to obtain an offset for fseek.

rewind (stream) is equivalent to fseek (stream, 0L, 0).

SEE ALSO:

lseek(4), fopen

DIAGNOSTICS:

fseek returns - 1 for improper seeks.

NAME:

getc, getchar, fgetc - Get Character or Word from Stream

SYNOPSIS:

```
#include<stdio.h>

int getc(stream)
FILE *stream;

int getchar()

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

DESCRIPTION:

getc returns the next character from the named input stream.

getchar() is identical to getc(stdin).

fgetc behaves like getc, but is a genuine function, not a macro; it may be used to save object text.

SEE ALSO:

fopen, putc, gets, scanf, fread, ungetc

DIAGNOSTICS:

These functions return the integer constant EOF at end of file or upon read error.

A stop with the message, "Reading bad file", means an attempt has been made to read from a stream that has not been opened for reading by fopen.

BUGS:

The end of file return from getchar is incompatible with that in earlier editions of UNIX.

Because it is implemented as a macro, getc treats a stream argument with side effects incorrectly. In particular, "getc(*f++);" doesn't work sensibly.

NAME:

getlogin - Get Login Name

SYNOPSIS:

```
char *getlogin();
```

DESCRIPTION:

getlogin returns a pointer to the login name for Multi-Terminal Monitor (MTM) users.

If getlogin is called within a process that is not attached to MTM, it returns NULL.

DIAGNOSTICS:

Returns NULL (0) if name not found.

BUGS:

The return values point to static data whose content is overwritten by each call.

NAME:

gets, fgets - Get a String from a Stream

SYNOPSIS:

```
#include<stdio.h>

char *gets(s)
char *s;

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

DESCRIPTION:

gets reads a string into s from the standard input stream stdin. The string is terminated by a newline character, which is replaced in s by a null character. gets returns its argument.

fgets reads n-1 characters, or up to a newline character, whichever comes first, from the stream into the string s. The last character read into s is followed by a null character. fgets returns its first argument.

SEE ALSO:

puts, getc, scanf, fread, ferror

DIAGNOSTICS:

gets and fgets return the constant pointer NULL upon end of file or error.

BUGS:

gets deletes a newline, fgets keeps it.

NAME:

hypot, cabs - Euclidean Distance

SYNOPSIS:

```
#include<math.h>

double hypot(x,y)
double x,y;

double cabs(z)
struct {double x,y;}z;
```

DESCRIPTION:

hypot and cabs return the following:

```
sqrt(x*x + y*y),
```

taking precautions against unwarranted overflows.

SEE ALSO:

exp for sqrt

NAME:

malloc, free, calloc - Main Memory Allocator

SYNOPSIS:

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

free(ptr)
char *ptr;

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

DESCRIPTION:

malloc and free provide a simple, general-purpose memory allocation package. malloc returns a pointer to a block of at least size bytes beginning on a word boundary.

The argument to free is a pointer to a block previously allocated by malloc; this space is made available for further allocation, but its contents are left undisturbed. Disorder will result if the space assigned by malloc is overrun or if some random number is handed to free.

malloc allocates the first, large enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls sbrk (see break(5)) to get more memory from the system when there is no suitable space already free.

calloc allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS:

malloc and calloc return a null pointer (0) if there is no available memory or if the area has been detectably corrupted by storing outside the bounds of a block.

BUGS:

When realloc returns 0, the block pointed to by ptr may be destroyed.

NAME:

mktemp - Make a Unique File Name

SYNOPSIS:

```
char *mktemp(template)
char *template;
```

DESCRIPTION:

mktemp replaces template by a unique file name, and returns the address of the template. The template should look like a file name with size trailing X's, which will be replaced with a unique letter.

NAME:

perror, sys_errlist, sys_nerr - System Error Messages

SYNOPSIS:

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION:

perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First, the argument string s is printed, then a colon, then the message and a newline. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable errno, which is set when errors occur but not cleared when nonerroneous calls are made.

To simplify variant formatting of messages, the vector of message strings sys_errlist is provided; errno can be used as an index in this table to get the message string without the newline. sys_nerr is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

NAME:

printf, fprintf, sprintf - Formatted Output Conversion

SYNOPSIS:

```
#include<stdio.h>

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

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

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

DESCRIPTION:

printf places output on the standard output stream, stdout. fprintf places output on the named output stream. sprintf places output in the string s, followed by the character '/0'.

Each of these functions converts, formats and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copies to the output stream and conversion specifications, each of which causes conversion and printing of the next successive arg printf.

Each conversion specification is introduced by the character %. Following the %, there may be:

- an optional minus sign '-' which specifies left adjustment of the converted value in the indicate field;
- an optional digit string specifying a field width; if the converted value has fewer characters than the field width, it will be padded with blanks on the left (or right, if the left adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '.' which serves to separate the field width from the next digit string;
- an optional digit string specifying a precision which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;

- the character l specifying that a following d, o, x or u corresponds to a long integer arg. (A capitalized conversion code accomplishes the same thing.)

- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer arg supplies the field width or precision.

The conversion characters and their meanings are:

dox	The integer arg is converted to decimal, octal or hexadecimal notation, respectively.
f	The float or double arg is converted to decimal notation in the style '[-]ddd.ddd', where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, six digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.
e	The float or double arg is converted in the style '[-]d.ddde+dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, six digits are produced.
g	The float or double arg is printed in style d, in style f or in style e, whichever gives full precision in minimum space.
c	The character arg is printed. Null characters are ignored.
s	arg is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing, all characters up to a null are printed.
u	The unsigned integer arg is converted to decimal and printed (the result will be in the range 0 to 4,294,967,295).
%	Print a '%', no argument is converted.

Never does a nonexistent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by printf are printed by putc(2).

EXAMPLES:

To print a date and time in the form 'Sunday, July 3, 10:02', where weekday and month are pointers to NUL-terminated strings:

```
printf("%s,%s %d,%02:%02d",weekday,month,day,hour,min);
```

To print to 5 decimals:

```
printf("pi = %.5f",4*atan(1.0));
```

SEE ALSO:

putc, scanf, ecvt

BUGS:

Very wide fields (>128 characters) fail.

NAME:

putc, putchar, fputc, putw - Put Character or Word on a Stream

SYNOPSIS:

```
#include<stdio.h>

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

putchar(c)

fputc(c, stream)
FILE *stream;

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

DESCRIPTION:

putc appends the character c to the named output stream. It returns the character written.

putchar(c) is defined as putc(c, stdout).

fputc behaves like putc, but is a genuine function rather than a macro. It may be used to save an object text.

putw appends word (i.e. int) w to the output stream. It returns the word written. putw neither assumes nor causes special alignment in the file.

The standard stream stdout is normally buffered only if the output does not refer to a terminal; this default may be changed by setbuf(2). The standard stream stderr is, by default, unbuffered unconditionally, but use of freopen (see fopen(2)) will cause it to become buffered; setbuf, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. fflush (see fclose(2)) may be used to force the block out early.

SEE ALSO:

fopen, fclose, getc, puts, printf, fread

DIAGNOSTICS:

These functions return the constant EOF upon error. Since this is a good integer, ferror(2) should be used to detect putw errors.

BUGS:

Because it is implemented as a macro, putc treats a stream argument with side effects improperly. In particular, 'putc(c,*f++);' doesn't work sensibly.

NAME:

puts, fputs - Put a String on a Stream

SYNOPSIS:

```
#include<stdio.h>

puts(s)
char *s;

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

DESCRIPTION:

puts copies the NUL-terminated string s to the standard output stream stdout and appends a newline character.

fputs copies the NUL-terminated string s to the named output stream.

Neither routine copies the terminal null character.

SEE ALSO:

fopen, gets, putc, printf, ferror

BUGS:

puts appends a newline, fputs does not.

NAME:

qsort - Quicker Sort

SYNOPSIS:

```
qsort(base,nel,width,compar)
char *base;
int (*compar) ();
```

DESCRIPTION:

qsort is an implementation of the quicker sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparision routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to or greater than 0 according as the first arguement is to be considered less than, equal to or greater than the second.

NAME:

rand, strand - Random Number Generator

SYNOPSIS:

```
strand(seed)
int seed;

rand()
```

DESCRIPTION:

rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo random numbers in the range from 0 to 2^{15} -1.

The generator is reinitialized by calling srand with 1 as argument. It can be set to a random starting point by calling srand with whatever you like as argument.

NAME:

scanf, fscanf, sscanf - Formatted Input Conversion

SYNOPSIS:

```
#include<stdio.h>

scanf(format [,pointer ]...)
char *s,*format;

fscanf (stream,format [, pointer]...)
FILE *stream;
char *format;

sscanf (s, format [, pointer]...)
char *s,*format
```

DESCRIPTION:

scanf reads from the standard input stream stdin. fscanf reads from the named input stream. sscanf reads from the character string s. Each function reads characters, interprets them according to a format and stores the results in its arguments. Each expects, as arguments, a control string format, described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.

2. An ordinary character (not %) which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of nonspace characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

%	indicates that a single '%' is expected in the input at this point; no assignment is done.
d	indicates that a decimal integer is expected; the corresponding argument should be an integer pointer.
o	indicates that an octal integer is expected; the corresponding argument should be a integer pointer.
x	indicates that a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
s	indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '/0', which will be added. The input field is terminated by a space character or a newline.
c	indicates that a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace character, try '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.
e	a floating point number is expected; the next
f	field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string 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.

a indicates a string not to be delimited by space characters. The left brackets define 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 circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^,the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be capitalized or preceeded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters e or f may be capitalized or preceded by l to indicate a pointer to double rather than to float. The conversion characters d, o and x may be preceeded by h to indicate a pointer to short rather than to int.

The scanf functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25 54.32E-1 thompson
```

will assign to i the value 25, x the value of 5.432, and name will contain 'thompson/0'. Or,

```
int i; float x; char name[50];
scanf("2d%f%*d%[124567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip '0123', and place the string '56/0' in name. The next call to getchar will return 'a'.

SEE ALSO:

atof, getc, printf

DIAGNOSTICS:

The scanf functions return EOF on end of input, and a short count for missing or illegal data items.

BUGS:

The success of literal matches and suppressed assignments is not directly determinable.

NAME:

setbuf - Assign Buffering to a Stream

SYNOPSIS:

```
#include <stdio.h>

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

DESCRIPTION:

setbuf is used after a stream has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer. If buf is the constant pointer NULL, I/O will be completely unbuffered.

A manifest constant BUFSIZ tells how large an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from malloc(2) upon the first getc or putc(2) on the file, except that output streams directed to terminals, and the standard error stream stderr are normally not buffered.

SEE ALSO:

fopen, getc, putc, malloc

NAME:

setjmp, longjmp - Nonlocal Goto

SYNOPSIS:

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION:

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in env for later use by longjmp. It returns value 0.

longjmp restores the environment saved by the last call of setjmp. It then returns in such a way that execution continues as if the call of setjmp had just returned the value val to the function that invoked setjmp, which must not itself have returned in the interim. All accessible data have values as of the time longjmp was called.

NAME:

sin, cos, tan, asin, acos, atan, atan2 - Trigonometric Functions

SYNOPSIS:

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

DESCRIPTION:

sin, cos and tan return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

acos returns the arc cosine in the range 0 to π .

atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

atan2 returns the arc tangent of x/y in the range $-\pi$ to π .

DIAGNOSTICS:

Arguments of magnitude greater than 1 cause asin and acos to return value 0. The value of tan at its singular points is a large number.

BUGS:

The value of tan for arguments greater than about 2**31 is indeterminate.

NAME:

sinh, cosh, tanh - Hyperbolic Functions

SYNOPSIS:

```
#include<math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION:

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS:

sinh and cosh return a large value of appropriate sign when the correct value would overflow.

NAME:

sleep - Suspend Execution for Interval

SYNOPSIS:

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION:

The current process is suspended from execution for the number of seconds specified by the arguent. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1 second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it occurs. The previous state of this signal is saved and restored. If the sleep time exceeds the time to the alarm signal, the process sleeps only until the signal would have occurred and the signal is sent 1 second later.

NAME:

stdio - Standard Buffered I/O Package

SYNOPSIS:

```
#include<stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION:

The functions described in Sections 2S constitute an efficient user-level buffering scheme. The inline macros getc and putc(3) handle characters quickly. The higher level routines gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf and fwrite all use getc and putc; they can be freely intermixed.

A file with associated buffering is called a stream, and is declared to be a pointer to a defined type FILE. fopen(2) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' NULL(0) designates no stream at all.

An integer constant EOF (-1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard I/O package must include the header file <stdio.h> of pertinent macro definitions. In general, the functions and constants mentioned in this chapter are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: getc, getchar, putc, putchar, feof, ferror and fileno.

SEE ALSO:

open(4), close (4), read (4), write(4)

DIAGONOSTICS:

The value EOF is returned uniformly to indicate that a FILE pointer has not been initialized with fopen, input(output) has been attempted on an output(input) stream, or a FILE pointer designates corrupt or otherwise unintelligible FILE data.

NAME:

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - String Operations

SYNOPSIS:

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

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

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

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

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

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

strlen(s)
char *s;

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

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

DESCRIPTION:

These functions operate on NUL-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string s2 to the end of string s1. strncat copies, at most, n characters. Both return a pointer to the NUL-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to or less than s2. strncmp makes the same comparison but looks at most n characters.

strcpy copies string s2 to s1, stopping after the null character has been moved. strncpy copies exactly n characters, truncating or null-padding s2; the target may not be NUL-terminated if the length of s2 is n or more. Both return s1.

strlen returns the number of nonnull characters in s.

Index (rindex) returns a pointer to the first (last) occurrence of character c in string s, or zero if c does not occur in the string.

NAME:

ttyname, isatty - Find Name of a Terminal

SYNOPSIS:

```
char *ttyname(fildes)

isatty(fildes)

ttyslot()
```

DESCRIPTION:

ttyname returns a pointer to the NUL-terminated name of the terminal device associated with file descriptor (fd) fildes.

isatty returns 1 if fildes is associated with a terminal device, 0 otherwise.

DIAGNOSTICS:

ttyname returns a null pointer (0) if fildes does not describe a terminal device.

BUGS:

The return value points to static data whose content is overwritten by each call.

NAME:

ungetc - Push Character Back into Input Stream

SYNOPSIS:

```
#include<stdio.h>

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

DESCRIPTION:

ungetc pushes the character c back on an input stream. That character will be returned by the next getc call on that stream. ungetc returns c.

One-character pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

fseek(2) erases all memory of pushed back characters.

SEE ALSO:

getc, setbuf, fseek

DIAGNOSTICS:

ungetc returns EOF if it cannot push a character back.

CHAPTER 3
IDRIS COMPATIBLE C RUN-TIME LIBRARY (RTL)

TABLE OF CONTENTS

NAME:

Conventions - Using C With the Standard Libraries

FUNCTION:

The current section, and the two that follow, document C callable functions provided on all systems supported by Whitesmiths, Ltd. All library functions follow a set of uniform coding conventions, which form an important part of the Whitesmiths C environment. These conventions should be mastered, the better to understand the descriptions following, to interface properly to the library functions, and, more generally, to write C in a portable manner.

Most standard conventions are supported at compile-time by the inclusion of a standard header file, std.h, which is separately documented in this section. The remainder are mainly described in the subsections on Style and Portability in Chapter 1. Here, however, are a few general cautions: every C program must contain a function named main, which is called at the outset and whose return signals the end of program execution; many library routines presume a conventional coding of main, documented in Chapter 4. Several of the "functions" described in this chapter are actually macros defined in the standard header. They appear on ordinary manual pages because, aside from certain side effects for which warning is served, they look to the programmer much like subprograms. On the other hand, there are a few secret library routines that are not documented anywhere in this manual; their names invariably begin with an underscore, to minimize accidental collisions with user-defined names.

The rest of this document provides a summary of the sections in a typical library function description.

NAME:

std.h - Standard Header File

SYNOPSIS:

```
#include <std.h>
```

FUNCTION:

All standard library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a standard header file, <std.h>, at the top of each program. The file defines a number of quasi-types and storage classes (in terms of the standard C types), various system parameters, the control structure used for buffered input/output (I/O) and some useful macros. The macros in <std.h> are each described in separate manual pages since, aside from certain curious side effects for which warning is served, macros look to the C programmer much like subroutines.

It is important to know these types and parameters, in order to understand manual pages for subroutines, to interface to the C library in a portable manner, and to code in good style. The principal definitions are as follows:

Quasi-types

- BITS - unsigned short, used as a set of 16 bits
- BOOL - int, tested only for nonzero, assigned YES or NO
- BYTES - unsigned int, for address arithmetic, indexing
- COUNT - short, for counting [-32,768, 32,768)
- DOUBLE - double precision floating point
- FILE - short, used for file descriptors (fds)
- LONG - long integer
- METACH - short, EOF or [0, 256)
- TBOOL - char, or unsigned char, used like BOOL
- TEXT - char, or unsigned char, containing printable text
- TINY - char, for counting [-128, 128)
- UCOUNT - unsigned short, for counting [0, 65,536)
- ULONG - unsigned long
- UTINY - unsigned char, for counting [0, 256)
- VOID - int, for functions returning nothing

Quasi-storage classes

- FAST - register
- GLOBAL - synonym for extern, used outside functions
- IMPORT - synonym for extern, used inside functions
- INTERN - synonym for static, used inside fuctions
- LOCAL - synonym for static, used outside fuctions

System Parameters

BUFSIZE - 512, the standard I/O buffer size
BWRITE - mode -1, opening for buffered writes
BYTMASK - 0377, mask for low byte of integer
EOF - -1, end of file metacharacter
FOREVER - for (; ;)
NO - BOOL 0
NULL - pointer 0
READ - mode 0, opening for read access
STDERR - FILE 2, the standard error output
STDIN - FILE 0, the standard input
STDOUT - FILE 1, the standard output
UPDATE - mode 2, opening for reading and writing
WRITE - mode 1, opening for writing
YES - BOOL 1

Control Structure for Input/Output

FIO - struct fio, for buffered I/O calls

Macros

(documented in manual pages)

abs
isalpha
isdigit
islower
isupper
iswhite
max
min
tolower
toupper

EXAMPLE:

```
/* THE MINIMUM PROGRAM
 * copyright (c) 1981 by Whitesmiths, Ltd.
 */
#include <std.h>

/* put string to STDOUT
 */
BOOL main()
    {
    write(STDOUT, "hello world\n", 12);
    return (YES);
    }
```

BUGS:

It is easy to forget about the macros, which cause unusual diagnostics when "redeclared". Using std.h and studio.h together may lead to severe difficulties.

NAME:

Cio - C I/O Subroutines

FUNCTION:

There are dozens of subroutines for performing I/O at various levels of sophistication. A brief guide to which groups best work together follows.

The simplest approach to I/O is to use putfmt for writing formatted output to the standard output; the odd error message can be sent via errfmt. It is easy to obtain simple input from the arguments passed to main by using getflags; and success or failure can be reported on program termination by exit, or the return value from main. If input must be read, getfmt makes it easy to read and encode items from the standard input under control of a format much like that use by the output routines.

Character-by-character or line-by-line, I/O is obtained by calls on getch, putch, getlin, and putlin. So long as output is text lines, i.e.; strings of characters terminated by newlines, buffering is automatic. These routines can be called interchangeably with getfmt and putfmt as well.

The standard header file <std.h> includes a declaration for the standard I/O control buffer, type FIO; all of the above routines quietly make use of the control buffers input and output, obtained as needed from the library. It is also possible to open and close files by name, and associate them with an FIO buffer, by calls to fopen, fcreate and fclose. Once established, an FIO buffer can be used for formatted output by calls on getf and putf. Character or line I/O under control of arbitrary FIO buffers can be obtained by calling getc, putc, getl and putl.

At a lower level, there are file descriptors (fds), numbers of type FILE (defined in the standard header) that are handed out by certain routines and used by others. Three fds are predefined: STDIN, STDOUT and STDERR, which are typically terminal input, terminal output and error output. Others can be obtained by opening filename arguments passed to main, using getfiles, then associating them with FIO buffers, using finit. Or, fds can be used directly with the lowest level I/O routines, described below.

The routines open and create generate new fds when they open files; close discredits an fd by ending its association with a file. A family of temporary files can be constructed from the root name returned by uname. Files can be removed from the file system by calling remove.

Fds are used by the lowest level routines (read and write) to move sequences of bytes between memory and files. Direct access is obtained by using lseek to read or write at random places in a file. Finally, the function putstr can be used to concatenate a sequence of strings to a specified file, which is useful for filing simple error messages.

BUGS:

No tutorial is available.

NAME:

FIO - The File I/O Structure

SYNOPSIS:

```
FIO stdin, stdout;
```

FUNCTION:

FIO is the type defined in <std.h> for the control buffers used by many of the C library I/O routines. Its elements are:

FILE _fd

holds the fd for the file with which I/O is performed.

COUNT _nleft

on input, tells how many characters are left undelivered in the buffer; on output, tells how many characters have been placed in the buffer for output. Setting _nleft to zero is sufficient to initialize an FIO buffer. Input routines set _nleft to -1 on end of file, as an indication that no further reads should be attempted.

COUNT _mode

is set to BWRITE, READ or WRITE to indicate the mode of operation.

TEXT *_pnext

on input, points to the next character to be delivered; on output, used to chain FIO buffers for draining on exit. If (_nleft == 0) on input, _pnext is undefined.

TEXT _buf[BUFSIZE]

is the character buffer, where BUFSIZE is 512.

BWRITE is a mode not recognized by the low-level interface routines. It is used to indicate buffered writing; i.e., output only on buffer-full or program exit. Normal output mode calls for draining the buffer whenever an output sequence ends with a newline.

All actual input using FIO control buffers is via getc or getl. All actual output is via putc or putl.

NAME:

abs - Find Absolute Value

SYNOPSIS:

```
abs(a)
```

FUNCTION:

abs obtains the absolute value of its argument. Since abs is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

abs is a numerical rvalue of the form ((a < 0) ? -a : a), suitably parenthesized.

EXAMPLE:

```
putfmt("balance %i%p\n", abs(bal), (bal < 0) ? "CR" : "");
```

BUGS:

Because it is a macro, abs cannot be called from non-C programs, nor can its address be taken. An argument with side effects may be evaluated more than once.

NAME:

alloc - Allocate Space on the Heap

SYNOPSIS:

```
TEXT *alloc(nbytes, link)
   BYTES nbytes, link
```

FUNCTION:

alloc allocates space on the heap for an item of size nbytes, then writes link in the zeroth integer location. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which pointer is guaranteed to be on a proper storage boundary for anything. The heap is enlarged as necessary; if space is exhausted, the message "out of heap space" is written to STDERR and an error exit is taken.

RETURNS:

If alloc returns, the pointer is guaranteed not to be NULL.

EXAMPLE:

To build a stack:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

top = alloc(sizeof (*top), top);    /* pushes a cell */
```

SEE ALSO:

buybuf, free, frelst, nalloc, sbreak(4)

BUGS:

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily clobbered. This number is related to the actual cell size in a most system-dependent fashion and should not be trusted.

An error may occur when attempting to allocate more than half of the address space at a time.

NAME:

amatch - Look for Anchored Match of Regular Expression

SYNOPSIS:

```
BYTES amatch(buf, n, idx, pat, psubs)
   TEXT *buf;
   BYTES n, idx;
   TEXT *pat;
   struct {
       TEXT *mtext;
       BYTES mlen;
       } *psubs;
```

FUNCTION:

amatch tests the n character buffer starting at buf[idx] for a match with the encoded pattern starting at pat; the match is constrained to match characters starting at buf[idx]. It is assumed that the pattern was built by the function pattern, whose manual page describes the notation for regular expressions accepted by these routines.

If (psubs is not NULL) then every balanced pair \(...\) within the pattern will have the substring it matches recorded at psubs[i], where i counts up from one for the left-most "\(" in the pattern. psubs[i] .mtext points at the first character of the matching substring, and psubs[i] .mlen is its length. psubs[0] always records the full match.

The pattern codes are a sequence of bytes with the values:

value	name	meaning
1	CCHAR	literal character follows
2	ANY	match anything but \n
3	SBOL	match beginning of line (0 width)
4	SEOL	match end of line, or just before ending \n
5	CLOSE	match following pattern zero or more times
6	CCL	character class follows (CCHARs or RANGEs)
7	NCCL	negated character class follows
8	RANGE	lower- and uppper-bound characters follow
9	CCLEND	character class ends
10	PEND	pattern end
19	RPAR	right parenthesis "\)", followed by a one-byte order number
20	LEFT	left parenthesis "\(", followed by a one-byte order number

These codes are only needed if patterns are to be built by hand.

RETURNS:

amatch returns the index of the right-most character of the match, if successful, else -1. The array at psubs is also filled in, if present.

EXAMPLE:

To match a variable pattern:

```
if (pattern(pbuf, av[1][0], &av[1][1]))
    while (n = getlin(buf, MAXBUF))
        if ((n = amatch(buf, n, 0, pbuf, NULL)) != -1)
            putlin(buf, m);
```

SEE ALSO:

match, pattern

NAME:

arctan - Arctangent

SYNOPSIS:

```
DOUBLE arctan(x)
   DOUBLE x;
```

FUNCTION:

arctan computes the angle in radians whose tangent is x, to full double precision. It works by folding x into the interval [0, 1], then interpolating from an eight entry table, using the sum of tangents formula and a fifth-order telescoped Taylor series approximation.

RETURNS:

arctan returns the nearest internal representation to arctan x, expressed as a double floating value in the interval $(-\pi/2, \pi/2)$.

EXAMPLE:

To find the phase angle of a vector:

```
theta = arctan(y / x) * 180.0 / π;
```

NAME:

bldks - Build Key Schedule From Key

SYNOPSIS:

```
TINY *bldks(ks, key)
   TINY ks[16][8];
   TEXT key[8];
```

FUNCTION:

bldks builds the key schedule used by the Data Encryption Standard algorithm for encrypting or decrypting data. All eight characters of key are used to form the key schedule, but the most significant bit (MSB) of each byte is ignored.

RETURNS:

bldks returns the address of ks, which contains the key schedule.

EXAMPLE:

To decrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);      while (read(STDIN, buf, 8) == 8)
    write(STDOUT, decrypt(buf, ks), 8);
```

SEE ALSO:

decrypt, encrypt

NAME:

btod - Convert Buffer to Double

SYNOPSIS:

```
BYTES btod(s, n, pdnum)
   TEXT *s;
   BYTES n;
   DOUBLE *pdnum;
```

FUNCTION:

btod converts the n character string starting at s into a double and stores it at pdnum. 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 at the end of the buffer or on the first unrecognizable character. Acceptable inputs match the pattern

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

where d is any decimal digit and e is 'e' or 'E'.

No checks are made against overflow, underflow or unusual character strings.

RETURNS:

btod returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pdnum.

EXAMPLE:

To convert a program's first command line argument into a double at dbl:

```
if (2 < ac)
    btod(av[1], lenstr(av[1]), &dbl);
else
    dbl = 0.0;
```

SEE ALSO:

dtento, dtoe, dtof

BUGS:

Nothing simple can be said about the properties of a number that has overflowed.

NAME:

btoi - Convert Buffer to Integer

SYNOPSIS:

```
BYTES btoi(s, n, pinum, base)
   TEXT *s;
   BYTES n, *pinum;
   COUNT base;
```

FUNCTION:

btoi converts the n character string starting at s into an integer and stores it at pinum. The string is taken as the text representation of a number in the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L', it is skipped over.

Acceptable characters are the decimal digits and letters, either upper- or lower-case, where the letter 'a' or 'A' has the value 10, as in the usual representation for hexadecimal. Letters with values greater than or equal to base are not acceptable digits. Thus, values of base from 1 to 36 are meaningful.

No checks are made against overflow, unreasonable values of base or unusual character strings.

RETURNS:

btoi returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pinum.

EXAMPLE:

```
BYTES num;

if (btoi(buf, size, &num, 10) != size) putstr(STDERR, "not a
decimal number\n", NULL);
```

SEE ALSO:

btol, btos, itob, ltob, stob

BUGS:

Nothing simple can be said about the properties of a number that has overflowed.

NAME:

btol - Convert Buffer to Long

SYNOPSIS:

```
BYTES btol(s, n, plnum, base)
   TEXT *s;
   BYTES n;
   LONG *plnum;
   COUNT base;
```

FUNCTION:

btol converts the n character string starting at s into a long integer and stores it at plnum. The string is taken as the text representation of a number in the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L', it is skipped.

Acceptable characters are the decimal digits and letters, either upper- or lower-case, where the letter 'a' or 'A' has the value 10, as in the usual representation for hexadecimal. If a letter has a value greater than or equal to base, it is not an acceptable digit. Thus, values of base from 1 to 36 are meaningful.

No checks are made against overflow, unreasonable values of base or unusual character strings.

RETURNS:

btol returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at plnum.

EXAMPLE:

```
LONG lnum;

if (btol(buf, size, &lnum, 16) != size)
    putstr(STDERR, "not a hexadecimal number\n", NULL);
```

SEE ALSO:

btoi, btos, itob, ltob, stob

BUGS:

Nothing simple can be said about the properties of a number that has overflowed.

NAME:

btos - Convert Buffer to Short Integer

SYNOPSIS:

```
BYTES btos(s, n, pinum, base)
   TEXT *s;
   BYTES n;
   COUNT *pinum, base;
```

FUNCTION:

btos converts the n character string starting at s into a short integer and stores it at pinum. The string is taken as the text representation of a number to the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L', it is skipped over.

Acceptable characters are the decimal digits and letters, either upper- or lower-case, where the letter 'a' or 'A' has the value 10, as in the usual representation for hexadecimal. Letters with values greater than or equal to base are not acceptable digits. Thus, values of base from 1 to 36 are meaningful.

No checks are made against overflow, unreasonable values of base or unusual character strings.

RETURNS:

btos returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pinum.

EXAMPLE:

```
COUNT snum;

if (btos(buf, size, &snum, 8) != size)
    putstr(STDERR, "not an octal number\n", NULL);
```

SEE ALSO:

btoi, btol, itob, ltob

BUGS:

Nothing simple can be said about the properties of a number that has overflowed.

NAME:

buybuf - Allocate a Cell and Copy in Text Buffer

SYNOPSIS:

```
TEXT *buybuf(s, n)
   TEXT *s;
   BYTES n;
```

FUNCTION:

buybuf allocates a cell of size n on the heap by calling alloc, then copies the n characters starting at s into it. If the heap is full, buybuf is terminated by alloc.

RETURNS:

The value returned is the pointer to the allocated cell.

EXAMPLE:

To read a text file into memory:

```
struct {
        TEXT *text;
        BYTES size;
        } lines[];

for (p = lines; 0 < (n = getlin(buf, BUFSIZE)); ++p)
    {
    p->text = buybuf(buf, n);
    p->size = n;
    }
```

SEE ALSO:

alloc, free, nalloc

BUGS:

There should be a way of dealing with heap overflow in buybuf.

NAME:

cmpbuf - Compare Two Buffers for Equality

SYNOPSIS:

```
BOOL cmpbuf(s1, s2, n)
   TEXT *s1, *s2;
   BYTES n;
```

FUNCTION:

cmpbuf compares two text buffers, character-by-character, for equality. The first buffer starts at s1, the second at s2; both are n characters long. s1 and s2 are said to be equal if the n characters in s1 and s2 are identical.

RETURNS:

The value returned is YES if the buffers are equal, else NO.

EXAMPLE:

```
if (cmpbuf(name, "include", 7))
    doinclude();
```

SEE ALSO:

cmpstr, prefix

NAME:

cmpstr - Compare Two Strings for Equality

SYNOPSIS:

```
BOOL cmpstr(s1, s2)
   TEXT *s1, *s2;
```

FUNCTION:

cmpstr compares two strings, character-by-character for equality. The first string starts at s1 and is terminated by a NUL '\0'; the second is likewise described by s2. The strings must match through and including their terminating NUL characters.

RETURNS:

The value returned is YES if the strings are equal, else NO.

EXAMPLE:

```
if (cmpstr(name, "include"))        doinclude();
```

SEE ALSO:

cmpbuf, prefix

NAME:

cos - Cosine in Radians

SYNOPSIS:

```
DOUBLE cos(x)
   DOUBLE x;
```

FUNCTION:

cos computes the cosine of x, expressed in radians, to full double precision. It works by scaling x in quadrants, then computing the appropriate sin or cos of an angle in the first half quadrant, using a sixth-order telescoped Taylor series approximation. 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, expressed as a double floating 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

NAME:

cpybuf - Copy One Buffer to Another

SYNOPSIS:

```
BYTES cpybuf(s1, s2, n)
   TEXT *s1, *s2;
   BYTES n;
```

FUNCTION:

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

RETURNS:

The value returned is n, the number of characters copied.

EXAMPLE:

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

```
n = cpybuf(buf, "first string", 12);
cpybuf(buf + n, ", second string", 15);
```

SEE ALSO:

cpystr

NAME:

cpystr - Copy Multiple Strings

SYNOPSIS:

```
TEXT *cpystr(ds, arg1, arg2, arg3, arg4, ..., NULL)
   TEXT *ds, *arg1, *arg2, *arg3, *arg4, ...;
```

FUNCTION:

cpystr concatenates a series of strings into the destination string ds. Each string begins at argx and is terminated by a NUL '\0'. The first character of arg2 is placed just after the last character (before the NUL) copied from arg1, etc. The series of string arguments is terminated by a NUL pointer argument. A NUL is appended to the final destination string to terminate it properly.

RETURNS:

The value returned is a pointer to the terminating NUL in the destination string.

EXAMPLE:

To concatenate string ss1 with " middle ", ss2 and " end." into buf:

```
cpystr(buf, ss1, " middle ", ss2, " end.", NULL);
```

BUGS:

There is no way to specify the size of the destination area, to prevent storage overwrites. Omitting the terminating NULL pointer is usually disastrous.

NAME:

decode - Convert Arguments to Text Under Format Control

SYNOPSIS:

```
BYTES decode(s, n, fmt, arg1, arg2, ...)
   TEXT *s;
   BYTES n;
   TEXT *fmt;
```

FUNCTION:

decode writes characters to the n character buffer starting at s exactly as if the contents were written to a file by putf, using the format string fmt and the zero or more arguments arg1, arg2,... It is not considered an error to generate more characters than will actually fit in the buffer; excess characters are simply discarded.

RETURNS:

decode returns the number of characters actually written in the buffer, a number between 0 and n, inclusive.

EXAMPLE:

To convert the integer symno to a symbolic name:

```
decode(&name, 6, "L%+05i", symno);
```

SEE ALSO:

dtoe, dtof, encode, putf, putfmt

NAME:

decrypt - Decode Encrypted Block of Text

SYNOPSIS:

```
TEXT *decrypt(data, ks)
   TEXT data[8];
   TINY ks[16][8];
```

FUNCTION:

decrypt converts the eight characters in the buffer data to decrypted form in place, using the key schedule constructed in ks by the function bldks. The Data Encryption Standard algorithm is used, taking bit 1 as the least significant bit (LSB) of data[0] and bit 64 as the MSB of data[7].

RETURNS:

decrypt returns a pointer to the start of data, which contains the decrypted text.

EXAMPLE:

To decrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);
while (read(STDIN, buf, 8) == 8)
    write(STDOUT, decrypt(buf, ks), 8);
```

SEE ALSO:

bldks, encrypt

NAME:

doesc - Process Character Escape Sequences

SYNOPSIS:

```
COUNT doesc(pp, magic)
   TEXT **pp, *magic;
```

FUNCTION:

doesc encodes the sequence of characters beginning at *pp, on the assumption that (*pp)[0] is an escape character, following the same escape conventions as the C compiler. It also updates the pointer at pp to point past the (variable length) escape sequence.

If ((*pp)[1] is NUL), the code value is (*pp)[0]; i.e., the escape character proper; this is the only escape sequence of length one. If ((*pp)[1] is a digit), then up to three digits are taken as the octal value of the code. If ((*pp)[1] is in the sequence "bfnrtv", in either case), the code is the corresponding member of the sequence (backspace, formfeed, newline, carriage return, horizontal tab, vertical tab). If (magic is not NULL) and (*pp)[1] is the ith character of the NUL terminated string at magic, the code is (-1 - i). Otherwise, the code is (*pp)[1].

In all cases, *pp is updated to point at the last character consumed.

RETURNS:

doesc returns the code obtained and updates the pointer *pp as necessary to point past the escape sequence.

EXAMPLE:

```
for (s = buf; *s; ++s)
    *t++ = (*s == '\\') ? doesc(&s, NULL) : *s++;
```

SEE ALSO:

mapchar

NAME:

dtento - Multiply Double by a Power of Ten

SYNOPSIS:

```
DOUBLE dtento(d, exp)
   DOUBLE d;
   COUNT exp;
```

FUNCTION:

dtento multiplies the double d by 10**exp. No check is made for overflow or underflow.

RETURNS:

dtento returns d * 10**exp as a double.

EXAMPLE:

To combine a fraction string and an exponent string:

```
btoi(fr, nfr, &intpart, 10);
btoi(sexp, nsexp, &exp, 10);
dbl = dtento((DOUBLE)intpart, exp - nfr);
```

SEE ALSO:

btod, dtoe, dtof

BUGS:

If the exponent is large in magnitude, dtento can loop for quite a long time. No special consideration is given (d == 0.0).

NAME:

dtoe - Convert Double to Buffer in Exponential Format

SYNOPSIS:

```
BYTES dtoe(s, dbl, p, q)
   TEXT *s;
   DOUBLE dbl;
   BYTES p, q;
```

FUNCTION:

dtoe converts the double number dbl to a text representation in the buffer starting at s, having the format:

```
[-]d*.d*e{+|-}d*
```

where d is a decimal digit. p specifies the number of digits to the left of the decimal point, and q the number to the right. There are either two or three digits in the exponent, depending upon the target machine.

RETURNS:

The value returned is the number of characters used to represent the double number.

EXAMPLE:

```
putfmt("area = %b\n", buf, dtoe(buf, area, 1, 5));
```

SEE ALSO:

dtof

NAME:

dtof - Convert Double to Buffer in Fixed Point Format

SYNOPSIS:

```
BYTES dtof(s, dbl, p, q)
   TEXT *s;
   DOUBLE dbl;
   BYTES p, q;
```

FUNCTION:

dtof converts the double number dbl to a text representation in the buffer starting at s, having the format:

```
[-]d*.d*
```

where d is a decimal digit, p specifies the maximum number of digits to the left of the decimal point and q the actual number to the right.

RETURNS:

The value returned is the number of characters used to represent the double number.

EXAMPLE:

```
putfmt("area = %b\n", buf, dtof(buf, area, 10, 5));
```

SEE ALSO:

dtoe

NAME:

encode - Convert Text to Arguments Under Format Control

SYNOPSIS:

```
COUNT encode(s, n, fmt, parg1, parg2, ...)
   TEXT *s;
   BYTES n;
   TEXT *fmt;
```

FUNCTION:

encode converts the contents of the n character buffer starting at s, using the format string at fmt and the argument pointers pargx, exactly as if the contents were read from a file by getf. It is particularly useful when multiple attempts must be made to read an input line.

RETURNS:

encode returns the number of arguments successfully converted, or EOF (-1) if end of buffer is encountered before any are converted.

EXAMPLE:

```
while (0 < (n = getlin(buf, BUFSIZE)))
    if (encode(buf, n, "x = %i", &x) <= 0 &&
        encode(buf, n, "y = %i", &y) <= 0)
        errfmt("unknown parameter %b\n", buf, n);
```

SEE ALSO:

btod, decode, getf, getfmt

NAME:

encrypt - Encode Block of text

SYNOPSIS:

```
TEXT *encrypt(data, ks)
   TEXT data[8];
   TINY ks[16][8];
```

FUNCTION:

encrypt converts the eight characters in the buffer data from encrypted form in place using the key schedule constructed in ks by the function bldks. The Data Encryption Standard (DES) algorithm is used, taking bit 1 as the LSB of data[0] and bit 64 as the MSB of data[7].

