C PROGRAMMER'S MANUAL

CGC 7900 SERIES
COLOR GRAPHICS COMPUTERS

Whitesmiths, Ltd.

C PROGRAMMERS' MANUAL

Release: 2.1

Date: March 1982

The C language was developed at Bell Laboratories by Dennis Ritchie; Whitesmiths, Ltd. has endeavored to remain as faithful as possible to his language specification. The external specifications of the Idris operating system, and of most of its utilities, are based heavily on those of UNIX, which was also developed at Bell Laboratories by Dennis Ritchie and Ken Thompson. Whitesmiths, Ltd. gratefully acknowledges the parentage of many of the concepts we have commercialized, and we thank Western Electric Co. for waiving patent licensing fees for use of the UNIX protection mechanism.

The successful implementation of Whitesmiths' compilers, operating systems, and utilities, however, is entirely the work of our programming staff and allied consultants.

For the record, UNIX is a trademark of Bell Laboratories; IAS, PDP-11, RSTS/E, RSX-11M, RT-11, VAX, VMS, and nearly every other term with an 11 in it all are trademarks of Digital Equipment Corporation; CP/M is a trademark of Digital Research Co.; MC68000 and VERSAdos are trademarks of Motorola Inc.; ISIS is a trademark of Intel Corporation; A-Natural and Idris are trademarks of Whitesmiths, Ltd. C is not.

C PROGRAMMERS' MANUAL

SECTIONS

SCOPE

This manual describes the C programming language, as implemented by Whitesmiths, Ltd., and the various library routines that make up the machine independent C environment. Section I introduces the C language, and Section II details the numerous functions callable from C to extend the power of the language. Section III lists the functions that interface to a given operating system, while Section IV describes the functions that interface to a given machine architecture. The distinction between Sections II, III, and IV, while of considerable importance to implementors, is probably academic to most programmers -- all functions in all of these sections are present as described on all systems supported by Whitesmiths, Ltd.

For documentation of the programming utilities, or information on each implementation of system or machine dependent features of the C support software, see the C Interface Manual for the appropriate machine.

I. 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 will perform 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 1978.

This section describes the current implementation, as succinctly, and, it is to be hoped, 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 compiler operation proper is found in the various C Interface manuals, which also contain the descriptions of subroutines used to communicate with various operating systems. The standard library of C callable functions is documented in the remaining sections of this manual.

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 deal with 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 whitespace. Whitespace consists of one or more non-printing characters, such as space, tab, or newline; its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms:

An identifier - consists of a letter or underscore, followed by zero or more letters, underscores, or digits. Uppercase letters are distinct from lowercase 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

A numeric constant - consists 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 nevertheless may contain the digits '8' and '9'. A non-zero 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.

A floating literal - consists 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.

A character constant - consists of a single quote, followed by zero or more character literals, followed by a second single quote. A character literal consists of a) any character except '\', newline, or single quote, the value being the ASCII representation of that character; b) 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 c) 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.

A string constant - is just like a character constant, 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 long-winded 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

```
<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>, and if it does 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 instance:

```
{ <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:

```
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 non-digit. 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 will be considered separately later on. The bulk of this discussion concerns the name space inhabited 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 it. 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 a) explicitly declared to be extern or static, or b) 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

Storage classes come in a variety of flavors, some with different seasoning depending on context.

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 few 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. If none, 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 a) efficient storage, such as the machine's fast registers, should be favored to hold the object, and b) 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 hence 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 are derived the composite forms struct, union, bitfield, pointer to, array of, and function returning. Recursive application of the rules for deriving composite types leads to a large, if not truly infinite, assortment of types.

[unsigned] char - is a byte integer, something just big enough to hold all of the characters in the machine's character set. It is promised that printable characters and common whitespace codes are small positive integers or zero.

[unsigned] short - is typically a two-byte integer, something just big enough to hold reasonable counts.

[unsigned] int - is either a short or a long, depending on the machine. It is promised to be big enough to count all the bytes in the machine's address space.

[unsigned] long - is typically a four-byte integer, something comfortably large.

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 types but function returning, array of unknown size, and struct of unknown content.

union - is an alternation of one or more members, the union being big enough and aligned well enough to accept any of its member types. Members may be any types 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. It is promised that 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. [N.B. 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

each having one of the forms:

```
[ <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; default is extern. The type <ty> may be a) a basic type, b) a struct or union declaration, described below, or c) an identifier earlier declared to be a typedef; 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 thus 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 */
```

Accepting these Truths is the first step on the path to Enlightenment.

Initializers come in two basic flavors, 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 it will merely be observed 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 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 a) for the first field specifier in a declaration, b) for the first bitfield specifier following a non-bitfield specifier, c) for a bitfield specifier that will not fit in the remaining space in the current unsigned int, or d) 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 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; 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 quietly 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. Thus

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

To eliminate a lurking ambiguity before it bites, "()" is always taken as function returning, and never as (unnecessary) parentheses around the omitted identifier.

EXAMPLE

Some simple declarations:

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

More elaborate:

```
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 an auto or register contains garbage.

