CP-6

C Language Reference

Worldw
Informa
Systems

Bull

CP-6

C LANGUAGE REFERENCE

SUBJECT

Reference Information for the Bull CP-6 C Language and Library

SPECIAL INSTRUCTIONS

This edition is a new publication.

SOFTWARE SUPPORTED

C Version B00 under CP-6 Operating System E00

DATE

June 1990

ORDER NUMBER

HA17-00

Worldwide
Information
Systems

Bull

Preface

This publication is a reference document for the B00 version of the *CP-6* C language compiler, running on the E00 version of the *CP-6* operating system. *CP-6* C is an enhanced version of the American National Standard for Information Systems — Programming Language C (ANSI X3.159-1989). This reference is intended for programmers familiar with the C language and with the *CP-6* operating environment.

UNIX is a registered trademark of AT&T.

The Bull Los Angeles Development Center Documentation Group authors, edits, reviews and creates laser print masters with integrated text and graphics using CP-6 CAP (Computer Aided Publication).

Readers of this document may report errors or suggest changes through a STAR on the CP-6 STARLOG system.

File No.: 1W13

HA17-00

Table of Contents

List of Tables

List of Figures

About This Manual

The contents of this manual are grouped into 20 sections and 6 appendixes, providing the following information:

Section 1 introduces the user to the *CP-6* C compiler, and describes compiler invocation, the translation and execution environments, and environmental considerations such as the character set and numerical limits.

Sections 2 through 8 describe the *CP-6* C language.

Section 2 presents the lexical elements of *CP-6* C.

Section 3 describes data conversion.

Section 4 describes expressions.

Section 5 describes data declarations.

Section 6 describes statements.

Section 7 describes external data definitions.

Section 8 describes preprocessing directives.

Sections 9 through 20 describe the *CP-6* C library.

Section 9 introduces the user to the *CP-6* C library, and describes headers, errors, limits, common definitions, library functions, and diagnostics.

Section 10 describes character handling functions.

Section 11 describes localization functions.

Section 12 describes mathematics functions.

Section 13 discusses nonlocal jumps.

Section 14 discusses signal handling.

Section 15 discusses variable arguments.

Section 16 describes input/output functions.

Section 17 describes general utility functions.

Section 18 describes string handling functions.

Section 19 describes date and time functions.

Section 20 describes *CP-6* extensions to the C library.

The appendixes are as follows:

Appendix A	summarizes the *CP-6* C language.
Appendix B	summarizes the *CP-6* C library functions.
Appendix C	describes how to debug C programs with DELTA.
Appendix D	describes how to interface PL-6 and assembler routines to *CP-6* C.
Appendix E	describes portability issues.
Appendix F	describes environmental limits of *CP-6* C.

On-Line HELP Facility

CP-6 C has an on-line HELP facility. CP-6 C programmers can display the syntax and descriptions of language elements and library functions, and related information. For a list of HELP topics for *CP-6* C from the system command level (!), enter the following at the terminal:

```
HELP (CC) TOPICS
```

Related Documents

Following is the list of related manuals:
CP-6 Programmer Reference, Order Number CE40
CP-6 DELTA Reference, Order Number CE39
CP-6 PL-6 Reference, Order Number CE44.

Manuals may be ordered using Form No. HB-2808 from:

Bull HN Information Systems Inc.
Customer Services Operation
Publications Order Entry
141 Needham Street
MA35/219
Newton Highlands, MA 02161 U.S.A.

or may be ordered by telephone:

(617) 552-5199

(ICN) 552-5199 (FAX)

Notation Conventions

In the syntax notation used in the Language sections (2-8), syntactic categories (nonterminals) are indicated by *italic* type, and literal words and character set members (terminals) by **bold** type. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "*opt*", so that

$\{ \ expression_{opt} \ \}$

indicates an optional expression enclosed in braces.

In the Library sections (9-20), literal words and character set members are indicated in **bold** type. In the Synopses, variables are shown in *italic* type.

Where *CP-6* IBEX commands and DELTA debugger commands are explained, command keywords are indicated by **bold** type; variables to be supplied by the user are indicated by *italic* type. The following notation is used to show optional and required elements IBEX and DELTA commands:

- Brackets (*[ ]*) enclose elements or lists of elements that are optional.
- Braces ({ }) enclose lists of values or keywords from which one value or keyword must be chosen.
- Within a list of optional or required choices, the OR bar (|) separates each value or keyword from the next.
- An ellipsis (...) indicates that a previous element may be repeated.

Section 1

Introduction to the C Compiler

The *CP-6* C compiler translates C source files into object files and uses a run-time library for the execution of C programs. Other *CP-6* tools such as the linker (LINK) are necessary to create runable programs.

The *CP-6* debugger, DELTA, is available for use in debugging and checking out programs. DELTA provides the ability to set breakpoints on either data or statements and to display C variables.

Compilation Environment

The following paragraphs describe the *CP-6* C compilation environment. C programs are typically located in *CP-6* consecutive or keyed files.

Structuring Programs

An entire C program need not be compiled all at the same time. The text of the program may be organized into one or more *source files.* A source file together with all the headers and source files included by the preprocessing directive **#include**, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a *object unit.* Previously compiled object units may be preserved as object files or in LEMUR libraries. The separate object units of a program communicate by calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Object units may be separately compiled and then later linked to produce an executable program.

Translation Phases

The precedence among the syntax rules of translation is specified by the following phases:

1. Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.
2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines.
3. The source file is decomposed into preprocessing tokens[1]and sequences of white-space characters (including comments). A source file may not end in a partial preprocessing token or comment. Each comment is replaced by one space character. New-line characters are retained. White-space characters other than new-line are retained.
4. Preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.
5. Each source character set member and escape sequence in character constants and string literals is converted.
6. Adjacent character string literal tokens are concatenated, and adjacent wide string literal tokens are concatenated.
7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated.
8. All external object and function references are resolved. All such translator output is collected into an object unit which contains information needed for execution in its execution environment.

Diagnostics

The C compiler writes diagnostic messages through **M$DO**. Each diagnostic message contains the source line on which the problem occurred and a message indicating the reason. The compiler produces warning messages and error messages. Warnings indicate something may be wrong; error messages indicate a failure to correctly compile the program. A successful compilation is indicated by the Step Condition Code (STEPCC) being set to zero.

Listings

The C compiler writes a listing of the source program through **M$LO**. The listing includes each line of the source file being compiled (and optionally the include files) along with any diagnostics associated with various source lines.

A summary of the compilation is provided which, at the end, indicates the number of diagnostic messages and the full names of all the include files.

[1] As described in Section 2, Lexical Elements, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of < within a **#include** preprocessing directive.

Compiling and Linking C Programs

The C compiler is invoked by the following IBEX command:

`!CC` *[source][,update] [*{`ON`|`OVER`|`INTO`}*[object][,list]] [(options)]*

where:

source specifies the file that contains C source code.

update specifies a file containing updates to the source file.

`ON` requests an error if the object or list file currently exists.

`OVER` specifies that the object or list file is to be overwritten if it exists.

`INTO` specifies that the object or list file is to be extended if it exists.

object specifies the disk file to contain the generated object code.

list specifies the file to receive the generated listing.

options specifies one or more compiler options, separated by commas. (These are described in the next subsection.)

Figure 1-1 provides an example of entering, compiling, linking, and running a C program.

```
!build hello:c
EDIT E02 here
    1.000 main()  {
    2.000   printf("Hello, world!\n");
    3.000 }
    4.000
!CC hello:c over hello:o,*:ls
CC.B00 here at 15:02 Tue Jan 2 1990
!LINK hello:o over hello
LINK E02 here
*  :SHARED_C.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!hello.
Hello, world!
!
```

Figure 1-1. C Program Example

C Compiler Options

The *CP-6* C compiler accepts the following options:

ANS*[I]* causes the use of extensions to the ANSI C language be flagged.

BU*[*ILTIN*]* *[* (*fun[,fun,...]*) *]* requests that the compiler recognize certain library functions and generate code to perform it without calling the library or with a special calling sequence that significantly speeds up the execution. If no function names are specified, all functions which the compiler knows how to build in are recognized; otherwise, only those specific functions will be recognized. If the source file includes the header file which defines these functions, they will be built in by default without specifying this option (see Section 9, Use of Library Functions). The possible builtin functions include the following: **abs**, **atan**, **atan2**, **cos**, **exp**, **fabs**, **log**, **log10**, **memchr**, **memcmp**, **memcpy**, **memmove**, **memrchr**, **memset**, **pow**, **sin**, **sinh**, **sqrt**, **strcat**, **strchr**, **strcmp**, **strcpy**, **strlen**, **tan**, and **tanh**.

CP6SRCH requests that include files be located by strictly following the search list in the order specified. The default is CP6SRCH.

DEF*[*INE*]* (*name1[=text][,name2[=text],...]*) defines preprocessor variables with the optionally provided text. *text* may be a number, an identifier, or a single quoted string.

DMAP requests a data map for declared variables. The data map displays the location, type, and size for each variable.

KR requests Kernighan and Ritchie language where ANSI differs.

LND*[*IRECT*]* requests the preprocessor to include line number directives. The default is LNDIRECT.

LO requests listing of the generated code. The default is NLO.

LS requests listing of the source input. The default is LS unless the IBEX command DONT LIST was entered.

LU requests listing of the updates. The default is NLU.

N*[*O*]* BU*[*ILTIN] *[* (*fun[,fun,...]*) *]* requests that the compiler refrain from building in certain library functions. If no function names are specified, all functions which the compiler knows how to build in are assumed; otherwise, only those specific functions will not be built in. Even if the source file includes the header file which defines these functions, they will not be built in. The possible built-in function names are listed under the BUILTIN option.

NDEF*[*INE*]* (*name [,name]*) causes the named predefined preprocessor variables to be undefined. The variable names affected may be TM_L66, TS_CP6, __L66__, _CP6_, __LINE__, __FILE__, __DATE__, __TIME__, and __STDC__.

NLND*[*IRECT*]* causes the preprocessor to suppress the inclusion of line number directives. The default is LNDIRECT.

NLO suppresses listing of the generated code. The default is NLO.

NLS suppresses listing of the source code. The default is LS.

NLU suppresses listing of the updates. The default is NLU.

NOPT/IMIZE/ [(*opt*[,*opt*,...])] requests that the compiler not perform various optimizations. If the individual optimizations are not explicitly specified, then the compiler does not perform any of its possible optimizations. The list of optimizations is the same as for the OPTIMIZE option.

NOU suppresses generation of an object file. The default is OU.

NPMAP suppresses the procedure map. The default is NPMAP.

NUI requests that the compiler not include updates. The default is NUI.

NWA/RN/ suppresses the listing of warning messages. The default is WARN.

OPT/IMIZE/ [(*opt*[,*opt*,...])] requests that the compiler perform various optimizations. If the individual optimizations are not explicitly requested, then the compiler performs all of its optimizations. The individual optimizations are as follows: PEEP/HOLE/, INL/INE], SUBE/XPR], STR/ENGTH/, LOOPB/RANCH], CON/STANT], PRO/PAGATION], and REG/ISTERS/. By default, the PEEPHOLE and REGISTER optimizations are performed.

OPTUI reports no error when an update file does not exist.

OU requests generation of an object unit. (This option may not be specified with the PREPROCESS option.) The default is OU.

PMAP requests a procedure map for functions. The procedure map displays the location, statement type, and approximate execution cost for each statement. The default is NPMAP.

PREP/ROCESS/ causes the compiler to preprocess the source and write it through M$OU.

{S/EA/RCH|SEAR/CH/ } (*fid*[,*fid*,...]) requests that the compiler locate include files by searching the specified accounts. If *fid* includes a filename part, then that part is prefixed to the include file name (or the include file name replaces the ? character in the *fid*). The compiler always adds the account :LIBRARY to the end of the search list. The default is CP6SRCH.

S/TATIC/ F/UNCTION/ S/UFFIX/ = '*string*' requests that *string* be used as the suffix added to static function names to make them unique. By default, *CP-6* C generates a unique name for every static function by using the name of the first nonstatic function or, if there are no nonstatic functions, the name of an extern variable defined in the file.

STR/INGS/ = {READ/ONLY/|WRITE/ABLE/} requests that the compiler put strings in write-protected memory or writeable memory. By default, a C program may not alter a character string constant. The default is STRINGS=READONLY.

UI requests that the compiler include updates. The default is NUI unless the UI filename is specified on the command line.

UNIXSRCH requests that nested include files be located by the UNIX method. If the file name is enclosed in angle brackets (<*filename*>), then the search is the same as for the CP6SRCH option. Otherwise, for file names enclosed in double quotes ("*filename*"), the search begins as if the search list were preceded by the search list entry used to locate the including source file. The default is CP6SRCH.

WA/RN/ requests the listing of warnings for recoverable errors. The default is **WARN**.

Execution Environment

The following paragraphs describe the *CP-6* C execution environment.

Program Startup

The function called at program startup is named **main**. *CP-6* C has no required prototype for this function. It can be defined with no parameters:

```
int main(void)  /*...*/
```

or with two parameters (referred to here as **argc** and **argv**, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[])  /*...*/
```

If they are defined, the parameters to the **main** function obey the following constraints:

- The value of **argc** is a positive integer.
- **argv[argc]** is a null pointer.
- The array members **argv[0]** through **argv[argc-1]** inclusive contain pointers to strings, which are tokens from the invocation line. The intent is to supply to the program information determined prior to program startup from the user or other programs.
- The string pointed to by **argv[0]** represents the *program name*. If the value of **argc** is greater than one, the strings pointed to by **argv[1]** through **argv[argc-1]** represent the *program parameters*.
- The parameters **argc** and **argv** and the strings pointed to by the **argv** array are modifiable by the program, and retain their last-stored values between program startup and program termination.

Program Execution

A program may use all the functions, macros, type definitions, and objects described in the library sections (9-20) of this manual.

C Run Unit Invocation

A C run unit can be invoked using either *CP-6* Standard Invocation or a nonstandard UNIX-like invocation. In both cases, command line arguments and options are passed to the C program via the **argc** and **argv** parameters to the **main** function. In either case, **argv**[0] contains the name of the run unit (as entered by the user) and **stderr** is opened to the current M$DO DCB setting.

CP-6 Standard Invocation

To perform *CP-6* standard invocation, the run unit must first be linked with the STDINVOC option. The syntax for *CP-6* standard invocation is:

!ru [dcb1,dcb2 [{on|over|into} *[dcb3][,dcb4]] [(options)]*

The *options* list contains one or more options separated by commas or white space. The entire *options* list including the parentheses is provided as a single **argv** string. When the program is executed, the C library opens **stdin** to the current M$SI DCB setting and **stdout** to the current M$LO DCB setting.

Example:

```
!blast OVER *OUT (LS,special,fizz)
```

If the run unit **blast** was linked with the options (DCB1=M$SI,DCB3=M$LO,STDINVOC), then in this example **stdout** is opened to the file *OUT using mode **w**. **stdin** is opened to the device ME using using mode **r** (if M$SI has not been set in IBEX). The parameters to the **main** function would have the following values:

```
argc == 2
argv[0] == "blast"
argv[1] == "(LS,special,fizz)"
argv[2] == (char *)0
```

When running C run units linked with standard invocation, the M$SI and M$LO DCBs are automatically opened as **stdin** and **stdout**. Other command line DCBs may be used by C programs, but the C library does not open them automatically. To open the other command line DCBs, the **fopen** or **freopen** function must be called with the file name argument "dcb=*dcbname*" or "#*n*", where *n* is a command line DCB number (1, 2, 3, or 4).

UNIX-Like Invocation

UNIX-like invocation occurs if the run unit is not linked with the STDINVOC option. The syntax is:

> !*ru [{ token [token...] | redirection_specification }...]*

where:

token is a contiguous sequence of non-white-space characters. White-space characters may be included in a token by enclosing them in single quotes. Single quotes are included in tokens by preceding them with a backslash character.

redirection_specification is of the form:
> { < | > | >> } *[white-space] token*
>
> "<" specifies input redirection. The following token is treated as a file name and the **stdin** stream is connected to that file for reading.
>
> ">" specifies output redirection. The following token is treated as a file name and the **stdout** stream is connected to that file for writing.
>
> ">>" specifies output append redirection. The following token is treated as a file name and the **stdout** stream is connected to that file for writing. If the file already exists, output is appended to the end of the file.

Tokens that are not part of a redirection specification are put into the **argv** list.

Example:

```
!ru >gorp fizz foo-bar
```

The **stdout** stream is opened to the file **gorp**. The **stdin** stream is opened to the **ME** device.

The parameters to the **main** function would have the values:

```
argc    == 3
argv[0] == "ru"
argv[1] == "fizz"
argv[2] == "foo-bar"
argv[3] == (char *)0
```

Program Termination

A return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument. If the **main** function executes a return that specifies no value, the step condition code STEPCC is set to 0.

Environmental Considerations

The following paragraphs describe the environment in effect during compilation.

Character Set

In a character constant or string literal, members of the character set may be represented by the character set or by *escape sequences* consisting of the backslash (\) followed by one or more characters. A byte with all bits set to 0, called the *null character*, terminates a character string literal.

The basic character set has the following members:

the 26 upper-case letters of the English alphabet:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

the 26 lower-case letters of the English alphabet:

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

the 10 decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

the following 32 graphic characters:

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

and the following white-space characters:

space-character
horizontal-tab
vertical-tab
form-feed

Trigraph Sequences

A *trigraph sequences* is a special 3-character group that is used to define a single character that is not part of the ISO ANSI code set. Table 1-1 shows the trigraph sequences with the corresponding single character each represents.

Trigraph	Meaning
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

Table 1-1. Trigraph Sequences

No other trigraph sequences exist. Each ? that does not begin one of the trigraphs listed above is not changed.

Example:

The following source line

```
printf("Eh???/n");
```

becomes (after replacement of the trigraph sequence ??/)

```
printf("Eh?\n");
```

Character Display Semantics

The *active position* is that location on a display device where the next character output by the `fputc` function would appear. The intent of writing a printable character (as defined by the `isprint` function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. If the active position is at the final position of a line (if there is one), CP-6 typically adds an automatic new line.

Alphabetic escape sequences representing nongraphic characters in the execution character set produce actions on display devices (terminals) described in Table 1-2.

Escape Sequence	Meaning
`\a` (*alert*)	Produces an audible or visible alert when displayed on a terminal. The active position is not changed.
`\b` (*backspace*)	Moves the active position to the previous position on the current terminal line. If the active position is at the initial position of a line, the behavior depends on the terminal type.
`\f` (*form feed*)	Moves the active position to the initial position at the start of the next logical page.
`\n` (*new line*)	Moves the active position to the initial position of the next line.
`\r` (*carriage return*)	Moves the active position to the initial position of the current line.
`\t` (*horizontal tab*)	Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior depends on the terminal type.
`\v` (*vertical tab*)	Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior depends on the terminal type.

Table 1-2. Alphabetic Escape Sequences

As shown in Table 1-3, each of these escape sequences produces a unique value which can be stored in a **char** object.

Escape Sequence	Value
`\a`	7
`\b`	8
`\f`	12
`\r`	13
`\t`	9
`\v`	11

Table 1-3. Escape Sequence Values

Signals and Interrupts

The functions in the library are not guaranteed to be re-entrant and may modify objects with static storage duration. This means that they should not be executed from signal handling functions.

Section 2

Lexical Elements

This section describes the lexical elements of the C language: keywords, identifiers, constants, string literals, operators, punctuators, header names, preprocessing numbers, and comments. These are shown in the syntax below. For each individual element, syntax, description, constraints, semantics, and examples are presented where appropriate.

Syntax:

```
    token:
            keyword
            identifier
            constant
            string-literal
            operator
            punctuator

    preprocessing-token:
            header-name
            identifier
            pp-number
            character-constant
            string-literal
            operator
            punctuator
            each non-white-space character that cannot be one of the above
```

Constraints:

Each preprocessing token that is converted to a token has the lexical form of a keyword, an identifier, a constant, a string literal, an operator, or a punctuator.

Semantics:

A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: *keywords*, *identifiers*, *constants*, *string literals*, *operators*, and *punctuators*. A *preprocessing token* is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: *header names*, *identifiers*, *preprocessing numbers*, *character constants*, *string literals*, *operators*, *punctuators*, and single non-white-space characters (except ' and ") that do not lexically match the other preprocessing token categories. Preprocessing tokens can be separated by *white space*; this consists of comments (described at the end of this section), *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Section 8, Preprocessing Directives, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token.

Examples:

The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating constant token), whether or not `E` is a macro name.

The program fragment `x+++++y` is parsed as `x ++ ++ + y` , which violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression.

Keywords

Syntax:

keyword: one of

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

Semantics:

The above tokens (entirely in lowercase) are reserved (in translation phases 7 and 8) for use as keywords and cannot be used otherwise.

Identifiers

Syntax:

identifier:
  nondigit
  identifier nondigit
  identifier digit

nondigit: one of