RETURNS:

encrypt returns a pointer to the start of data, which contains the encrypted text.

EXAMPLE:

To encrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);
   while (0 < (n = read(STDIN, buf, 8)))
       {
       while (n < 8)
           buf[n++] = '\0';
       write(STDOUT, encrypt(buf, ks), 8);
       }
```

SEE ALSO:

bldks, decrypt

NAME:

enter - Enter a Control Region

SYNOPSIS:

```
BYTES enter(pfn, arg)
   BYTES (*pfn)();
   BYTES arg;
```

FUNCTION:

enter establishes a new "control region"; i.e., a function invocation that can be terminated early by a leave call, then performs the sequence

```
leave((*pfn)(arg));
```

i.e.; the function pointed at by pfn is called with the specified arg; its return value is used as the argument of a call to leave. The control region may be terminated before (*pfn) returns by a call to leave in (*pfn) or in any of its dynamic descendants. In any case, the first leave call encountered disestablishes the new control region and causes enter to return with the value specified by the argument to that call to leave.

Control regions may be nested to any depth.

RETURNS:

enter returns the value of the argument to the first leave call encountered, or the value of the function at pfn if no leave was executed.

EXAMPLE:

To restart a function after each error message:

```
while (s = enter(&func, file))
    putstr(STDERR, s, "\n", NULL);
```

so that one can write in, say, one of func's dynamic descendants:

```
if (counterr)
    leave("missing parameter");
```

SEE ALSO:

_raise(5), _when(5), leave

BUGS:

There is no way to pass to the function (*pfn) more than one argument.

NAME:

errfmt - Format Output to Error File

SYNOPSIS:

```
VOID errfmt(fmt, arg1, arg2, ...)
   TEXT *fmt;
```

FUNCTION:

errfmt performs formatted output to STDERR, in much the same way as putf. Output is performed by multiple calls directly to write, which may be inefficient for large volumes of output but is least likely to lose diagnostics when a program malfunctions.

RETURNS:

Nothing. An error exit occurs if any writes fail.

EXAMPLE:

```
errfmt("can't open file %p\n", fname);
```

SEE ALSO:

putf, putfmt

NAME:

error - Print Error Message and Exit

SYNOPSIS:

```
VOID error (s1, s2)
   TEXT *s1, s2;
```

FUNCTION:

error prints an error message to STDERR, consisting of the program name _pname, a colon and space, the strings at s1 and s2 and a newline. It then takes an error exit. Either s1 or s2 may be NULL.

RETURNS:

error never returns to its caller.

EXAMPLE:

```
if ((fd = open(file, READ, 0)) < 0)
    error("can't open ", file);
```

SEE ALSO:

_pname(4), exit(4)

NAME:

exp - Exponential

SYNOPSIS:

```
DOUBLE exp(x)
   DOUBLE x;
```

FUNCTION:

exp computes the exponential of x to full double precision. It works by expressing x/ln 2 as an integer plus a fraction in the interval (-1/2, 1/2]. The exponential of the fraction is approximated by a ratio of two seventh-order polynomials.

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, a range error condition is raised; if that is inhibited, the largest representable value is returned.

EXAMPLE:

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

SEE ALSO:

_range(5), ln

NAME:

fclose - Close a File Controlled by FIO Buffer

SYNOPSIS:

```
FIO *fclose(pfio)
   FIO *pfio;
```

FUNCTION:

fclose closes the file under control of the FIO buffer at pfio. If the control buffer was initialized with a mode of WRITE or BWRITE, any remaining output is drained before closing and the control buffer is removed from the list of buffers to be drained on program exit.

RETURNS:

fclose returns pfio if the file was successfully closed, else NULL. An error exit is taken if (pfio == NULL).

SEE ALSO:

fcreate, finit, fopen

NAME:

fcreate - Create a File and Initialize a Control Buffer

SYNOPSIS:

```
FIO *fcreate(pfio, fname, mode)
   FIO *fcreate;
   TEXT *fname;
   COUNT mode;
```

FUNCTION:

fcreate creates a file with name fname and specified mode, and if successful, initializes the control buffer at pfio for proper operation with the file. mode should have one of the values BWRITE, READ or WRITE.

RETURNS:

fcreate returns pfio, if successful, else NULL. An error exit is taken if (pfio == NULL).

EXAMPLE:

```
if (!fcreate(&fio, "file", READ))
    errfmt("can't create file\n");
```

SEE ALSO:

create(4), fclose, finit, fopen

NAME:

fill - Propagate Fill Character Throughout Buffer

SYNOPSIS:

```
BYTES fill(s, n, c)
   TEXT *s, c;
   BYTES n;
```

FUNCTION:

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

RETURNS:

fill returns n.

EXAMPLE:

To write a 512-byte buffer of NULs:

```
write(fd, buf, fill(buf, BUFSIZE, '\0'));
```

SEE ALSO:

squeeze

NAME:

finit - Initialize an FIO Control Buffer

SYNOPSIS:

```
FIO *finit(pfio, fd, mode)
   FIO *pfio;
   FILE fd;
   COUNT mode;
```

FUNCTION:

finit initializes the FIO control buffer at pfio for proper operation with the file specified by fd, in the mode specified by mode. If (mode == BWRITE) the control buffer is set up for buffered writes, to be drained only when the buffer is full or the program exits. If (mode == READ) the control buffer is set up for reading. Otherwise (mode == WRITE) of necessity and the control buffer is set up for writing; writes will be buffered as for BWRITE only if lseek calls are acceptable with the specified fd, an indication that the output is a file and not an interactive device or a pipeline. Unbuffered output is drained whenever a segment of output ends with a newline character or on program termination.

RETURNS:

finit returns pfio. An error exit occurs if (pfio == NULL).

EXAMPLE:

To adapt stdout for most effective buffering strategy:

```
finit(&stdout, STDOUT, WRITE);
```

SEE ALSO:

fclose, fcreate, fopen

BUGS:

No check is made for (mode == UPDATE), which may or may not work satisfactorily.

NAME:

fopen - Open a File and Initialize a Control Buffer

SYNOPSIS:

```
FIO *fopen(pfio, fname, mode)
   FIO *fopen;
   TEXT *fname;
   COUNT mode;
```

FUNCTION:

fopen opens a file with name fname and specified mode, and if successful, initializes the control buffer at pfio for proper operation with the file. mode should have one of the values BWRITE, READ or WRITE.

RETURNS:

fopen returns pfio, if successful, else NULL. An error exit is taken if (pfio == NULL).

EXAMPLE:

```
if (!fopen(&fio, file, READ))
    errfmt("can't open %p\n", file);
```

SEE ALSO:

fclose, fcreate, finit, open(4)

NAME:

fread - Read Until Full Count

SYNOPSIS:

```
COUNT fread(fd, buf, size)
   FILE fd;
   TEXT *buf;
   BYTES size;
```

FUNCTION:

fread reads up to size characters from the file specified by fd into the buffer starting at buf. It does so by making repeated calls to read until an end of file is encountered or until size characters have been read. Thus, fread should be used whenever an entire record must be read at once, since read reserves the right to return a short count at all times.

RETURNS:

Unless end of file is encountered, fread always returns size; otherwise the value returned is between 0 and size, inclusive.

EXAMPLE:

To copy a file in integral records:

```
while (fread(STDIN, buf, RECSIZE) == RECSIZE)
    write(STDOUT, buf, RECSIZE);
```

SEE ALSO:

read(4)

NAME:

free - Free Space on the Heap

SYNOPSIS:

```
TEXT *free(pcell, link)
   TEXT *pcell;
   TEXT *link;
```

FUNCTION:

free returns an allocated cell to the heap for subsequent reuse, then returns link to the caller. The cell pointer pcell must have been obtained by an earlier alloc call; otherwise the heap will become corrupted. free tries to defend itself as best as it can against subversive calls and will take an error exit if it does not like what it is given. The message "bad free call" is written to STDERR if free is given the address of a pcell that has never been allocated, or if for some reason, the size field has been corrupted. "freeing a free cell" is displayed when free is called with an address within the free chain. A NULL pcell is explicitly allowed, however, and is ignored.

RETURNS:

If free returns, its value is guaranteed to be link, which is otherwise unused by free.

EXAMPLE:

To pop a stack item:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

top = free(top, top->prev); /* pops a cell */
```

SEE ALSO:

alloc, frelst, nalloc, sbreak(4)

BUGS:

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily destroyed. 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 later activity on the stack to come to grief, at least on some systems.

NAME:

frelst - Free a List of Allocated Cells

SYNOPSIS:

```
struct list *frelst(plist, pstop)
   struct list {struct list *next; ...} *plist, *pstop;
```

FUNCTION:

frelst walks a linked list that has been built with calls to alloc, freeing each cell on the list. Any type of cells can occur on the list, in any combination, as long as the first entry in each structure is a pointer used to link to the next cell. A NULL next pointer or one equal to pstop terminates the list.

RETURNS:

frelst returns the pointer that terminates the list, either NULL or pstop.

EXAMPLE:

```
struct list {
    struct list *next;
    ...} *list;

list = frelst(list, NULL);
```

SEE ALSO:

alloc, free

BUGS:

Freeing a list that was not made from calls on alloc can be disastrous.

NAME:

getbfiles - Collect Files from Command Line

SYNOPSIS:

```
FILE getbfiles(pac, pav, dfd, efd, rsize)
   BYTES *pac, rsize;
   TEXT ***pav;
   FILE dfd, efd;
```

FUNCTION:

getbfiles examines the file arguments passed to a command and opens files as needed for reading. The arguments to examine are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the command name and any flags have been skipped, for instance, by calling getflags. If there are no arguments left on the first call to getbfiles (*pac == 0), the default fd, dfd, is returned and all subsequent calls will fail. Otherwise, each call to getbfiles will inspect the next argument in sequence.

If a filename matches the string "-", dfd is returned. Otherwise, an attempt is made to open the file for reading with the record size specified by rsize. If the file is to contain arbitrary binary data, as opposed to printable ASCII text, rsize should be nonzero. On success, the fd of the opened file is returned. If the open fails, efd is returned instead. After the last filename is processed, all calls to getbfiles will fail. It is up to the calling program to close any files opened by getbfiles.

RETURNS:

getbfiles returns either an fd obtained as described above, or the failure code -1; the first call to getbfiles will never return failure. *pac and *pav are updated on each call to reflect the number of arguments left to encode. To signal end of arguments, *pac is set to -1. If the returned fd is not dfd, the name of the file under consideration (successfully opened or not) is at (*pav)[-1].

EXAMPLE:

To walk a list of binary files:

```
BYTES ac;
TEXT **av;

while (0 <= (fd = getbfiles(&ac, &av, STDIN, STDERR, 1)))
    if (fd == STDERR)
        errfmt("can't read %p\n", av[-1]);
    else
        {
        process(fd);
        close(fd);
        }
```

SEE ALSO:

getfiles, getflags, open(4)

NAME:

getc - Get a Character from Input Buffer

SYNOPSIS:

```
METACH getc(pfio)
   FIO *pfio;
```

FUNCTION:

getc obtains the next input character, if any, from the file controlled by the FIO buffer at pfio; if end of file has been encountered, a code is returned that is distinguishable from any character.

RETURNS:

getc returns the character as zero (for '\0') or a small positive integer; end of file is signalled by the code EOF (-1). An error exit occurs if any reads fail, or if (pfio == NULL).

EXAMPLE:

To copy a file, character by character:

```
while (putc(&stdout, getc(&stdin)) != EOF)
    ;
```

SEE ALSO:

getch, putc, putch

NAME:

getch - Get a Character from Input Buffer stdin

SYNOPSIS:

```
METACH getch()
```

FUNCTION:

getch obtains the next input character, if any, from the file controlled by the FIO buffer stdin; if end of file has been encountered a code is returned that is distinguishable from any character.

RETURNS:

getch returns the character as zero (for NUL) or a small positive integer; end of file is signalled by the code EOF (-1). An error exit occurs if any reads fail.

EXAMPLE:

To copy a file, character by character:

```
while (putch(getch()) != EOF)
    ;
```

SEE ALSO:

getc, putc, putch

NAME:

getf - Read Formatted Input

SYNOPSIS:

```
COUNT getf(pfio, fmt, arg1, arg2, ...)
   FIO *pfio;
   TEXT *fmt;
   ...
```

FUNCTION:

getf reads input text from the file controlled by the buffer at pfio, and parses it according to the control format string starting at fmt, in order to assign converted values to a series of variables, each pointed at by one ot the arguments arg1,... The format string consists of newlines and literal text to be matched, interspersed with <field-specifier>s that determine how the input text is to be read and how it is to be converted before assignment. Input is consumed on a line-by-line basis. The number of lines consumed in any one call is typically equal to the number of newlines encountered in the format string, plus one if any character follows the last newline encountered in the format. An exception to this may occur if "% " appears in the format string; this sequence matches arbitrary whitespace, even extending across multiple lines.

For example:

```
getf(&stdin, "%i\n%i\n%i", &arg1, &arg2, &arg3);
```

obtains values for the three integers arg1, arg2 and arg3 from three successive lines of stdin, while:

```
getf(&stdin, "%i%i%i", &arg1, &arg2, &arg3);
```

obtains values for the three integers arg1, arg2 and arg3 from three whitespace separated fields on a single line of stdin.

Matching of literal text occurs on a character-by-character basis. If the charater in the format string does not match the next character to be consumed on the input line, the scan is terminated. A newline character in the format string matches any characters remaining in the current input line, up to and including the terminating newline, if any. Since a newline is consumed only by a literal match, by "% ", or (implicitly) by the end of the format string, an embedded '\n' is the most controlled way of reading multiple lines with one call to getf.

A <field-specifier> takes the form:

```
%[+z|-z][#]<field-code>
```

That is, a <field-specifier> consists of a literal '%', followed by an optional "+z" or "-z", where z can be any character, followed by an optional field width #, and is terminated by a <field-code>. A "+z", if present, calls for the stripping of any left fill with the fill character z, while "-z" calls for the stripping of any right fill with z. A #, if present, specifies the total width in characters of the field to be input, and is either a decimal integer, or the letter 'n'. If an 'n' is given, then the value of the next argument from the argument list is taken to specify the field width.

To read a nine-character field left-filled with '*', and interpret it as a floating point number:

```
getf(&stdin, "%+*9f", &argl);
```

or:

```
getf(&stdin, "%+*nf", 9, &argl);
```

The number of characters to consume during a field conversion is given by the width specifier, if present. If there are fewer than that many characters before the next newline, the rest of the line is consumed. If no width is specified, leading whitespace is skipped and the following group of nonwhite characters is taken to be the field; at least one nonwhite character must be present. The characters actually converted are the contents of the field less any fill characters. If no fill character is given, getf presumes the field is left-filled with spaces.

Text input of

```
$ 100.53
```

can be read as two integers with:

```
getf(&stdin, " %6i.%2i", &dollars, &cents);
```

or as a single double with either:

```
getf(&stdin, "%+ 10d", &cash);
```

or

```
getf(&stdin, " %d", &cash);
```

A <field-code> is composed of a <modifier>, a <specifier> or both. The <specifier> defines how the input field is to be converted, and is one of the following:

- c = char integer
- s = short integer
- i = integer
- l = long integer
- p = NUL-terminated string
- b = buffer of specified length
- d = double
- f = float
- x = padding only (no conversion)

A <modifier> causes the input to an integer variable to be interpreted as:

- a = ASCII bytes, in decreasing order of numerical significance
- h = hexadecimal (with or without a leading "0x")
- o = octal (with or without a leading '0')
- u = unsigned decimal

If no <specifier> is given, it is presumed to be 'i', and a <modifier> given from the above series will be taken to apply to the implied integer field. If a <specifier> of 'c', 's', 'i' or 'l' is given with no <modifier>, the input is interpreted as signed decimal.

In addition, an optional precision modifier, ".#", limits the number of characters that may be input with a <specifier> of 'p' or 'b', and is permitted but ignored with 'd' and 'f', for compatibility with putf. Like the field width specifier, the precision modifier # may be either an explicit integer, or an 'n', to make use of the next argument value in sequence.

Hence a <field-code> usually consists of one of the following combinations of <specifier> and <modifier>:

```
[a|h|o|u]{c|s|i|l}    /* integer input */
[.#]{b|p|d|f}         /* precision ignored for f and d */
{a|h|o|u}             /* default specifier is i */
{x}                   /* just skip field */
```

Any other character in the place of a <field-code> is taken as a single literal character to be matched in the input line. Thus, a '%' may be scanned with the specifier "%%" and a '\n' may be scanned, without skipping characters in the input line, by using the specifier "%\n". Hence, while "% " and "\n" have special meaning, "%\n" and " " each match only one character.

Each <field-specifier> given in the format string requires the argument list following to contain in identical sequence a pointer to a datum of the appropriate type; the pointer argument is used to assign a correctly converted field.

The following would read an int in hex, a char-sized value as ASCII, and a short as signed decimal, all of them optionally separated by whitespace:

```
getf(&stdin, "%8h%ac%s", &addr, &code, &offset);
```

Any integer field may contain leading whitespace, even after the stripping of fill characters, as well as an optional [+|-] sign, and an optional trailing [l|L] (which is C notation for a long constant). No unexpected conversion character may occur or the scan is terminated before the corresponding argument is assigned.

The 'a' modifier treats the input as a sequence of characters and converts it to a base 256 number whose digits are the characters; the argument gets assigned the value represented by the low-order bytes of that number.

Entire text strings may be assigned to arguments under the 'p' or 'b' field code. In the first case, the argument is a pointer to the start of a string, and input characters are copied into that string with a terminating NUL; in the second case, the argument is also a pointer to text but characters are copied in without the terminating NUL, and the number of characters copied is assigned using the argument following the pointer as a pointer to integer. In either case, the number of characters actually copied will be no more than the precision modifier, if it is present and nonzero.

For example, exactly one character of an 80-character input line could be assigned to str with:

```
getf(&stdin, "%80.np", 1, str);
```

Floating point numbers may be read in using 'd' for double variables and 'f' for float. In either case, the input may be in either fixed point or scientific notation (see btod). Leading whitespace will be skipped, even after the stripping of fill characters. The precision modifier is ignored.

The 'x' field code consumes no arguments; it is a convenient way to skip over text.

RETURNS:

getf returns the number of arguments successfully assigned, or EOF if end of file is encountered on input before any argument has been converted. An error exit occurs if (pfio == NULL).

EXAMPLE:

Given the code:

```
FIO input;
TEXT buf1[BUFSIZE], buf2[10];
BYTES nargs, x, y, z;

nargs = getf(&input, "%b%-*i%.6p%4i", &buf1, &x, &y, &buf2, &z);
if (nargs != 5)
    putstr(STDERR, "bad input format\n", NULL);
```

The input line:

```
LINE 17** IDENTIFIER 263
```

would assign:

```
"LINE" to buf1, with no trailing NUL
4 to x
17 to y
"IDENTI" to buf2, with trailing NUL
263 to z
```

SEE ALSO:

btod, encode, getfmt

NAME:

getfiles - Collect Text Files from Command Line

SYNOPSIS:

```
FILE getfiles(pac, pav, dfd, efd)
   BYTES *pac;
   TEXT ***pav;
   FILE dfd, efd;
```

FUNCTION:

getfiles examines the file arguments passed to a command and opens text files as needed for reading. The arguments to examine are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the command name and any flags have been skipped, for instance, by calling getflags. If there are no arguments left on the first call to getfiles (*pac == 0), the default fd, dfd, is returned and all subsequent calls will fail. Otherwise each call to getfiles will inspect the next argument in sequence.

If a filename matches the string "-", dfd is returned. Otherwise an attempt is made to open the file for reading as a text file; on success, the fd of the opened file is returned. If the open fails, efd is returned instead. After the last filename is processed, all calls to getfiles will fail. It is up to the calling program to close any files opened by getfiles.