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 will 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. Thus

```
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 creature being initialized. It is often permissible, however, to write an initializer by eliding 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 hard enough 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 the <expression> produces no useful side effect, i.e., "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 non-zero 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 innermost "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 doesn't 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> - So long as <expression> evaluates to a non-zero 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> may thus be executed zero or more times.

do <statement> while (<expression>); - The <statement> is executed and, so long as <expression> evaluates to non-zero 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, then so long as <ex2> evaluates to non-zero 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 innermost 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 innermost 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 outermost 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 multi-level 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 thus 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 scalar. 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, to wit: 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, the 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

the pointer to produce a result of type pointer. Otherwise 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 ... 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 before 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 non-zero, 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 evaluatable 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 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 on the manual page for the pp command, described elsewhere in 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 non-whitespace 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 non-digit; 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 non-whitespace character following the identifier or its parameter list, and ends with the last non-whitespace 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 just "" 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, else 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 constraint 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 recognizably a command causes a diagnostic.

Expansion of non-command 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 blow up.

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 about five hundred lines of code, it should be split into files each no bigger than that. 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:

```
/* 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 Section II of this manual.

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 all caps 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 clumped 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 half dozen or so lines of pseudocode that best summarize the algorithm. There is seldom a 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 multi-level break, which is not provided in C; but this seldom proves to be a prudent thing to do 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 shoddy 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 half a dozen local variables. Naturally there are exceptions to all these guidelines, but they are just that — exceptions.

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

```
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 in

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

which is demonstrably true when c is a digit.

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 high-handed dicta have proved their worth many times over:

Each external declaration should begin (with optional storage class and mandatory type) at the beginning of a line, immediately following its ex-

planatory 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 first extern (IMPORT), then register (FAST), then auto (no storage class specifier), then static (INTERN) declarations; these are sorted alphabetically by type within storage class and alphabetically by name within type. Comma separated lists may be used, so 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 explicit exception of the else-if chain, each subordinate <statement> is indented one tab stop further to the right than its controlling statement. Without this rule, 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 offender is

```
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 (of 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 remarkably easy, most of the time. When machine dependencies creep in, however, they can be extremely difficult to dig out; and trying to keep them out of new code can often lead the programmer to paranoid extremes. Herewith a set of rules to follow that eliminate nearly all the ghastlies before they bite.

First and foremost, use the portable library. Pretend the C Interface Manuals don't exist, and fall out of love with the endearing peculiarities of your current host system. It will change.

If your program processes text files, assume that carriage returns, NULs, and other funny characters that don't 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 non-zero. A third argument of 1 is always acceptable, and will not lead to storage inefficiencies in the target file. Sadly, 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 all over the map. A long integer, '3210' for instance, can read out as '3210', '0123', or '2301' on three popular computers, where '0' is the least significant byte. The best rule is never to write a multi-byte 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 says filenames should be no more elaborate than "xxxxxx.yy". Believe it. Better yet, use uname to build temp file names and avoid wiring any other file names into code. That's what the command line is for, and what getfiles is designed to help with. 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 of silly 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 lvalueness creeps in, as when assigning to a register or argument, or taking the address of an argument.

Never assume that assigning to one of these creatures, or applying a cast such as (char), will perform any sort of truncation; it won't, at least not below integer. And while it is nice to be able to declare an argument to be char for the sake of documenting its likely range, it is easy to forget when taking its address that what you want is a pointer to int, not a pointer to char. Storing a char in part of an int may work a little bit right on some computers, but it will surely cause trouble sooner or later. 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. If you know how big you want a datum, regardless of what machine the program is run on, use the aliases for {char, short, long}: i.e., {TINY, COUNT, LONG} or the unsigned versions {UTINY, UCOUNT, ULONG}. If something must hold a pointer, declare it as such by all means; 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. Remember that case switch values are ints; hence, only short values are portable for case labels.