```
_  a  b  c  d  e  f  g  h  i  j  k  l  m
   n  o  p  q  r  s  t  u  v  w  x  y  z
   A  B  C  D  E  F  G  H  I  J  K  L  M
   N  O  P  Q  R  S  T  U  V  W  X  Y  Z
$
```

digit: one of

```
0  1  2  3  4  5  6  7  8  9
```

Description:

An identifier is a sequence of nondigit characters, including the underscore (_) and the lower-case and upper-case letters, and digits. The first character is a nondigit character. In *CP-6* C, "$" is permitted to occur in an identifier. Use the ANSI option to restrict identifiers.

Constraints:

In translation phases 7 and 8, an identifier may not consist of the same sequence of characters as a keyword.

Semantics:

An identifier denotes an object, a function, or one of the following entities described in Sections 4 through 8: a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. A member of an enumeration is called an *enumeration constant.* Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

The default maximum identifier length is 64 characters.

Implementation Limits:

CP-6 C uses only the first 64 characters of an *internal name* (a macro name or an identifier that does not have external linkage) or *external name* (an identifier that has external linkage). In names, the case of letters is significant.

Scopes of Identifiers

An identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

A label name is the only kind of identifier that has *function scope*. It can be used (in a `goto` statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a `:` and a statement). Label names must be unique within a function.

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the object unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the `}` that closes the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an outer declaration of a lexically identical identifier exists in the same name space, it is hidden until the current scope terminates, after which it again becomes visible.

Two identifiers have the same scope if and only if their scopes terminate at the same point.

Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

Linkages of Identifiers

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*. There are three kinds of linkage: external, internal, and none.

In the set of object units and libraries that constitutes an entire program, each instance of a particular identifier with *external linkage* denotes the same object or function. Within one object unit, each instance of an identifier with *internal linkage* denotes the same object or function. Identifiers with *no linkage* denote unique entities.

If the declaration of an identifier for an object or a function has file scope and contains the storage-class specifier `static`, the identifier has internal linkage.

If the declaration of an identifier for an object or a function contains the storage-class specifier `extern`, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier `extern`.

Within a object unit, the same identifier must not appear with both internal and external linkage.

Name Spaces of Identifiers

If more than one declaration of a particular identifier is visible at any point in a object unit, the uses that refer to different entities are specified by the syntactic context. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

- *Label names* (specified by the syntax of the label declaration and use).
- *Tags* of structures, unions, and enumerations (specified by following any[1]of the keywords `struct`, `union`, or `enum`).
- *Members* of structures or unions; each structure or union has a separate name space for its members (specified by the type of the expression used to access the member via the `.` or `->` operator).
- All other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Storage Durations of Objects

An object has a *storage duration* that determines its lifetime. There are two storage durations: static and automatic.

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier `static`, has *static storage duration.* For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists and retains its last-stored value throughout the execution of the entire program.[2]

[1] There is only one name space for tags even though three are possible.

[2] In the case of a volatile object, the last store may not be explicit in the program.

An object whose identifier is declared with no linkage and without the storage-class specifier `static` has *automatic storage duration.* Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated, or on a jump from outside the block to a labeled statement in the block or in an enclosed block. If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block suspends but does not end execution of the enclosing block. Calling a function suspends but does not end execution of the block containing the call.) The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

Types

The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

An object declared as type `char` is large enough to store any member of the character set. If a member of the required source character set enumerated in Section 1 is stored in a `char` object, its value is guaranteed to be positive. If other quantities are stored in a `char` object, the value is truncated and treated as non-negative integers.

There are four *signed integer types*, designated as `signed char`, `short int`, `int`, and `long int`. (The signed integer and other types may be designated in several additional ways, as described in Section 5, Data Declarations.) The types `short int`, `int`, and `long int` are the same size in *CP-6* C.

An object declared as type `signed char` occupies the same amount of storage as a "plain" `char` object. A "plain" `int` object occupies 4 bytes (36 bits), which is large enough to contain any value in the range `INT_MIN` to `INT_MAX` as defined in the header `<limits.h>`. In the list of signed integer types above, the range of values of each type is a subrange of the values of the next type in the list.

For each of the signed integer types, there is a corresponding (but different) *unsigned integer type* (designated with the keyword `unsigned`) that uses the same amount of storage (including sign information) and has the same alignment requirements. The range of non-negative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[3] A

[3] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

There are three *floating types*, designated as `float`, `double`, and `long double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is the same as the set of values of the type `long double`. A `float` occupies four bytes; a `double` or a `long double` occupies eight bytes.

The type `char`, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. Even though two or more basic types have the same internal representation, they are nevertheless different types.

The three types `char`, `signed char`, and `unsigned char` are collectively called the *character types*.

An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed.

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows:

- An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called "array of T". The construction of an array type from an element type is called "array type derivation".
- A *structure type* describes a sequentially allocated nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.
- A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.
- A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is T, the function type is sometimes called "function returning T". The construction of a function type from a return type is called "function type derivation".
- A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called "pointer to T". The construction of a pointer type from a referenced type is called "pointer type derivation".

These methods of constructing derived types can be applied recursively.

The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integral types. Floating types* are represented by using hexadecimal floating point numbers.

Integral and floating types are collectively called *arithmetic types.* Arithmetic types and pointer types are collectively called *scalar types.* Array and structure types are collectively called *aggregate types.*[4]

An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in Section 5, Data Declarations) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

Array, function, and pointer types are collectively called *derived declarator types.* A *declarator type derivation* from a type T is the construction of a derived declarator type from T by the application of an array, a function, or a pointer type derivation to T.

A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

Any type so far mentioned is an *unqualified type.* Each unqualified type has three corresponding *qualified versions* of its type: a *const-qualified* version, a *volatile-qualified* version, and a version having both qualifications. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[5]A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

A pointer to **void** has the same representation and alignment requirements as a pointer to a character type. Similarly, pointers to qualified or unqualified versions of compatible types have the same representation and alignment requirements.[6]Pointers to other types need not have the same representation or alignment requirements.

[4] Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[5] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

[6] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

Examples:

The type designated as "`float *`" has type "pointer to `float`". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type — its type is "pointer to const-qualified `float`" and is a pointer to a qualified type.

Finally, the type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float` . Its type category is array.

Compatible Type and Composite Type

Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in Section 5, Data Declarations, for type specifiers, type qualifiers, and declarators.[7]Moreover, two structure, union, or enumeration types declared in separate object units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members must be in the same order; for two structures or unions, the bit-fields must have the same widths; for two enumerations, the members must have the same values.

All declarations that refer to the same object or function must have compatible type.

A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

- If one type is an array of known size, the composite type is an array of that size.
- If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.
- If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

Examples:

Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

[7] Two types need not be identical to be compatible.

Constants

Syntax:

> *constant:*
> > *floating-constant*
> > *integer-constant*
> > *enumeration-constant*
> > *character-constant*

Constraints:

The value of a constant must be in the range of representable values for its type.

Semantics:

Each constant has a type, determined by its form and value, as detailed in Section 5, Data Declarations.

Floating Constants

Syntax:

> *floating-constant:*
> > *fractional-constant exponent-part*$_{opt}$ *floating-suffix*$_{opt}$
> > *digit-sequence exponent-part floating-suffix*$_{opt}$
>
> *fractional-constant:*
> > *digit-sequence*$_{opt}$ **.** *digit-sequence*
> > *digit-sequence* **.**
>
> *exponent-part:*
> > **e** *sign*$_{opt}$ *digit-sequence*
> > **E** *sign*$_{opt}$ *digit-sequence*
>
> *sign:* one of
> > **+ -**
>
> *digit-sequence:*
> > *digit*
> > *digit-sequence digit*
>
> *floating-suffix:* one of
> > **f l F L**

Description:

A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (`.`), followed by a digit sequence representing the fraction part. The components of the exponent part are an `e` or `E` followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part must be present; either the period or the exponent part must be present.

Semantics:

The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled. If the scaled value is in the range of representable values (for its type), the result is the smaller representable value immediately adjacent to the nearest representable value.

An unsuffixed floating constant has type `double`. If suffixed by the letter `f` or `F`, it has type `float`. If suffixed by the letter `l` or `L`, it has type `long double`.

Integer Constants

Syntax:

integer-constant:
  decimal-constant integer-suffix$_{opt}$
  octal-constant integer-suffix$_{opt}$
  hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
  nonzero-digit
  decimal-constant digit

octal-constant:
  `0`
  octal-constant octal-digit

hexadecimal-constant:
  `0x` *hexadecimal-digit*
  `0X` *hexadecimal-digit*
  hexadecimal-constant hexadecimal-digit

nonzero-digit: one of
  `1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

Description:

An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix `0` optionally followed by a sequence of the digits `0` through `7` only. A hexadecimal constant consists of the prefix `0x` or `0X` followed by a sequence of the decimal digits and the letters `a` (or `A`) through `f` (or `F`) with values 10 through 15 respectively.

Semantics:

The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal: `int`, `long int`, `unsigned long int`; unsuffixed octal or hexadecimal: `int`, `unsigned int`, `long int`, `unsigned long int`; suffixed by the letter `u` or `U`: `unsigned int`, `unsigned long int`; suffixed by the letter `l` or `L`: `long int`, `unsigned long int`; suffixed by both the letters `u` or `U` and `l` or `L`: `unsigned long int`

Enumeration Constants

Syntax:

enumeration-constant:
identifier

Semantics:

An identifier declared as an enumeration constant has type `int`.

Character Constants

Syntax:

character-constant:
  ' *c-char-sequence* '
  L' *c-char-sequence* '

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except
    the single quote ('), backslash (\), or new-line character
  escape-sequence

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  `\'  \"  \?  \\`
  `\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
  \ *octal-digit*
  \ *octal-digit octal-digit*
  \ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
  `\x` *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

Description:

An integer character constant is a sequence of one or more multibyte characters enclosed in single quotes, as in `'x'` or `'ab'`. A wide character constant is the same, except prefixed by the letter L. With a few exceptions detailed in Table 2-1, the elements of the sequence are any members of the source character set.

The single quote ('), double quote ("), question mark (?), backslash (\), and arbitrary integral values are representable as the escape sequences shown in the following table:

Character Constant	Escape Sequence
single quote (')	\'
double quote (")	\"
question mark (?)	\?
backslash (\)	\\
octal integer	\octal digits
hexadecimal integer	\x*hexadecimal digits*

Table 2-1. Escape Sequences

The double quote (") and question mark (?) are representable either by themselves or by the escape sequences \" and \?, respectively; the single quote (') and backslash (\) are represented by the escape sequences \' and \\, respectively.

The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

In addition, certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lower-case letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[8]No other escape sequence may be used.

Semantics:

An integer character constant has type `int`. The value of an integer character constant containing a single character that maps into a member of the character set is the numerical value of the representation of the mapped character interpreted as an integer. An integer character constant containing up to four characters may be used to specify an integer value. If an escape sequence is not recognized in a character constant, the backslash (\) is removed from the string. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type `char` whose value is that of the single character or escape sequence is converted to type `int`.

A wide character constant has type `wchar_t`, an integral type defined in the `<stddef.h>` header as `char`. The value of a wide character constant containing a single multibyte character that maps into a member of the extended execution character set is the *wide character* (code) corresponding to that multibyte character, as defined by the `mbtowc` function, with a locale of `"C"`.

[8] The semantics of these characters are discussed in Section 1.

String Literals

Syntax:

string-literal:
    "*s-char-sequence*$_{opt}$"
    L"*s-char-sequence*$_{opt}$"

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char:
    any member of the source character set except
        the double-quote ("), backslash (\), or new-line character
    escape-sequence

Description:

A character string literal is a sequence of zero or more multibyte characters enclosed in double quotes, as in "**xyz**". A wide string literal is the same, except prefixed by the letter L.

The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, except that the single quote (') is representable either by itself or by the escape sequence (\'), but the double quote (") is representable by the escape sequence (\").

Semantics:

In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string literal tokens, are concatenated into a single multibyte character sequence.

In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[9]The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals and wide string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence.

Identical string literals may not be distinct. The program should not attempt to modify a string literal because, by default, *CP-6* C puts strings in read-only memory.

[9] A character string literal need not be a string (see Section 18), because a null character may be embedded in it by a \0 escape sequence.

Examples:

This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are `\x12` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

Operators

Syntax:

operator: one of

```
[  ]  (  )  .  ->
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
?  :
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #  ##
```

Constraints:

The operators `[ ]`, `( )`, and `? :` occur in pairs, possibly separated by expressions. The operators `#` and `##` may appear in macro-defining preprocessing directives only.

Semantics:

An operator specifies an operation to be performed (an *evaluation*) that yields a value, yields a designator, produces a side effect, or a combination thereof. An *operand* is an entity on which an operator acts.

Punctuators

Syntax:

punctuator: one of

```
[  ]  (  )  {  }  *  ,  :  =  ;  ...  #
```

Constraints:

The punctuators `[ ]`, `( )`, and `{ }` occur in pairs, possibly separated by expressions, declarations, or statements. The punctuator `#` may appear in preprocessing directives only.

Semantics:

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value. Depending on context, the same symbol may also represent an operator or part of an operator.

Header Names

Syntax:

header-name:
    <h-char-sequence>
    "q-char-sequence"

h-char-sequence:
    h-char
    h-char-sequence h-char

h-char:
    any member of the source character set except
        the new-line character and >

q-char-sequence:
    q-char
    q-char-sequence q-char

q-char:
    any member of the source character set except
        the new-line character and "

Constraints:

Header name preprocessing tokens may appear only within a `#include` preprocessing directive.

Semantics:

The sequences in both forms of header names are mapped into *CP-6* file names representing headers or external source file names, as specified in Section 8, Preprocessing Directives.

The characters ', \, ", or /* should not appear in the sequence between the < and > delimiters. Similarly, the characters ', \, or /* should not appear in the sequence between the " delimiters.[10]

Examples:

The following sequence of characters:

```
0x3<1/a.h>1e2
#include </a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a { on the left and a } on the right):

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include}  {<1/a.h>}
{#}{define}  {const}{.}{member}{@}{$}
```

[10] Thus, sequences of characters that resemble escape sequences should not be used.

Preprocessing Numbers

Syntax:

pp-number:
- *digit*
- . *digit*
- *pp-number digit*
- *pp-number nondigit*
- *pp-number* e *sign*
- *pp-number* E *sign*
- *pp-number* .

Description:

A preprocessing number begins with a digit optionally preceded by a period (.) and may be followed by letters, underscores, digits, periods, and e+, e-, E+, or E- character sequences.

Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics:

A preprocessing number does not have type or a value; it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

Comments

Except within a character constant, a string literal, or a comment, the characters /* introduce a comment. The contents of a comment are examined only to find the characters */ that terminate it.[11]

[11] Thus comments do not nest.

Section 3

Data Conversions

Several operators convert operand values from one type to another automatically. This section specifies the result that occurs from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). This section summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in Section 4, Expressions.

Conversion of an operand value to a compatible type causes no change to the value or the representation.

Arithmetic Operands

The types of arithmetic operands are described in the following paragraphs.

Characters and Integers

A **`char`**, a **`short int`**, an **`int`** bit-field, their signed or unsigned varieties, or an object that has enumeration type may be used in an expression wherever an **`int`** or **`unsigned int`** may be used. If an **`int`** can represent all values of the original type, the value is converted to an **`int`**; otherwise it is converted to an **`unsigned int`**. These are called the *integral promotions.*[1] All other arithmetic types are unchanged by the integral promotions.

The integral promotions preserve value including sign. As discussed As discussed under Types in Section 2, a "plain" **`char`** is treated as unsigned.

[1] The integral promotions are applied only as part of the usual arithmetic conversions; to certain argument expressions; to the operands of the unary +, -, and ~ operators; and to both operands of the shift operators; as specified by their respective sections.

Signed and Unsigned Integers

When an integer is converted to another integral type and the value can be represented by the new type, its value is unchanged.

When a signed integer is converted to an unsigned integer with equal or greater size and the value of the signed integer is non-negative, its value is unchanged. Otherwise, if the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.[2]

When an integer is demoted to an unsigned integer with smaller size, the result is the non-negative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size. When an integer is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, the result is truncated if the value cannot be represented.

Floating and Integral

When a value of floating type is converted to integral type, the fractional part is discarded.[3]

When a value of integral type is converted to floating type and the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is the nearest lower value.

Floating Types

When a `float` is promoted to `double` or `long double`, or a `double` is promoted to `long double`, its value is unchanged.

When a `double` is demoted to `float` or a `long double` is demoted to `double` or `float`, and the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is the nearest lower value.

Usual Arithmetic Conversions

Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the ***usual arithmetic conversions***, as follows:

[2] In a two's-complement representation, there is no actual change in the bit pattern except filling the high-order bits with copies of the sign bit if the unsigned integer has greater size.

[3] The remaindering operation performed when a value of integral type is converted to unsigned type is not performed. Thus the range of portable floating values is [0, `U`*type*`_MAX` +1).

- First, if either operand has type `long double`, the other operand is converted to `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to `double`.
- Otherwise, if either operand has type `float`, the other operand is converted to `float`.
- Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:
  - If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
  - Otherwise, if one operand has type `long int` and the other has type `unsigned int`, and a `long int` can represent all values of an `unsigned int`, the operand of type `unsigned int` is converted to `long int`; if a `long int` cannot represent all the values of an `unsigned int`, both operands are converted to `unsigned long int`.
  - Otherwise, if either operand has type `long int`, the other operand is converted to `long int`.
  - Otherwise, if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
  - Otherwise, both operands have type `int`.

The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type.[4]

Other Operands

The following paragraphs describe the lvalue, function designator, and void expressions, and pointers.

Lvalues and Function Designators

An *lvalue* is an expression (with an object type or an incomplete type other than `void`) that designates an object.[5]When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

[4] The cast and assignment operators still perform their specified conversions.

[5] The name "lvalue" comes originally from the assignment expression `E1 = E2`, in which the left operand `E1` must be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this manual described as the "value of an expression".

An obvious example of an lvalue is an identifier of an object. As a further example, if `E` is a unary expression that is a pointer to an object, `*E` is an lvalue that designates the object to which `E` points.

Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; otherwise the value has the type of the lvalue. An lvalue that has an incomplete type and does not have array type should not be used.

Except when it is the operand of the `sizeof` operator or the unary `&` operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with `wchar_t`, an lvalue that has type "array of *type*" is converted to an expression that has type "pointer to *type*" that points to the initial element of the array object and is not an lvalue.

A *function designator* is an expression that has function type. Except when it is the operand of the `sizeof` operator[6]or the unary `&` operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

void

The (nonexistent) value of a *void expression* (an expression that has type `void`) may not be used in any way, and implicit or explicit conversions (except to `void`) may not be applied to such an expression. If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded. (A void expression is evaluated for its side effects.)

Pointers

A pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result compares equal to the original pointer.

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers compare equal.

An integral constant expression with the value 0, or such an expression cast to type `void *`, is called a *null pointer constant.* If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

Two null pointers, converted through possibly different sequences of casts to pointer types, compare equal.

[6] **Because this conversion does not occur, the operand of the `sizeof` operator remains a function designator and violates the constraint in Section 4, Expressions.**

Section 4

Expressions

An *expression* is a sequence of operators and operands that specifies computation of a value, designates an object or a function, generates side effects, or performs a combination thereof.

Between the previous and next sequence point, an object may have its stored value modified at most once by the evaluation of an expression. The prior value is accessed only to determine the value to be stored.[1]

Except as indicated by the syntax[2]or otherwise specified later (for the function-call operator `()`, `&&`, `||`, `?:`, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as *bitwise operators*) have operands of integral type. These operators return values that depend on the internal representations of integers.

---

[1] This paragraph renders non-portable statement expressions such as