RETURNS:

getfiles returns either an fd obtained as described above, or the failure code -1; the first call to getfiles will never return failure. *pac and *pav are updated on each call to reflect the number of arguments left to encode. To signal end of arguments, *pac is set to -1. If the returned fd is not dfd, the name of the file under consideration (successfully opened or not) is at (*pav)[-1].

EXAMPLE:

To walk a list of files:

```
BYTES ac;
TEXT **av;

while (0 <= (fd = getfiles(&ac, &av, STDIN, STDERR)))
    if (fd == STDERR)
        errfmt("can't read %p\n", av[-1]);
    else
        {
        process(fd);
        close(fd);
        }
```

SEE ALSO:

getbfiles, getflags, open(4)

NAME:

getflags - Collect Flags from Command Line

SYNOPSIS:

```
TEXT *getflags(pac, pav, fmt, arg1, arg2, ...)
   BYTES *pac;
   TEXT ***pav;
   TEXT *fmt;
   ...
```

FUNCTION:

getflags encodes the flag arguments passed to a command and sets the flags, counts, character variables and string names specified by a format string. The arguments to encode are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the first argument is a command name, to be skipped. Each succeeding argument is taken as a set of one or more flags if a) it begins with '-' or '+' and b) it is not the string "-" or "--". A leading '-' is otherwise skipped over on each command argument.

fmt is a concatenation of descriptors that determine how each of the succeeding arguments arg1,... is to be interpreted. A descriptor is a sequence of match characters, terminated by a ',', a '<' or by the '\0' or ':' that terminates the format string. Format characters have the following effect:

'*' always matches the rest of the current argument, if any left, or all of the succeeding argument, if present, or a null string otherwise. The value of the match is a (non-NULL) pointer to the start of the matched string.

'?' always matches the next argument character, if any, or a NUL character. The value of the match is the matched character, taken as an integer constant.

'#' tries to parse, as an integer, the remainder of the current argument, if any left, or all of the succeeding argument. The value of the match is the decimal value of the string, if it does not begin with a '0', or its hexadecimal value if it begins with '0x' or '0X', or its octal value otherwise. An error occurs if no argument is found, or if it cannot be completely scanned as an integer with the selected base.

'##' same as single # except that target is assumed to be a long instead of an int.

',' delivers a successful match value to the corresponding argument (in sequence) pointed at by argl,.... If no match, the command arguments are rescanned, from the last successful match, using the descriptor following.

'>' behaves just like ',', except the corresponding argument pointer is taken as a pointer to a structure of the form:

```
struct { BYTES ntop; TEXT *val[MAX]; }
args {MAX};
```

If (0 < ntop) ntop is decremented and the value is delivered to val[ntop]; otherwise an error occurs.

'\0' behaves just like ',', except that if there is no successful match an error occurs.

':' if a colon is encountered in the format string before a flag is successfully matched, then the NUL-terminated string following the colon is written to STDERR, preceded by "usage: <pname> " and followed by a newline, where <pname> is the name by which the current program was invoked. getflags then terminates, reporting failure. Any occurrence of an 'F' in the diagnostic string is replaced with a slightly expanded representation of the flag format string preceding the colon. For example, the format string "a*>+b,c?,z:F <files>" would produce the error message:

```
usage: pname -[a*^ +b c? z] <files>
```

Any other character causes a successful match only if the next command line character is identical to it. The value of the match is a boolean YES.

The rules by which getflags parses flag arguments impose two significant constraints on how flags are ordered within the format string. Any flag whose name is a prefix of the name of another flag must appear in the format string after the longer flag. This also implies that unnamed flags, such as "-#" or "-##" or "-*", must be given last.

RETURNS:

getflags returns a pointer to the remaining command argument string, if an error occurs and no colon is found in the format string; otherwise an error causes diagnostic output and an error exit from the program. If all flag arguments are successfully scanned, getflags returns NULL. The values pointed at by pac and pav are updated to reflect the number of arguments consumed; "--" is consumed as a flags terminator, while "-" is taken as a potential special file name and is not consumed. One or more values should be delivered to locations pointed at by the argl,...

Note that all locations pointed at are assumed to be ints or pointers, except that a "##" descriptor expects a long, and a '>' expects a structure as described above.

EXAMPLE:

To accept the line:

```
cmd +3 -3 -f filename -mx -b0x10000 <files> ...
```

one might write:

```
BOOL mxflag {NO};
BYTES mcnt {0};
BYTES from {0};
BYTES to {0};
LONG bias {0};
TEXT *fname "default";

COUNT main(ac, av)
BYTES ac;
TEXT **av;
{
getflags(&ac, &av, "b##,f*,mx,m#,+#,#:F <files>",
     &bias, &fname, &mxflag, &mcnt, &from, &to))
```

SEE ALSO:

getfiles, usage

BUGS:

A "##" descriptor cannot be used with the stacking operation '>'.

NAME:

getfmt - Format Input from stdin

SYNOPSIS:

```
COUNT getfmt(fmt, arg1, arg2, ...)
   TEXT *fmt;
```

FUNCTION:

getfmt reads formatted input from the file controlled by the FIO buffer stdin, in exactly the same way as getf.

RETURNS:

getfmt returns the number of arguments successfully converted, or EOF (-1) if end of file is encountered before any are converted. An error exit occurs if any reads fail.

EXAMPLE:

```
for (lsum = 0; 0 < getfmt("%l", &lnum); )
     lsum =+ lnum;
```

SEE ALSO:

encode, getf, stdin

NAME:

getl - Get a Text Line into the Input Buffer

SYNOPSIS:

```
BYTES getl(pfio, s, n)
   FIO *pfio;
   TEXT *s;
   BYTES n;
```

FUNCTION:

getl copies characters, from the file controlled by the FIO buffer at pfio, to the n character buffer starting at s. Characters are copied until a newline is copied, end of file is reached or n characters have been copied.

RETURNS:

getl returns a count of the number of characters copied, which will be between 1 and n unless end of file has been encountered, from which time on all getl calls will return zero. An error exit occurs if any reads fail, or if (pfio == NULL).

EXAMPLE:

To copy a file, line by line:

```
while (putl(&stdout, buf, getl(&stdin, buf, BUFSIZE)))
    ;
```

SEE ALSO:

getlin, putl, putlin

NAME:

getlin - Get a Text Line from stdin

SYNOPSIS:

```
BYTES getlin(s, n)
   TEXT *s;
   BYTES n;
```

FUNCTION:

getlin copies characters from the file controlled by the FIO buffer stdin, to the n character buffer starting at s. Characters are copied until a newline is copied, end of file is reached or n characters have been copied.

RETURNS:

getlin returns a count of the number of characters copied, which will be between 1 and n unless end of file has been encountered, from which time on all getlin calls will return zero. An error exit occurs if any read fails.

EXAMPLE:

To copy a file, line by line:

```
while (putlin(buf, getlin(buf, BUFSIZE)))
    ;
```

SEE ALSO:

getl, putl, putlin, stdin

NAME:

inbuf - Find First Occurrence in Buffer of Character in Set

SYNOPSIS:

```
BYTES inbuf(p, n, s)
   TEXT *p, *s;
   BYTES n;
```

FUNCTION:

inbuf scans the n-character buffer starting at p for the first instance of a character in the NUL terminated set s. If the NUL character is to be part of the set, it must be the first character in the set.

RETURNS:

inbuf returns the index of the first character in p that is also in the set s, or n if no character in the buffer is in the set.

EXAMPLE:

To blank out imbedded NUL characters:

```
while ((i = inbuf(buf, n, "\0")) < n)
    buf[i] = ' ';
```

SEE ALSO:

instr, notbuf, notstr, scnbuf, scnstr, subbuf, substr

NAME:

instr - Find First Occurrence in String of Character in Set

SYNOPSIS:

```
BYTES instr(p, s)
   TEXT *p, *s;
```

FUNCTION:

instr scans the NUL terminated string starting at p for the first occurrence of a character in the NUL terminated set s.

RETURNS:

instr returns the index of the first character in p that is also contained in the set s, or the index of the terminating NUL if none.

EXAMPLE:

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

```
while (string[i = instr(string, "'{|}~")])
    string[i] = '@';
```

SEE ALSO:

inbuf, notbuf, notstr, scnbuf, scnstr, subbuf, substr

NAME:

isalpha - Test for Alphabetic Character

SYNOPSIS:

```
BOOL isalpha(c)
```

FUNCTION:

isalpha tests whether its argument is an alphabetic character, either lower- or upper-case. Since isalpha is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

isalpha is a Boolean rvalue.

EXAMPLE:

To find the end points of an alpha string:

```
if (isalpha(*first))
    for (last = first; isalpha(*last); ++last)
        ;
```

SEE ALSO:

isdigit, islower, isupper, iswhite, tolower, toupper

BUGS:

Because it is a macro, isalpha cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME:

isdigit - Test for Digit

SYNOPSIS:

```
BOOL isdigit(c)
```

FUNCTION:

isdigit tests whether its argument is a decimal digit, i.e., between '0' and '9' inclusive. Since isdigit is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

isdigit is a Boolean rvalue.

EXAMPLE:

To convert a digit string to a number:

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

SEE ALSO:

isalpha, islower, isupper, iswhite, tolower, toupper

BUGS:

Because it is a macro, isdigit cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME:

islower - Test for Lower-Case Character

SYNOPSIS:

```
BOOL islower(c)
```

FUNCTION:

islower tests whether its argument is a lower-case character. Since islower is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

islower is a Boolean rvalue.

EXAMPLE:

To convert to uppercase:

```
if (islower(c))
    c =+ 'A' - 'a';      /* but see toupper () */
```

SEE ALSO:

isalpha, isdigit, isupper, iswhite, tolower, toupper

BUGS:

Because it is a macro, islower cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME:

isupper - Test For Upper-Case Character

SYNOPSIS:

```
BOOL isupper(c)
```

FUNCTION:

isupper tests whether its argument is an upper-case character. Since isupper is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

isupper is a Boolean rvalue.

EXAMPLE:

To convert to lowercase:

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

SEE ALSO:

isalpha, isdigit, islower, iswhite, tolower, toupper

BUGS:

Because it is a macro, isupper cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME:

iswhite - Test for Whitespace Character

SYNOPSIS:

```
BOOL iswhite(c)
```

FUNCTION:

iswhite tests whether its argument is a nonprinting character code, i.e., whether its ASCII value is at or below that of ' ' (040) or at or above that of DEL (0177). Note that both NUL '\0' and newline '\n' qualify as whitespace. Since iswhite is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS:

iswhite is a Boolean rvalue.

EXAMPLE:

To skip whitespace:

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

SEE ALSO:

isalpha, isdigit, islower, isupper, tolower, toupper

BUGS:

Because it is a macro, iswhite cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME:

itob - Convert Integer to Text in Buffer

SYNOPSIS:

```
BYTES itob(s, i, base)
   TEXT *s;
   ARGINT i;
   COUNT base;
```

FUNCTION:

itob converts the integer i to a text representation in the buffer starting at s. The number is represented in the base specified, using lower-case letters beginning with 'a' to specify digits from 10 on. If (0 < base) the number i is taken as unsigned; otherwise if (base < 0) negative numbers have a leading minus sign and are converted to -base; if (base == 0) it is taken as -10. Only magnitudes of base between 2 and 36 are generally meaningful, but no check is made for reasonableness.

RETURNS:

The value returned is the number of characters used to represent the integer, which in hexadecimal can vary from four to eight digits, plus sign, depending upon the target machine.

EXAMPLE:

To output i in decimal:

```
write(STDOUT, buf, itob(buf, i, 10));
```

SEE ALSO:

btoi, btol, ltob, stob

BUGS:

The length of the buffer is not specifiable. If (|base| == 1), the program can bomb; if (36 < |base|) unpredictable characters can be inserted in the buffer.

NAME:

itols - Convert Integer to Leading Low-Byte String

SYNOPSIS:

```
TEXT *itols(s, val)
   TEXT *s;
   COUNT val;
```

FUNCTION:

itols writes the integer val into the two-byte string at s, with the least significant byte at s[0] and the next least significant byte at s[1]. No stronger storage boundary than that required for char is demanded of s.

A number of de facto standard file formats have arisen on machines that represent integers internally in this fashion; itols provides a machine-independent way of writing such files.

RETURNS:

itols writes the two bytes at s and returns s as the value of the function.

EXAMPLE:

To write a library header:

```
struct {
    TEXT name[14];
    COUNT size;
    } *p;

write(STDOUT, p->name, 14);
write(STDOUT, itols(buf, p->size), 2);
```

SEE ALSO:

lstoi, lstol, ltols

NAME:

leave - Leave a Control Region

SYNOPSIS:

```
VOID leave(val)
   BYTES val;
```

FUNCTION:

leave causes an exit from the control region established by the most recent enter call. Execution resumes as if enter had just performed a return with value val. Any number of functions may be terminated early by a leave call, as long as all are dynamic descendants of at least one enter call. The control region is disestablished by the call to leave.

RETURNS:

leave will never return to its caller; instead val is used as the return value of the most recent call to enter. If no instance of enter is currently active, leave writes an error message to STDERR and takes an error exit.

EXAMPLE:

To restart a function after each error message:

```
while (s = enter(&func, file))
    putstr(STDERR, s, "\n", NULL);
```

so that one can write in, say, one of func's dynamic descendants:

```
if (counterr)
    leave("missing parameter");
```

SEE ALSO:

_raise(5), _when(5), enter

NAME:

lenstr - Find Length of a String

SYNOPSIS:

```
BYTES lenstr(s)
   TEXT *s;
```

FUNCTION:

lenstr 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.

EXAMPLE:

To output a string:

```
write(STDOUT, s, lenstr(s));
```

NAME:

ln - Natural Logarithm

SYNOPSIS:

```
DOUBLE ln(x)
   DOUBLE x;
```

FUNCTION:

ln computes the natural log of x to full double precision. It works by expressing x as a fraction in the interval [1/2, 1), times an integer power of two. The logarithm of the fraction is approximated by a sixth-order telescoped Taylor series approximation.

RETURNS:

ln returns the nearest internal representation to ln x, expressed as a double floating value. If x is negative or zero, a domain error condition is raised.

EXAMPLE:

```
arcsinh = ln(x + sqrt(x * x + 1));
```

SEE ALSO:

_domain(5), exp

NAME:

lower - Convert Characters in Buffer to Lower-Case

SYNOPSIS:

```
BYTES lower(s, n)
   TEXT *s;
   BYTES n;
```

FUNCTION:

lower converts the n characters in buffer starting at s to their lower-case equivalent, if possible.

RETURNS:

lower returns n.

EXAMPLE:

```
buf[lower(buf, size)] = '\0';
```

SEE ALSO:

tolower

NAME:

lstoi - Convert Leading Low-Byte String to Integer

SYNOPSIS:

```
COUNT lstoi(s)
   TEXT *s;
```

FUNCTION:

lstoi converts the two-byte string at s into an integer, on the assumption that the leading byte is the least significant part of the integer. No stronger storage boundary than that required for char is demanded of s.

A number of de facto standard file formats have arisen on machines that represent integers internally in this fashion; lstoi provides a machine-independent way of reading such files.

RETURNS:

lstoi returns the integer representation of the two-byte integer at s.

EXAMPLE:

To read a library header:

```
struct {
    TEXT name[14];
    COUNT size;
    } *p;

read(STDIN, p, 16);
p->size = lstoi(&p->size);
```

SEE ALSO:

itols, lstol, ltols

NAME:

lstol - Convert Filesystem Date to Long

SYNOPSIS:

```
LONG lstol(s)
   TEXT *s;
```

FUNCTION:

lstol converts the four-byte string at s into a long, on the assumption that the bytes are ordered 2, 3, 0, 1, where 0 is the least significant byte. This bizarre order is used to represent dates in IDRIS filesystems, due to their PDP-11 origins. No stronger storage boundary than that required for char is demanded of s.

RETURNS:

lstol returns the long representation of the four-byte integer at s.

EXAMPLE:

```
time = lstol(&pi->n_actime);
```

SEE ALSO:

itols, lstoi, ltols

NAME:

ltob - Convert Long to Text in Buffer

SYNOPSIS:

```
BYTES ltob(s, l, base)
   TEXT *s;
   LONG l;
   COUNT base;
```

FUNCTION:

ltob converts the long l to a text representation in the buffer starting at s. The number is represented in the base specified, using lower-case letters beginning with 'a' to specify digits from 10 on. If (0 << base), the number l is taken as unsigned; otherwise if (base < 0), negative numbers have a leading minus sign and are converted to -base; if (base == 0), it is taken as -10. Only values of base between 2 and 36 in magnitude are generally meaningful, but no check is made for reasonableness.

RETURNS:

The value returned is the number of characters used to represent the long, which, in hexadecimal, can be up to eight digits plus sign.

EXAMPLE:

To output l as an unsigned decimal number:

```
write(STDOUT, buf, ltob(buf, l, 10));
```

SEE ALSO:

btoi, btol, itob, stob

BUGS:

The length of the buffer is not specifiable. If (|base| == 1) the program can bomb; if (36 < |base|) indeterminate characters can be inserted in the buffer.

NAME:

ltols - Convert Long to Filesystem Date

SYNOPSIS:

```
TEXT *ltols(plong, lo)
   TEXT *plong;
   LONG lo;
```

FUNCTION:

ltols writes the four bytes of the long lo into the buffer, starting at plong, in the order 2, 3, 0, 1, where 0 is the least significant byte. This order is used to represent dates in IDRIS filesystems.

RETURNS:

ltols writes the four bytes at plong and returns plong as its value.

EXAMPLE:

```
ltols(&pi->n_actime, time);
```

SEE ALSO:

itols, lstoi, lstol

NAME:

mapchar - Map Single Character to Printable Representation

SYNOPSIS:

```
VOID mapchar(c, ptr)
   TEXT c, *ptr;
```

FUNCTION:

mapchar writes a visible representation of the character c into a two-byte buffer pointed at by ptr. A printable character (including space through '~') is written as a space followed by the character. Other codes appear as:

CHARACTER	BECOMES
[0, 07]	\0 - \7
backspace	\b
tab	\t
newline	\n
vertical tab	\v
formfeed	\f
carriage return	\r
all other values	\?

RETURNS:

Nothing. mapchar writes two characters at ptr[0] and ptr[1].

EXAMPLE:

To output a visible representation of an arbitrary character, one might write:

```
TEXT c, str[2];

mapchar(c,str);
putfmt("%4b\n", str, 2);
```

SEE ALSO:

doesc

NAME:

match - Match a Regular Expression

SYNOPSIS:

```
BOOL match(buf, n, pat)
   TEXT *buf;
   BYTES n;
   TEXT *pat;
```

FUNCTION:

match tests the n character buffer starting at buf for a match with the encoded pattern starting at pat. It is assumed that the pattern was built by the function pattern, whose manual page describes the notation for regular expressions accepted by these routines.

RETURNS:

match returns YES if the pattern matches.

EXAMPLE:

To test a line for the presence of three colons:

```
if (match(line, n, pattern(pbuf, '\0', ":*:*:")))
    return (YES);
```

SEE ALSO:

pattern

NAME:

max - Test for Maximum

SYNOPSIS:

```
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:

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

SEE ALSO:

min

BUGS:

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 more than just once.

NAME:

min - Test for Minimum

SYNOPSIS:

```
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:

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

SEE ALSO:

max

BUGS:

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 other than just once.

NAME:

mkord - Make an Ordering Function

SYNOPSIS:

```
COUNT (*mkord(keyarray, lnordrule))()
   TEXT **keyarray, *lnordrule;
```

FUNCTION:

mkord uses the encoded text strings pointed at by lnordrule and the elements of keyarray to produce a function, suitable for use with sort, that compares two text buffers for lexical order. The function produced can be declared (symbolically, at least) as:

```
COUNT ordfun(i, j, ppa)
    BYTES i, j;
    struct {
        UCOUNT len;
        TEXT buf[len];
        } ***ppa;
```

That is, ppa is a pointer to an array of pointers to structures, each of which consists of a two-byte buffer length len, followed by the text buffer proper. The function is expected to compare the text in the structure pointed at by (*ppa)[i] with that in the structure pointed at by (*ppa)[j], returning a negative number if the first is less than the second, zero if the two compare equal, and positive otherwise.

keyarray is a NULL-terminated list of "keys", or ordering rules to be used by ordfun, listed in reverse order of application; i.e., keyarray[0] specifies a rule that is applied only if keyarray[1] is NULL or if it (and all higher rules) says that the two text buffers compare equal, on a given call to ordfun.