There are two other important malleable types besides BYTES. 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 you know a constant must

be long on any machine, be sure to force it long; ~0 can be different from ~0L.

To end on a positive note, there are some things you can depend on across implementations. There will always be at least three registers available, for instance. 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. And 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. And it is possible to parameterize code efficiently by writing expressions like:

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

Only <short test> or <long test> will actually be generated as runtime code; the rest 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 hews closely to that standard, save for minor changes in emphasis. There are also several available implementations of C that differ in more important respects. Herewith a summary of the things to look out for.

THE STANDARD

- The major deviation is that this compiler requires each external declaration to be explicitly initialized exactly once among all the files that comprise a C program; the standard permits external declarations to remain uninitialized.
- 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.

SYSTEM DEPENDENCIES

Since this implementation produces assembler code for the target system, there is some variation in naming caused by assembler limitations. There may be as few as six, but never more than eight significant characters in external identifiers; often only one case of letters is significant. For specific differences, see the C Interface Manual for the relevant target machine.

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 per se (they should "never happen") and hence should be reported to the maintainers. Here is a summary of the diagnostics that can be produced by erroneous C programs:

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) - stuff inside () unrecognizable.
bad field width - negative or larger than word size.
bad flag - see manual page for p1.
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 modifier
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.
```

II. Portable C Runtime Library

TABLE OF CONTENTS

inbuf	find first occurrence in buffer of character in set
instr	find first occurrence in string of character in set
isalpha	test for alphabetic character
isdigit	test for digit
islower	test for lowercase character
isupper	test for uppercase character
iswhite	test for whitespace character
itob	convert integer to text in buffer
itols	convert integer to leading low-byte string
leave	leave a control region
lenstr	find length of a string
ln	natural logarithm
lower	convert characters in buffer to lowercase
lstoi	convert leading low-byte string to integer
lstol	convert filesystem date to long
ltob	convert long to text in buffer
ltols	convert long to filesystem date
mapchar	map single character to printable representation
match	match a regular expression
max	test for maximum
min	test for minimum
mkord	make an ordering function
nalloc	allocate space on the heap
notbuf	find first occurrence in buffer of character not in set
notstr	find first occurrence in string of character not in set
ordbuf	compare two NUL padded buffers for lexical order
pathnm	complete a pathname
pattern	build a regular expression pattern
prefix	test if one string is a prefix of the other
putc	put a character to output buffer
putch	put a character to stdout buffer
putf	output arguments formatted
putfmt	format arguments to stdout
putl	put a text line from buffer
putlin	put a text line to stdout
putstr	copy multiple strings to file
remark	print non-fatal error message
scnbuf	scan buffer for character
scnstr	scan string for character
sin	sine in radians
sort	sort items in memory
sqrt	real square root
squeeze	delete specified character from buffer
stdin	the standard input control buffer
stdout	the standard output control buffer
stob	convert short to text in buffer
subbuf	find occurrence of substring in buffer
substr	find occurrence of substring
tolower	convert character to lowercase if necessary
toupper	convert character to uppercase if necessary
usage	output standard usage information

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 Section I of this manual. Here, however, are a few general caveats: 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 Section III of this manual. Several of the "functions" described in this section 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 blow-by-blow summary of the sections in a typical library function description. For clarity, it is presented as a psuedo-manual page, with the remarks on each section of a real page appearing under the normal heading for that section. This page also points out where the conventions just mentioned are likely to rear their heads. The sections follow:

NAME

title - a name and concise description for the function
(The function is called from C by the name given.)

SYNOPSIS

Here the returned type and argument list of the function are given precisely as they would appear in a C program defining the function. Almost always, the types of arguments and function come from the set of psuedo-types defined by the standard header. These psuedo-types are for the most part simple equivalents to types pre-defined by C, renamed to increase their mnemonic value, to isolate machine dependencies, to promote disciplined coding practices, or (usually) to achieve some combination of all three.

FUNCTION