```
i = ++i + 1;
```

while allowing

```
i = i + 1;
```

[2] The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator are those expressions defined in Primary Expressions through Additive Operators, in this section. The exceptions are cast expressions as operands of unary operators and an operand contained between any of the following pairs of operators: grouping parentheses `()`, subscripting brackets `[]`, function-call parentheses `()`, and the conditional operator `?:`. These are all described in this section.

Within each major subsection, the operators have the same precedence. Left- or right-associativity is indicated in each subsection by the syntax for the expressions discussed therein.

An *exception* can occur during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type). Normally, an exception condition is raised and the program is aborted.

For each expression and operator discussed in this section, syntax, description, constraints, semantics, and examples are presented where appropriate.

Primary Expressions

Syntax:

> *primary-expression:*
> > *identifier*
> > *constant*
> > *string-literal*
> > (*expression*)

Semantics:

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).

A constant is a primary expression. Its type depends on its form and value, as detailed in Section 2, Lexical Elements.

A string literal is a primary expression. It is an lvalue with type as detailed in Section 2, Lexical Elements.

A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

Postfix Operators

Syntax:

> *postfix-expression:*
> > *primary-expression*
> > *postfix-expression* [*expression*]
> > *postfix-expression* (*argument-expression-list*$_{opt}$)
> > *postfix-expression* . *identifier*
> > *postfix-expression* -> *identifier*
> > *postfix-expression* ++
> > *postfix-expression* --
>
> *argument-expression-list:*
> > *assignment-expression*
> > *argument-expression-list* , *assignment-expression*

Array Subscripting

Constraints:

One of the expressions has type "pointer to object *type*", the other expression has integral type, and the result has type "*type*".

Semantics:

A postfix expression followed by an expression in square brackets (`[]`) is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*(E1+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2`-th element of `E1` (counting from zero).

Successive subscript operators designate an element of a multidimensional array object. If `E` is an *n*-dimensional array ($n \geq 2$) with dimensions $i \times j \times \ldots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n - 1)$-dimensional array with dimensions $j \times \ldots \times k$. If the unary `*` operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to $(n - 1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

Examples:

Consider the array object defined by the following declaration:

```
        int x[3][5];
```

Here, `x` is a 3×5 array of `int`s; more precisely, `x` is an array of three element objects, each of which is an array of five `int`s. In the expression `x[i]`, which is equivalent to `(*(x+(i)))`, `x` is first converted to a pointer to the initial array of five `int`s. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five `int` objects. The results are added and indirection is applied to yield an array of five `int`s. When used in the expression `x[i][j]`, that in turn is converted to a pointer to the first of the `int`s, so `x[i][j]` yields an `int`.

Function Calls

Constraints:

The expression that denotes the called function[3]has type pointer to function returning `void` or returning an object type other than an array type.

If the expression that denotes the called function has a type that includes a prototype, the number of arguments must agree with the number of parameters. Each argument has a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics:

A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration

```
    extern int identifier();
```

appeared.[4]

An argument can be an expression of any object type. In preparing for the call to a function, the arguments are evaluated and each parameter is assigned the value of the corresponding argument.[5]The value of the function call expression is specified in Section 6, Statements.

If the expression that denotes the called function has a type that does not include a prototype, the integral promotions are performed on each argument, and arguments that have type `float` are promoted to `double`. These are called the *default argument promotions....* The number of arguments should agree with the number of parameters. The function should be defined with a type that does not include a prototype, and the types of the arguments after promotion should be compatible with those of the parameters after promotion. If the function is defined with a type that includes a prototype and the types of the arguments after promotion are not compatible with the types of the

[3] Most often, this is the result of converting an identifier that is a function designator.

[4] That is, an identifier with block scope declared to have external linkage with type function without parameter information and returning an `int`. It should not be defined as having type "function returning `int`".

[5] A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is converted to a parameter with a pointer type as described in Section 7, External Data Definitions.

parameters, or if the prototype ends with an ellipsis (`, ...`), an error is reported if there is not a type conversion that produces the correct type.

If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments. If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, an error is reported and default conversions occur.

No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

The order of evaluation of the function designator, arguments, and subexpressions within the arguments is unspecified, but there is a sequence point before the actual call.

Recursive function calls are permitted, both directly and indirectly through any chain of other functions.

Examples:

In the function call

```
        (*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` may be called in any order. All side effects are completed before the function pointed to by `pf[f1()]` is entered.

Structure and Union Members

Constraints:

The first operand of the `.` operator has a qualified or unqualified structure or union type, and the second operand names a member of that type.

The first operand of the `->` operator has type "pointer to qualified or unqualified structure" or "pointer to qualified or unqualified union", and the second operand names a member of the type pointed to.

Semantics:

A postfix expression followed by a dot (`.`) and an identifier designates a member of a structure or union object. The value is that of the named member and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

A postfix expression followed by an arrow (`->`) and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.[6]If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is nonportable.[7]One special guarantee is made in order to simplify the use of unions. If a union contains several structures that share a common initial sequence, and if the union object currently contains one of these structures, the common initial part of any of them may be inspected. Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

Examples:

If `f` is a function returning a structure or union, and `x` is a member of that structure or union, `f().x` is a valid postfix expression but is not an lvalue.

The following is a valid fragment:

```
union {
      struct {
            int       alltypes;
      } n;
      struct {
            int       type;
            int       intnode;
      } ni;
      struct {
            int       type;
            double    doublenode;
      } nf;
} u;
/*...*/
u.nf.type = 1;
u.nf.doublenode = 3.14;
/*...*/
if (u.n.alltypes == 1)
      /*...*/ sin(u.nf.doublenode) /*...*/
```

[6] If `&E` is a valid pointer expression (where `&` is the "address-of" operator, which generates a pointer to its operand) the expression `(&E)->MOS` is the same as `E.MOS`.

[7] The "byte orders" for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately-sized array of character type).

Postfix Increment and Decrement Operators

Constraints:

The operand of the postfix increment or decrement operator has qualified or unqualified scalar type and is a modifiable lvalue.

Semantics:

The result of the postfix ++ operator is the value of the operand. After the result is obtained, the value of the operand is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment later in this section for information on constraints, types, conversions, and the effects of operations on pointers. The side effect of updating the stored value of the operand occurs between the previous and the next sequence point.

The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

Unary Operators

Syntax:

```
unary-expression:
        postfix-expression
        ++ unary-expression
        -- unary-expression
        unary-operator cast-expression
        sizeof unary-expression
        sizeof ( type-name )

unary-operator: one of
        &  *  +  -  ~  !
```

Prefix Increment and Decrement Operators

Constraints:

The operand of the prefix increment or decrement operator has qualified or unqualified scalar type and is a modifiable lvalue.

Semantics:

The value of the operand of the prefix ++ operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`. See the discussions of additive operators and compound assignment later in this section for information on constraints, types, side effects, conversions, and the effects of operations on pointers.

The prefix -- operator is analogous to the prefix ++ operator, except that the value of the operand is decremented.

Address and Indirection Operators

Constraints:

The operand of the unary `&` operator is either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

The operand of the unary `*` operator has pointer type.

Semantics:

The result of the unary `&` (address-of) operator is a pointer to the object or function designated by its operand. If the operand has type "*type*", the result has type "pointer to *type*".

The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator cannot be predicted.[8]

Unary Arithmetic Operators

Constraints:

The operand of the unary `+` or `-` operator has arithmetic type; of the `~` operator, integral type; and of the `!` operator, scalar type.

Semantics:

The result of the unary `+` operator is the value of its operand. The integral promotion is performed on the operand, and the result has the promoted type.

The result of the unary `-` operator is the negative of its operand. The integral promotion is performed on the operand, and the result has the promoted type.

The result of the `~` operator is the bitwise complement of its operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set).

[8] It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `*&E` is a function designator or an lvalue equal to `E`.

If `*P` is an lvalue and `T` is the name of an object pointer type, the cast expression `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object that has automatic storage duration when execution of the block with which the object is associated has terminated.

The integral promotion is performed on the operand, and the result has the promoted type. The expression `~E` is equivalent to `(ULONG_MAX-E)` if `E` is promoted to type `unsigned long`, and to `(UINT_MAX-E)` if `E` is promoted to type `unsigned int`. (The constants `ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>`.)

The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, or 1 if the value of its operand compares equal to 0. The result has type `int`. The expression `!E` is equivalent to `(0==E)`.

`sizeof` Operator

Constraints:

The `sizeof` operator may not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an lvalue that designates a bit-field object.

Semantics:

The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand, which is not itself evaluated. The result is an integer constant.

When applied to an operand that has type `char`, `unsigned char`, or `signed char`, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[9]When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

The value of the result depends upon the argument type, and its type (an unsigned integral type) is `size_t` defined in the `<stddef.h>` header.

Examples:

A principal use of the `sizeof` operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to `void`. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The `alloc` function ensures that its return value is aligned suitably for conversion to a pointer to `double`.

Another use of the `sizeof` operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

[9] When applied to a parameter declared to have array (or function) type, the `sizeof` operator yields the size of the pointer obtained by converting as in Section 3, Data Conversion; see Section 7, External Data Definitions.

Cast Operators

Syntax:

cast-expression:
  unary-expression
  (*type-name*) *cast-expression*

Constraints:

Unless it specifies void type, the type name specifies qualified or unqualified scalar type and the operand has scalar type.

Semantics:

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast.*[10] A cast that specifies no conversion has no effect on the type or value of an expression.

Conversions that involve pointers (other than as permitted by the constraints in this section) are specified by means of an explicit cast; they have *CP-6*specific aspects, as follows:

- A pointer may be converted to an integral type. The size of integer required is an `int` and the value is zero if the pointer is the `NULL` pointer; otherwise, it is the pointer value exclusive `OR`'ed with octal `06014`.
- An arbitrary integer may be converted to a pointer. The result is a `NULL` pointer if the integer is zero; otherwise, it is the integer value exclusive `OR`'ed with octal `06014`.
- A pointer to an object or incomplete type may be converted to a pointer to a different object type or a different incomplete type. The resulting pointer might not be valid if it is improperly aligned for the type pointed to. It is guaranteed, however, that a pointer to an object of a given alignment may be converted to a pointer to an object of the same alignment or a less strict alignment and back again; the result compares equal to the original pointer (an object that has character type has the least strict alignment).
- A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result compares equal to the original pointer. A converted pointer must not be used to call a function that has a type that is not compatible with the type of the called function.

[10] A cast does not yield an lvalue. Thus a cast to a qualified type has the same effect as a cast to the unqualified version of the type.

Multiplicative Operators

Syntax:

> *multiplicative-expression:*
> > *cast-expression*
> > *multiplicative-expression* * *cast-expression*
> > *multiplicative-expression* / *cast-expression*
> > *multiplicative-expression* % *cast-expression*

Constraints:

Each of the operands has arithmetic type. The operands of the % operator have integral type.

Semantics:

The usual arithmetic conversions are performed on the operands.

The result of the binary * operator is the product of the operands.

The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, the value of the second operand may not be zero, or the `SIGFPE` signal is raised (see Section 14).

When integers are divided and the division is inexact, and both operands are positive, the result of the / operator is the largest integer less than the algebraic quotient; the result of the % operator is positive. If either operand is negative, the result of the / operator is the largest integer less than or equal to the algebraic quotient. The sign of the result of the `a%b` operator is negative if `a` is negative. If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` equals `a`.

Additive Operators

Syntax:

> ***additive-expression:***
> > ***multiplicative-expression***
> > ***additive-expression* + *multiplicative-expression***
> > ***additive-expression* - *multiplicative-expression***

Constraints:

For addition, either both operands have arithmetic type, or one operand is a pointer to an object type and the other has integral type. (Incrementing is equivalent to adding 1.)

For subtraction, one of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to qualified or unqualified versions of compatible object types.
- The left operand is a pointer to an object type, and the right operand has integral type. (Decrementing is equivalent to subtracting 1.)

Semantics:

If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

The result of the binary + operator is the sum of the operands.

The result of the binary - operator is the difference resulting from the subtraction of the second operand from the first.

For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression `P` points to the *i*-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value *n*) point to, respectively, the *i+n*-th and *i-n*-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation should not produce an overflow. Unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object, the result should not be used as an operand of the unary * operator.

When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The size of the result is an `int`, and its type is `ptrdiff_t` defined in the `<stddef.h>` header. Both pointers should point to elements of the same array object, or one past the last element of the same array object.

Bitwise Shift Operators

Syntax:

shift-expression:
  additive-expression
  shift-expression >> *additive-expression*
  shift-expression << *additive-expression*

Constraints:

Each of the operands has integral type.

Semantics:

The integral promotions are performed on each of the operands. The type of the result is that of the promoted left operand. The value of the right operand should not be negative or greater than or equal to the width in bits of the promoted left operand.

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is `E1` multiplied by the quantity 2 raised to the power `E2`, reduced modulo `ULONG_MAX+1` if `E1` has type `unsigned long`, `UINT_MAX+1` otherwise. (The constants `ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>`.)

The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a non-negative value, the value of the result is the integral part of the quotient of `E1` divided by the quantity 2 raised to the power `E2`. If `E1` has a signed type and a negative value, vacated bit positions of `E1` are filled with 1's.

Relational Operators

Syntax:

relational-expression:
  shift-expression
  relational-expression < *shift-expression*
  relational-expression > *shift-expression*
  relational-expression <= *shift-expression*
  relational-expression >= *shift-expression*

Constraints:

One of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to qualified or unqualified versions of compatible object types.
- Both operands are pointers to qualified or unqualified versions of compatible incomplete types.

Semantics:

If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare higher than pointers to members declared earlier in the structure. Pointers to array elements with larger subscript values compare higher than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. The objects pointed to should be members of the same aggregate or union object, with the following exception. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares higher than `P`, even though `Q+1` does not point to an element of the array object.

If two pointers to an object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal. If two pointers to an object or incomplete types compare equal, both point to the same object, or both point one past the last element of the same array object.[11]

Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) yields 1 if the specified relation is true and 0 if it is false.[12]The result has type `int`.

Equality Operators

Syntax:

equality-expression:
 relational-expression
 equality-expression `==` *relational-expression*
 equality-expression `!=` *relational-expression*

Constraints:

One of the following must hold:

[11] Invalid pointer operations, such as accesses outside array bounds, should be avoided because of possible adverse effects on subsequent comparisons.

[12] The expression `a<b<c` is not interpreted as in ordinary mathematics. As the syntax indicates, it means `(a<b)<c`; in other words, "if `a` is less than `b` compare 1 to `c`; otherwise compare 0 to `c`".

- Both operands have arithmetic type.
- Both operands are pointers to qualified or unqualified versions of compatible types.
- One operand is a pointer to an object or incomplete type and the other is a qualified or unqualified version of `void`.
- One operand is a pointer and the other is a null pointer constant.

Semantics:

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.[13]Where the operands have types and values suitable for the relational operators, the semantics detailed in Relational Operators, earlier in this section, apply.

If two pointers to object or incomplete types are both null pointers, they compare equal. If two pointers to object or incomplete types compare equal, they both are null pointers, or both point to the same object, or both point one past the last element of the same array object. If two pointers to function types compare equal, either both are null pointers or both point to the same function. If one of the operands is a pointer to an object or incomplete type and the other has type pointer to a qualified or unqualified version of `void`, the pointer to an object or incomplete type is converted to the type of the other operand.

Bitwise AND Operator

Syntax:

AND-expression:
equality-expression
AND-expression **&** *equality-expression*

Constraints:

Each of the operands has integral type.

Semantics:

The usual arithmetic conversions are performed on the operands.

The result of the binary **&** operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

[13] Because of the precedences, "a<b == c<d" is 1 whenever a<b and c<d have the same truth value.

Bitwise Exclusive OR Operator

Syntax:

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression ^ *AND-expression*

Constraints:

Each of the operands has integral type.

Semantics:

The usual arithmetic conversions are performed on the operands.

The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

Bitwise Inclusive OR Operator

Syntax:

inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression | *exclusive-OR-expression*

Constraints:

Each of the operands has integral type.

Semantics:

The usual arithmetic conversions are performed on the operands.

The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

Logical AND Operator

Syntax:

> *logical-AND-expression:*
> > *inclusive-OR-expression*
> > *logical-AND-expression* **&&** *inclusive-OR-expression*

Constraints:

Each of the operands has scalar type.

Semantics:

The **&&** operator yields 1 if both of its operands compare unequal to 0; otherwise it yields 0. The result has type `int`.

Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.

Logical OR Operator

Syntax:

> *logical-OR-expression:*
> > *logical-AND-expression*
> > *logical-OR-expression* || *logical-AND-expression*

Constraints:

Each of the operands has scalar type.

Semantics:

The || operator yields 1 if either of its operands compare unequal to 0; otherwise it yields 0. The result has type `int`.

Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares unequal to 0, the second operand is not evaluated.

Conditional Operator

Syntax:

conditional-expression:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*

Constraints:

The first operand has scalar type.

One of the following must hold for the second and third operands:

- Both operands have arithmetic type.
- Both operands have compatible structure or union types.
- Both operands have void type.
- Both operands are pointers to qualified or unqualified versions of compatible types.
- One operand is a pointer and the other is a null pointer constant.
- One operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`.

Semantics:

The first operand is evaluated; there is a sequence point after its evaluation. The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the value of the second or third operand (whichever is evaluated) is the result.[14]

If both the second and third operands have arithmetic type, the usual arithmetic conversions are performed to bring them to a common type, and the result has that type. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands. Furthermore, if both operands are pointers to compatible types or differently qualified versions of a compatible type, the result has the composite type; if one operand is a null pointer constant, the result has the type of the other operand. Otherwise, one operand is a pointer to `void` or a qualified version of `void`, in which case the other operand is converted to type pointer to `void`, and the result has that type.

[14] A conditional expression does not yield an lvalue.

Assignment Operators

Syntax:

assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression
assignment-operator: one of

```
= *= /= %= += -= <<= >>= &= ^= |=
```

Constraints:

An assignment operator has a modifiable lvalue as its left operand.

Semantics:

An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand occurs between the previous and the next sequence point.

The order of evaluation of the operands is unspecified.

Simple Assignment

Constraints:

One of the following must hold:[15]

- The left operand has qualified or unqualified arithmetic type, and the right has arithmetic type.
- The left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right.
- Both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of `void`; the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- The left operand is a pointer, and the right is a null pointer constant.

[15] The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in Section 3, Data Conversion) that changes lvalues to "the value of the expression", which removes any type qualifiers from the type category of the expression.

Semantics:

In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap must be exact and the two objects must have qualified or unqualified versions of a compatible type.

Examples:

In the program fragment

```
int f(void);
char c;
/*...*/
/*...*/ ((c = f()) == -1) /*...*/
```

the `int` value returned by the function may be truncated when stored in the `char` and then converted back to `int` width prior to the comparison. Since "plain" `char` has the same range of values as `unsigned char` (and `char` is narrower than `int`), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, the variable `c` should be declared as `int`.

Compound Assignment

Constraints:

For the operators += and -= only, either the left operand is a pointer to an object type and the right has integral type, or the left operand has qualified or unqualified arithmetic type and the right has arithmetic type.

For the other operators, each operand has arithmetic type consistent with those allowed by the corresponding binary operator.

Semantics:

A *compound assignment* of the form `E1` *op* `=E2` differs from the simple assignment expression `E1=E1` *op* `(E2)` only in that the lvalue `E1` is evaluated only once.

Comma Operator

Syntax:

expression:
assignment-expression
expression , assignment-expression

Semantics:

The left operand of a comma operator is evaluated as a void expression; there is a sequence point after its evaluation. The right operand is then evaluated; the result has its type and value.[16]

Examples:

As indicated by the syntax, in contexts where a comma is a punctuator (in lists of arguments to functions and lists of initializers), the comma operator as described in this section cannot appear. On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Constant Expressions

Syntax:

constant-expression:
conditional-expression

Description:

A *constant expression* will be evaluated during translation rather than run time, and accordingly may be used in any place that a constant may be used.

Constraints:

Constant expressions do not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within the operand of a `sizeof` operator.[17]

Each constant expression evaluates to a constant that is in the range of representable values for its type.

[16] A comma operator does not yield an lvalue.

[17] The operand of a `sizeof` operator (described earlier in this section) is not evaluated, and thus any operator described in this section may be used.

Semantics:

An expression that evaluates to a constant is required in several contexts.[18]

An *integral constant expression* has integral type and can only have operands that are integer constants, enumeration constants, character constants, `sizeof` expressions, and floating constants that are the immediate operands of casts. Cast operators in an integral constant expression can only convert arithmetic types to integral types, except as part of an operand to the `sizeof` operator.

More latitude is permitted for constant expressions in initializers. Such a constant expression evaluates to one of the following:

- An arithmetic constant expression.
- A null pointer constant.
- An address constant.
- An address constant for an object type plus or minus an integral constant expression.

An *arithmetic constant expression* has arithmetic type and can only have operands that are integer constants, floating constants, enumeration constants, character constants, and `sizeof` expressions. Cast operators in an arithmetic constant expression can only convert arithmetic types to arithmetic types, except as part of an operand to the `sizeof` operator.

An *address constant* is a pointer to an lvalue designating an object of static storage duration, or to a function designator; it is created explicitly, using the unary `&` operator, or implicitly, by the use of an expression of array or function type. The array-subscript (`[]`) and member-access (`.` and `->`) operators; the address (`&`) and indirection (`*`) unary operators; and pointer casts may be used in the creation an address constant, but the value of an object may not be accessed by use of these operators.

The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.

[18] An integral constant expression must be used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a case constant. Further constraints that apply to the integral constant expressions used in conditional-inclusion preprocessing directives are discussed in Section 8, Preprocessing Directives.

Section 5

Data Declarations

This section describes the data declarations of the C language: storage-class specifiers, type specifiers, type qualifiers, declarators, type names, type definitions, and initializers. For each declaration, syntax, constraints, semantics, and examples are presented where appropriate.