Each of the keys, as well as lnordrule, is a NUL-terminated string that specifies a rule (as shown below) for ordering two text buffers. lnordrule is the key tried last by ordfun; it also specifies the default method of comparison for any keys in keyarray that don't explicitly state a method. Thus, if keyarray[0] is NULL, lnordrule alone specifies the ordering.

Strings in lnordrule and keyarray take the form:

```
[adln][b][r][t?][#.#-#.#]
```

Where:

a - compares character-by-character in ASCII collating sequence. A missing character compares lower than any ASCII code.

b - skips leading whitespace.

d - compares character-by-character in dictionary collating sequence; i.e., characters other than letters, digits or spaces are omitted, and case distinctions among letters are ignored.

l - compares character-by-character in ASCII collating sequence, except that case distinctions among letters are ignored.

n - compares by arithmetic value, treating each buffer as a numeric string consisting of optional whitespace, optional minus sign and digits with an optional decimal point.

r - reverses the sense of comparisons.

t? - uses ? as the tab character for determining offsets (described below).

#.#- #.# describes offsets from the start of each text buffer for the beginning (first character used) and, after the minus '-', for the end (first character not used) of the text to be considered by the rule. The number before each dot '.' is the number of tab characters to skip, and the number after each dot is the number of characters to skip thereafter. Thus, in the string "abcd=efgh", with '=' as the tab character, the offset "1.2" would point to 'g', and "0.0 would point to 'a'. A missing number # is taken as zero; a missing final pair "-#.#" points just past the last of the text in each of the buffers to be compared. If the first offset is past the second offset, the buffer is considered empty.

If no tab character is specified, for each tab to be skipped a string of spaces, followed by nonspaces other than newlines, is skipped instead. Thus, in the string " ABC DEF GHI", the offset "3" would point to the space just after 'I'.

Only one of 'a', 'd', 'l' or 'n' may be present in a rule, and no more than ten ordering rules can be specified by keyarray.

RETURNS:

If all keys make sense, mkord returns a pointer to an internal ordering function as described above; otherwise it returns NULL. Various internal tables are rewritten, on each call to mkord, so only one ordering function may be defined at a time.

EXAMPLE:

```
#define MAXKEY  10
INTERN struct {
BYTES n;
TEXT *key[MAXKEY+1];
} kstack {MAXKEY};  /* kstack.key[MAXKEY] is always NULL */

getflags(&ac, &av, "+*):+[#.#-#.# a b d l n r t?]", &kstack);
order = mkord(&kstack.key[kstack.n], "a");
...
sort(nlines, order, &swapfn, linptrs);
```

SEE ALSO:

sort

BUGS:

It's useful, but difficult to use.

NAME:

nalloc - Allocate Space on the Heap

SYNOPSIS:

```
TEXT *nalloc(nbytes, link)
   BYTES nbytes;
   TEXT *link;
```

FUNCTION:

nalloc allocates space on the heap for an item of size nbytes, then writes link in the zeroth integer location. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which pointer is guaranteed to be on a proper storage boundary for anything. The heap is grown as necessary.

RETURNS:

nalloc returns a pointer to the allocated cell if successful; otherwise, it returns a NULL pointer.

EXAMPLE:

To build a stack:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

    top = nalloc(sizeof (*top), top);   /* pushes a cell */
```

SEE ALSO:

alloc, free, sbreak(4)

BUGS:

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily clobbered. This number is related to the actual cell size in a most system- dependent fashion and should not be trusted.

NAME:

notbuf - Find First Occurrence in Buffer of Character Not In Set

SYNOPSIS:

```
BYTES notbuf(p, n, s)
   TEXT *p, *s;
   BYTES n;
```

FUNCTION:

notbuf scans the n-character buffer starting at p for the first instance of a character not in the NUL-terminated set starting at s. If the NUL character is to be part of the set, it must be the first character in the set.

RETURNS:

notbuf returns the index of the first character in p not contained in the set s, or the value n if all buffer characters are in the set.

EXAMPLE:

To check that an input string contains only digits:

```
if (notbuf(buf, n, "0123456789") < n)
    errfmt("illegal number\n");
```

SEE ALSO:

inbuf, instr, lenstr, notstr, scnbuf, scnstr, subbuf, substr

NAME:

notstr - Find First Occurrence in String of Character Not In Set

SYNOPSIS:

```
BYTES notstr(p, s)
   TEXT *p, *s;
```

FUNCTION:

notstr scans the NUL-terminated string starting at p for the first occurrence of a character not in the NUL-terminated set starting at s.

RETURNS:

notstr returns the index of the first character in p not contained in the set s, or the index of the terminating NUL if all are in s.

EXAMPLE:

To check a string for non-numeric characters:

```
if (str[notstr(str, "0123456789")])
    errfmt("illegal number\n");
```

SEE ALSO:

inbuf, instr, notbuf, scnbuf, scnstr, subbuf, substr

NAME:

ordbuf - Compare Two NUL-Padded Buffers for Lexical Order

SYNOPSIS:

```
COUNT ordbuf(s1, s2, n)
   TEXT *s1, *s2;
   COUNT n;
```

FUNCTION:

ordbuf 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.

Note that encoded numbers, such as int or double, seldom sort properly when treated as text strings.

RETURNS:

The value returned is -1 when s1 is lower, 0 when s1 equals s2, and +1 when s2 is lower.

EXAMPLE:

```
sort(nthings, &ordbuf, &swap, &data);
```

SEE ALSO:

sort

NAME:

pathnm - Complete a Pathname

SYNOPSIS:

```
TEXT *pathnm(buf, n1, n2)
   TEXT *buf, *n1, *n2;
```

FUNCTION:

pathnm builds a pathname in buf that is derived from the pair of NUL-terminated names pointed at by n1 and n2.

If the name pointed at by n2 ends in ':' or ']' then the longest suffix of the string pointed at by n1 that does not contain a ':', ']', or '/' is appended to it in buf. If the string pointed at by n2 does not end in ':' or ']', then a '/' followed by the same suffix is appended to the n2 string in buf.

Thus, the following results are obtained:

n1	n2	buf
x	y	y/x
x	a:	a:x
x	[2,3]	[2,3]x
z/x	y	y/x
a:x	b:	b:x
:f1:x	:f2:	:f2:x

This scheme is designed to be maximally convenient on numerous operating systems, provided that truly esoteric filenames, such as "a/3:", are avoided.

RETURNS:

pathnm returns the concatenation of n2, possibly a '/', and the suffix of n1, NUL terminated in the area pointed at by buf. The value of the function is always buf.

BUGS:

There is no way to specify the size of buf, which must be at least lenstr(n1) + lenstr(n2) + 2 characters.

NAME:

pattern - Build a Regular Expression Pattern

SYNOPSIS:

```
TEXT *pattern(pat, delim, p)
   TEXT *pat, delim, *p;
```

FUNCTION:

pattern builds an encoded pattern in the string buffer starting at pat, suitable for use with amatch or match in matching regular expressions. The pattern is encoded from the string p, which should be terminated by an unescaped instance of delim, but which must be NUL terminated to prevent an ill-formed pattern from compounding the code. It is assumed that p points just past the left delimiter.

Code values for the encoded pattern are listed in the manual page for amatch; simple usage, however, requires no knowledge of these inner workings. It is sufficient to know that the encoded string at pat will never occupy more than twice as many bytes as the string p, counting delimiters at both ends.

A regular expression is a shorthand notation for a sequence of target characters contained in a temporary file line. These characters are said to "match" the regular expression. The following regular expressions are allowed:

- An ordinary character is considered a regular expression which matches that character.
- The character sequences "\b", "\f", "\n", "\r", "\t", "\v", in upper- or lower-case, are regular expressions each representing the single character cursor movements of ⟨backspace⟩, ⟨formfeed⟩, ⟨newline⟩, ⟨return⟩, ⟨tab⟩, ⟨vertical tab⟩, respectively. Additionally, any single character in the character set may be represented by the form "\ddd" where ddd is the one to three digit octal representation of the character; this is the safest way to match most nonprinting characters, and the only way to match ASCII NUL (\0).
- A '?' matches any single character except a ⟨newline⟩.
- A '^' as the leftmost character of a series of regular expressions constrains the match to begin at the beginning of the line.

- A '^' following a character matches zero or more occurrences of that character. This pattern may, thus, match a null string which occurs at the beginning of a line, between pairs of characters or at the end of the line. A '^' enclosed in "\(" and "\)", or following either a '\' or an initial '^', is taken as a literal '^', however.

- A '^' in any position other than the ones mentioned above is taken as a literal '^'.

- A '*' matches zero or more characters, not including <newline>. It is conceptually identical to the sequence "?^".

- A character string enclosed in square brackets "[]" matches a single character which may be any of the characters in the bracketed list, but no other. However, if the first character of the string is a '!', this expression matches any character except <newline> and the ones in the bracketed list. A range of characters in the character collating sequence may be indicated by the sequence of <lowest character>, '-', <highest character>. ([z-a] will not work and it is ignored.) Thus, [ej-maE] is a regular expression which will match one character that may be E, a, e, j, k, l or m. When matching a literal "-", the "-" must be the first or last character in the bracketed list; otherwise it is taken to specify a range of characters.

- A regular expression enclosed between the sequences "\(" and "\)" tags this expression in a way useful for substitutions, but otherwise has no effect on the characters the expression matches. (See the s command for further explanation.)

- A concatenation of regular expressions matches the concatenation of strings matched by individual regular expressions. In other words, a regular expression composed of several "sub expressions" will match a concatenation of the strings implied by each of the individual "subexpressions".

- A '$' as the right-most character after a series of regular expressions constrains the match, if any, to end at the end of the line prior to the <newline>.

- A null regular expression standing alone stands for the last regular expression encountered.

Note that arbitrary grouping and alternation are not fully supported by this notation, as the text patterns utilized are not the full class of regular mathematical expressions.

RETURNS:

pattern returns pat, if no syntax errors are found in p, else NULL.

EXAMPLE:

```
pattern(pbuf, '0', "^??????T ");
while (match(buf, n = getlin(buf, MAXBUF), pbuf))
    putlin(buf, n);
```

SEE ALSO:

match

NAME:

prefix - Test if One String is a Prefix of the Other

SYNOPSIS:

```
BOOL prefix(s1, s2)
   TEXT *s1, *s2;
```

FUNCTION:

prefix compares two strings, character-by-character, for equality. The first string starts at s1 and is terminated by a NUL '\0'; the second is likewise described by s2. The strings must match up to but not including the NUL terminating the second string; i.e., s2 must be a prefix of s1.

RETURNS:

The value returned is YES if s2 is a prefix of s1, else NO.

EXAMPLE:

```
if (prefix(line, "#include ")
    doinclude();
```

SEE ALSO:

cmpbuf, cmpstr

NAME:

putc - Put a Character to Output Buffer

SYNOPSIS:

```
COUNT putc(pfio, c)
   FIO *pfio;
   COUNT c;
```

FUNCTION:

If c is not negative, it is treated as a character to be copied to the file controlled by the FIO buffer at pfio; otherwise putc simply ensures that all characters in the buffer are written out. It may be necessary to explicitly drain the output buffer in this fashion if putc is used to buffer output, unless pfio has been initialized by finit which then will take care to drain the output buffer on exit from the user program. If the pfio buffer has been opened for WRITE, the output buffer is drained whenever a newline is encountered.

RETURNS:

putc returns c. An error exit occurs if any writes fail, or if (pfio == NULL).

EXAMPLE:

To copy a file, character by character:

```
while (putc(&stdout, getc(&stdin)) != EOF)
    ;
```

SEE ALSO:

finit, getc, getch, putch

BUGS:

Arbitrary characters, as opposed to ASCII text, are often sign-extended to make negative integers; these quietly disappear on putc calls.

NAME:

putch - Put a Character to stdout Buffer

SYNOPSIS:

```
COUNT putch(c)
   COUNT c;
```

FUNCTION:

If c is not negative, it is treated as a character to be copied to the file controlled by the FIO buffer stdout; otherwise putch simply ensures that all characters in the buffer are written out. It should not be necessary to explicitly drain the stdout buffer in this fashion if putch is used to buffer text output.

RETURNS:

putch returns c. An error exit occurs if any writes fail.

EXAMPLE:

To copy a file, character by character:

```
while (putch(getch()) != EOF)
    ;
```

SEE ALSO:

finit, getc, getch, putc, stdout

BUGS:

The stdout buffer is drained only when the character written is a newline. If stdout has not been explicitly initialized before use by the call

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If nontext output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used. Arbitrary characters, as opposed to ASCII text, are often sign-extended to make negative integers; these quietly disappear on putch calls unless masked properly.

NAME:

putf - Output Arguments Formatted

SYNOPSIS:

```
VOID putf(pfio, fmt, arg1, arg2, ...)
   FIO *pfio;
   TEXT *fmt;
   ...
```

FUNCTION:

putf converts a series of arguments arg1,... to text, which is output to the file controlled by the FIO buffer at pfio, under control of a format string at fmt. The format string consists of literal text to be output, interspersed with <field-specifier>s that determine how the arguments are to be interpreted and how they are to be converted for output.

A <field-specifier> takes the form:

```
%[+z|-z][#]<field-code>
```

That is, a <field-specifier> consists of a literal '%', followed by an optional "+z" or "-z", where z can be any character, followed by an optional field width #, and is terminated by a <field-code>. If present, a "+z" calls for the field to be left-filled with the character z, while "-z" calls for the field to be right-filled with the character z. If present, a # specifies total width in characters of the field to be output, and is either a decimal integer, or the letter 'n'. If an 'n' is given, then the value of the next argument from the argument list is taken to specify the field width.

For example, if arg1 is a double with the value 100.53, then:

```
putf(&stdout, "%+*9.2f", arg1);
putf(&stdout, "%+*n.nf", 9, 2, arg1);
```

both will output:

```
***100.53
```

If the number of characters needed to represent the output item is less than the field width, fill characters are used to left or right pad the item up to the field width. By default, left fill with spaces is used. The default field width is zero.

A <field-code> is composed of a <modifier>, a <specifier> or both. The <specifier> defines how an output field is to be represented, and is one of the following:

```
c = char integer
s = short integer
i = integer
l = long integer
p = NUL-terminated string
b = buffer of specified length
d = double output in scientific notation (e.g., 1.00e+00)
f = double output in fixed point notation (e.g., 1.00)
x = fill characters (usually spaces) only
```

A <modifier> causes an integer value to be output as:

```
a = ASCII characters
h = hexadecimal (no leading "0x")
o = octal (no leading '0')
u = unsigned decimal
```

If no <specifier> is given, it is presumed to be 'i', and a <modifier> given from the above series will be taken to apply to the implied integer field. If a <specifier> of 'c', 's', 'i' or 'l' is given with no <modifier>, the associated value is output in signed decimal.

In addition, an optional precision modifier, ".#", limits the number of characters actually output with a <specifier> of 'p' or 'b', and specifies the number of fractional digits output with a <specifier> of 'd' or 'f'. Like the field width specifier, the precision modifier # may be either an explicit integer, or an 'n', to make use of the next argument value in sequence.

Hence a <field-code> usually consists of one of the following combinations of <specifier> and <modifier>:

```
[a|h|o|u]{c|s|i|l}      /* integer output */
[.#]{b|p|d|f}           /* string or floating output */
{a|h|o|u}               /* default specifier is i */
{x}                     /* just output fill characters */
```

Any other character in the place of a <field-code> is taken as a single literal character to be output, permitting a '%' to be output with a "%%" specification.

The 'a' modifier treats the integer as a sequence of characters of the appropriate length, and outputs the characters in descending order of their ignificance within the number. This permits multi-yte binary data to be written to a file in a host-independent manner.

A string of characters may be output under the 'p' field code, if it is NUL-terminated, or under 'b' if its length is known.

If arg2 is a vector containing the 11-character NUL-terminated string "hello world", either of these calls would output the string:

```
putf(&stdout, "%p\n", arg2);
putf(&stdout, "%b\n", arg2, 11);
```

In the first case, the argument is a pointer to the beginning of the string; in the second case two arguments are used, one a pointer to the start of the string and the second an integer specifying its length. In either case, the number of characters actually output will be no more than the precision modifier, if it is present and nonzero.

A double (or float) number may be output with 'd' or 'f', the precision modifier specifying the number of characters to the right of the decimal point. For the 'd' field code, the number is written in the scientific notation form:

```
[-]#.#*e{+|-}#*
```

There is always one digit to the left of the decimal point; there are either two or three digits in the exponent, depending on the target machine. The 'f' field code prints the number in fixed point format, i.e., without exponent. In either case, no more than 24 characters will be output.

For example, if arg1 is a double with the value 100.53, then:

```
putf(&stdout, "%1.4d\n", arg1);
```

would output it as:

```
1.0053e+02
```

while:

```
putf(&stdout, "$%6.nf", 2, arg1);
```

would output it in fixed point notation as:

```
$100.53
```

The 'x' field code consumes no arguments; it is a convenient way to output pure filler.

RETURNS:

Nothing. An error exit occurs if any writes fail, or if (pfio == NULL).

EXAMPLE:

```
putf(&stdout, "%i errors in file %p\n", nerrors, fname);
```

SEE ALSO:

decode, dtoe, dtof, errfmt, putfmt

BUGS:

A call with more <field-specifier>s than argument variables will produce unpedictable results.

NAME:

putfmt - Format Arguments to stdout

SYNOPSIS:

```
VOID putfmt(fmt, arg1, arg2, ...)
   TEXT *fmt;
```

FUNCTION:

putfmt writes formatted output to the file controlled by the FIO buffer stdout, using the format string at fmt and the arguments argx, in exactly the same way as putf.

RETURNS:

Nothing. An error exit occurs if any writes fail.

EXAMPLE:

```
putfmt("%i:%p\n", lineno, line);
```

SEE ALSO:

decode, errfmt, finit, putf, stdout

BUGS:

The stdout buffer is drained only when the last character written is a newline. If stdout has not been explicitly initialized before use by the call

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If nontext output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used.

NAME: putl - Put a Text Line from Buffer

SYNOPSIS:

```
BYTES putl(pfio, s, n)
   FIO *pfio;
   TEXT *s;
   BYTES n;
```

FUNCTION:

putl copies characters from the n character buffer starting at s to the file controlled by the FIO buffer at pfio.

RETURNS:

putl returns n. An error exit occurs if any writes fail, or if (pfio == NULL).

EXAMPLE:

To copy a text file, line-by-line:

```
while (putl(&stdout, buf, getl(&stdin, buf, BUFSIZE)))
    ;
```

SEE ALSO:

getl, getlin, putlin

NAME:

putlin - Put a Text Line to stdout

SYNOPSIS:

```
BYTES putlin(s, n)
   TEXT *s;
   BYTES n;
```

FUNCTION:

putlin copies characters from the n character buffer starting at s to the file controlled by the FIO buffer stdout.

RETURNS:

putlin returns n. An error exit occurs if any writes fail.

EXAMPLE:

To copy a text file, line by line:

```
while (putlin(buf, getlin(buf, BUFSIZE)))
    ;
```

SEE ALSO:

finit, getl, getlin, putl, stdout

BUGS:

The stdout buffer is drained only when the last character written is a newline. If stdout has not been explicitly initialized before use by the call:

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If nontext output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used.

NAME:

putstr - Copy Multiple Strings to File

SYNOPSIS:

```
VOID putstr(fd, arg1, arg2,  ..., NULL)
   FILE fd;
   TEXT *arg1, *arg2,  ...;
```

FUNCTION:

putstr writes a series of strings out to a file with descriptor fd. Each string begins at arg1,... and is terminated by a NUL '\0'. The series of string arguments is terminated by a NULL pointer argument. For each string, putstr invokes lenstr to discover its size and issues a call directly to write; therefore, putstr should only be used for low volume output.

RETURNS:

Nothing.

EXAMPLE:

```
putstr(STDERR, fname, ": bad format\n", NULL);
```

SEE ALSO:

lenstr, write(4)

BUGS:

Forgetting the terminating NULL pointer is usually disastrous.

NAME:

remark - Print Nonfatal Error Message

SYNOPSIS:

```
VOID remark(s1, s2);
   TEXT *s1, *s2;
```

FUNCTION:

remark prints an error message to STDERR, consisting of the concatenation of strings s1 and s2, followed by a newline. It then returns to the caller for further processing.

RETURNS:

Nothing.

EXAMPLE:

```
if ((fd = open(name, READ, 0)) < 0)
    remark("can't open: ", name);
```

SEE ALSO:

errfmt, error, putstr

NAME:

scnbuf - Scan Buffer for Character

SYNOPSIS:

```
BYTES scnbuf(s, n, c)
   TEXT *s;
   BYTES n;
   TEXT c;
```

FUNCTION:

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

RETURNS:

scnbuf returns the index of the first character that matches c, or n if none.

EXAMPLE:

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

```
if ((n = scnbuf(keybuf, KEYSIZ, *s)) != KEYSIZ)
    *s = subst[n];
```

SEE ALSO:

inbuf, instr, notbuf, notstr, scnstr, subbuf, substr

NAME:

scnstr - Scan String for Character

SYNOPSIS:

```
BYTES scnstr(s, c)
   TEXT *s, c;
```

FUNCTION:

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

RETURNS:

scnstr returns the index of the first character that matches c, or the index of the terminating NUL if none does.

EXAMPLE:

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

```
if (s[n = scnstr(keystr, *s)])
    *s = subst[n];
```

SEE ALSO:

inbuf, instr, notbuf, notstr, scnbuf, subbuf, substr

NAME:

sin - Sine in Radians

SYNOPSIS:

```
DOUBLE sin(x)
   DOUBLE x;
```

FUNCTION:

sin computes the sine of x, expressed in radians, to full double precision. It works by scaling x in quadrants, then computing the appropriate sin or cos of an angle in the first half quadrant, using a sixth-order telescoped Taylor series approximation. If the magnitude of x is too large to contain a fractional quadrant part, the value of sin is 0.

RETURNS:

sin returns the nearest internal representation to sin x, expressed as a double floating 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:

cos

NAME:

sort - Sort Items in Memory

SYNOPSIS:

```
VOID sort(n, ordf, excf, base)
   ARGINT n;
   COUNT (*ordf)();
   VOID (*excf)();
   TEXT *base;
```

FUNCTION:

sort orders n items in memory using the quicksort algorithm. It decides whether items i and j are in order by performing the call

```
(*ordf)(i, j, &base);
```

where i and j are both guaranteed to be in the range [0, n). If (item i is to sort less than item j) then the value returned must be less than zero; otherwise if (item i is to sort equal to item j) then the value returned must be zero; the value is otherwise unconstrained.

To exchange two items, sort makes the call

```
(*excf)(i, j, &base);
```

and henceforth presumes that the items are interchanged.

Note that it is the address of base that is passed to both functions. This permits multiple parameters to follow base in the original argument list, which can be accessed as members of a structure pointed to by &base, providing the structure is declared with careful knowledge of how C passes arguments. For ordering and exchange functions in the know, base can also simply be ignored.

RETURNS:

Nothing. The items are sorted in place.

EXAMPLE:

To sort an array of short integers in ascending order:

```
COUNT iord(i, j, pa)
    COUNT i, j, **pa;
    {
    return ((*pa)[i] - (*pa)[j]);
    }

VOID iswap(i, j, pa)
    COUNT i, j, **pa;
    {
    COUNT t;

    t = (*pa)[i], (*pa)[i] = (*pa)[j], (*pa)[j] = t;
    }
VOID isort(a, n)
    COUNT a[], n;
    {
    sort(n, &iord, &iswap, a);
    }
```

BUGS:

It cannot sort more than half of memory, i.e.; n is taken as signed and must be positive.

NAME:

sqrt - Real Square Root

SYNOPSIS:

```
DOUBLE sqrt(x)
   DOUBLE x;
```

FUNCTION:

sqrt computes the square root of x to full double precision. It works by expressing x as a fraction in the interval [1/2, 1), times an integer power of two. The square root of the fraction is obtained by three iterations of Newton's method, using a quadratic approximation as a starting value.

RETURNS:

sqrt returns the nearest internal representation to sqrt x, expressed as a double floating value. If x is negative, a domain error condition is raised.

EXAMPLE:

To find the magnitude of a vector:

```
mag = sqrt(x * x + y * y);
```

SEE ALSO:

_domain(5), exp

NAME:

squeeze - Delete Specified Character from Buffer

SYNOPSIS:

```
BYTES squeeze(s, n, c)
   TEXT c, *s;
   BYTES n;
```

FUNCTION:

squeeze deletes character c from the n-character buffer starting at s, and compresses it in place.

RETURNS:

squeeze returns the number of chararcters remaining in s, which is in the interval [0, n].

EXAMPLE:

To write out a buffer after stripping off NULs and carriage returns:

```
write(STDOUT, buf, squeeze(buf, squeeze(buf, BUFSIZE, '\0'), '\r'));
```

SEE ALSO:

fill

NAME:

stdin - The Standard Input Control Buffer

SYNOPSIS:

```
FIO stdin;
```

FUNCTION:

stdin is an FIO control buffer initialized for input from STDIN.

EXAMPLE:

To count lines:

```
for (nl = 0; getl(&stdin, buf, BUFSIZE); ++nl)
    ;
```

SEE ALSO:

stdout

NAME:

stdout - The Standard Output Control Buffer

SYNOPSIS:

```
FIO stdout;
```

FUNCTION:

stdout is an FIO control buffer initialized for output to STDOUT.

EXAMPLE:

```
putl(&stdout, outbuf, outsiz);
```

SEE ALSO:

finit, stdin

BUGS:

stdout should not be used for nontext output unless initialized before use by:

```
finit(&stdout, STDOUT, BWRITE);
```

NAME:

stob - Convert Short to Text in Buffer

SYNOPSIS:

```
BYTES stob(s, i, base)
   TEXT *s;
   COUNT i;
   COUNT base;
```

FUNCTION:

stob converts the short i to a text representation in the buffer starting at s. The number is represented in the base specified, using lower-case letters beginning with 'a' to specify digits from ten on. If (0 < base), the number i is taken as unsigned; otherwise if (base < 0), negative numbers have a leading minus sign and are converted to -base; if (base == 0), it is taken as -10. Only magnitudes of base between 2 and 36 are generally meaningful, but no check is made for reasonableness.

RETURNS:

The value returned is the number of characters used to represent the short, which, in hexadecimal, can be up to four digits plus sign.

EXAMPLE:

To output i in decimal:

```
write(STDOUT, buf, stob(buf, i, 10));
```

SEE ALSO:

btoi, btol, btos, itob, ltob

BUGS:

The length of the buffer is not specifiable. If (|base| == 1) the program can bomb; if (36 < |base|) unpredictable characters can be inserted in the buffer.

NAME:

subbuf - Find Occurrence of Substring in Buffer

SYNOPSIS:

```
BYTES subbuf(s, ns, p, np)
   TEXT *s, *p;
   BYTES ns, np;
```

FUNCTION:

subbuf scans the buffer starting at s of size ns, and looks for the first occurrence of the substring at p of size np.

RETURNS:

The value returned is the index in s of the left-most character in the substring if subbuf is successful; otherwise, ns is returned.

EXAMPLE:

```
for(p = buf, i = size; (j = subbuf(p, i, "\r\n", 2)) < i;
    p =+ j + 2, i =- j + 2)
    {
    write(fd, p, j);
    write(fd, "\n", 1);
    }
```

SEE ALSO:

inbuf, instr, match, notbuf, notstr, scnbuf, scnstr, substr

NAME:

substr - Find Occurrence of Substring

SYNOPSIS:

```
BYTES substr(s, p)
   TEXT *s, *p;
```

FUNCTION:

substr scans the string starting at s, and looks for the first occurrence of the substring at p.

RETURNS:

The value returned is the index in s of the left-most character in the substring if substr is successful; otherwise, the index of the terminating NUL is returned.

EXAMPLE:

```
if (line[substr(line, "Page")])
    putfmt("%s: %\n", lno / 66 + 1, line);
```

SEE ALSO:

inbuf, instr, match, notbuf, notstr, scnbuf, scnstr, subbuf

NAME:

tolower - Convert Character to Lower-Case if Necessary

SYNOPSIS:

```
tolower(c)
```

FUNCTION:

tolower converts an upper-case letter to its lower-case equivalent, leaving all other characters unscathed.

RETURNS:

tolower is a numerical rvalue guaranteed not to be an upper-case character.

EXAMPLE:

To accumulate a hexadecimal digit:

```
if ('a' <= c && c <= 'f' || 'A' <= c && c <= 'F')
    sum = sum * 10 + tolower(c) + (10 - 'a');
```

SEE ALSO:

isalpha, isdigit, islower, isupper, iswhite, toupper

BUGS:

Because it is a macro, tolower cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated more than once.

NAME:

toupper - Convert Character to Upper-Case if Necessary

SYNOPSIS:

```
toupper(c)
```

FUNCTION:

toupper converts a lower-case letter to its upper-case equivalent, leaving all other characters unscathed.

RETURNS:

toupper is a numerical rvalue guaranteed not to be a lower-case character.

EXAMPLE:

To convert a character string to upppercase letters:

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

SEE ALSO:

isalpha, isdigit, islower, isupper, iswhite, tolower

BUGS:

Because it is a macro, toupper cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated more than once.

NAME:

usage - Output Standard Usage Information

SYNOPSIS:

```
COUNT usage(msg)
   TEXT *msg;
```

FUNCTION:

usage outputs to STDERR the string "usage: <pname> ", followed by the string pointed to by msg, where <pname> is the name by which the current program was invoked. If msg is terminated with a newline, usage immediately takes an error exit.

RETURNS:

If usage returns to the caller, its value is the number of characters output to STDERR.

EXAMPLE:

```
if (1 < aflag + bflag + nflag)
usage("-[a b n] <files>\n");
```

SEE ALSO:

_pname(4), getflags

CHAPTER 4
C SYSTEM INTERFACE LIBRARY

TABLE OF CONTENTS

NAME:

Cint - C Interface to Operating System

FUNCTION:

C programs operating in user mode under any operating system may assume the existence of several functions which implement program entry/exit and low-level input/output (I/O). This section documents these functions, plus several critical presumptions that can be made about the environment supplied, in the most portable of terms. Details of actual implementations may be found in the various C interface manuals; but these are best ignored if portability is considered a virtue.

Each C program must provide a function main(), detailed on a separate manual page that has access to the command line used to invoke the program. Returning from main, or calling exit(), terminates program execution and reports, at most, one bit of status, success or failure, to the invoker.

C programs may assume the existence of three open text files: STDIN (file descriptor (fd) 0), STDOUT (fd 1), and and STDERR (fd 2). The first may be used with read() and close(); the latter two may be used with write() and close().

The standard input, STDIN and standard output, STDOUT may be redirected on the command line (transparently to the program); the standard error file, STDERR is a reliable destination for error messages. The following conventions apply to I/O:

filename	is a string, hence a NUL terminated hence a pointer to char when used as an argument. For maximum portability, a filename should consist of letters, of one case only, and digits. The first character should be a letter and there should be no more than six characters, optionally followed by a '.' and no more than two more letters.
file descriptor	is a short integer (type FILE in the standard header std.h) that is guaranteed to be non-negative. Its value should be otherwise assumed to be magic.
mode	is a short integer that specifies reading (mode == 0), writing (mode == 1) or updating (mode == 2). When std.h or stdio.h are included, mode may be combined with IASCII, IBINARY or IRAW as, for example, IASCII|1 to obtain ASCII write mode. (See Appendix A for details on mode.)

binary file — looks to a C program like a sequence of characters. There is no record structure and all character codes are allowed. Trailing NULs may be provided by some operating systems.

text file — is much like a binary file, except it is assumed to contain printable text that may be mapped between internal and external forms. Most programs deal with such files of printable text, where a line structure is imposed (internally) by the presence of a newline (ASCII line feed) character at the end of each line. Lines can be assumed never to be longer than 512 characters, counting the terminating newline, nor should a text file ever be produced whose last line has no newline at the end.

Space is reserved for each program to grow a stack, or LIFO list option call argument lists and automatic storage frames, and a heap, structured data area. Heap is purchased in (not necessarily contiguous) chunks by calls on sbreak(), and is never given back during program execution. Stack and heap must often contend for the same (limited) space, so an otherwise correct C program may terminate early, or misbehave, because insufficient space was allotted.

Note that all objects in C are presumed to have non-NULL addresses; the system is never to bind an external identifier to the value zero. The system interface ensures that address zero never occurs on the stack or heap, as well. In fact, the addresses -1 and +1 are also discouraged, since some functions treat these values as codes for discredited pointers, much like NULL (0).

NAME:

main - Enter a C Program

SYNOPSIS:

```
BOOL main(ac, av)
   BYTES ac;
   TEXT **av;
```

FUNCTION:

main is the function called to initiate a C program; hence every user program must contain a function called main. Its arguments are a sequence of NUL-terminated strings, pointed at by the first ac elements of the array av, obtained from the command line used to invoke the programs. By convention, ac is always at least one, av[0] is the name by which the program has been invoked, and av[1], if present, is the first argument string, etc. Program execution is terminated by returning from main, or by an explicit call to exit. In either case, one bit of status is returned to the invoker to signify whether the program ran successfully.

RETURNS:

main returns zero if successful, otherwise nonzero.

EXAMPLE:

```
/*  ECHO ARGUMENTS TO STDOUT
 *  copyright (c) 1980 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    if (1 < ac)
        {
        putstr(STDOUT, *++av, NULL);
        for (--ac, ++av; --ac; ++av)
            putstr(STDOUT, " ", *av, NULL);
        write(STDOUT, "\n", 1);
        }
    return (YES);
    }
```

NAME:

_pname - Program Name

SYNOPSIS:

```
TEXT _pname;
```

FUNCTION:

_pname is the (NUL-terminated) name by which the program was invoked, if that can be determined from the command line, or the name provided by the C programmer, if present, or the name "error", delivered up by a waiting library module. The library definition is used only if no definition of _pname is provided by the C program and/or the compile-time name is not overridden at run-time.

It is used primarily for labelling diagnostic printouts.

SEE ALSO:

error(3)

NAME:

brk, sbrk, break - Change Core Allocation

SYNOPSIS:

```
char *brk(addr)
char *sbrk(incr)
```

FUNCTION:

brk sets the system's idea of the lowest location not used by the program (called the break) to addr. Locations not less than addr are still in the address space but not available for use.

In the alternate function sbrk, incr more bytes are added to the program's data space and a pointer to the start of the new area is returned.

Break performs the function of brk. The name of the routine differs from that in C for historical reasons.

SEE ALSO:

malloc(2), end(2)

DIAGNOSTICS:

Zero is returned if the break could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break.

NAME:

chdir - Change Default Volume and Account

SYNOPSIS:

```
chdir(dirname)
char *dirname;
```

FUNCTION:

Dirname is the address of a null-terminated string containing an optional volume name terminated with a ":" followed by an optional account preceded by a "/". Chdir causes this volume and account to be used with filenames not beginning with a volume or ending with an account.

SEE ALSO

Appendix A

DIAGNOSTICS:

Zero is returned if the call is successful; -1 is returned if the given name is not that of a volume and/or account.

NAME:

close - Close a File

SYNOPSIS:

```
FILE close(fd)
   FILE fd;
```

FUNCTION:

close closes the file associated with the file descriptor fd, making the fd available for future open or create calls.

RETURNS:

close returns the now useless file descriptor, if successful, or a negative number.

EXAMPLE:

To copy an arbitrary number of files:

```
while (fd = getfiles(&ac, &av, STDIN, -1))
    {
    while (0 < (n = read(fd, buf, BUFSIZE)))
        write(STDOUT, buf, n);
    close(fd);
    }
```

SEE ALSO:

create, open, remove, uname

NAME:

creat - Create a New File

SYNOPSIS:

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

FUNCTION:

creat creates a new file or prepares to rewrite an existing file called name, given as the address of a null-terminated string. If the file does not exist, it is given mode mode. See Appendix A for construction of the mode argument.

If the file does exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing and its fd is returned.

The mode given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then, if a second instance of the program attempts a creat, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO:

write(4), close(4), open (4)

DIAGNOSTICS:

The value -1 is returned if: the file does not exist and is unwritable; the file is a directory; there are already too many files open.

NAME:

create - Open an Empty Instance of a File

SYNOPSIS:

```
FILE create(fname, mode, rsize)
   TEXT *fname;
   COUNT mode;
   BYTES rsize;
```

FUNCTION:

create makes a new file fname, if it did not previously exist, or truncates the existing file to zero length. If (mode == 0), the file is opened for reading, else if (mode == 1), it is opened for writing, else (mode ==2), of necessity and the file is opened for updating (reading and writing).

If the file is to contain arbitrary binary data, as opposed to printable ASCII text, the record size rsize should be nonzero. Not all systems behave well if a textfile is created for updating.

RETURNS:

create returns an fd for the created file or a negative number.

EXAMPLE:

```
if ((fd = create("xeq", WRITE, 1)) < 0)
    write(STDERR, "can't create xeq\n", 17);
```

SEE ALSO:

close, open, remove, uname

NAME:

ctime, localtime, gmtime, asctime, timezone - Convert Date and Time to ASCII

SYNOPSIS:

```
chqr *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

setzone (zone, min, dst)
int zone, min, dst;
```

FUNCTION:

ctime converts a time pointed to by clock such as returned by time(4) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 10:03:52 1073\n\0
```

localtime and gmtime return pointers to structures containing the broken-down time. localtime corrects for the time zone and possible daylight savings time; gmtime converts to GMT, which is the time UNIX uses. asctime converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {/*see ctime(4) */

        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday - 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults _timezon and _dst (see bugs below) to determine the time zone and whether the standard USA daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

setzone sets the current time zone and a flag which indicates whether daylight savings time takes arguments which give setzone or the zone in hours or minutes west of Greenwich. (To go east of Greenwich negative values should be used.) The final argument, dst, indicates whether this time zone is subject to the daylight savings time transformation.

SEE ALSO:

time(4)

BUGS:

The return values point to static data whose content is overwritten by each call.

The default values of zone and dst which affect the translation of time are 5 and 1, respectively. This means that the run-time system thinks the OS/32 clock is measuring time in USA eastern standard time and that daylight savings applies. In order to get the proper time from gmtime for other time zones, either setzone must be called, or _timezon, _tzmins and _dst must be patched to the proper values in cinit.obj.

NAME:

envir - C Run-Time Environment

SYNOPSIS:

```
#include <envir.h>

extern struct _ENVIR *_envir;
```

FUNCTION:

_envir is a pointer in cinit which points to a structure which is filled in at start-up with information used by the run-time library (RTL).

Details of the various fields are given in Appendix A.

DESCRIPTION:

```
/*
 *   Environment header file
 *   describes the storage area in cinit
 */
struct _ENVIR {

    /* this group is set by cinit, _cinit or _stdass and are
       used by various parts of the run-time library */

    int _stksz; /* number of bytes reserved for stack default 8k */
    int *_stktop; /* top of the stack (only used during startup) */
    int *_stkend; /* bottom of stack area */
    int *_heaptop; /* top of the heap (only used during startup) */
    int _maxlu;    /* maximum number of lu's available */
    struct inode *_lutab; /* pointer to table of lu descriptors */
    int _tskops;   /* task options word */
    int _sysops;   /* system options word */
    int _uacc;     /* user private account number */
    int _gacc;     /* user group account number   */
    char *_uid;    /* user's signod id (mtm only) */
    char *_task_id; /* filename from which task was loaded */
    char *_def_dev; /* the name of the default console dev */

    /* the next group are set by chdir */

    char *_def_acc; /* default account number for open */
    char *_def_vol; /* default volume name for open    */
```

```
    /* the next group are set by setzone (see ctime(4) */

    int _timezone;  /* timezone in hours west of Greenwich */
    int _tzmins;    /* alternate in minutes                */
    int _dst;       /* flag indicating if daylight saving applies */
};
```

SEE ALSO:

Appendix A, time(4), ctime(4)

NAME:

exit - Terminate Process

SYNOPSIS:

```
exit(status)
init status;

_exit(status)
int status;
```

FUNCTION:

exit is the normal means of terminating a process. exit closes all the process's and notifies the parent process if it is executing a wait. The low-order eight bits of status are available to the parent process.