Generally, this section contains three parts, which may or may not be easily distinguishable. An opening sentence or two states the utility of the function: usually what it does, not how it does it. Next comes a summary of each argument to the function, including its gen-

eral effect on the operation of the function. Each argument is called by the same name as was given in the function synopsis. These names are usually mnemonic, to make citations distinct but still self-defining. Finally, an additional paragraph (or more) may provide more details of how the routine works, or how specific argument values affect it. This last component, for better or worse, has traditionally been kept to a remarkable minimum.

Note that heavy use is made of the conventional symbol NULL to refer to a zero-valued pointer, and of NUL to refer to the ASCII code zero, or "\0"; other symbols defined in the standard header may also crop up from time to time. Be prepared for them.

Ranges of numbers are often represented as in mathematics: "(0,4)" is the open interval "1,2,3", whereas "[0,4]" is the closed interval "0,1,2,3,4". Thus, "[0,4)" is not a typo, but shorthand for the half-open interval "0,1,2,3".

RETURNS

This section describes the range of possible return values for the function, and under what circumstances one value will be returned rather than another. Any location in the calling program altered via pointer access from the function is also documented here, though usually it has already been mentioned in the preceding section.

EXAMPLE

Here are shown one or more typical calls to the documented function, and (sometimes) their results. The examples are designed to be brief and evocative, or even useful as code fragments directly interpolated into user programs. Often, related functions have been given similar examples, either to emphasize the differences in usage between routines that perform similar tasks, or to show conventional patterns of use that apply across a family of routines.

SEE ALSO

This section lists related functions that could be profitably examined in conjunction with the current one. Functions documented in the same manual section are simply listed by name; functions documented elsewhere in the same manual are listed followed by the number of the section containing them.

If a description seems a bit impenetrable after a first reading, by all means look at the other ones mentioned here. Likewise if the current function doesn't do exactly what is wanted; a different one may come closer.

BUGS

Known inconsistencies or shortcomings in the documented routine are mentioned here. Most often, these relate to insufficient checking of value sensitive user-supplied parameters; the deviance of a calling sequence from the ideal is also sometimes mentioned. Finally, notice is always given here if a "function" is in reality a macro.

SEE ALSO

Portability(I), Style(I), main(III), std.h

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 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. Herewith the principal definitions:

Quasi-types

BITS - unsigned short, used as a set of 16 bits
BOOL - int, tested only for non-zero, 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
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 input/output 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 input/output 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 bizarre diagnostics when "redeclared".

NAME

Cio - C input/output subroutines

FUNCTION

There are dozens of subroutines for performing input/output at various levels of sophistication. Herewith a brief guide to which groups best work together:

The simplest approach to input/output 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 at a time, or line at a time, input/output 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 input/output 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 input/output under control of arbitrary FIO buffers can be obtained by calling getc, putc, getl, and putl.

At a lower level, there are creatures called "file descriptors", magic numbers of type FILE (defined in the standard header) that are handed out by certain routines and used by others. Three file descriptors 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, file descriptors can be used directly with the lowest level input/output routines, described below.

The routines open and create generate new file descriptors when they open files; close discredits a file descriptor 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 filesystem by calling remove.

File descriptors 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 handy for putting simple error messages.

BUGS

A tutorial is sorely needed.

NAME

FIO - the file input/output 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 input/output routines. Its elements are:

FILE _fd - holds the file descriptor for the file with which input/output 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 other than just 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 grown as necessary; if space is exhausted "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(III)

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.

Attempting to allocate more than half of the address space at a time is flaky.

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 leftmost "\(" 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 need be known only if patterns are to be built by hand.

RETURNS

amatch returns the index of the rightmost 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 / pi;
```