Syntax:

> *declaration:*
> > *declaration-specifiers init-declarator-list*$_{opt}$;
>
> *declaration-specifiers:*
> > *storage-class-specifier declaration-specifiers*$_{opt}$
> > *type-specifier declaration-specifiers*$_{opt}$
> > *type-qualifier declaration-specifiers*$_{opt}$
>
> *init-declarator-list:*
> > *init-declarator*
> > *init-declarator-list* , *init-declarator*
>
> *init-declarator:*
> > *declarator*
> > *declarator* = *initializer*

Constraints:

A declaration must declare at least a declarator, a tag, or the members of an enumeration.

If an identifier has no linkage, there may be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified later in this section.

All declarations in the same scope that refer to the same object or function must specify compatible types.

Semantics:

A *declaration* specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a *definition.*[1]

The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, an initializer, or both. The declarators contain the identifiers (if any) being declared.

If an identifier for an object is declared with no linkage, the type for the object must be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer.

Storage-Class Specifiers

Syntax:

storage-class-specifier:
```
        typedef
        extern
        static
        auto
        register
```

Constraints:

At most one storage-class specifier may be given in the declaration specifiers in a declaration.

Semantics:

The `typedef` specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in Type Definitions, later in this section. The meanings of the various linkages and storage durations are discussed in Section 2, Lexical Elements.

A declaration of an identifier for an object with storage-class specifier `register` suggests that access to the object be as fast as possible.[2]

The declaration of an identifier for a function that has block scope can have no explicit storage-class specifier other than `extern`.

[1] Function definitions have a different syntax, described in Section 7, External Data Definitions.

[2] *CP-6* C treats any `register` declaration simply as an `auto` declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier `register` may not be computed, either explicitly (by use of the unary `&` operator as discussed in Section 4, Expressions) or implicitly (by converting an array name to a pointer as discussed in Section 3, Data Conversion). Thus the only operator that can be applied to an array declared with storage-class specifier `register` is `sizeof`.

Type Specifiers

Syntax:

```
type-specifier:
        void
        char
        short
        int
        long
        float
        double
        signed
        unsigned
        struct-or-union-specifier
        enum-specifier
        typedef-name
```

Constraints:

Each list of type specifiers must be one of the following sets (delimited by commas, when there is more than one set on a line); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers:

- `void`
- `char`
- `signed char`
- `unsigned char`
- `short`, `signed short`, `short int`, or `signed short int`
- `unsigned short` or `unsigned short int`
- `int`, `signed`, `signed int`, or no type specifiers
- `unsigned` or `unsigned int`
- `long`, `signed long`, `long int`, or `signed long int`
- `unsigned long` or `unsigned long int`
- `float`
- `double`
- `long double`
- struct-or-union specifier
- enum-specifier
- typedef-name

Semantics:

Specifiers for structures, unions, and enumerations are discussed later in this section, as are declarations of typedef names. The characteristics of the other types are discussed in Section 2, Lexical Elements.

Each of the above comma-separated sets designates the same type. In some C compilers, the field type `signed int` (or `signed`) may differ from `int` (or no type specifiers).

Structure and Union Specifiers

Syntax:

struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
    struct-or-union identifier

struct-or-union:
    `struct`
    `union`

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
    type-specifier specifier-qualifier-list$_{opt}$
    type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
    struct-declarator
    struct-declarator-list , *struct-declarator*

struct-declarator:
    declarator
    declarator$_{opt}$: *constant-expression*

Constraints:

A structure or union may not contain a member with incomplete or function type. Hence it may not contain an instance of itself (but may contain a pointer to an instance of itself).

The expression that specifies the width of a bit-field is an integral constant expression that has non-negative value that may not exceed the number of bits in an ordinary object of compatible type. If the value is zero, the declaration may not have a declarator.

Semantics:

As discussed in Section 2, Lexical Elements, a structure is a type consisting of a sequence of named members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of named members, whose storage overlaps.

Structure and union specifiers have the same form.

The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within an object unit. The struct-declaration-list is a sequence of declarations for the members of the structure or union. The struct-declaration-list should contain at least one named member. The type is incomplete until after the } that terminates the list.

A member of a structure or union may have any object type. In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[3]its width is preceded by a colon.

A bit-field must have type `int`, `unsigned int`, or `signed int`. A "plain" `int` bit-field is treated as a "`signed int`". A bit-field is interpreted as an integral type consisting of the specified number of bits.

CP-6 C allocates bit fields in units of 36-bit words. If enough space remains, a bit-field that immediately follows another bit-field in a structure is packed into adjacent bits of the same unit. If insufficient space remains, a bit-field that does not fit is put into the next unit. The order of allocation of bit-fields within a unit is high-order to low-order. The addressable storage unit is word-aligned.

A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[4]As a special case of this, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

Each non-bit-field member of a structure or union object is aligned according to its type.

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may therefore be unnamed holes within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

There may also be unnamed padding at the end of a structure or union, as necessary to achieve the appropriate alignment were the structure or union to be an element of an array.

[3] The unary & (address-of) operator may not be applied to a bit-field object; thus there are no pointers to or arrays of bit-field objects.

[4] An unnamed bit-field structure member is useful for padding to conform to externally-imposed layouts.

Enumeration Specifiers

Syntax:

enum-specifier:
  enum *identifier$_{opt}$* { *enumerator-list* }
  enum *identifier*

enumerator-list:
  enumerator
  enumerator-list , *enumerator*

enumerator:
  enumeration-constant
  enumeration-constant = *constant-expression*

Constraints:

The expression that defines the value of an enumeration constant must be an integral constant expression that has a value representable as an `int`.

Semantics:

The identifiers in an enumerator list are declared as constants that have type `int` and may appear wherever such are permitted.[5] An enumerator with = defines its enumeration constant as the value of the constant expression. If the first enumerator has no =, the value of its enumeration constant is 0. Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

Each enumerated type is compatible with type `int`.

Examples:

```
enum hue { chartreuse, burgundy, claret=20, winedark };
/*...*/
enum hue col, *cp;
/*...*/
col = claret;
cp = &col;
/*...*/
/*...*/ (*cp != burgundy) /*...*/
```

makes `hue` the tag of an enumeration, and then declares `col` as an object that has that type and `cp` as a pointer to an object that has that type. The enumerated values are in the set {0, 1, 20, 21}.

[5] Thus, the identifiers of enumeration constants declared in the same scope are all distinct from each other and from other identifiers declared in ordinary declarators.

Tags

Semantics:

A type specifier of the form

> *struct-or-union identifier* { *struct-declaration-list* }

or

> **enum** *identifier* { *enumerator-list* }

declares the identifier to be the *tag* of the structure, union, or enumeration specified by the list. The list defines the *structure content*, *union content*, or *enumeration content.* If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared structure, union, or enumerated type. Subsequent declarations in the same scope must omit the bracketed list.

If a type specifier of the form

> *struct-or-union identifier*

occurs prior to the declaration that defines the content, the structure or union is an incomplete type.[6]It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.[7]If the type is to be completed, another declaration of the tag in the same scope (but not in an enclosed block, which declares a new type known only within that block) defines the content. A declaration of the form

> *struct-or-union identifier* ;

specifies a structure or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

A type specifier of the form

> *struct-or-union* { *struct-declaration-list* }

or

> **enum** { *enumerator-list* }

specifies a new structure, union, or enumerated type, within the object unit, that can only be referred to by the declaration of which it is a part.[8]

[6] A similar construction with **enum** does not exist and is not necessary as there can be no mutual dependencies between the declaration of an enumerated type and any other type.

[7] It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in Section 2, Lexical Elements.) The specification must be complete before such a function is called or defined.

[8] Of course, when the declaration is of a typedef name, subsequent declarations can make use of the typedef name to declare objects having the specified structure, union, or enumerated type.

Examples:

This mechanism allows declaration of a self-referential structure:

```
struct tnode {
        int count;
        struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be an object of the given type and `sp` to be a pointer to an object of the given type. With these declarations, the expression `sp->left` refers to the left `struct tnode` pointer of the object to which `sp` points; the expression `s.right->count` designates the `count` member of the right `struct tnode` pointed to from `s`.

The following alternative formulation uses the `typedef` mechanism:

```
typedef struct tnode TNODE;
struct tnode {
        int count;
        TNODE *left, *right;
};
TNODE s, *sp;
```

To illustrate the use of prior declaration of a tag to specify a pair of mutually-referential structures, the declarations

```
struct s1 { struct s2 *s2p; /*...*/ }; /* D1 */
struct s2 { struct s1 *s1p; /*...*/ }; /* D2 */
```

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to *it*, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the otherwise vacuous declaration

```
struct s2;
```

may be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

Type Qualifiers

Syntax:

```
type-qualifier:
        const
        volatile
```

Constraints:

The same type qualifier may not appear more than once in the same specifier list or qualifier list, either directly or via one or more `typedefs`.

Semantics:

The properties associated with qualified types are meaningful only for expressions that are lvalues.[9]

No attempt should be made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type. Nor should an attempt be made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type.[10]

An object that has volatile-qualified type is assumed to be modified in ways unknown to the compiler or have other unknown side effects.

If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. The specification of a function type should not include any type qualifiers, since they are ignored.[11]

For two qualified types to be compatible, both must have the identically qualified version of a compatible type. The order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

Examples:

An object declared

```
extern const volatile int real_time_clock;
```

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented within this compilation unit.

The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

[9] *CP-6* C often places a `const` object that is not `volatile` in a read-only region of storage.

[10] This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

[11] Both of these can only occur through the use of `typedefs`.

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  /* the object ncs is modifiable */
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; /* array of array of const int */
int *pi;
const int *pci;

ncs = cs;       /* valid */
cs = ncs;       /* violates modifiable lvalue constraint for = */
pi = &ncs.mem;  /* valid */
pi = &cs.mem;   /* violates type constraints for = */
pci = &cs.mem;  /* valid */
pi = a[0];      /* invalid: a[0]  has type "const int *" */
```

Declarators

Syntax:

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  (*declarator*)
  direct-declarator [*constant-expression*$_{opt}$]
  direct-declarator (*parameter-type-list*)
  direct-declarator (*identifier-list*$_{opt}$)

pointer:
  * *type-qualifier-list*$_{opt}$
  * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
  type-qualifier
  type-qualifier-list type-qualifier

parameter-type-list:
  parameter-list
  parameter-list , . . .

parameter-list:
  parameter-declaration
  parameter-list , *parameter-declaration*

parameter-declaration:
  declaration-specifiers declarator

declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    identifier-list , *identifier*

Semantics:

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

In the following subsections, consider a declaration

```
T D1
```

where T contains the declaration specifiers that specify a type *T* (such as `int`) and D1 is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

If, in the declaration "T D1", D1 has the form

identifier

then the type specified for *ident* is *T*.

If, in the declaration "T D1", D1 has the form

```
( D )
```

then *ident* has the type specified by the declaration "T D". Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complex declarators may be altered by parentheses.

Pointer Declarators

Semantics:

If, in the declaration "T D1", D1 has the form

* *type-qualifier-list*$_{opt}$ D

and the type specified for *ident* in the declaration "T D" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*". For each type qualifier in the list, *ident* is a so-qualified pointer.

For two pointer types to be compatible, both must be identically qualified and both must be pointers to compatible types.

Examples:

The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value":

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of an object pointed to by `ptr_to_constant` may not be modified through that pointer, but `ptr_to_constant` itself may be changed to point to another object. Similarly, the contents of the `int` pointed to by `constant_ptr` may be modified, but `constant_ptr` itself always points to the same location.

The declaration of the constant pointer `constant_ptr` may be clarified by including a definition for the type "pointer to `int`":

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares `constant_ptr` as an object that has type "const-qualified pointer to `int`".

Array Declarators

Constraints:

The expression delimited by `[` and `]` (which specifies the size of an array) is an integral constant expression that has a value greater than zero.

Semantics:

If, in the declaration "`T D1`", `D1` has the form

`D[`*constant-expression*$_{opt}$`]`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[12] If the size is not present, the array type is an incomplete type.

For two array types to be compatible, both must have compatible element types, and if both size specifiers are present, they must have the same value.

Examples:

```
float fa[11], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers.

Note the distinction between the declarations:

```
extern int *x;
extern int y[];
```

The first declares `x` to be a pointer to `int`; the second declares `y` to be an array of `int` of unspecified size (an incomplete type), the storage for which is defined elsewhere.

[12] When several "array of" specifications are adjacent, a multidimensional array is declared.

Function Declarators (including Prototypes)

Constraints:

A function declarator may not specify a return type that is a function type or an array type.

The only storage-class specifier that may occur in a parameter declaration is `register`.

An *identifier list* in a function declarator that is not part of a function definition must be empty.

Semantics:

If, in the declaration "`T D1`", `D1` has the form

`D(`*parameter-type-list*`)`

or

`D(`*identifier-list*$_{opt}$`)`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning *T*".

A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[13]The special case of `void` as the only item in the list specifies that the function has no parameters.

In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a function definition specifies that the function has no parameters. The empty list in a function declarator that is not part of a function definition specifies that no information about the number or types of the parameters is supplied.

For two function types to be compatible, both must specify compatible return types.[14]The parameter type lists, if both are present, must agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters must have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the

[13] The macros defined in the `<stdarg.h>` header (Section 15) may be used to access arguments that correspond to the ellipsis.

[14] If both function types are "old style", parameter types are not compared.

parameter list cannot have an ellipsis terminator, and the type of each parameter must be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both must agree in the number of parameters, and the type of each prototype parameter must be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in Section 7, External Data Definitions. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

Examples:

The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function `f` with no parameters returning an `int`, a function `fip` with no parameter specification returning a pointer to an `int`, and a pointer `pfi` to a function with no parameter specification returning an `int`. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the pointer result to yield an `int`. In the declarator `(*pfi)()`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator which is then used to call the function; it returns an `int`.

If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions `f` and `fip` have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer `pfi` has block scope and no linkage.

Here are two more intricate examples:

```
int (*apfi[3])(int *x, int *y);
```

declares an array `apfi` of three pointers to functions returning `int`. Each of these functions has two parameters that are pointers to `int`. The identifiers `x` and `y` are declared for descriptive purposes only and go out of scope at the end of the declaration of `apfi`. The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function `fpfi` that returns a pointer to a function returning an `int`. The function `fpfi` has two parameters: a pointer to a function returning an `int` (with one parameter of type `long`), and an `int`. The pointer returned by `fpfi` points to a function that has one `int` parameter and accepts zero or more additional arguments of any type.

Type Names

Syntax:

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ direct-abstract-declarator

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression$_{opt}$*]
direct-abstract-declarator$_{opt}$ (*parameter-type-list$_{opt}$*)

Semantics:

In several contexts it is desired to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[15]

Examples:

The constructions

```
(a)    int
(b)    int *
(c)    int *[3]
(d)    int (*)[3]
(e)    int *()
(f)    int (*)(void)
(g)    int (*const [])(unsigned int, ...)
```

name respectively the types (a) `int`, (b) pointer to `int`, (c) array of three pointers to `int`, (d) pointer to an array of three `int`s, (e) function with no parameter specification returning a pointer to `int`, (f) pointer to function with no parameters returning an `int`, and (g) array of an unspecified number of constant pointers to functions, each with one parameter that has type `unsigned int` and an unspecified number of other parameters, returning an `int`.

[15] As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification", rather than redundant parentheses around the omitted identifier.

Type Definitions

Syntax:

typedef-name:
identifier

Semantics:

In a declaration whose storage-class specifier is `typedef`, each declarator defines an identifier to be a typedef name that specifies the type specified for the identifier in the way described at the beginning of this section. A `typedef` declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in `T` (known as *T*), and the identifier in `D` has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of `D`. A typedef name shares the same name space as other identifiers declared in ordinary declarators. If the identifier is redeclared in an inner scope or is declared as a member of a structure or union in the same or an inner scope, the type specifiers cannot be omitted in the inner declaration.

Examples:

After

```
typedef int MILES, KLICKSP();
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
complex x;
complex z, *zp;
```

are all valid declarations. The type of `distance` is `int`, that of `metricp` is "pointer to function with no parameter specification returning `int`", and that of `x` and `z` is the specified structure; `zp` is a pointer to such a structure. The object `distance` has a type compatible with any other `int` object.

After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type `t1` and the type pointed to by `tp1` are compatible. Type `t1` is also compatible with type `struct s1`, but is not compatible with the types `struct s2`, `t2`, the type pointed to by `tp2`, and `int`.

The following obscure constructions:

```
typedef signed int t;
typedef int plain;
struct tag {
      unsigned t:4;
      const t:5;
      plain r:5;
};
```

declare a typedef name `t` with type `signed int`, a typedef name `plain` with type `int`, and a structure with three bit-field members: one named `t` that contains values in the range [0,15], an unnamed const-qualified bit-field which (if it could be accessed) would contain values in at least the range [-15,+15], and one named `r` that contains values in the range [-16,+15]. The first two bit-field declarations differ in that `unsigned` is a type specifier (which forces `t` to be the name of a structure member), while `const` is a type qualifier (which modifies `t` which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function `f` is declared with type "function returning `signed int` with one unnamed parameter with type pointer to function returning `signed int` with one unnamed parameter with type `signed int`", and an identifier `t` with type `long`.

On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the `signal` function specify exactly the same type, the first without making use of any typedef names:

```
typedef void fv(int);
typedef void (*pfv)(int);
void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Initialization

Syntax:

initializer:
  assignment-expression
  { *initializer-list* }
  { *initializer-list* , }

initializer-list:
  initializer
  initializer-list , *initializer*

Constraints:

There may be no more initializers in an initializer list than there are objects to be initialized.

The type of the entity to be initialized is an object type or an array of unknown size.

All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type are constant expressions.

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration may not have an initializer for the identifier.

Semantics:

An initializer specifies the initial value stored in an object.

All unnamed structure or union members are ignored during initialization.

If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant. If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

The initializer for a scalar is a single expression, optionally enclosed in braces. The initial value of the object is that of the expression; the same type constraints and conversions as for simple assignment apply.

A brace-enclosed initializer for a union object initializes the member that appears first in the declaration list of the union type.

The initializer for a structure or union object that has automatic storage duration is either an initializer list as described below, or is a single expression that has compatible structure or union type. In the latter case, the initial value of the object is that of the expression.

The rest of this section deals with initializers for objects that have aggregate or union type.

An array of character type may be initialized by a character string literal, optionally enclosed in braces. Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

An array with element type compatible with **wchar_t** may be initialized by a wide string literal, optionally enclosed in braces. Successive codes of the wide string literal (including the terminating zero-valued code if there is room or if the array is of unknown size) initialize the elements of the array.

Otherwise, the initializer for an object that has aggregate type is a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order. The initializer for an object that has union type is a brace-enclosed initializer for the first member of the union.

If the aggregate contains members that are aggregates or unions, or if the first member of a union is an aggregate or union, the rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the members of the subaggregate or the first member of the contained union. Otherwise, only enough initializers from the list are taken to account for the members of the subaggregate or the first member of the contained union. Any remaining initializers are left to initialize the next member of the aggregate of which the current subaggregate or contained union is a part.

If there are fewer initializers in a brace-enclosed list than there are members of an aggregate, the remainder of the aggregate is initialized implicitly the same as objects that have static storage duration.

If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements. At the end of its initializer list, the array no longer has incomplete type.

Examples:

The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

```
float y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of `y` (the array object `y[0]`), namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, so `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by:

```
float y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float z[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `z` as specified and initializes the rest with zeros.

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed but legal initialization. It defines an array with two element structures: `w[0].a[0]` is 1 and `w[1].a[0]` is 2; all the other elements are zero.

The declaration

```
short q[4][3][2] = {
        { 1 },
        { 2, 3 },
        { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0]` does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for `q[1][0]` and `q[2][0]` do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
        1, 0, 0, 0, 0, 0,
        2, 3, 0, 0, 0, 0,
        4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
        {
                { 1 },
        },
        {
                { 2, 3 },
        },
        {
                { 4, 5 },
                { 6 },
        }
};
```

in a fully-bracketed form.

Note that the fully-bracketed and minimally-bracketed forms of initialization are, in general, less likely to cause confusion.

Finally, the declaration

```
char s[] = "abc", t[3] = "abc";
```

defines "plain" `char` array objects `s` and `t` whose elements are initialized with character string literals. This declaration is identical to

```
    char s[] = { 'a', 'b', 'c', '\0' },
         t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
    char *p = "abc";
```

defines `p` with type "pointer to `char`" that is initialized to point to an object with type "array of `char`" with length 4 whose elements are initialized with a character string literal. If any attempt is made to use `p` to modify the contents of the array, the `SIGSEGV` signal is raised. The compilation option `STRING=WRITEABLE` allows strings to be overwritten.