This call can never return.

The C function exit may cause cleanup actions before the final "sys exit". The function _exit circumvents all cleanup.

NAME:

getuid, getgid, geteuid, getegid - Get User and Group Identity

SYNOPSIS:

```
getuid()
geteuid()
getgid()
getegid()
```

FUNCTION:

getuid and geteuid both return the current multi-terminal monitor (MTM) user account number as an int. getgid and getegid do the same for the group account.

NAME:

lseek, tell - Move Read/Write Pointer

SYNOPSIS:

```
long lseek(fildes, offset, whence)
long offset;

long tell(fildes)
```

FUNCTION:

The fd refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

- If whence is 0, the pointer is set to offset bytes.
- If whence is 1, the pointer is set to the current position plus offset bytes.
- If whence is 2, the pointer is set to the end of the file plus offset bytes.

The returned value is the resulting pointer location.

The obsolete function tell(fildes) is identical to lseek(fildes,OL, 1).

Seeking far beyond the end of a file, then writing, creates a gap or "hole", which occupies no physical space and reads as zeros.

SEE ALSO:

open(4), creat(4), fseek(2)

DIAGNOSTICS:

-1 is returned for an undefined fd or seek to a position before the beginning of file.

BUGS:

lseek is a no-op on terminals and printers. lseek also may position incorrectly if used on binary files opened for append (see Appendix A).

NAME:

onexit - Call Function on Program Exit

SYNOPSIS:

```
VOID (*onexit())(pfn)
   VOID (*(*pfn)())();
```

FUNCTION:

onexit registers the function pointed at by pfn, to be called on program exit. The function at pfn is obliged to return the pointer returned by the onexit call, so that any previously registered functions can also be called.

RETURNS:

onexit returns a pointer to another function; it is guaranteed to be non-NULL.

EXAMPLE:

To register the function thisguy:

```
GLOBAL VOID (*(*nextguy)())(), (*thisguy())();

if (!nextguy)
     nextguy = onexit(&thisguy);
```

SEE ALSO:

exit

BUGS:

The type declarations defy description, and are still wrong.

NAME:

open - Open a File

SYNOPSIS:

```
FILE open(fname, mode)
   TEXT *fname;
   COUNT mode;
```

FUNCTION:

open opens a file fname and assigns a fd to it. If (mode == 0), the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing). Mode may also occur in combined form with IASCII, IBINARY and IRAW. (See Appendix A for details.)

RETURNS:

open returns an fd for the opened file, or a negative number, if unsuccessful.

EXAMPLE:

```
if ((fd = open("xeq", WRITE, 1)) < 0)
    write(STDERR, "can't open xeq\n", 16);
```

SEE ALSO:

close, create

NAME:

pause - Pause Process

SYNOPSIS:

```
pause( )
```

FUNCTION:

pause performs an OS/32 supervisor call 2 (SVC2) pause.

NAME:

read - Read Characters From a File

SYNOPSIS:

```
COUNT read(fd, buf, size)
   FILE fd;
   TEXT *buf;
   BYTES size;
```

FUNCTION:

read reads up to size characters from the file specified by fd into the buffer starting at buf.

RETURNS:

If an error occurs, read returns a negative number; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive, which is the number of characters actually read into buf.

EXAMPLE:

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    write(STDOUT, buf, n);
```

SEE ALSO:

write

NAME:

remove - Remove a File

SYNOPSIS:

```
FILE remove(fname)
   TEXT *fname;
```

FUNCTION:

remove removes the file fname; on most systems, this is an irreversible act.

RETURNS:

remove returns zero, if successful, or a negative number.

EXAMPLE:

```
if (remove(uname()) < 0)
    putstr(STDERR, "can't remove temp file\n", NULL);
```

NAME:

sbreak - Set System Break

SYNOPSIS:

```
TEXT *sbreak(size)
   ARGINT size;
```

FUNCTION:

sbreak moves the system break, at the top of the data area, algebraically up by size bytes, rounded-up as necessary to placate memory management hardware. There is no guarantee that successive calls to sbreak will deliver contiguous areas of memory, nor can all systems safely accept a call with negative size.

RETURNS:

If successful, sbreak returns a pointer to the start of the added data area; otherwise the value returned is NULL.

EXAMPLE:

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
    {
    putstr(STDERR, "not enough room!\n", NULL);
    exit(NO);
    }
```

NAME:

time - Get Date and Time

SYNOPSIS:

```
long time(0)

long time(tloc)
long *tloc;
```

FUNCTION:

time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If tloc is non-null, the return value is also stored in the place to which tloc points.

SEE ALSO:

ctime(4), envir(4)

BUGS:

Time on OS/32 attempts to return GMT. However, because OS/32 time is always local time, time must convert back to GMT using _timezone, _tzmins and _dst. All time routines will work without these being correctly set except gmtime (see ctime(4) and envir(4) for details).

NAME:

uname - Create a Unique Filename

SYNOPSIS:

```
TEXT *uname()
```

FUNCTION:

uname returns a pointer to the start of a NUL-terminated name which is likely not to conflict with normal user filenames. The name may be modified by a letter suffix (but not in place!), so that a family of process-unique files may be dealt with. The name may be used as the first argument to a create, or subsequent open, call, so long as any such files created are removed before program termination. It is considered bad manners to leave scratch files lying about.

RETURNS:

uname returns the same pointer on every call during a given program invocation. The pointer will never be NULL.

EXAMPLE:

```
if ((fd = create(uname(), WRITE, 1)) < 0)
    putstr(STDERR, "can't create sort temp\n", NULL);
```

SEE ALSO:

close, create, open, remove

NAME:

unlink - Remove Directory Entry

SYNOPSIS:

```
unlink(name)
char *name;
```

FUNCTION:

Name points to a null-terminated string. unlink removes the entry for the file pointed to by name from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO:

rm(1), link(5)

DIAGNOSTICS:

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super user).

NAME:

write - Write Characters to a File

SYNOPSIS:

```
COUNT write(fd, buf, size)
   FILE fd;
   TEXT *buf;
   COUNT size;
```

FUNCTION:

write writes size characters starting at buf to the file specified by fd.

RETURNS:

If an error occurs, write either returns a negative number or a number other than size; otherwise size is returned.

EXAMPLE:

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    if (write(STDOUT, buf, n) != n)
        {
        putsts(STDERR, "write error\n", NULL);
        exit(NO);
        }
```

SEE ALSO

read

CHAPTER 5
C MACHINE INTERFACE LIBRARY

TABLE OF CONTENTS

NAME:

Conventions of the C Machine Interface Library

FUNCTION:

The functions and variables documented in this section are usable just like any of those in Chapters 3 and 4, but need not be known to the typical C programmer. Rather, they are called upon by higher level functions to perform machine-dependent operations, to provide machine- dependent information, or merely to provide an important service with efficiency and/or extra precision.

They are isolated in a separate section to avoid cluttering an already extensive collection of useful functions with arcana, and to show prospective implementors what is required in the way of low-level support for a new machine. Note that Chaper 4 serves much the same purpose for implementors of new operating system interfaces.

NAME:

_addexp - Scale Double Exponent

SYNOPSIS:

```
DOUBLE _addexp(d, n, msg)
   DOUBLE d;
   COUNT n;
   TEXT *msg;
```

FUNCTION:

_addexp effectively multiplies the double d by two raised to the power n, although it endeavors to do so by some speedy ruse. If the double result is too large in magnitude to be represented by the machine, _range is called with msg.

RETURNS:

_addexp returns the double result d * (1 << n), or any value returned by _range.

EXAMPLE:

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    COUNT n;

    n = _unpack(&x);
    x = newton(x);
    if (n & 1)
        x =* SQRT2;
    return (_addexp(x, n >> 1, "can't happen"));
    }
```

SEE ALSO:

_frac, _range, _unpack

NAME:

_domain - Report Domain Error

SYNOPSIS:

```
VOID _domain(msg)
   TEXT *msg;
```

FUNCTION:

_domain is called by math functions to report a domain error; i.e., the fact that an input value lies outside the set of values over which the function is defined. It copies msg to _domerr, then calls _raise for the condition _domerr. This exception, if not caught, results in an error exit that prints the NUL-terminated string at msg to STDERR, followed by a newline.

There is no way of inhibiting domain errors, though any code using _when to handle them may choose to ignore their occurrence.

RETURNS:

_domain never returns to its caller. It may return from an instance of _when that is willing to handle a domain error; otherwise the program exits, reporting failure.

EXAMPLE:

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    if (x < 0)
        _domain("negative argument to sqrt");
    ...
```

SEE ALSO:

_domerr, _raise, _range, _when

NAME:

_domerr - Domain Error Condition

SYNOPSIS:

```
TEXT *_domerr
```

FUNCTION:

_domerr is the condition raised when a domain error occurs; i.e., when a math function discovers that an input value lies outside the set of values over which the function is defined.

SEE ALSO:

_domain, _raise, _ranerr

NAME:

_dtens - Powers of Ten

SYNOPSIS:

```
DOUBLE _dtens[];
```

FUNCTION:

_dtens is an array of doubles with values 1, 10, 100, 10**4, 10**8, etc. up to the largest such number the machine can represent. The number of entries in _dtens is recorded in the variable _ntens.

SEE ALSO:

_ntens

NAME:

_dzero - Double Zero

SYNOPSIS:

```
DOUBLE _dzero;
```

FUNCTION:

_dzero is a double zero, provided for convenience more than necessity.

SEE ALSO:

_huge, _tiny

NAME:

_frac - Extract Integer from Fraction Part

SYNOPSIS:

```
COUNT _frac(pd, mul)
   DOUBLE *pd, mul;
```

FUNCTION:

_frac forms the double product of *pd and mul, then partitions it into an integer plus a double fraction in the interval [-1/2, 1/2], delivers the fractional part to *pd and the low bits of the integer part as the value of the function. If the integer part cannot be properly represented as a COUNT, it is truncated on the left without remark.

RETURNS:

_frac returns the low bits of the integer part of the product (*pd * mul) as the value of the function and writes the fractional part of the product at *pd.

EXAMPLE:

```
DOUBLE sind(x)
    DOUBLE x;
    {
    COUNT n;

    n = _frac(&x, 1.0/90.0);
    ...
```

SEE ALSO:

_addexp, _unpack

NAME:

_huge - Largest Double Number

SYNOPSIS:

```
DOUBLE _huge
```

FUNCTION:

_huge is the largest representable double number.

SEE ALSO:

_dzero, _tiny

NAME:

_norm - Convert Double to Normalized Text String

SYNOPSIS:

```
COUNT _norm(s, d, prec)
   TEXT *s;
   DOUBLE d;
   BYTES prec;
```

FUNCTION:

_norm factors the double d into a double in the interval [0.1, 1) or zero, and an integral power of ten. The first prec digits of the fraction are written as text characters in the buffer starting at s. If the number is negative on entry, it is forced positive.

RETURNS:

The value of the function on return is the power of ten to which the fraction string in s must be raised to give the value of d. If d is zero, all characters in s are '0's and the value returned is zero.

SEE ALSO:

_round

NAME:

_ntens - Number of Powers of Ten

SYNOPSIS:

```
COUNT _ntens;
```

FUNCTION:

_ntens is the number of elements in the array _dtens, which holds various powers of ten as double numbers.

SEE ALSO:

_dtens

NAME:

_poly - Compute Polynomial

SYNOPSIS:

```
DOUBLE _poly(d, tab, n)
   DOUBLE d, *tab;
   COUNT n;
```

FUNCTION:

_poly computes the polynomial of order n in the independent variable d, using the coefficients in the table pointed to by tab. Horner's method is used, taking tab[0] as the coefficient of the highest power of d, so the value computed is:

```
tab[n] + d * (tab[n-1] + d * (...  + d * tab[0]))
```

No precautions are taken against overflow or underflow.

RETURNS:

_poly returns the double value of the polynomial of order n in d.

EXAMPLE:

```
return (x * _poly(x * x, coeffs, 6));
```

NAME:

_raise - Raise an Exception

SYNOPSIS:

```
VOID _raise(ptr, cx)
   TEXT **ptr, **cx;
```

FUNCTION:

_raise signals the presence of a condition that must be handled by an earlier call to _when. The _when/_raise mechanism is used to perform a broad spectrum of stack manipulations normally beyond the scope of the C language, including: ADA exception handling, Pascal nonlocal goto's, IDRIS process switching, editor interrupt fielding and math error reporting.

The handler to be first considered is specified by ptr. If ptr is -1 or NULL, the latest _when call is used as the start of a search for a willing handler; otherwise ptr must have been set by an earlier _when call to specify that call as the starting point of the search.

If cx is NULL or -1, then the first handler encountered returns to its caller with the value zero; otherwise cx must match a condition argument of one of the registered handlers to be considered, or at some level it must be handled by a NULL terminating a list of condition arguments.

The return from _when caused by a _raise call cleans up the stack if either ptr or cx is NULL. Otherwise, the handler for that _when call remains on the stack and is made the latest of the chain of handlers.

RETURNS:

_raise never returns to its caller. It returns from the latest willing _when call with registers, stack, and handler chain restored to that level; the value returned by _when is nonnegative. The handler chain is initialized to a single catchall handler which calls error to print an error message, and takes an error exit. If the condition can be interpreted as the address of a pointer to a NUL-terminated string, then that string, followed by a newline, is used as the error message; otherwise the message is "unchecked condition".

EXAMPLE:

To exit on end of file:

```
TEXT *endfile {"unchecked end of file"};

VOID readrec(buf)
    TEXT *buf;
    {
    if (fread(STDIN, buf, 80) != 80)
        _raise(NULL, &endfile);
    }
 ...
    switch(_when(NULL, &endfile, NULL))
        {
    case 1:
        oneof();
        }
```

SEE ALSO:

_when, error(3), enter(3), leave(3)

BUGS:

You are not expected to understand this.

NAME:

_ranerr - Range Error Condition

SYNOPSIS:

```
TEXT *_ranerr
```

FUNCTION:

_ranerr is the condition raised when a range error occurs; i.e., when a math routine discovers that a return value is too large to represent. Unlike most conditions, the range condition may be inhibited from time to time by writing a nonzero value in _ranerr.

SEE ALSO:

_domerr, _range

NAME:

_range - Report Range Error

SYNOPSIS:

```
DOUBLE _range(msg)
   TEXT *msg;
```

FUNCTION:

_range is called by math functions to report a range error; i.e., the production of an output value that cannot be represented properly by the machine. If _ranerr is NULL, _range copies msg to _ranerr, then calls _raise for the condition _ranerr. This exception, if not caught, results in an error exit that prints the NUL-terminated string at msg to STDERR, followed by a newline.

If _ranerr is not NULL, the condition is not raised, and _range returns to its caller.

RETURNS:

If _range returns to its caller, the value returned is the largest double that can be represented by the machine; otherwise the _ranerr condition is raised and _range does not return to its caller. It may return from an instance of _when that is willing to handle a range error; otherwise the program exits, reporting failure.

EXAMPLE:

```
if (_lnhuge < x)
    _range("exp overflow");
```

SEE ALSO:

_domain, _ranerr, _raise, _when

NAME:

_round - Round Off a Fraction String

SYNOPSIS:

```
COUNT _round(s, n, prec)
   TEXT *s;
   BYTES n, prec;
```

FUNCTION:

_round rewrites the n character buffer starting at s as a properly rounded string of prec digits. If prec is outside the buffer, or if (s[prec] < '5'), no action is taken. Otherwise, the next character to the left is incremented and carries are propagated. All '9's is rewritten as '1000...' to prec digits.

RETURNS:

_round returns 1 if all '9's rounded up, otherwise zero.

SEE ALSO:

_norm

BUGS:

No check is made for nondigits in the buffer.

NAME:

_tiny - Smallest Double Number

SYNOPSIS:

```
DOUBLE _tiny
```

FUNCTION:

_tiny is the smallest positive representable double number larger than zero.

SEE ALSO:

_dzero, _huge

NAME:

_unpack - Extract Fraction from Exponent Part

SYNOPSIS:

```
COUNT _unpack(pd)
   DOUBLE *pd;
```

FUNCTION:

_unpack partitions the double at *pd, which should be nonzero, into a fraction in the interval [1/2, 1) times two raised to an integer power, delivers the fraction to *pd and returns the integer power as the value of the function.

RETURNS:

_unpack returns the power of two exponent of the double at pd as the value of the function and writes the fraction at *pd. The exponent is generally meaningless if d is zero.

EXAMPLE:

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    COUNT n;

    n = _unpack(&x);
    x = newton(x);
    if (n & 1)
        x =* SQRT2;
    return (_addexp(x, n >> 1));
    }
```

SEE ALSO:

_addexp, _frac

NAME:

_when - Handle Exceptions

SYNOPSIS:

```
COUNT _when(ptr, c1, c2, ..., cend)
   TEXT **ptr, **c1, **c2, ..., **cend;
```

FUNCTION:

_when registers a willingness to handle certain exceptions that may be raised by calls to _raise. The _when/_raise mechanism is used to perform a broad spectrum of stack manipulations normally beyond the scope of the C language, including: ADA exception handling, Pascal nonlocal goto's, IDRIS process switching, editor interrupt fielding, and math error reporting.

The call to _when causes its argument list and certain nonvolatile registers to be left on the stack, where they are made the latest part of a chain of condition handlers. Should a subsequent call to _raise report a condition that is to be handled by this part of the chain, control flow resumes with a return from _when, indicating which condition has been raised. Upon every return, all register variables are restored to their values at the time of the initial call to _when. The _raise call may cause the stack to be cleaned up as part of the return from _when; this is a mandatory prelude to returning from any function that calls _when.

If ptr is not NULL, it is used as the address of a pointer that should be set to point at the latest part of the handler chain; this value may be used by subsequent _raise calls to specify this particular call to _when instead of the normal top of the handler chain. ptr is also used when the stack is cleaned up on return, as the address at which to write the condition being handled.

The conditions c1, c2, etc., each may assume any value except NULL or -1, although there is a strong presumption that the value is a valid data space address of a pointer to a NUL-terminated string of characters. A -1 is taken as a cend that indicates no further conditions, while a NULL is taken as a cend that will handle any condition. The left-most condition argument that will handle a given condition, in the latest part of the handler chain, is chosen to handle the condition.

_when should never be used except as the lone operand in a switch statement, and all _when calls must be carefully coordinated with appropriate _raise calls.

RETURNS:

_when returns -1 upon return from its initial setup. It returns zero on a cleanup return that reports no condition. Otherwise it returns the ordinal position, within the argument list, of the condition it is handling; a one indicates c1, two means c2, etc. If cend is NULL, its ordinal position will be returned for any condition not otherwise handled.

The stack is cleared, and a non-NULL ptr is used to return the second argument to _raise, if either argument to _raise was NULL or if a NULL cend is handling the condition.

EXAMPLE:

To field interrupts interactively:

```
VOID endup()
    {
    putstr(STDOUT, "?\n", NULL);
    _raise(NULL, NULL);
    }

 ...

FOREVER
    {
    onintr(&endup);
    _when(NULL, NULL);
    if (edit() == EOF)
        exit(YES);
    }
```

SEE ALSO:

_raise, enter(3), leave(3)

BUGS:

You are not expected to understand this.