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 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 completely silly 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 completely silly 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 lowercase, 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 completely silly 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 completely silly 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. Forgetting 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 of data[0] and bit 64 as the most significant bit 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 algorithm is used, taking bit 1 as the least significant bit of data[0] and bit 64 as the most significant bit 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(IV), _when(IV), 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(III), exit(III)

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(IV), 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(III), 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(III)

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(III)

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 the looks of 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(III)

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. 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 types of cells can occur on the list, in any combination, so 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

If a list is freed that was not made from calls on alloc, all hell can break loose.

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 file descriptor 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 non-zero. On success the file descriptor 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 a file descriptor 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 file descriptor 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(III)

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 of 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", &arg1);
```

or:

```
getf(&stdin, "%+*nf", 9, &arg1);
```

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 non-white characters is taken to be the field; at least one non-white 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 ter-

minating 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 1 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 file descriptor 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 file descriptor 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 a file descriptor 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 file descriptor 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(III)

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 doesn't 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 arg1, ... 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 arg1, ...

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) a newline is copied, b) end of file is reached, or c) 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) a newline is copied, b) end of file is reached, or c) 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 lowercase character

SYNOPSIS
BOOL islower(c)

FUNCTION
islower tests whether its argument is a lowercase 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 uppercase character

SYNOPSIS
BOOL isupper(c)

FUNCTION
isupper tests whether its argument is an uppercase 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 non-printing 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 lowercase 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|) funny 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 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, so 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(IV), _when(IV), 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 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(IV), exp

NAME
lower - convert characters in buffer to lowercase

SYNOPSIS
```
BYTES lower(s, n)
   TEXT *s;
   BYTES n;
```

FUNCTION
lower converts the n characters in buffer starting at s to their lowercase 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 less 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, thanks 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|) funny 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 bizarre order is used to represent dates in Idris filesystems, thanks to their PDP-11 origins.

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

amatch, 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 other 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 non-spaces 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 complicated.

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 zeroeth 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(III)

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 to it in buf is appended the longest suffix of the string pointed at by n1 that does not contain a ':', ']', or '/'. 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 confounding 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 non-printing 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] won't work; 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 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 "sub-expressions".

A '$' as the rightmost 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 expressions beloved by mathematicians.

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

amatch, 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 non-text 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>. A "+z", if present, calls for the field to be left-filled with the character z, while "-z" calls for the output of right fill with z. A #, if present, specifies the 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 significance within the number. This permits multi-byte 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 non-zero.

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 non-text 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 non-text 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(III)

BUGS

Forgetting the terminating NULL pointer is usually disastrous.

NAME
remark - print non-fatal 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 can't 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(IV), 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 non-text 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 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 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|) funny 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 leftmost 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

amatch, 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 leftmost 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
amatch, inbuf, instr, match, notbuf, notstr, scnbuf, scnstr, subbuf

NAME
tolower - convert character to lowercase if necessary

SYNOPSIS
tolower(c)

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

RETURNS
tolower is a numerical rvalue guaranteed not to be an uppercase 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 other than just once.

NAME
toupper - convert character to uppercase if necessary

SYNOPSIS
toupper(c)

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

RETURNS
toupper is a numerical rvalue guaranteed not to be a lowercase 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 other than just 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(III), getflags

III. 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 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 0), STDOUT (file descriptor 1), and STDERR (file descriptor 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:

A filename - is a string, hence a NUL terminated array of characters, 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.

A 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.

A mode - is a short integer that specifies reading (mode == 0), writing (mode == 1), or updating (mode == 2). No other values are defined.

A binary file - looks to a C program like a sequence of characters, period. There is no record structure and all character codes are allowed. Trailing NULs may be provided, free of charge, by some operating systems.

A 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 linefeed) 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 of function call argument lists and automatic storage frames, and a heap, or unstructured 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 often must contend for the same (limited) space, so an otherwise correct C program may terminate early, or (sadly) misbehave, because insufficient space was allotted.

Note that all objects in C are presumed to have non-NULL addresses; the system is obliged 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 YES (or non-zero) if successful, otherwise NO (zero).

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 runtime.

It is used primarily for labelling diagnostic printouts.

SEE ALSO

error(II)

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

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 non-zero. Not all systems behave well if a textfile is created for updating.

RETURNS

create returns a file descriptor 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

exit - terminate program execution

SYNOPSIS

```
VOID exit(success)
   BOOL success;
```

FUNCTION

exit calls all functions registered with onexit, closes all files, and terminates program execution. exit is called with a non-zero (YES) to indicate success, or a zero (NO) to indicate unsuccessful termination; not all systems provide a recipient for this information.

RETURNS

exit will never return to its caller.

EXAMPLE

```
if ((fd = open("file", READ, 0)) < 0)
    {
    write(STDERR, "can't open file\n", 16);
    exit(NO);
    }
```

SEE ALSO

onexit

NAME

lseek - set file read/write pointer

SYNOPSIS

```
COUNT lseek(fd, offset, sense)
   FILE fd;
   LONG offset;
   COUNT sense;
```

FUNCTION

lseek uses the long offset provided to modify the read/write pointer for the binary file fd, under control of sense. If (sense == 0) the pointer is set to the byte offset, which should be positive. If (sense == 1) the byte offset is algebraically added to the current pointer. Other values of sense are extremely system dependent.

The call lseek(fd, 0L, 1) is guaranteed to leave the file pointer unmodified and, more important, to succeed only if lseek calls are both acceptable and meaningful for the fd specified. Other lseek calls may appear to succeed, but without effect, as when rewinding a terminal.

RETURNS

lseek returns the file descriptor if successful, or a negative number.

EXAMPLE

To read a 512-byte block:

```
BOOL getblock(buf, blkno)
    TEXT *buf;
    BYTES blkno;
    {

    lseek(STDIN, (LONG)blkno << 9, 0);
    return (read(STDIN, buf, BUFSIZE) != BUFSIZE);
    }
```

NAME

mkexec - make file executable

SYNOPSIS

```
BOOL mkexec(fname)
   TEXT *fname;
```

FUNCTION

mkexec converts the file fname to executable form. This may entail renaming the file by adding or replacing a system dependent suffix (or "extent") to fname; or it may simply involve altering access attributes. It is used by program constructors (loaders, linkers, task builders) to bless a successful product.

RETURNS

mkexec returns true if successful, otherwise false.

EXAMPLE

```
if (load1() && load2())
    return (mkexec(xfile));
```

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

onintr - capture interrupts

SYNOPSIS

```
VOID onintr(pfn)
   VOID (*pfn)();
```

FUNCTION

onintr ensures that the function at pfn is called on the occurrence of an interrupt generated from the keyboard of a controlling terminal. (Typing a delete DEL, or sometimes a ctl-C ETX, performs this service on many systems.) Any earlier call to onintr is overriden.

The function is called with one integer argument, whose value is always zero, and must not return; if it does, a message is output to STDERR and an immediate error exit is taken.

If (pfn is NULL) then the interrupt is disabled (turned off), assuming that the system supports such an operation. A disabled interrupt is not, however, turned on by a subsequent call with pfn not NULL. Systems that support nothing resembling a keyboard interrupt behave as if the interrupt were disabled at program startup, i.e., the function at pfn is never called.

RETURNS

Nothing.

EXAMPLE

A common use of onintr is to ensure a graceful exit on early termination:

```
VOID rmtemp()
    {
    remove(uname());
    }
    ...
    onexit(&rmtemp);
    onintr(&exit);
```

Still another use is to provide a way of terminating long printouts, as with an interactive editor:

```
    while (!enter(docmd, NULL))
        putstr(STDOUT, "?\n", NULL);
    ...
VOID docmd()
    {
    onintr(&leave);
    ...
```

SEE ALSO

enter(II), leave(II), onexit

NAME

open - open a file

SYNOPSIS

```
FILE open(fname, mode, rsize)
   TEXT *fname;
   COUNT mode;
   BYTES rsize;
```

FUNCTION

open opens a file fname and assigns a file descriptor 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).

If the file is to contain arbitrary binary data, as opposed to printable ASCII text, the record size rsize should be non-zero. Not all systems behave well if a text file is opened for updating.

RETURNS

open returns a file descriptor 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

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

uname - create a unique file name

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
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, writes 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)
           {
           putstr(STDERR, "write error\n", NULL);
           exit(NO);
           }
```

SEE ALSO
read

IV. C System 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 Section II or Section III, 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 a) to avoid cluttering an already extensive collection of useful functions with arcana, and b) to show prospective implementors what is required in the way of low level support for a new machine. Note that Section III 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

_fcan - canonicalize floating point datum

SYNOPSIS

```
COUNT _fcan(pd)
   TEXT *pd;
```

FUNCTION

_fcan is a machine dependent routine required by the C code generators to translate native double floating data to a canonical format. Each code generator can then translate from canonical to target machine format, irrespective of the host environment.

The canonical form is an array of eight characters stored in place of the double number at pd. pd[0] is zero if the number is positive, else 0200; pd[1] is the most significant byte of the fraction, with an assumed binary point to the left of its most significant bit; the remaining fraction bytes are stored in descending order of significance at pd[2] through pd[7]. If the number is nonzero, the most significant (0200) bit of pd[1] is set, so that the fraction is in the half-open interval [1/2, 1).

It is assumed that the number at pd is normalized on entry to _fcan.

RETURNS

_fcan returns the power of two by which the fraction must be multiplied to give the proper value. The sign and fraction bytes are written in place of the double number.

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) a double in the interval [0.1, 1) or zero, and b) 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(II), enter(II), leave(II)

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 non-digits 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 non-volatile 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 leftmost condition argument that will handle a given condition, in the latest part of the handler chain, is chosen to handle the condition.

Since _when plays fast and loose with the stack, it should never be used except as the lone operand in a switch statment, and all _when calls must be carefully coordinated with appropriate _raise calls to stay sane.

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 a) either argument to _raise was NULL or b) 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(II), leave(II)

BUGS

You are not expected to understand this.