Section 6

Statements

This section describes the statement types of the C language: labeled, compound, expression and null, selection, iteration, and jump. For each statement type or statement, syntax, constraints, semantics, and examples are presented where appropriate.

Syntax:

> *statement:*
>> *labeled-statement*
>> *compound-statement*
>> *expression-statement*
>> *selection-statement*
>> *iteration-statement*
>> *jump-statement*

Semantics:

A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression: an initializer, the expression in an expression statement, the controlling expression of a selection statement (`if` or `switch`), the controlling expression of a `while` or `do` statement, each of the three (optional) expressions of a `for` statement, and the (optional) expression in a `return` statement.

Labeled Statements

Syntax:

> *labeled-statement:*
>> *identifier* : *statement*
>> **`case`** *constant-expression* : *statement*
>> **`default`** : *statement*

Constraints:

A `case` or `default` label appears only in a `switch` statement. Further constraints on such labels are discussed under the `switch` statement, later in this section.

Semantics:

Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Compound Statement or Block

Syntax:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

Semantics:

A *compound statement* (also called a *block*) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations.initializations (as discussed in Section 2, Lexical Elements). The initializers of objects that have automatic storage duration are evaluated, and the values are stored in the objects in the order in which their declarators appear in the object unit.

Expression and Null Statements

Syntax:

expression-statement:
expression$_{opt}$;

Semantics:

The expression in an expression statement is evaluated as a void expression for its side effects.[1]

A *null statement* (consisting of just a semicolon) performs no operations.

Examples:

If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/*...*/
(void)p(0);
```

[1] Such as assignments, and function calls which have side effects.

In the program fragment

```
char *s;
/*...*/
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

A null statement may also be used to carry a label just before the closing } of a compound statement, as shown in the following:

```
while (loop1) {
        /*...*/
        while (loop2) {
                /*...*/
                if (want_out)
                        goto end_loop1;
                /*...*/
        }
        /*...*/
end_loop1: ;
}
```

Selection Statements

Syntax:

selection-statement:
  `if (` *expression* `)` *statement*
  `if (` *expression* `)` *statement* `else` *statement*
  `switch (` *expression* `)` *statement*

Semantics:

A selection statement selects among a set of statements depending on the value of a controlling expression.

`if` Statement

Constraints:

The controlling expression of an `if` statement has scalar type.

Semantics:

In both forms, the first substatement is executed if the expression is not equal to 0. In the `else` form, the second substatement is executed if the expression is equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

An `else` is associated with the lexically immediately preceding `else`-less `if` that is in the same block (but not in an enclosed block).

switch Statement

Constraints:

The controlling expression of a **switch** statement has integral type. The expression of each **case** label is an integral constant expression. No two of the **case** constant expressions in the same **switch** statement may have the same value after conversion. There may be at most one **default** label in a **switch** statement. (Any enclosed **switch** statement may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.)

Semantics:

A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement.

The integral promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

Examples:

In the artificial program fragment

```
switch (expr)
{
        int i = 4;
        f(i);
case 0:
        i = 17;    /* falls through into default code */
default:
        printf("%d\n", i);
}
```

the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

Iteration Statements

Syntax:

```
iteration-statement:
        while ( expression ) statement
        do statement while ( expression ) ;
        for ( expression_opt ; expression_opt ; expression_opt ) statement
```

Constraints:

The controlling expression of an iteration statement has scalar type.

Semantics:

An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0.

`while` Statement

The evaluation of the controlling expression takes place before each execution of the loop body.

`do` Statement

The evaluation of the controlling expression takes place after each execution of the loop body.

`for` Statement

Except for the behavior of a `continue` statement in the loop body, the statement

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

and the sequence of statements

```
expression-1 ;
while (expression-2) {
        statement
        expression-3 ;
}
```

are equivalent.[2]

Both *expression-1* and *expression-3* may be omitted. Each is evaluated as a void expression. An omitted *expression-2* is replaced by a nonzero constant.

Jump Statements

Syntax:

```
    jump-statement:
            goto identifier ;
            continue ;
            break ;
            return expression_opt ;
```

Semantics:

A jump statement causes an unconditional jump to another place.

goto Statement

Constraints:

The identifier in a goto statement names a label located somewhere in the enclosing function.

Semantics:

A goto statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

Examples:

It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation must be at the head of the loop (to allow it to be reached by `continue` statements, for example).

[2] Thus *expression-1* specifies initialization for the loop. *expression-2*, the controlling expression, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0. *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

The code for these assumptions is as follows:

```
/*...*/
goto first_time;
for (;;) {
        /* determine next operation */
        /*...*/
        if (need to reinitialize) {
                /* reinitialize-only code */
                /*...*/
        first_time:
                /* general initialization code */
                /*...*/
                continue;
        }
        /* handle other operations */
        /*...*/
}
```

continue Statement

Constraints:

A **continue** statement appears only in or as a loop body.

Semantics:

A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/*...*/) {     do {                    for (/*...*/) {
  /*...*/               /*...*/                 /*...*/
  continue;             continue;               continue;
  /*...*/               /*...*/                 /*...*/
contin: ;             contin: ;               contin: ;
}                     } while (/*...*/);      }
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin;**.[3]

[3] Following the contin: label is a null statement.

`break` Statement

Constraints:

A **`break`** statement appears only in or as a switch body or loop body.

Semantics:

A **`break`** statement terminates execution of the smallest enclosing **`switch`** or iteration statement.

`return` Statement

Constraints:

A **`return`** statement with an expression may not appear in a function whose return type is **`void`**.

Semantics:

A **`return`** statement terminates execution of the current function and returns control to its caller. A function may have any number of **`return`** statements, with and without expressions.

If a **`return`** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type.

A **`return`** statement without an expression should not be executed when the value of the function call is used by the caller. Reaching the } that terminates a function is equivalent to executing a **`return`** statement without an expression.

Section 7

External Data Definitions

This section describes the function definitions and external object definitions of the C language. Syntax, constraints, semantics, and examples are presented where appropriate to the definition types.

Syntax:

object-unit:
  external-declaration
  object-unit external-declaration

external-declaration:
  function-definition
  declaration

Constraints:

The storage-class specifiers `auto` and `register` may not appear in the declaration specifiers in an external declaration.

There may be no more than one external definition for each identifier declared with internal linkage in an object unit. If an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a `sizeof` operator), there must be exactly one external definition for the identifier in the object unit.

Semantics:

As discussed in Section 1 under Compilation Environment, the unit of program text after preprocessing is an object unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in Section 5, Data Declarations, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

An *external definition* is an external declaration that is also a definition of a function or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a `sizeof` operator), somewhere in the entire program there must be exactly one external definition for the identifier.[1]

[1] Thus, if an identifier declared with external linkage is not used in an expression, there need not be an external definition for it.

Function Definitions

Syntax:

function-definition:
declaration-specifiers$_{opt}$ declarator declaration-list$_{opt}$ compound-statement

Constraints:

The identifier declared in a function definition (which is the name of the function) must have a function type, as specified by the declarator portion of the function definition.[2]

The return type of a function must be `void` or an object type other than array.

The storage-class specifier, if any, in the declaration specifiers may be either `extern` or `static`.

If the declarator includes a parameter type list, the declaration of each parameter must include an identifier (except for the special case of a parameter list consisting of a single parameter of type `void`, in which there cannot be an identifier). A declaration list cannot follow.

If the declarator includes an identifier list, each declaration in the declaration list must have at least one declarator, and those declarators must declare only identifiers from the identifier list. An identifier declared as a typedef name cannot be redeclared as a parameter. The declarations in the declaration list cannot contain a storage-class specifier other than `register` and cannot contain initializations.

Semantics:

The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same object unit. If the declarator

[2] The type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);          /* type F is "function of no arguments returning int"  */
F f, g;                       /* f and g both have type compatible with F */
F f { /*...*/ }               /* WRONG: syntax/constraint error */
F g() { /*...*/ }             /* WRONG: declares that g returns a function */
int f(void) { /*...*/ }       /* RIGHT: f has type compatible with F */
int g() { /*...*/ }           /* RIGHT: g has type compatible with F */
F *e(void) { /*...*/ }        /* e returns a pointer to a function */
F *((e))(void) { /*...*/ }    /* same: parentheses irrelevant */
int (*fp)(void);              /* fp points to a function that has type F */
F *Fp;                        /* Fp points to a function that has type F */
```

includes an identifier list, the types of the parameters may be declared in a following declaration list. Any parameter that is not declared has type `int`.

A function that accepts a variable number of arguments cannot be defined without a parameter type list that ends with the ellipsis notation.

On entry to the function, the value of each argument expression is converted to the type of its corresponding parameter, as if by assignment to the parameter. Array expressions and function designators as arguments are converted to pointers before the call. A declaration of a parameter as "array of *type*" is adjusted to "pointer to *type*", and a declaration of a parameter as "function returning *type*" is adjusted to "pointer to function returning *type*", as in Section 3, Data Conversion. The resulting parameter type is an object type.

Each parameter has automatic storage duration. Its identifier is an lvalue.[3]

Examples:

```
extern int max(int a, int b)
{
        return a > b ? a : b;
}
```

Here, `extern` is the storage-class specifier and `int` is the type specifier (each of which may be omitted, as they are the defaults); `max(int a, int b)` is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
        return a > b ? a : b;
}
```

Here, `int a, b;` is the declaration list for the parameters (which may be omitted, as it is the default). The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

To pass one function to another, the following could be used:

```
int f(void);
/*...*/
g(f);
```

Note that `f` must be declared explicitly in the calling function, as its appearance in the expression `g(f)` was not followed by `(`.

[3] A parameter is in effect declared at the head of the compound statement that constitutes the function body, and therefore may not be redeclared in the function body (except in an enclosed block).

Then the definition of g might read:

```
g(int (*funcp)(void))
{
        /*...*/ (*funcp)() /* or funcp() ... */
}
```

or, equivalently,

```
g(int func(void))
{
        /*...*/ func() /* or (*func)() ... */
}
```

External Object Definitions

Semantics:

If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier `static`, constitutes a *tentative definition.* If an object unit contains one or more tentative definitions for an identifier, and the object unit contains no external definition for that identifier, then the behavior is exactly as if the object unit contains a file scope declaration of that identifier, with the composite type as of the end of the object unit and with an initializer equal to 0.

If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type cannot be an incomplete type.

Examples:

```
int i1 = 1;          /* definition, external linkage */
static int i2 = 2; /* definition, internal linkage */
extern int i3 = 3; /* definition, external linkage */
int i4;              /* tentative definition, external linkage */
static int i5;      /* tentative definition, internal linkage */
int i1;              /* valid tentative definition, refers to previous */
int i2;              /* Linkages of Identifiers, in Section 2, linkage
disagreement */ int i3;          /* valid tentative definition, refers to
previous */
int i4;              /* valid tentative definition, refers to previous */
int i5;              /* Linkages of Identifiers, in Section 2, linkage
disagreement */
extern int i1;      /* refers to previous, whose linkage is external */
extern int i2;      /* refers to previous, whose linkage is internal */
extern int i3;      /* refers to previous, whose linkage is external */
extern int i4;      /* refers to previous, whose linkage is external */
extern int i5;      /* refers to previous, whose linkage is internal */
```

Section 8

Preprocessing Directives

This section describes the types and uses of the preprocessing directives of the C language: conditional inclusion, source file inclusion, macro replacement, line control, error directive, pragma directive, null directive, and predefined macro names. The syntax of the directives follows; constraints, semantics, and examples are presented where appropriate for the various directive types.

Syntax:

> *preprocessing-file:*
>   *group*$_{opt}$
>
> *group:*
>   *group-part*
>   *group group-part*
>
> *group-part:*
>   *pp-tokens*$_{opt}$ *new-line*
>   *if-section*
>   *control-line*
>
> *if-section:*
>   *if-group elif-groups*$_{opt}$ *else-group*$_{opt}$ *endif-line*
>
> *if-group:*
>   **`# if`** *constant-expression new-line group*$_{opt}$
>   **`# ifdef`** *identifier new-line group*$_{opt}$
>   **`# ifndef`** *identifier new-line group*$_{opt}$
>
> *elif-groups:*
>   *elif-group*
>   *elif-groups elif-group*
>
> *elif-group:*
>   **`# elif`** *constant-expression new-line group*$_{opt}$

else-group:
  `# else` *new-line* *group*$_{opt}$

endif-line:
  `# endif` *new-line*

control-line:
  `# include` *pp-tokens new-line*
  `# define` *identifier replacement-list new-line*
  `# define` *identifier lparen identifier-list*$_{opt}$) *replacement-list new-line*
  `# undef` *identifier new-line*
  `# line` *pp-tokens new-line*
  `# error` *pp-tokens*$_{opt}$ *new-line*
  `# pragma` *pp-tokens*$_{opt}$ *new-line*
  `#` *new-line*

lparen:
  the left parenthesis character without preceding white space

replacement-list:
  pp-tokens$_{opt}$

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

Description:

A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that is the first character in the source line (optionally after white space) and is ended by the next new-line character.[1]

Constraints:

The only white-space characters that may appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

[1] Thus preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator later in this section).

Semantics:

The compiler can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting object unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

Conditional Inclusion

Constraints:

The expression that controls conditional inclusion must be an integral constant expression except that it may not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[2]and it may contain unary operator expressions of the form

    defined *identifier*

or

    defined (*identifier*)

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), or to 0 if it is not.

Each preprocessing token that remains after all macro replacements have occurred is in the lexical form of a token.

Semantics:

Preprocessing directives of the forms

    # if *constant-expression new-line* $group_{opt}$
    # elif *constant-expression new-line* $group_{opt}$

check whether the controlling constant expression evaluates to nonzero.

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text. The token **defined** is not generated as a result of this replacement process; use of the **defined** unary operator must match one of the two specified forms prior to macro replacement. After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers are replaced with the pp-number 0, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated

[2] Because the controlling constant expression is evaluated during translation phase 4, an identifier either is or is not a macro name.

according to the rules under Constant Expressions, in Section 4, using arithmetic that has the ranges specified under Numerical Limits in Appendix F, except that `int` and `unsigned int` act as if they have the same representation as, respectively, `long` and `unsigned long`. This includes interpreting character constants, which may involve converting escape sequences into execution character set members. The numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression.[3] Also, a single-character character constant is unsigned.

Preprocessing directives of the forms

> `# ifdef` *identifier new-line group*$_{opt}$
> `# ifndef` *identifier new-line group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined` *identifier* and `#if !defined` *identifier*, respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped, and skipped directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals. The rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[4]

Source File Inclusion

Constraints:

A `#include` directive identifies a header or source file that is to be processed.

Semantics:

A preprocessing directive of the form

> `# include` *<h-char-sequence> new-line*

[3] Thus the constant expression in the following `#if` directive and `if` statement is guaranteed to evaluate to the same value in these two contexts:

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

[4] As indicated by the syntax, a preprocessing token may not follow a `#else` or `#endif` directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

searches the directories specified by the command line `SEARCH` option (which always terminates with the current file management account and the `:LIBRARY` account) for a file identified uniquely by the *fid* between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header.

A preprocessing directive of the form

> `# include` "*q-char-sequence*" *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched as if the directive read was

> `# include` <*h-char-sequence*> *new-line*

with the identical contained sequence (including > characters, if any) from the original directive unless the `UNIXSRCH` option was specified. The effect of the `UNIXSRCH` option is to start the search at the search list entry used to locate the file containing the include directive.

A preprocessing directive of the form

> `# include` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements must match one of the two previous forms.[5]

The characters within the `include` source file designation are treated as a *CP-6* file identifier, with the following restrictions:

1. The case of the source file designator is significant.
2. The *filename* portion of the *CP-6 fid* is limited to 31 characters.
3. All "\" characters with the exception of the character preceding the file name are turned into "?" characters.
4. If the file identifier contains a "." character, the identifier is first treated as if it contained an account designation. If the corresponding file exists, it is included.
5. Otherwise, all "." characters in the file identifier are turned into ":" characters, and the search list is used to locate the file.

[5] Note that adjacent string literals are not concatenated into a single string literal (see Translation Phases in Section 1); thus an expansion that results in two string literals is an invalid directive.

A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to a nesting limit of 10 (see Translation Limits, in Appendix F).

Examples:

The most common uses of **#include** preprocessing directives are as in the following:

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

This example illustrates a macro-replaced **#include** directive:

```
#if VERSION == 1
      #define INCFILE  "vers1.h"
#elif VERSION == 2
      #define INCFILE  "vers2.h"
                                  /* and so on */
#else
      #define INCFILE  "versN.h"
#endif
/*...*/
#include INCFILE
```

Macro Replacement

Constraints:

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as a macro without use of lparen (an *object-like* macro) may be redefined by another **#define** preprocessing directive, provided that the second definition is an object-like macro definition and the two replacement lists are identical.

An identifier currently defined as a macro using lparen (a *function-like* macro) may be redefined by another **#define** preprocessing directive, provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

The number of arguments in an invocation of a function-like macro must agree with the number of parameters in the macro definition, and a) preprocessing token must terminate the invocation.

A parameter identifier in a function-like macro is uniquely declared within its scope.

Semantics:

The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form

> `# define` *identifier replacement-list new-line*

defines an object-like macro that causes each subsequent instance of the macro name[6]to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the form

> `# define` *identifier lparen* $identifier\text{-}list_{opt}$) *replacement-list new-line*

defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a "(" as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ")" preprocessing token, skipping intervening matched pairs of left and right parentheses preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. Any argument (before argument substitution) should consist of at least one preprocessing token. Sequences of preprocessing tokens should not be within the list of arguments that would otherwise act as preprocessing directives.

[6] Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see Translation Phases in Section 1), they are never scanned for macro names or parameters.

Argument Substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the object unit; no other preprocessing tokens are available.

# Operator

Constraints:

Each # preprocessing token in the replacement list for a function-like macro is followed by a parameter as the next preprocessing token in the replacement list.

Semantics:

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants. A \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters). The replacement that results is a valid character string literal.

## Operator

Constraints:

A ## preprocessing token may not appear at the beginning or at the end of a replacement list for either form of macro definition.

Semantics:

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

For both object-like and function-like macro invocations, before the replacement list is re-examined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. The resulting token is available for further macro replacement.

Rescanning and Further Replacement

After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned with the rest of the source file's preprocessing tokens for more macro names to replace.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. If any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

Scope of Macro Definitions

A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the object unit.

A preprocessing directive of the form

```
# undef identifier new-line
```

causes the specified identifier to no longer be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

Examples:

The simplest use of this facility is to define a "manifest constant", as in:

```
#define TABSIZE 100
int table[TABSIZE];
```

The example below defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

To illustrate the rules for redefinition and re-examination, the sequence

```
#define x    3
#define f(a) f(x * (a))
#undef  x
#define x    2
#define g    f
#define z    z[0]
#define h    g(~
#define m(a) a(w)
#define w    0,1
#define t(a) a
f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
      (f)^m(m);
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
```

To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)       # s
#define xstr(s)      str(s)
#define debug(s, t)  printf("x" # s "= %d, x" # t "= %s", \
                         x ## s, x ## t)
#define INCFILE(n)   vers ## n  /* from previous #include example */
#define glue(a, b)   a ## b
#define xglue(a, b)  glue(a, b)
#define HIGHLOW      "hello"
#define LOW          LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4')  /* this goes away */
      == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
```

```
    fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
    #include "vers2.h"    (after macro replacement, before file access)
    "hello";
    "hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

And finally, to demonstrate the redefinition rules, the following sequence is valid:

```
    #define OBJ_LIKE      (1-1)
    #define OBJ_LIKE      /* white space */ (1-1) /* other */
    #define FTN_LIKE(a)   ( a )
    #define FTN_LIKE( a )(           /* note the white space */ \
                            a /* other stuff on this line
                              */ )
```

But the following redefinitions are invalid:

```
    #define OBJ_LIKE     (0)     /* different token sequence */
    #define OBJ_LIKE     (1 - 1) /* different white space */
    #define FTN_LIKE(b)  ( a )   /* different parameter usage */
    #define FTN_LIKE(b)  ( b )   /* different parameter spelling */
```

Line Control

Constraints:

The string literal of a #line directive, if present, is a character string literal.

Semantics:

The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 in translation phase 1 (see Translation Phases, in Section 1) while processing the source file to the current token.

A preprocessing directive of the form

`# line` *digit-sequence new-line*

behaves as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence may not specify zero, nor a number greater than 32767.

A preprocessing directive of the form

`# line` *digit-sequence* " *s-char-sequence*$_{opt}$ " *new-line*

sets the line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

A preprocessing directive of the form

> `# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements must match one of the two previous forms and is then processed as appropriate.

The compiler uses the line numbers and source file name for printing error messages, and makes them available to the programmer through the built-in macros `__LINE__` and `__FILE__`.