APPENDIX A
OS/32 FILE SYSTEM INPUT/OUTPUT (I/O) INTERFACE AND RUN-TIME ENVIRONMENT

INTRODUCTION:

The C language was the original language of UNIX operating systems. For this reason, the run-time environment of C has a strong relationship to UNIX. This has created the need for some special compromises in providing an equivalent run-time system for OS/32.

Two different run-time libraries (RTLs) are supplied with the OS/32 C compiler, an EDITION VII compatible library and an IDRIS compatible library. However, they are not usually compatible with one another, as they contain duplicate routines which differ in function. The user should be careful to use only one set of routines in any single program.

The OS/32 C language has also been enhanced to permit linkage to FORTRAN VII subroutines and functions, and to allow reference to FORTRAN COMMONS.

EXAMPLES:

```
main ()
{
        fortran int FUNC ();
        common   {
                    int a;
                    int b;
                 } ABC;
        ABC.a = FUNC(ABC.a, ABC.b);
}
```

Note that FORTRAN function names and COMMON name must be given in upper-case and are limited to seven characters of significance.

When Linking to FORTRAN VII functions, the user must take care to Link in the C FORTRAN initializer (CFINIT.OBJ) instead of the standard C initializer (CINIT.OBJ) and to compile with the -Z option specified to the C preprocessor.

The UNIX file system, from which C's run-time system evolved, has a number of characteristics, among them are:

1. Only one file type.
2. Data may be accessed anywhere in a file, in any length without any record length considerations.
3. A file may grow to any size, at any time.
4. File sharing is controlled by file protection bits.
5. ASCII and binary files are treated identically (packed).

The OS/32 file system has an even larger number of characteristics, among them are:

1. Four types of files: indexed, contiguous, unbuffered indexed and extendable contiguous.
2. Data may be accessed only at logical record boundaries. The logical record length is fixed at file creation time.
3. Contiguous files may not grow in size.
4. File sharing is controlled both via protection codes and at open time.
5. ASCII and binary files are treated differently.
6. Files may be preassigned.

In order to deal with these wide differences and still provide a UNIX-like interface, a number of assumptions have been made concerning the OS/32 file type, record length, etc. Several of these assumptions may be overridden by the use of the mode arguments on open, creat and fopen calls or by use of the global variables described below.

The user can make use of the mode argument to force image (raw) mode I/O on a particular file descriptor (fd) and to set the ASCII or binary attribute of a file.

MODE SETTING	EFFECT
READ	open file read-only
WRITE	open file write-only
UPDATE	open file for read and write
IASCII	open file as ASCII file
IBINARY	open file as binary file
IRAW	open file in image mode

EXAMPLE:

```
fd = open ("con:", UPDATE|IASCII|IRAW);
```

VARIABLE NAME	MEANING	DEFAULT
int fstype;	file type 3 = unbuffered indexed 2 = indexed 1 = extendable contiguous 0 = contiguous	indexed
int fsrecl;	file record length	256
int fsrkey;	file read key	0
int fswkey;	file write key	0
int fsblksz;	file size for contiguous files	256
int fsacp;	file account priveledges (see below)	NO
int fsshare;	shared/exclusive access 0 = shared access 1 = exclusive access	shared
int ftype;	ASCII/binary mode 0 = determined by record length 1 = ASCII 2 = Binary	determined by record length

FILE CREATION:

The default file created by a call to creat is indexed, 256-byte logical record length and blocking factor of 1. If the mode argument to the creat call is 0, then the variables fstype, fsrecl, fsrkey, fswkey, fsblksz and fsacp listed above are used. The defaults may be overriden by setting these variables before a call to creat. These values are not automatically reset after the call - they maintain their values until changed again by the user.

When a program is running, automatic adjustment is made as to whether file account numbers rather than only file classes are supported. If the program was linked with account privileges, account numbers will be accepted anywhere a file class was permitted.

FILE ACCESS:

When a file is opened via a call to open or creat, then it will be opened for access in the following ways, depending on the value of fsshare above. The following list indicates the access modes used:

OPEN MODE	FSSHARE = 1	FSSHARE = 0
0 (read)	ERO	SRO
1 (write)	ERW	SRW
2 (update)	ERW	SRW
creat call	ERW	SRW

ASCII/BINARY MODE

When a file is opened by open, or creat then ASCII or binary mode is determined either by the mode of the open or creat or by a combination of the setting of the variable ftype and the logical length of the file:

FTYPE VALUE	RECORD LENGTH	ACCESS MODE
0	not 256	ASCII
	256	Binary
1	N/A	ASCII
2	N/A	Binary

Reading from ASCII files (via a call to read) is treated in the following manner:

1. All CRs are changed to new-lines.
2. All characters within a record following a CR are ignored.
3. A full record (one whose record length if filled with characters none of which is a CR) will have an implied carriage return added to it.
4. Any number of characters may be requested in a read. The software will hide the existence of records and record length from the user.
5. The user may lseek to a position within an ASCII file. That position is calculated by counting the number of real characters in the records. Note that this is rather slow.

Writing to ASCII files (via a call to write) is treated in the following manner:

1. All new-lines are changed to CRs and cause the end of the current logical record.

2. Characters following the CR within a written record are set to spaces.

3. Any number of characters may be written, the output is buffered until a record is full or a newline is written.

As in the EDITION VII environment, fd numbers 0, 1 and 2 are set to standard input, standard output and error output. These fds are attached to logical units 0, 1 and 2, respectively. These may be redirected to/from files on the command line (see section below). They are opened in ASCII mode or preassigned. One additional call applies to the case of these fds (and any other fd desired): fsraw. When this routine is called with appropriate arguments, a read request to the fd will be performed in image mode and will be satisfied as soon as the requested number of characters is read or a CR is encountered. Further, the CR is not mapped to a line feed. Output to fds set into raw mode will cause any writes performed to be effected immediately in image mode. When an fd is not in raw mode, input and output requests are buffered until CR or newline or until the buffer is full.

Reading from binary files is treated in the following manner:

1. No character mapping occurs.

2. Any number of characters may be read; input is buffered within the software.

3. The user may lseek to any position within the file by reference to the byte number.

Writing to binary files is treated in the following manner:

1. No character mapping occurs.

2. Any number of characters may be written; output is buffered within the software.

Only wait input/output (I/O) is utilized.

For stack allocation, the C initializers (CINIT.OBJ and CFINIT.OBJ) contain a constant _stksz which is used to establish the maximum size of the stack at startup. This value is, by default, set to 8 kb. The only way to adjust the maximum stack size is to modify _stksz to a different value using the OS/32 PATCH utility.

EXAMPLE:

```
- OS/32 PATCH
> OBJECT CINIT.OBJ
> GET *OBJ
> EXA 0,2
- 0000 YYYY
> EXA YYYY,2
- 0000 2000
> MOD YYYY,0,4000
> SAVE
> END
```

Under EDITION VII, system time is kept in seconds since midnight January 1, 1970, while system time on OS/32 is kept as the local date and time. In order to make the gmtime routine work properly, the user must set variables in the C initializer routines which give the local time zone and a flag indicating whether daylight savings time transformations apply in the local time zone.

The variables can be accessed either with the OS/32 PATCH Utility or by use of the pointer _envir and the envir.h include file.

VARIABLE		FUNCTION	DEFAULT
_timezon	(TIMEZONE for PATCH)	hours west of Greenwich	5(EST)
_tzmins	(TZMINS for PATCH)	minutes west of Greenwich	0
_dst	(DST for PATCH)	does daylight savings apply	1 (YES)

Only one of _timezon or _tzmins should be set.

EXAMPLE:

```
#include <envir.h>
main ()
{
 extern struct _ENVIR *_envir;
   _envir->_timzon=6;  /* CST */
   _envir->_dst   =0;  /* no daylight savings */
        .
        .
        .
}
```

Finally, to signal End Of File (EOF) on a terminal, use Control D. If a BIOC terminal is used, Control D is interpreted as Echo-only. In this instance, the user must use Control T Control D to signal EOF on a terminal.

APPENDIX B
OPERATING PROCEDURES

TABLE OF CONTENTS

INTRODUCTION:

The following sections describe commands which are supplied with the OS/32 C Compiler. They enable the multi-terminal monitor (MTM) user to compile, assemble, link and list C programs. In addition to these commands, the C Compiler package also includes an enhancement to the OS/32 program development system (EOU.CSS) which permits the use of C with that system. For the purposes of the program development system, the C language environment is C and the C extension is .c. All features of the program development system are described in the OS/32 Multi-Terminal Monitor (MTM) Reference Manual.

The most important of the commands is CC, which involves all of the others except LISTER. Use of this command permits the user to compile, assemble and link any C source program which consists of a single file. If CC has been used to create the executable version of a C program, it may have been run by entering its name followed by any arguments separated by blanks. UNIX-style redirection of STDIN and STDOUT is also permitted.

EXAMPLE:

```
CC  cprogram
cprogram -abc <file.in >file.out
```

NAME:

CC - Compile, Assemble and Link a C Program

SYNOPSIS:

CC name[,option]

FUNCTION:

CC takes a C source language file as input and compiles, assembles, and links the input C program to produce an executable file. In addition, it creates a command substitution system (CSS) file which permits the resulting program to be run by simply invoking its file name. (See introduction).

The arguments are:

name — is the name of a C source language file without an extension. The extension is assumed to be .c.

option — is one of the following:

1 - Stop compilation after running the C preprocessor. Note that the resulting file is not ASCII text. To produce ASCII output from the preprocessor, see pp in this appendix. Output is left on name.1.

2 - stop compilation after running the C compiler first pass. Output is left on name.2.

cal - stop compilation after running the C compiler second pass. The output is in CAL/32 assembly language on name.cal.

obj - stop compilation after assembly. Output is left on name.obj. Note that this is the option that must be used if the C source program consists of several files. Linking must then be performed manually. See LN in this appendix.

u - Compile, assemble and link using the EDITION VII compatible run-time library (RTL). This is the default value of option. Output is left on name.tsk.

w - Compile, assemble and link using the IDRIS compatible RTL. As above, output is left on name.tsk.

NAME

CU - Assemble a C Program

SYNOPSIS:

```
CU    name
```

FUNCTION:

CU takes the assembly language file which is the output of ptwo and assembles it using CAL/32 R01-01 which distinguishes between upper- and-lower case. Note that C programs which are not assembled with this version of CAL/32 will not function properly.

The argument to CU is name with no extension. The extension is presumed to be .CAL.

CU invokes the assembler with the following options - lcase ur erl del nlist squez=3 ncros

EXAMPLE:

```
CU grep
```

NAME:

LISTER - List Source and Errors

SYNOPSIS:

LISTER -[e 0 1 i* l# w#] <file> [<errors>]

FUNCTION:

lister produces a line numbered listing of the <file> on stdout. The following options may be used:

-e	expands include files found in the source
-0	intersperses errors from pass 0 of the C compiler with the lines of the source at the appropriate point. The errors are assumed to be found in the <errors> file which must be included with this option.
-1	intersperses errors from pass 1 of the C compiler with the lines of the source at the appropriate point. The errors are assumed to be found in the <errors> file which must be included with this option.
-i*	the search path for the #include "..." files. See chdir (4) for the form of the prefix on OS/32.
-l#	number of lines per page. (Default is 66)
-w#	number of characters per line. (Default is 80)

EXAMPLE:

LISTER -e -1 -i/usr/include plfunc.c plfunc.erl

SEE ALSO:

PP, PONE

NAME:

LN - Link a C Program to Run on OS/32

SYNOPSIS:

```
LN    name, option
```

FUNCTION:

LN builds a link command file, then loads and runs LINK/32 using that command file to build a runnable C program. By default, the resulting program has float and double float support and a default workspace of 8kb for stack and heap growth. The command file INCLUDE's CINIT.OBJ, the C run-time initilization routine, then INCLUDE's a single user C object file name.obj, LIB's against LIBU.OBJ, the Edition VII compatible library, or LIBW.OBJ, the IDRIS compatible library, and then LIB's against LIBE.OBJ, the OS/32 interface library and finally against the system math library. A short link map is built on file name.MAP and the task is built on name.TSK.

The arguments to LN are:

name	is the name of C object file without extension. The extension is assumed to be .OBJ.
option	is either "w" or "u". If "w" is specified the IDRIS compatible C RTL is resolved against. If "u" is specified the Edition VII compatible library is resolved against. The default is "u".

If LN is not used to build the C task, then the following conventions must be observed:

1. OP FL,DFL must be specified.
2. CINIT.OBJ or CFINIT.OBJ must be included
3. When editing against run-time libraries, the following order must be observed:

```
LIB    LIBU.OBJ/S
LIB    LIBW.OBJ/S
LIB    LIBE.OBJ/S
LIB    PEMATH.OBJ/S
```

NAME:

PP - Preprocess Defines and Includes

SYNOPSIS:

```
PP  -[c d* i* o* p? s? x 6] <files>
```

FUNCTION:

PP is the preprocessor used by the C compiler to perform #define, #include and other functions signalled by a #, before actual compilation begins. It can be used to advantage, however, with most language processors. The flag options are:

-c	don't strip out /* comments */ nor continue lines that end with \.
-d*	where * has the form name=def, define name with the definition string def before reading the input; if#=def is omitted, the definition is taken as "1". The name and def must be in the argument, i.e.; no blanks are permitted unless the argument is quoted. Up to ten definitions may be entered in this fashion.
-i*	change the prefix used the #include "filename" from the default "" to the string *. Multiple prefixes to be tried in order may be specified, separated by the character "¦". See chdir(4) for the form of the prefix on OS/32.
-o*	write the output to the file * and write error messages to STDOUT. Default is STDOUT for output and STDERR for error messages. On many systems (other than Idris), the -o option is mandatory with -x because STDOUT is interpreted as a text file and, hence, becomes corrupted.
-p?	change the preprocessor control character for "#" to the character ?.
-s?	change the secondary preprocessor control character fro. "@" to the character ?.
-x	put out lexemes for input to the C compiler pone, not lines of text.
-v	report source file and line number during processing.

-6 put out extra newlines and/or SOH ('\1') codes to keep source line numbers correct for UNIX/V6 compiler or ptc.

-z permit FORTRAN function declarations and common declarations.

PP processes the named files, or STDIN if none are given, in the order specified, writing the resultant text to STDOUT.

Preprocessor actions are described in detail in Chapter 1 of this manual.

The presence of a secondary preprocessor control character permits two levels of parameters. For instance, the invocation

```
PP -c-p@
```

will expand define and ifdef conditionals, leaving all the commands and comments intact; invoking PP with no arguments would expand both @ and # commands. The flag -s# would effectively disable the secondary control character.

EXAMPLE:

The standard style for writing C programs is:

```
/* name of program
 */
#include <std.h>
#define MAXN        100
COUNT things [MAXN];
etc.
```

The use of uppercase identifiers is not required by PP, but is strongly recommended to distinguish parameters from normal program identifiers and keywords.

SEE ALSO:

PONE

BUGS:

Unbalanced quotes ' or " may not occur in a line, even in the absence of the -x flag. Floating constants longer than 38 digits may compile incorrectly on some host machines.

NAME:

PONE - Parse C Programs

SYNOPSIS:

PONE - [a b# c = l m n# o# r# u] <file>

FUNCTION:

PONE is the parsing pass of the C compiler. It accepts a sequential file of lexemes from the preprocessor PP and writes a sequential file of flow graphs and parse trees, suitable for input to a machine-dependent code generator PTWO. The operation of PONE is largely independent of any target machine. The flag options are:

-a	compile code for machines with separate address and data registers.
-b#	enforce storage boundaries according to #, which is reduced modulo 4. A bound of 0 leaves no hols in structures or auto allocations; a bound of 1 (default) requires short, int and longer data to begin on an even bound; a bound of 2 is the same as 1, except that 4- to 8- byte data are forced to a multiple of 4 byte boundary; a bound of 3 is the same as 2, except that 8 byte data (doubles) are forced to a multiple of 8- byte boundary.
-c	ignore case distinctions in testing external identifiers for equality, and map all names to lowercase on output. By default, case distinctions-matter.
-d	use standard C floats and doubles. By default float and double are distinct.
-e	do not force loading of extern references that are declared but never defined or used in an expression. Default is to load all externs declared.
-l	take integers and pointers to be 4 bytes long. Default is 2 bytes.
-m	treat each struct/union as a separate name space, and require x.m to have x a structure with m one of its members.

-n#	ignore characters after the first # in testing external identifiers for equality. Default is 7; maximum is 8.
o#	write the output to the file # and write error messages to STDOUT. Defalt is STOUT for output and STDERR for error messages.
-r#	assign no more than the # variables to registers at any one time, where # is reduced modulo 4. Default is 3 register variable.
-u	take "string" as array of unsigned char, not array or char.
-v	report file being processed and line in increments of 10 characters.

If <file> is present, it is used as the input file instead of the default STDIN. On many systems (other than IDRIS/UNIX), the o option and <file> are mandatory because STDIN and STDOUT are interpreted as text files and, hence, become corrupted.

EXAMPLE:

PONE is usually between PP and some version of PTWO, as in the following

```
PP -xv tenol file.o
PONE -ulv -b2 -n8 -otemp2 temp1
PTWO -mv -ofile.s temp2
```

SEE ALSO:

PP

NAME:

PTWO - Generate Code for OS/32 C Programs

SYNOPSIS:

```
PTWO -[o# p s x#]  <file>
```

FUNCTION:

PTWO is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from PONE and writes a sequential file of assembly language statements suitable for input to the CAL assembler.

The flags are:

-d	use standard C conventions on floating point (i.e.; all arithmetic is double and arguments are double.
-o#	write the output to the file and write error messages to STDOUT. Default is STDOUT for output and STDERP for error messages.
p	emit debugging output
-s	put out source lines preceeding appropriate code in assembly file.
-x#	mark the three virtual sections for Function (04), Literals (02) and Variables (01), to the two physical sections Code (bit is a one) and Data (bit is a zero). Thus, "-x4" is for separate I/D space, "-x6" is for read-only memory/random access memory (ROM/RAM) code, and " x7" is for compiling tables into ROM. Default is 4.
-n	generate code for OS/32
-v	report source file and relevant line number during processing.
-8	generate compatible code for 8/32, 7/32.

If <file> is present, it is used as the input file instead of the default STDIN. On many systems (other than IDRIS), <file> is mandatory, because STDIN in interpreted as a text file and, hence, becomes corrupted.

Files output from pone for use with the code generator should be generated with: -m signifying OS/32 as a target; -l since pointers are long; and -b2 flag, since only the value of "-b2" is acceptable.

Whenever possible, labels in the emitted code are followed by a comment which gives the source line from which the code immediately following obtains, along with a running count of the number of words of code produced for a given function body. +s will cause the actual # source text to be output.

EXAMPLE:

PTWO usually follows up and PONE, as follows:

```
PP -xv  -otempl file.c
PONE -ulv -b2 -n8 -otemp2 templ
PTWO -o file.s temp2
```

SEE ALSO:

PONE

BIBLIOGRAPHY

Kernighan, Brian W., Ritchie, Dennis M. The C Programming Language. New Jersey: Prentice-Hall, Inc., 1978.

INDEX

U,V

W,X,Y,Z

PERKIN-ELMER

PUBLICATION COMMENT FORM

We try to make our publications easy to understand and free of errors. Our users are an integral source of information for improving future revisions. Please use this postage paid form to send us comments, corrections, suggestions, *etc.*

1. Publication number _____

2. Title of publication _____

3. Describe, providing page numbers, any technical errors you found. Attach additional sheet if neccessary.

   _____

   _____

   _____

4. Was the publication easy to understand? If no, why not?

   _____

5. Were illustrations adequate? _____

   _____

   _____

6. What additions or deletions would you suggest? _____

   _____

   _____

7. Other comments: _____

   _____

   _____

From _____ Date _____

Position/Title _____

Company _____

Address _____

_____

_____

STAPLE

STAPLE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 22 OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

PERKIN-ELMER

Data Systems Group
106 Apple Street
Tinton Falls, NJ 07724

ATTN:
TECHNICAL SYSTEMS PUBLICATIONS DEPT.

FOLD

FOLD

STAPLE

STAPLE

6433-1