Error Directive

Semantics:

A preprocessing directive of the form

> `# error` *pp-tokens*$_{opt}$ *new-line*

produces a diagnostic message that includes the specified sequence of preprocessing tokens.

Pragma Directive

Semantics:

A preprocessing directive of the form

> `# pragma` *pp-tokens*$_{opt}$ *new-line*

behaves in a system-dependent manner. Any pragma that is not recognized is ignored.

CP-6 Preprocessing Pragmas

The *CP-6* Preprocessing pragmas provide control over the listing output. Listings may be controlled by turning listing on or off, turning listing of include files on and off, skipping lines, and ejecting the current page.

Syntax:

listing-control
  # pragma LIST ON *new-line*
  # pragma LIST OFF *new-line*

include-file-listing-control
  # pragma LIST INCLUDE_ON *new-line*
  # pragma LIST INCLUDE_OFF *new-line*

listing-space-control
  # pragma LIST SPACE *integer-decimal-constant*$_{opt}$ *new-line*
  # pragma LIST EJECT *new-line*

Constraints:

The LIST keyword is used to signify the start of the listing pragmas. This keyword is immediately followed by a second keyword which selects the required listing state or operation. The SPACE listing operation may optionally be followed by an integer constant count of lines to space.

Semantics:

The listing generator can be in a state where listing output is (or is not) being generated, and in a state where include files will (or will not) be listed. These states are controlled by the following LIST pragma controls:

Control	Meaning
ON	Default. Listing output will be generated.
OFF	Listing output will not be generated for subsequent lines (up to a LIST_ON pragma).
INCLUDE_ON	Listing output will be generated for header files.
INCLUDE_OFF	Default. Listing output will not be generated for header files.

Explicit blank space may be requested in listing output by using the SPACE and EJECT LIST pragmas as follows:

Control	Meaning
EJECT	The next source line will be displayed on the next output page.
SPACE	The next source line will be displayed after 1 (the default) or *integer-decimal-constant* blank lines.

Null Directive

Semantics:

A preprocessing directive of the form

    # *new-line*

has no effect.

Predefined Macro Names

The following macro names are predefined by *CP-6* C:

`__LINE__` Is the line number of the current source line (a decimal constant).

`__FILE__` Is the presumed name of the source file (a character string literal).

`__DATE__` Is the date of translation of the source file (a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10).

`__TIME__` Is the time of translation of the source file (a character string literal of the form `"hh:mm:ss"`, as in the time generated by the `asctime` function).

`__STDC__` Is the decimal constant 1, intended to indicate a conforming ANSI C implementation.

`_CP6_` Is defined when compiling on a *CP-6* system. Its value is 1.

`_L66_` Is defined when compiling on a *CP-6* system. Its value is 1.

`TS_CP6` When the strict ANSI conformance option is not specified, this variable is defined with a value of 1.

`TM_L66` When the strict ANSI conformance option is not specified, this variable is defined with a value of 1.

The values of the predefined macros (except for `__LINE__` and `__FILE__`) remain constant throughout the object unit.

None of these macro names, nor the identifier **`defined`**, may be the subject of a **`#define`** or a **`#undef`** preprocessing directive. All predefined macro names begin with a leading underscore followed by an upper-case letter or a second underscore (unless the strict ANSI conformance option is not specified).

Section 9

Introduction to the C Library

This section presents an overview of the *CP-6* C library headers, definitions, and functions.

Definitions of Terms

A *string* is a contiguous sequence of characters terminated by and including the first null character. A "pointer to" a string is a pointer to its initial (lowest addressed) character. The "length" of a string is the number of characters preceding the null character, and its "value" is the sequence of the values of the contained characters, in order.

A *letter* is a printing character in the execution character set corresponding to any of the 52 required lower-case and upper-case letters in the source character set, listed in Section 1.

The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[1]It is represented in the text and examples by a period.

Headers

Each library function is declared in a *header*, whose contents are made available by the **#include** preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use.

[1] The functions that make use of the decimal-point character are `localeconv`, `fprintf`, `fscanf`, `printf`, `scanf`, `sprintf`, `sscanf`, `vfprintf`, `vprintf`, `vsprintf`, `atof`, and `strtod`.

ANSI Standard Headers

The standard headers are:

`<assert.h>`	`<locale.h>`	`<stddef.h>`
`<ctype.h>`	`<math.h>`	`<stdio.h>`
`<errno.h>`	`<setjmp.h>`	`<stdlib.h>`
`<float.h>`	`<signal.h>`	`<string.h>`
`<limits.h>`	`<stdarg.h>`	`<time.h>`

Headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including `<assert.h>` depends on the definition of `NDEBUG`. If used, a header must be included outside of any external declaration or definition, and it must first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if the identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program may not have any macros with names lexically identical to keywords currently defined prior to the inclusion.

CP-6 C Headers

These header files provide access to *CP-6* specific structures for *CP-6* Host functions:

`<b$dcb_c.h>`	`<fileinfo.h>`	`<xu_macro_c.h>`
`<b$jit_c.h>`	`<memory.h>`	`<xu_perr_c.h>`
`<b$roseg.h>`	`<uts_name.h>`	`<xu_subs_c.h>`
`<b$tcb_c.h>`	`<valloc.h>`	`<xux$interface_m.h>`
`<cp_6_subs.h>`	`<xu_cp6_c.h>`	

Reserved Identifiers

Each header declares or defines all identifiers listed in its associated section, and optionally declares or defines identifiers which are always reserved either for any use or for use as file scope identifiers.

Identifiers are reserved as follows:

- All identifiers that begin with an underscore and either an upper-case letter or another underscore are always reserved for any use.
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
- Each macro name listed in any of the following sections is reserved for any use if any of its associated headers is included.
- All identifiers with external linkage in any of the following sections are always reserved for use as identifiers with external linkage.[2]
- Each identifier with file scope listed in any of the following sections is reserved for use as an identifier with file scope in the same name space if any of its associated headers is included.

No other identifiers are reserved. If the program declares or defines an identifier with the same name as an identifier reserved in that context (other than as described under Use of Library Functions later in this section), section), compilation errors (or runtime errors) may result.[3]

Errors `<errno.h>`

The header `<errno.h>` defines several macros, all relating to the reporting of error conditions.

The macros are

```
EDOM
ERANGE
```

which expand to integral constant expressions with distinct nonzero values; and

```
errno
```

which expands to a modifiable lvalue that has type `int`, the value of which is set to a positive error number by several library functions. `errno` is an identifier declared with external linkage. A program should not define an identifier with the name `errno`.

The value of `errno` is zero at program startup, but is never set to zero by any library function.[4]The value of `errno` may be set to nonzero by a library function call whether or not there is an error.

[2] The list of reserved identifiers with external linkage includes errno, setjmp, and va_end.

[3] Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names must not be defined if an associated header, if any, is included.

[4] Thus, a program that uses errno for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call.

Limits `<float.h>` and `<limits.h>`

The headers `<float.h>` and `<limits.h>` define several macros that expand to various limits and parameters.

The macros, their meanings, and the constraints (or restrictions) on their values are listed under Numerical Limits, Appendix F.

Common Definitions `<stddef.h>`

The following types and macros are defined in the standard header `<stddef.h>`. Some are also defined in other headers, as noted in their respective sections.

The types are

```
typedef int ptrdiff_t;
```

which is the signed integral type of the result of subtracting two pointers;

```
typedef int size_t;
```

which is the unsigned integral type of the result of the `sizeof` operator; and

```
typedef char wchar_t;
```

which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

The macros are

```
#define NULL (void*)0
```

which expands to a null pointer constant; and

`offsetof(`*type*, *member-designator*`)`

which expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* is such that given

`static` *type* `t;`

then the expression `&(t.`*member-designator*`)` evaluates to an address constant. (The specified member must not be a bit-field.)

Use of Library Functions

An argument to a function must not have an invalid value (such as a value outside the domain of the function, a pointer outside the address space of the program, or a null pointer). If a function argument is described as being an array, the pointer actually passed to the function must have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid. Some functions declared in a header may be defined as a macro, so library functions should not be declared explicitly if their headers are included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that inhibits expansion of a macro function name. For the same syntactic reason, the address of a library function can be taken even if it is also defined as a macro. The use of **#undef** to remove any macro definition will also ensure that an actual function is referred to.

Library functions can also be declared, either explicitly or implicitly, and used without including the associated header. A function that accepts a variable number of arguments should be declared either explicitly or by including its associated header.

Examples:

The function atoi may be used in any of several ways:

- By use of its associated header (possibly generating a macro expansion):

```
#include <stdlib.h>
const char *str;
/*...*/
i = atoi(str);
```

- By use of its associated header (assuredly generating a true function reference):

```
#include <stdlib.h>
#undef atoi
const char *str;
/*...*/
i = atoi(str);
```

  or

```
#include <stdlib.h>
const char *str;
/*...*/
i = (atoi)(str);
```

- By explicit declaration:

```
extern int atoi(const char *);
const char *str;
/*...*/
i = atoi(str);
```

- By implicit declaration:

```
const char *str;
/*...*/
i = atoi(str);
```

Diagnostics <assert.h>

The header `<assert.h>` defines the **assert** macro and refers to another macro,

```
NDEBUG
```

which is *not* defined by `<assert.h>`. If **NDEBUG** is defined as a macro name at the point in the source file where `<assert.h>` is included, the **assert** macro is defined simply as

```
#define assert(ignore) ((void)0)
```

The **assert** macro is implemented as a macro, not as an actual function. The macro definition may not be suppressed in order to access an actual function.

assert Macro

Synopsis:

```
#include <assert.h>
void assert(int expression);
```

Description:

The **assert** macro puts diagnostics into programs. When it is executed, if the value of the **expression** is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number — the latter are respectively the values of the preprocessing macros __FILE__ and __LINE__) on the standard error file **stderr**.[5]It then calls the **abort** function.

Returns:

The **assert** macro returns no value.

[5] The message is written with the form:

```
Assertion failed in file xyz:c, line nnn: expression
**** C run-time error
**** Exceptional condition "Abort signal" occurred
**** XBI-00197-7 The program issued an abort signal (SIGABRT).
   Traceback follows:
XBI_RAISE_SIGNAL+.40 / TSX1 XBI_DEFAULT_SIGNAL_HANDLER
raise+.40 / TSX1 XBI_RAISE_SIGNAL
abort+.6 / TSX1 raise
_assert+.33 / TSX1 abort
main:10,,.11 [MISC] / TSX1 _assert
```

```
main:0,,.1 [INITIALIZE] / TSXO __XBI_CSTARTUPD
Bottom frame
  M$ERR issued by user.
```

Section 10

Character Handling <ctype.h> Functions

The header `<ctype.h>` declares several functions useful for testing and mapping characters. In all cases the argument is an `int`, the value of which is representable as an **`unsigned char`** or equal to the value of the macro `EOF`. The argument must not have any other value.

The term *printing character* refers to a member of the values from `0x20` (space) through `0x7E` (tilde), each of which occupies one printing position on a display device. The term *control character* refers to a member of the values from `0` (`NUL`) through `0x1F` (`US`) and the value `0x7F` (`DEL`).

Character Testing Functions

The functions in this subsection return nonzero (true) if and only if the value of the argument `c` conforms to that in the description of the function.

`isalnum` Function

Synopsis:

```
#include <ctype.h>
int isalnum(int c);
```

Description:

The **`isalnum`** function returns 1 for any character for which **`isalpha`** or **`isdigit`** is true; otherwise, it returns 0.

`isalpha` Function

Synopsis:

```
#include <ctype.h>
int isalpha(int c);
```

Description:

The **`isalpha`** function returns 1 for any character for which **`isupper`** or **`islower`** is true; otherwise, it returns 0.

iscntrl Function

Synopsis:

```
#include <ctype.h>
int iscntrl(int c);
```

Description:

The `iscntrl` function returns 1 for any control character; otherwise, it returns 0.

isdigit Function

Synopsis:

```
#include <ctype.h>
int isdigit(int c);
```

Description:

The `isdigit` function returns 1 for any decimal-digit character (as defined under Character Set in Section 1); otherwise, it returns 0.

isgraph Function

Synopsis:

```
#include <ctype.h>
int isgraph(int c);
```

Description:

The `isgraph` function returns 1 for any printing character except space (' '); otherwise, it returns 0.

islower Function

Synopsis:

```
#include <ctype.h>
int islower(int c);
```

Description:

The `islower` function returns 1 for any character that is a lower-case letter; otherwise it returns 0.

isprint Function

Synopsis:

```
#include <ctype.h>
int isprint(int c);
```

Description:

The `isprint` function returns 1 for any printing character including space (`' '`); otherwise, it returns 0.

ispunct Function

Synopsis:

```
#include <ctype.h>
int ispunct(int c);
```

Description:

The `ispunct` function returns 1 for any printing character that is neither space (`' '`) nor a character for which `isalnum` is true; otherwise, it returns 0.

isspace Function

Synopsis:

```
#include <ctype.h>
int isspace(int c);
```

Description:

The `isspace` function returns 1 for any character that is a standard white-space character; otherwise, it returns 0. The standard white-space characters are space (`' '`), form feed (`'\f'`), new-line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`).

isupper Function

Synopsis:

```
#include <ctype.h>
int isupper(int c);
```

Description:

The `isupper` function returns 1 for any character that is an upper-case letter; otherwise it returns 0.

isxdigit Function

Synopsis:

```
#include <ctype.h>
int isxdigit(int c);
```

Description:

The isxdigit function returns 1 for any hexadecimal-digit character (as defined in Section 2, Lexical Elements); otherwise, it returns 0.

Character Case Mapping Functions

The tolower and toupper functions control character case conversions.

tolower Function

Synopsis:

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

Description:

The tolower function converts an upper-case letter to the corresponding lower-case letter.

Returns:

If the argument is a character for which isupper is true and there is a corresponding character for which islower is true, the tolower function returns the corresponding character; otherwise the argument is returned unchanged.

toupper Function

Synopsis:

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

Description:

The toupper function converts a lower-case letter to the corresponding upper-case letter.

Returns:

If the argument is a character for which islower is true and there is a corresponding character for which isupper is true, the toupper function returns the corresponding character; otherwise the argument is returned unchanged.

Section 11

Localization <locale.h> Functions

The header **<locale.h>** declares two functions, one type, and defines several macros.

The type is

```
    struct lconv
```

which contains members related to the formatting of numeric values. The structure contains the following members. The semantics of the members and their normal ranges are explained under Numeric Formatting Convention Inquiry, later in this section. In the "C" locale, the members have the values specified in the comments.

```
    char *decimal_point;       /* "." */
    char *thousands_sep;       /* "" */
    char *grouping;            /* "" */
    char *int_curr_symbol;     /* "" */
    char *currency_symbol;     /* "" */
    char *mon_decimal_point;   /* "" */
    char *mon_thousands_sep;   /* "" */
    char *mon_grouping;        /* "" */
    char *positive_sign;       /* "" */
    char *negative_sign;       /* "" */
    char int_frac_digits;      /* CHAR_MAX */
    char frac_digits;          /* CHAR_MAX */
    char p_cs_precedes;        /* CHAR_MAX */
    char p_sep_by_space;       /* CHAR_MAX */
    char n_cs_precedes;        /* CHAR_MAX */
    char n_sep_by_space;       /* CHAR_MAX */
    char p_sign_posn;          /* CHAR_MAX */
    char n_sign_posn;          /* CHAR_MAX */
```

The macros defined are NULL described under Common Definitions in Section 9); and

```
    LC_ALL
    LC_COLLATE
    LC_CTYPE
    LC_MONETARY
    LC_NUMERIC
    LC_TIME
```

which expand to integral constant expressions with distinct values, suitable for use as the first argument to the **setlocale** function.

Locale Control

Locale control is accomplished with the setlocale function, as described below.

setlocale Function

Synopsis:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description:

The setlocale function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The setlocale function may be used to change or query the program's entire current locale or portions thereof. The value LC_ALL, for category, names the program's entire locale; the other values for category name only a portion of the program's locale. LC_COLLATE affects the behavior of the strcoll and strxfrm functions. LC_CTYPE affects the behavior of the character-handling functions[1]and the multibyte functions. LC_MONETARY affects the monetary formatting information returned by the localeconv function. LC_NUMERIC affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the localeconv function. LC_TIME affects the behavior of the strftime function.

A value of "C" for locale specifies the minimal environment for C translation; a value of "" for locale specifies the native environment which is "C".

The locales supported by *CP-6* C are C (the default locale), USA, and ENGLISH_CANADA.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

Returns:

If a pointer to a string is given for locale and the selection can be honored, the setlocale function returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, the setlocale function returns a null pointer and the program's locale is not changed.

A null pointer for locale causes the setlocale function to return a pointer to the string associated with the category for the program's current locale; the program's locale is not changed.

The pointer to string returned by the setlocale function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to may not be modified by the program, but may be overwritten by a subsequent call to the setlocale function.

[1] The only functions in Section 10 whose behavior is not affected by the current locale are isdigit and isxdigit.

Numeric Formatting Convention Inquiry

The localeconv function is used in formatting numeric quantities as described below.

localeconv Function

Synopsis:

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description:

The localeconv function sets the components of an object with type struct lconv with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type char * are pointers to strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are non-negative numbers, any of which can be CHAR_MAX to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
    The decimal-point character used to format nonmonetary quantities.

char *thousands_sep
    The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char *grouping
    A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char *int_curr_symbol
    The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds.* The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
    The local currency symbol applicable to the current locale.

char *mon_decimal_point
    The decimal point used to format monetary quantities.

char *mon_thousands_sep
    The separator for groups of digits before the decimal point in formatted monetary quantities.

char *mon_grouping
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
The string used to indicate a non-negative formatted monetary quantity.

char *negative_sign
The string used to indicate a negative formatted monetary quantity.

char int_frac_digits
The number of fractional digits (those after the decimal point) to be displayed in an internationally formatted monetary quantity.

char frac_digits
The number of fractional digits (those after the decimal point) to be displayed in a formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a non-negative formatted monetary quantity.

char p_sep_by_space
Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a non-negative formatted monetary quantity.

char n_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a negative formatted monetary quantity.

char n_sep_by_space
Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

char p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

0 Parentheses surround the quantity and `currency_symbol`.

1 The sign string precedes the quantity and `currency_symbol`.

2 The sign string succeeds the quantity and `currency_symbol`.

3 The sign string immediately precedes the `currency_symbol`.

4 The sign string immediately succeeds the `currency_symbol`.

Returns:

The **`localeconv`** function returns a pointer to the filled-in object. The structure pointed to by the return value should not be modified by the program, and may be overwritten by a subsequent call to the **`localeconv`** function. In addition, calls to the **`setlocale`** function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Section 12

Mathematics `<math.h>` Functions

The header `<math.h>` declares several mathematical functions and defines one macro. The functions take double-precision arguments and return double-precision values. Integer arithmetic functions and conversion functions are discussed in Section 17, General Utilities.

The macro defined is

```
HUGE_VAL
```

which expands to a positive `double` expression, not representable as a `float`.

Treatment of Error Conditions

The behavior of each mathematical function is defined for all representable values of its input arguments. Each function executes as if it were a single operation, without generating any externally visible exceptions.

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors. On a domain error, the function returns an indeterminate value; the value of the macro `EDOM` is stored in `errno`.

Similarly, a *range error* occurs if the result of the function cannot be represented as a `double` value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign (except for the `tan` function) as the correct value of the function; the value of the macro `ERANGE` is stored in `errno`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero.

Trigonometric Functions

The trigonometric functions are described in the following subsections.

acos Function

Synopsis:

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

Description:

The **acos** function computes the principal value of the arc cosine of **x**. A domain error occurs for arguments not in the range [-1, +1].

Returns:

The **acos** function returns the arc cosine in the range [0, π] radians.

asin Function

Synopsis:

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

Description:

The **asin** function computes the principal value of the arc sine of **x**. A domain error occurs for arguments not in the range [-1, +1].

Returns:

The **asin** function returns the arc sine in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

atan Function

Synopsis:

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

Description:

The **atan** function computes the principal value of the arc tangent of **x**.

Returns:

The **atan** function returns the arc tangent in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

atan2 Function

Synopsis:

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

Description:

The atan2 function computes the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.

Returns:

The atan2 function returns the arc tangent of y/x, in the range $[-\pi, +\pi]$ radians.

cos Function

Synopsis:

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

Description:

The cos function computes the cosine of x (measured in radians).

Returns:

The cos function returns the cosine value.

sin Function

Synopsis:

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

Description:

The sin function computes the sine of x (measured in radians).

Returns:

The sin function returns the sine value.

tan Function

Synopsis:

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

Description:

The tan function returns the tangent of x (measured in radians).

Returns:

The tan function returns the tangent value.

Hyperbolic Functions

The cosh, sinh, and tanh functions are described below.

cosh Function

Synopsis:

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

Description:

The cosh function computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Returns:

The cosh function returns the hyperbolic cosine value.

sinh Function

Synopsis:

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

Description:

The sinh function computes the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

Returns:

The sinh function returns the hyperbolic sine value.

tanh Function

Synopsis:

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

Description:

The tanh function computes the hyperbolic tangent of x.

Returns:

The tanh function returns the hyperbolic tangent value.

Exponential and Logarithmic Functions

The exponential and logarithmic functions are described in the following subsections.

exp Function

Synopsis:

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

Description:

The exp function computes the exponential function of x. A range error occurs if the magnitude of x is too large.

Returns:

The exp function returns the exponential value.

frexp Function

Synopsis:

```
#include <math.h>
double frexp(double value, int *exp);
```

Description:

The frexp function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the int object pointed to by exp.

Returns:

The frexp function returns the value x, such that x is a double with magnitude in the interval of $[\frac{1}{2},1)$ or zero, and value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.

`ldexp` Function

Synopsis:

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

Description:

The `ldexp` function multiplies a floating-point number by an integral power of 2. A range error may occur.

Returns:

The `ldexp` function returns the value of `x` times 2 raised to the power `exp`.

`log` Function

Synopsis:

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

Description:

The `log` function computes the natural logarithm of `x`. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Returns:

The `log` function returns the natural logarithm.

`log10` Function

Synopsis:

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

Description:

The `log10` function computes the base-ten logarithm of `x`. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Returns:

The `log10` function returns the base-ten logarithm.

modf Function

Synopsis:

```
#include <math.h>
double modf(double value, double *iptr);
```

Description:

The modf function breaks the argument value into integral and fractional parts, each of which has the same sign as value. It stores the integral part as a double in the object pointed to by iptr.

Returns:

The modf function returns the signed fractional part of value.

Power Functions

The pow and sqrt functions are described below.

pow Function

Synopsis:

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

Description:

The pow function computes x raised to the power y. A domain error occurs if x is negative and y is not an integral value or when x is zero and y is less than or equal to zero. A range error occurs if the result value is too large to represent.

Returns:

The pow function returns the value of x raised to the power y.

sqrt Function

Synopsis:

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

Description:

The `sqrt` function computes the non-negative square root of `x`. A domain error occurs if the argument is negative.

Returns:

The `sqrt` function returns the value of the square root.

Nearest Integer, Absolute Value, and Remainder Functions

The `ceil`, `fabs`, `floor`, and `fmod` functions are described below.

ceil Function

Synopsis:

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

Description:

The `ceil` function computes the smallest integral value not less than `x`.

Returns:

The `ceil` function returns the smallest integral value not less than `x`, expressed as a `double`.

fabs Function

Synopsis:

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

Description:

The `fabs` function computes the absolute value of a floating-point number `x`.

Returns:

The `fabs` function returns the absolute value of `x`.

`floor` Function

Synopsis:

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

Description:

The `floor` function computes the largest integral value not greater than `x`.

Returns:

The `floor` function returns the largest integral value not greater than `x`, expressed as a `double`.

`fmod` Function

Synopsis:

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

Description:

The `fmod` function computes the floating-point remainder of `x/y`.

Returns:

The `fmod` function returns the value `x - `*i*` * y`, for some integer *i* such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`. If `y` is zero, the `fmod` function returns zero.

Section 13

Non-Local Jumps <setjmp.h>

The header **<setjmp.h>** defines the macro **setjmp**, and declares one function and one type, for bypassing the normal function call and return discipline. This facility is useful for dealing with unusual conditions encountered in a low-level function of a program.

The type declared is

```
jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment.

setjmp is a macro. The macro definition may not be suppressed in order to access an actual function.

Calling Environment

The **setjmp** macro and the **longjmp** function save and restore the calling environment, respectively, as described below.

setjmp Macro

Synopsis:

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

Description:

The **setjmp** macro saves its calling environment in its **jmp_buf** argument for later use by the **longjmp** function.

Returns:

If the return is from a direct invocation, the **setjmp** macro returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** macro returns a nonzero value.

Environmental Constraints:

An invocation of the **setjmp** macro should appear only in one of the following contexts:

- The entire controlling expression of a selection or iteration statement.
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.
- The operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement.
- The entire expression of an expression statement (possibly cast to `void`).

`longjmp` Function

Synopsis:

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

Description:

The **`longjmp`** function restores the environment saved by the most recent invocation of the **`setjmp`** macro in the same invocation of the program, with the corresponding **`jmp_buf`** argument. If there has been no such invocation, or if the function containing the invocation of the **`setjmp`** macro has terminated execution[1]in the interim, the behavior is undefined.

All accessible objects have values as of the time **`longjmp`** was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **`setjmp`** macro that do not have volatile-qualified type and have been changed between the **`setjmp`** invocation and **`longjmp`** call are indeterminate.

As it bypasses the usual function call and return mechanisms, the **`longjmp`** function executes correctly in contexts of interrupts, signals, and any of their associated functions. However, the **`longjmp`** function may not be invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal).

Returns:

After **`longjmp`** is completed, program execution continues as if the corresponding invocation of the **`setjmp`** macro had just returned the value specified by **`val`**. The **`longjmp`** function cannot cause the **`setjmp`** macro to return the value 0; if **`val`** is 0, the **`setjmp`** macro returns the value 1.

[1] For example, by executing a `return` statement or because another `longjmp` call has caused a transfer to a `setjmp` invocation in a function earlier in the set of nested calls.

Section 14

Signal Handling `<signal.h>`

The header `<signal.h>` declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

The type defined is

```
sig_atomic_t
```

which is the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The macros defined include

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values. These values have a type compatible with the second argument to and the return value of the `signal` function; each value compares unequal to the address of any declarable function. Additional macros, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition, are as follows:

```
SIGABRT
SIGALRM
SIGFPE
SIGHUP
SIGILL
SIGINT
SIGSEGV
SIGTERM
SIGUSR1
SIGUSR2
```

All signal numbers are positive.

Signal Handling Macros

The three signal handling macros are described here along with their uses.

SIG_DFL Macro

This macro is a constant expression whose type is "pointer to function". This value is suitable as the second argument to the `signal` function to request the default signal handler. The default signal handler will display a description of the signal, display the call history, and stop execution when a signal using this handler is `raised`.

SIG_ERR Macro

This macro is a constant expression whose type is "pointer to function". This value is returned by the `signal` function when the signal request cannot be honored. This is not a valid signal handler and therefore is not accepted as the second argument of the `signal` function.

SIG_IGN Macro

This macro is a constant expression whose type is "pointer to function". The value is suitable as the second argument to the `signal` function to request ignoring of a signal. This signal handler keeps itself as the handler and returns. This signal handler is only permitted to handle the following signals:

```
SIGABRT
SIGALRM
SIGINT
SIGTERM
SIGUSR1
SIGUSR2
```

Signal Types

The various types of signals and their use are explained below.

SIGABRT Signal

Description:

This signal is raised by calling the `abort` function or by assertion failures which use the `assert` macro.

Semantics:

The initial handler for this signal is `SIG_DFL`. If the signal handler returns to its caller, this signal is immediately repeated to the new signal handler which by default will cause program termination. The signal handling function may `longjmp` to continue execution. The default handling for this signal is to display a C run-time error message, display the call history at the time of the signal, and stop execution.

SIGALRM Signal

Description:

This signal is reserved for future use.

SIGFPE Signal

Description:

This signal is raised for division by zero, fixed point overflow, floating point overflow and floating point underflow.

Semantics:

The initial state of this signal is `SIG_DFL`. If the signal handler returns to its caller, this signal is immediately repeated to the new signal handler which by default will cause program termination (via `SIG_DFL` handling). The signal handling function may `longjmp` to continue execution. The default handling for this message is to display a C run-time error message, display the call history at the time of the signal, and stop execution.

If the signal handler for this signal is set to `SIG_IGN`, the result of the erroneous operation is set to an appropriate value and program execution continues after the point of the error. The "appropriate" value depends upon the fault that occurred. For example, when overflow occurs, the value is set to the largest value that can be represented.

SIGHUP Signal

Description:

This signal is raised when the user's terminal is disconnected from the system.

Semantics:

The default handler for this signal is `SIG_IGN` which causes the disconnect event to be ignored and terminates program execution. If the signal handler returns to its caller, execution continues at the point of the signal. When the `SIGHUP` signal handler is called, the signal handler for `SIGHUP` is set to `SIG_IGN`. This event may be used to complete any database accesses or save the current status of the process.

SIGILL Signal

Description:

This signal is raised if the program tries to execute an illegal or nonexistent instruction.

Semantics:

The default handler for this signal is `SIG_DFL`. If the signal handler returns to its caller, the signal is immediately repeated to the new signal handler which by default will cause program termination via `SIG_DFL`. The signal handling function may `longjmp` to continue execution.

The default handling for this signal is to display a C run-time error message, display the call history at the time of the signal, and stop execution.

SIGINT Signal

Description:

This signal is raised when a time-sharing user interrupts the program either by using the **break** key or the **escape-B** sequence.

Semantics:

The default signal handler is `SIG_IGN` which causes the command processor to get control when a break event occurs. The signal handler may return to continue execution at the point of interruption, or `longjmp` to continue execution at a location specified by an earlier `setjmp`.

After the `SIGINT` handler is entered, it must immediately establish a new handler for this signal or all subsequent break events will be ignored.

SIGSEGV Signal

Description:

This signal is raised when the program makes an invalid memory access. This includes the following traps: security faults, page faults, programmed faults and various other hardware faults.

Semantics:

The initial handler for this signal is `SIG_DFL`. If the signal handler returns to its caller, this signal is immediately raised to the new signal handler which by default will cause program termination through the `SIG_DFL` handler. The signal handling function may `longjmp` to continue execution.

The default handling for this signal is to display a C run-time error message, display the call history at the time of the signal, and stop execution.

SIGTERM Signal

Description:

This signal is raised when the program exits.

Semantics:

The default handler for this signal is `SIG_IGN` which causes the exit event to be ignored and terminates program execution. If the signal handler returns to its caller, execution continues at the point of the signal.

The **exit** function and **return** from the **main** function do not raise `SIGTERM`. In addition, termination due to the default signal handler does not raise `SIGTERM`. All other exits cause `SIGTERM` to be raised.

When the `SIGTERM` signal handler is called, the signal handler for `SIGTERM` is set to `SIG_IGN`.

This signal may be raised for many reasons including

1. A non-C function executing an **M$EXIT**, **M$ERR** or **M$XXX** monitor call.
2. A monitor service request that had an error which did not specify error handling.
3. Invocation of another program via the **M$LDTRC** monitor service.
4. An exceptional condition which could not be handled. This will not normally occur.
5. The operator "errored" the user.
6. A resource limit (such as `CPU` time, or output) was exceeded.
7. The operator "aborted" the user.

The SIGUSR1 and SIGUSR2 Signals

Description:

These signals are provided for use by C programs. These signals are not raised by the C library under any conditions.

Semantics:

The initial handler for these signals is `SIG_DFL`. If the signal handler returns to its caller, execution resumes at the point the event was raised. The signal handling function may establish a new handler for the signal and may also `longjmp` to continue execution.

The default handling for these signals is to display a C run-time error message, display the call history at the time of the signal, and stop execution.

Signal Handling and Sending

The `signal` and `raise` functions specify signal handling and sending, respectively, as described below.

`signal` Function

Synopsis:

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

Description:

The `signal` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise, `func` must point to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if `func` points to a function, first the equivalent of `signal(sig, SIG_DFL);` is executed. Next the equivalent of `(*func)(sig);` is executed. The function `func` may terminate by executing a `return` statement or by calling the `abort`, `exit`, or `longjmp` function. If `func` executes a `return` statement and the value of `sig` was `SIGFPE` or any value corresponding to a computational exception, the behavior is undefined. In any case, the program will resume execution at the point at which it was interrupted.

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler calls any function in the standard library other than the `signal` function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler) or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call to the `signal` function results in a `SIG_ERR` return, the value of `errno` is indeterminate.[1]

At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

is executed for `SIGINT` and `SIGTERM`; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals.

Returns:

If the request can be honored, the `signal` function returns the value of `func` for the most recent call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

`raise` Function

Synopsis:

```
#include <signal.h>
int raise(int sig);
```

Description:

The `raise` function sends the signal `sig` to the executing program. The following signals may be raised:

```
SIGABRT
SIGALRM
SIGINT
SIGTERM
SIGUSR1
SIGUSR2
```

Attempting to raise any other signals results in an error.

Returns:

The `raise` function returns zero if successful, nonzero if unsuccessful.

[1] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

Section 15

Variable Arguments <stdarg.h>

The header **<stdarg.h>** declares a type and defines three macros, for advancing through a list of arguments whose number and types are not known to the called function when it is compiled.

A function may be called with a variable number of arguments of varying types. As described in Section 7, External Data Definitions, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

The type declared is

```
va_list
```

which is a type suitable for holding information needed by the macros **va_start**, **va_arg**, and **va_end**. If access to the varying arguments is desired, the called function should declare an object (referred to as **ap** in this section) having type **va_list**. The object **ap** may be passed as an argument to another function.

Variable Argument List Access Macros

The **va_start**, **va_end**, and **va_arg** macros described in this section are not actual functions. The **va_start** and **va_end** macros must be invoked in order to access a function. The **va_start** and **va_end** macros are invoked in the function accepting a varying number of arguments, if access to the varying arguments is desired.

va_start Macro

Synopsis:

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

Description:

The **va_start** macro must be invoked before any access to the unnamed arguments.

The **va_start** macro initializes **ap** for subsequent use by **va_arg** and **va_end**.

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). The parameter *parmN* may not be declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions.

Returns:

The **va_start** macro returns no value.

va_arg Macro

Synopsis:

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

Description:

The **va_arg** macro expands to an expression that has the type and value of the next argument in the call. The parameter **ap** must be initialized by the **va_start** macro. Each invocation of **va_arg** modifies **ap** so that the values of successive arguments are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. There must be an actual next argument, and *type* must be compatible with the type of the actual next argument (as promoted according to the default argument promotions).

Returns:

The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

va_end Macro

Synopsis:

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

Description:

The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of **va_start** that initialized the **va_list ap**. The **va_end** macro modifies **ap** so that it is no longer usable (without an intervening invocation of **va_start**).

Returns:

The **va_end** macro returns no value.

Examples:

The function **f1** gathers into an array a list of arguments that are pointers to strings (but not more than **MAXARGS** arguments), then passes the array as a single argument to function **f2**. The number of pointers is specified by the first argument to **f1**.

```
#include <stdarg.h>
#define MAXARGS     31
void f1(int n_ptrs, ...)

        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;
        if (n_ptrs > MAXARGS)
              n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no < n_ptrs)
              array[ptr_no++] = va_arg(ap, char *);
        va_end(ap);
        f2(n_ptrs, array);
```

Each call to **f1** should have visible the definition of the function or a declaration such as:

```
void f1(int, ...);
```

Section 16

Input/Output <stdio.h> Functions

Introduction

The header `<stdio.h>` declares three types, several macros, and many functions for performing input and output.

The types declared are `size_t` (described under Common Definitions, in Section 9);

```
FILE
```

which is an object type that holds the information needed to control an I/O stream, including its file position indicator, a pointer to its associated buffer, an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

```
fpos_t
```

which is an object type that holds the information needed to uniquely specify a position within a file.

The macros include `NULL` (described under Common Definitions, in Section 9) and the following:

```
_IOFBF
_IOLBF
_IONBF
```

which expand to integral constant expressions with distinct values, for use as the third argument to the `setvbuf` function;

```
BUFSIZ
```

which expands to an integral constant expression, which is the size of the buffer used by the `setbuf` function;

```
EOF
```

which expands to a negative integral constant expression that is returned by several functions to indicate *end-of-file* (that is, no more input from a stream);

```
FOPEN_MAX
```

which expands to an integral constant expression that is the maximum number of files that can be open simultaneously;

```
FILENAME_MAX
```

which expands to an integral constant expression that is the size needed for an array of `char` large enough to hold the longest file name string that can be opened;

```
L_tmpnam
```

which expands to an integral constant expression that is the size needed for an array of `char` large enough to hold a temporary file name string generated by the `tmpnam` function;

```
SEEK_CUR
SEEK_END
SEEK_SET
```

which expand to integral constant expressions with distinct values, for use as the third argument to the `fseek` function;

```
TMP_MAX
```

which expands to an integral constant expression that is the minimum number of unique file names generated by the `tmpnam` function;

```
stderr
stdin
stdout
```

which are expressions of type "pointer to `FILE`" that point to the `FILE` objects associated, respectively, with the standard error, standard input, and standard output streams.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.

A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. The last line does not require a terminating new-line character when writing a file, as a new line is automatically added when the file is read. Data read in from a text stream will compare equal to the data that were earlier written out to that stream only if: the data consists only of printable characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character.

Space characters that are written out immediately before a new-line character may not appear when read in.

A binary stream is an ordered sequence of characters that transparently record internal data. Data read in from a binary stream compare equal to the data that were earlier written out to that stream.

Stream Buffering

The C Library allows either *fully buffered*, *line buffered*, or *unbuffered* to be specified for a stream. The actual meaning of these attributes is likely to vary between C implementations. *Fully buffered* is the default when a stream is opened. A different buffering attribute can be requested by calling the `setbuf` or `setvbuf` functions.

Fully buffered requests the use of `FSFA` (Fast Sequential File Access) routines. If the stream cannot be opened with `FSFA`, then *line buffered* is used.

Line buffered requests the use of **`M$READ`** and **`M$WRITE`** to read and write entire lines of text.

Unbuffered is treated as *line buffered* when I/O is directed to a file since *unbuffered* I/O is not meaningful in a record-oriented file system. Terminal I/O is the only case in which *unbuffered* I/O may be used. It is not possible to efficiently implement *unbuffered* input on *CP-6* systems, so when this behavior is required, specially written `PL-6` routines are necessary.

Unbuffered output can be useful when writing to a terminal. If *unbuffered* is specified on a stream connected to a terminal, the stream is opened with *CP-6* mode **`ORG=TERMINAL`**. However, if *CP-6* C were to immediately write every character sent to the terminal as it was generated to an *unbuffered* stream, there would be excessive overhead. For example, if a large quantity of output were generated using `putc` function calls, there would be an **`M$WRITE`** monitor call for each character written. Because of the poor performance that this behavior would produce, a compromise between efficiency and functionality is provided. Functions that normally generate more than one character of output flush output immediately; functions in this category include `printf`, `fputs`, `puts` and `fwrite`. Functions that generate a single character of output (`fputc`, `putc` and `putchar`) buffer their output. Buffered output is flushed when a function that flushes output immediately is called or when the `fflush` function is executed. To ensure that output appears before a newline character is written, the `fflush` function must be executed at appropriate places in the program.

The `fflush` function only flushes output directed to the terminal device. If a stream is connected to a file, *CP-6* C waits until a newline character is written before writing the new record.

CP-6 C provides the ability to set the prompt to be used on terminal reads. The prompt is set when any of the stream-accessing functions is used (fgetc, getc, fgets, or fread). When a read from the terminal is about to take place, *CP-6* C first checks to see if the stream from which the read is going to occur has any queued output pending. If it does, or if stdout has queued output to the terminal pending, then that output is used to set the prompt. This prompt remains until another read occurs with queued terminal output.

Mapping Text Streams to the CP-6 File System

Mapping from a C text stream into a *CP-6* file transforms each line in the text stream into a record. Newline characters do not appear in a file; each record implicitly ends with a newline character.

Text files created by C are consecutive files; however, a text stream may read from any type of *CP-6* file.

The size of the buffer used to read text streams may be specified using setvbuf function. The buffer should be large enough to handle the longest record to to be read or written in the file.

CP-6 C does not support the "r+" and "w+" modes on text streams.[1]When these modes are specified on a text stream, the stream is opened as a binary stream instead of a text stream. Append-Update mode ("a+") is handled properly as this requests that all writes are done at the end of the stream regardless of the position of the stream at the time of the write. When opening a file in "a+" mode, the type of the file is examined to determine if the file should be opened as a binary or text stream. If the file type is cb (or if the file does not exist), the file is opened as a binary stream; otherwise, it is opened as a text stream.

[1] *CP-6* C does not support read-update or write-update mode for text streams because there are two cases that cannot be handled correctly:

1. A newline character cannot be written at a position that contains a non-newline character. To maintain a consistent file structure it would be necessary to split the record containing the character into two records. As records cannot be inserted into a consecutive file, this cannot be done.
2. A non-newline character cannot be written at a position that contains a newline character. To maintain a consistent file structure, it would be necessary to join the record containing the character with the following record. As records cannot be deleted in a consecutive file, this cannot be done.

Text Stream Positioning

The **fseek** function provides the ability to position to a specified position in a stream. Positioning in text streams is very restricted; it is only possible to position to points that have been previously "remembered" by a call to the **ftell** function or to the beginning or the end of the file.

Mapping Binary Streams to the CP-6 File System

The basic operations on binary streams are reading, writing and positioning.

Positioning works differently for binary streams than for text streams. The **fseek** function is used to position to any absolute or relative position in a binary stream. Binary streams are implemented in *CP-6* C using keyed files with fixed size records. Because the records have a fixed size, it is possible to translate a character position in a binary stream to a record number and position within the record.

Binary files created by C are keyed files with edit keys that start at 0.001 and increment by 0.001. It is possible to read binary files produced by non-C programs. To do this:

1. The file must have fixed size records (the last record in the file can be short).
2. The **setvbuf** function must be used to provide a buffer to the stream that is the same size as the records in the file.
3. If the file is keyed, it must have appropriate keys. The following COPY command puts correct keys onto a file:

```
!COPY file OVER file (ln(.001,.001),ty=cb)
```

If a binary stream is open to a device, then TRANS=YES is specified on all reads and writes.

Binary Stream Buffering

When a binary stream is connected to a file, the only type of buffering provided is *line buffered* in which M$READ and M$WRITE are used to read and write entire records.

Unbuffered is available for streams connected to a device. If a terminal device is connected, the stream is opened with ORG=TERMINAL and the terminal attribute ACTONTRN is set. This allows read activation to occur when a newline (CR) character is read.

As is the case for text files, functions that generate a single character of output buffer their output; functions that normally generate more than one character of output flush it immediately.

Files

A stream is associated with an external file, which may be a physical device, by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start of the file, unless the file is opened with append mode, in which case it is positioned at the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input takes place as if characters were read by successive calls to the `fgetc` function; all output takes place as if characters were written by successive calls to the `fputc` function.

Binary files are not truncated, except as defined under the `fopen` function later in this section. A write on a text stream causes all records in the associated file to be deleted beyond that point.

A file is disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are written) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after closing the associated file, including the standard text streams. A file on which no characters have been written by an output stream will actually exist, with no data.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the `main` function returns to its original caller, or if the `exit` function is called, all open files are closed, hence all output streams are flushed before program termination.

The address of the `FILE` object used to control a stream is significant; a copy of a `FILE` object will not serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream does not refer to an interactive device.

Functions that open additional, nontemporary files require a *file name*, which is a string containing a *CP-6* file identifier.

Operations on Files

The functions that perform operations on files are described below.

`remove` Function

Synopsis:

```
#include <stdio.h>
int remove(const char *filename);
```

Description:

The **`remove`** function causes the file whose name is the string pointed to by **`filename`** to be deleted. If the **`remove`** function is successful, a subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the **`remove`** function deletes the file when all readers have closed the stream, or reports an error if the file is being updated.

Returns:

The **`remove`** function returns zero if the operation succeeds, nonzero if it fails.

`rename` Function

Synopsis:

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description:

The **`rename`** function causes the file whose name is the string pointed to by **`old`** to be henceforth known by the name given by the string pointed to by **`new`**. The file named **`old`** is no longer accessible by that name. If a file named by the string pointed to by **`new`** exists prior to the call to the **`rename`** function, an error occurs. The **`rename`** function may not change the account in which the file resides.

Returns:

The **`rename`** function returns zero if the operation succeeds; it returns nonzero if it fails,[2]in which case if the file existed previously it is still known by its original name.

[2] Among the reasons the **`rename`** function may fail are that the file is open or that it is necessary to copy its contents to rename it.

tmpfile Function

Synopsis:

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description:

The **tmpfile** function creates a temporary binary file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, an open temporary file is removed. The file is opened for update with "**wb+**" mode.

Returns:

The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

tmpnam Function

Synopsis:

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

Description:

The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file. Files created using strings generated by the **tmpnam** function are temporary only in the sense that they are star files that are deleted at logoff time. It is still advisable to use the **remove** function to remove such files when their use is ended, and before program termination.

The **tmpnam** function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, an error is reported by returning a **NULL** pointer.

Returns:

If the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the **tmpnam** function modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam chars**; the **tmpnam** function writes its result in that array and returns the argument as its value. The value of **TMP_MAX** is 999.

File Access Functions

The file access functions are described in the following subsections.

`fclose` Function

Synopsis:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description:

The `fclose` function causes the stream pointed to by **`stream`** to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

Returns:

The `fclose` function returns zero if the stream was successfully closed, or `EOF` if any errors were detected.

`fflush` Function

Synopsis:

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description:

If **`stream`** points to an output stream or an update stream in which the most recent operation was not input, the **`fflush`** function causes any unwritten data for that stream to be written to the file.

If **`stream`** is a null pointer, the `fflush` function performs this flushing action on all streams.

Returns:

The `fflush` function returns `EOF` if a write error occurs; otherwise it returns zero.

fopen Function

Synopsis:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Description:

The **fopen** function opens the file whose name is the string pointed to by **filename**, and associates a stream with it.

The argument **mode** points to a string beginning with one of the following sequences:

r	Open text file for reading.
w	Truncate to zero length or create text file for writing.
a	Append; open or create text file for writing at end-of-file.
rb	Open binary file for reading.
wb	Truncate to zero length or create binary file for writing.
ab	Append; open or create binary file for writing at end-of-file.
r+	Open text file for update (reading and writing).
w+	Truncate to zero length or create text file for update.
a+	Append; open or create text file for update, writing at end-of-file.
r+b *or* rb+	Open binary file for update (reading and writing).
w+b *or* wb+	Truncate to zero length or create binary file for update.
a+b *or* ab+	Append; open or create binary file for update, writing at end-of-file.

Opening a file with read mode ('r' as the first character in the **mode** argument) fails if the file does not exist or cannot be read.

Opening a file with append mode ('a' as the first character in the **mode** argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **fseek** function.

When a file is opened with update mode ('+' as the second or third character in the above list of **mode** argument values), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the **fflush** function or to a file positioning function (**fseek** , **fsetpos**, or **rewind**), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode opens (or creates) a binary stream.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

The *CP-6* version of the **fopen** function permits a specific DCB to be opened by supplying an illegal file name in one of two forms which has special interpretation:

#n In this case, the *n* must be one of the integers "1", "2", "3" or "4". This form requests that the stream be opened to one of the command line file names (by convention, "#1" is the input file position, "#2" is the update file position, "#3" is the object file position and "#4" is the listing file position).

dcb=*name* In this case, the DCB name is explicitly supplied. The DCB to be opened can be arranged from the command line by using *CP-6* **link** options or it could be a DCB for which the user has specified an IBEX **set** command.

When a DCB is explicitly requested, the **mode** argument to **fopen** is treated differently when the DCB has been "**set**" with one of the following options:

Fun=Create,Exist=Oldfile If these options are specified (the command line verb INTO can implicitly request it), the file is opened in append mode ("**a**") regardless of the mode specified on the call.

Fun=Create,Exist=Error If the mode is "**w**" then the **fopen** successfully opens the file only if the file does not currently exist. This mode can be requested implicitly by using the command line verb ON or TO.

The *CP-6* **fopen** function also provides the ability to specify a number of open options in the **mode** parameter. The **mode** string must begin with the sequence outlined previously which indicates the basic kind of operations that are required upon the stream. The additional options are used to further control the actual behavior of the stream (*CP-6* device or file type dependent). The initial mode may optionally be followed by a blank and keywords separated by blanks request various options. The "**fopen** mode keywords" table contains the keywords accepted by **fopen**. If a keyword is found which is not recognized, the rest of the string is ignored.

Keyword	Meaning
oldfile	If creating, update original file if it exists.
newfile	If creating, replace original file if it exists.
error	If creating, report an error if the file exists.
named	If creating, create a permanent file.
scratch	If creating, create a temporary file.
ctg	If creating, catalog file immediately.
none	Allow multiple readers or one updater.
sharein	Allow one updater and multiple readers.
all	Allow multiple updaters and readers.

Table 16-1. **fopen** *Mode Keywords*

Keyword	Meaning
d800	For tape opens, set density to 800 bpi.
d1600	For tape opens, set density to 1600 bpi.
d6250	For tape opens, set density to 6250 bpi.
terminal	Program will supply all terminal positioning.
x364	Translate X3.64 controls for terminal type.
ur	If creating, create a unit-record file.
keyed	If creating, create a keyed file.
consec	If creating, create a consecutive file.
comp	If creating, compress records in disk file.

Table 16-1. `fopen` *Mode Keywords (part 2)*

Returns:

The `fopen` function returns a pointer to the `FILE` object controlling the stream. If the open operation fails, `fopen` returns a null pointer.

`freopen` Function

Synopsis:

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode,
     FILE *stream);
```

Description:

The `freopen` function opens the file whose name is the string pointed to by `filename` and associates the stream pointed to by `stream` with it. The `mode` argument is used just as in the `fopen` function.[3]

The `freopen` function first attempts to close any file that is associated with the specified stream. Errors encountered while closing the stream are not reported. The error and end-of-file indicators for the stream are cleared.

Returns:

The `freopen` function returns a null pointer if the open operation fails. Otherwise, `freopen` returns the value of `stream`.

[3] The primary use of the `freopen` function is to change the file associated with a standard text stream (`stderr`, `stdin`, or `stdout`), as those identifiers are not modifiable lvalues to which the value returned by the `fopen` function may be assigned.

setbuf Function

Synopsis:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Description:

Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values _IOFBF for **mode** and BUFSIZ for **size**, or, if **buf** is a null pointer, with the value _IONBF for **mode**.

Returns:

The **setbuf** function returns no value.

setvbuf Function

Synopsis:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description:

The **setvbuf** function may be used only after the stream pointed to by **stream** has been associated with an open file and before any other operation is performed on the stream. The argument **mode** determines how **stream** will be buffered, as follows: _IOFBF causes input/output to be fully buffered; _IOLBF causes input/output to be line buffered; _IONBF causes input/output to be unbuffered. If **buf** is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function.[4]The argument **size** specifies the size of the array. The contents of the array at any time are indeterminate.

Returns:

The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for **mode** or if the request cannot be honored.

Formatted Input/Output Functions

The formatted input/output functions are described in the following subsections.

[4] The buffer must have a lifetime at least as great as the open stream, so the stream must be closed before a buffer that has automatic storage duration is deallocated upon block exit.

`fprintf` Function

Synopsis:

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Description:

The `fprintf` function writes output to the stream pointed to by `stream`, under control of the string pointed to by `format` which specifies how subsequent arguments are converted for output. There must be sufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

The format must be a character string. The format is composed of zero or more directives: ordinary characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described below, has been given) to the field width. The field width takes the form of an asterisk `*` (described below) or a decimal integer.[5]
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions; the number of digits to appear after the decimal-point character for `e`, `E`, and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of characters to be written from a string in `s` conversion. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described below) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. A precision may not appear with any other conversion specifier.
- An optional `h` specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integral promotions, and its value will be converted to `short int` or `unsigned short int` before printing); an optional `h` specifying that a following `n` conversion specifier applies to a pointer to a `short int` argument; an optional `l` (ell) specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; an optional `l` specifying that a following `n` conversion specifier applies to a pointer to a `long int` argument; or an optional `L` specifying that a following `e`, `E`, `f`, `g`, or `G` conversion specifier applies to a `long double` argument. An `h`, `l`, or `L` may not appear with any other conversion specifier.
- A character that specifies the type of conversion to be applied.

[5] Note that `0` is taken as a flag, not as the beginning of a field width.

A field width, or precision, or both, may be indicated by an asterisk instead of a digit string. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field. If this flag is omitted, the result will be right-justified.

+ The result of a signed conversion will always begin with a plus or minus sign. If this flag is omitted, the result will begin with a sign only when a negative value is converted.

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

# The result is to be converted to an "alternate form". For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (or `X`) conversion, a nonzero result will have `0x` (or `0X`) prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros will *not* be removed from the result. Other conversions do not use this flag.

0 For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and - flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. Other conversions do not use this flag.

The conversion specifiers and their meanings are:

`d,i` The `int` argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

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

`f` The `double` argument is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

`e,E` The `double` argument is converted in the style *[-]d.ddd*e±*dd*, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

`g,G` The `double` argument is converted in style `f` or `e` (or in style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

`c` The `int` argument is converted to an `unsigned char`, and the resulting character is written.

`s` The argument must be a pointer to an array of character type.[6] Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.

`p` The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, as unsigned octal.

`n` The argument is a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call. No argument is converted.

`%` A `%` is written. No argument is converted. The complete conversion specification is `%%`.

[6] No special provisions are made for multibyte characters.

An argument may not be, or point to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer using `%p` conversion).

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns:

The `fprintf` function returns the number of characters written to the output stream, or a negative value if an output error occurred.

Environmental Limit:

The maximum number of characters produced by any single conversion is 509.

Examples:

The following prints a date and time in the form "Sunday, July 3, 10:02", where `weekday` and `month` are pointers to strings:

```
#include <stdio.h>
/*...*/
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
```

The following prints π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/*...*/
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

`fscanf` Function

Synopsis:

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Description:

The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. There must be sufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

The format is a character string. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional decimal integer that specifies the maximum field width.
- An optional `h`, `l` (ell), or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` must be preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by `l` if it is a pointer to `long int`. Similarly, the conversion specifiers `o`, `u`, and `x` must be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by `l` if it is a pointer to `unsigned long int`. Finally, the conversion specifiers `e`, `f`, and `g` must be preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. An `h`, `l`, or `L` must not appear with any other conversion specifier.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `fscanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `fscanf` function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

- Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier.
- An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails. This condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.
- Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails. This condition is a matching failure. Unless assignment suppression was indicated by an `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. This object should have an appropriate type, and the result of the conversion must be represented in the space provided.

The following conversion specifiers are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the `base` argument. The corresponding argument is a pointer to integer.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the `base` argument. The corresponding argument is a pointer to integer.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the `base` argument. The corresponding argument is a pointer to unsigned integer.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the `base` argument. The corresponding argument is a pointer to unsigned integer.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the `base` argument. The corresponding argument is a pointer to unsigned integer.

`e,f,g` Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument is a pointer to floating.

`s` Matches a sequence of non-white-space characters.[7]The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

`[` Matches a nonempty sequence of characters[8]from a set of expected characters (the *scanset*). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) constitute the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the `-` is treated as an ordinary character.

[7] No special provisions are made for multibyte characters.

[8] No special provisions are made for multibyte characters.

c Matches a sequence of characters[9] of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p Matches an unsigned octal integer, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fprintf` function. The corresponding argument is a pointer to a pointer to `void`. If the input item is a value converted earlier during the same program execution, the pointer that results should compare equal to that value.

n No input is consumed. The corresponding argument is a pointer to integer into which is written the number of characters read from the input stream so far by this call to the `fscanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `fscanf` function.

% Matches a single `%`; no conversion or assignment occurs. The complete conversion specification is `%%`.

The conversion specifiers `E`, `G`, and `X` are also valid and behave the same as, respectively, `e`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure. Otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns:

The `fscanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `fscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Examples:

The call:

```
#include <stdio.h>
/*...*/
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

[9] No special provisions are made for multibyte characters.

with the input line:

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

will assign to *n* the value 3, to *i* the value 25, and to *x* the value 5.432; and *name* will contain `thompson\0`. Or:

```
#include <stdio.h>
/*...*/
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to *i* the value 56 and to *x* the value 789.0, will skip 0123, and *name* will contain `56\0`. The next character read from the input stream will be `a`.

The following accepts repeatedly from `stdin` a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/*...*/
int count; float quant; char units[21], item[21];
while (!feof(stdin) && !ferror(stdin)) {
      count = fscanf(stdin, "%f%20s of %20s",
            &quant, units, item);
      fscanf(stdin,"%*[^\n]");
}
```

If the `stdin` stream contains the following lines:

```
2 quarts of oil
-12.8 degrees Celsius
lots of luck
10.0 LBS of fertilizer
100 ergs of energy
```

the execution of the above example is analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; /* "C" fails to match "o" */
count = 0; /* "l" fails to match "%f" */
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "fertilizer");
count = 3;
count = 0; /* "100e" fails to match "%f" */
count = EOF;
```

printf Function

Synopsis:

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

Description:

The printf function is equivalent to fprintf with the argument stdout interposed before the arguments to printf.

Returns:

The printf function returns the number of characters written to stdout, or a negative value if an output error occurred.

scanf Function

Synopsis:

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

Description:

The scanf function is equivalent to fscanf with the argument stdin interposed before the arguments to scanf.

Returns:

The scanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the scanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

sprintf Function

Synopsis:

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

Description:

The sprintf function is equivalent to fprintf, except that the argument s specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum. Copying must not take place between objects that overlap.

Returns:

The sprintf function returns the number of characters written into the array, not counting the terminating null character.

sscanf Function

Synopsis:

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

Description:

The sscanf function is equivalent to fscanf, except that the argument s specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the fscanf function. Copying must not take place between objects that overlap.

Returns:

The sscanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the sscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vfprintf Function

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

Description:

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by arg, which is initialized by the va_start macro (and possibly subsequent va_arg calls).

Returns:

The vfprintf function returns the number of characters written to the output stream, or a negative value if an output error occurred.

Examples:

The following shows the use of the vfprintf function in a general error-reporting routine:

```
#include <stdarg.h>
#include <stdio.h>
void error(char *function_name, char *format, ...)
{
```

```
            va_list args;
            va_start(args, format);
            /* print out name of function causing error */
            fprintf(stderr, "ERROR in %s: ", function_name);
            /* print out remainder of message */
            vfprintf(stderr, format, args);
            va_end(args);
    }
```

vprintf Function

Synopsis:

```
    #include <stdarg.h>
    #include <stdio.h>
    int vprintf(const char *format, va_list arg);
```

Description:

The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which is initialized by the **va_start** macro (and possibly subsequent **va_arg** calls).

Returns:

The **vprintf** function returns the number of characters written to **stdout**, or a negative value if an output error occurred.

vsprintf Function

Synopsis:

```
    #include <stdarg.h>
    #include <stdio.h>
    int vsprintf(char *s, const char *format, va_list arg);
```

Description:

The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by **arg**, which is initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). Copying must not take place between objects that overlap.

Returns:

The **vsprintf** function returns the number of characters written into the array, not counting the terminating null character.

Character Input/Output Functions

The character input/output functions are described in the following subsections.

`fgetc` Function

Synopsis:

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description:

The `fgetc` function obtains the next character (if present) as an `unsigned char` converted to an `int`, from the input stream pointed to by `stream`, and advances the associated file position indicator for the stream.

Returns:

The `fgetc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetc` returns `EOF`. If a read error occurs, the error indicator for the stream is set and `fgetc` returns `EOF`.[10]

`fgets` Function

Synopsis:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Description:

The `fgets` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns:

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

[10] An end-of-file and a read error can be distinguished by use of the `feof` and `ferror` functions.

fputc Function

Synopsis:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description:

The **fputc** function writes the character specified by c (converted to an **unsigned char**) to the output stream pointed to by **stream**, at the position indicated by the associated file position indicator for the stream, and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns:

The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns EOF.

fputs Function

Synopsis:

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Description:

The **fputs** function writes the string pointed to by **s** to the stream pointed to by **stream**. The terminating null character is not written.

Returns:

The **fputs** function returns EOF if a write error occurs; otherwise it returns a non-negative value.

getc Function

Synopsis:

```
#include <stdio.h>
int getc(FILE *stream);
```

Description:

The getc function is equivalent to fgetc, except that since it is implemented as a macro, it evaluates stream more than once, so that argument should never be an expression with side effects.

Returns:

The getc function returns the next character from the input stream pointed to by stream. If the stream is at end-of-file, the end-of-file indicator for the stream is set and getc returns EOF. If a read error occurs, the error indicator for the stream is set and getc returns EOF.

getchar Function

Synopsis:

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

Description:

The getchar function is equivalent to getc with the argument stdin.

Returns:

The getchar function returns the next character from the input stream pointed to by stdin. If the stream is at end-of-file, the end-of-file indicator for the stream is set and getchar returns EOF. If a read error occurs, the error indicator for the stream is set and getchar returns EOF.

gets Function

Synopsis:

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

Description:

The **gets** function reads characters from the input stream pointed to by **stdin**, into the array pointed to by **s**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

Returns:

The **gets** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

putc Function

Synopsis:

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description:

The **putc** function is equivalent to **fputc**, except that since it is implemented as a macro, it evaluates **stream** more than once, so that argument should never be an expression with side effects.

Returns:

The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

putchar Function

Synopsis:

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

Description:

The putchar function is equivalent to putc with the second argument stdout.

Returns:

The putchar function returns the character written. If a write error occurs, the error indicator for the stream is set and putchar returns EOF.

puts Function

Synopsis:

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

Description:

The puts function writes the string pointed to by s to the stream pointed to by stdout, and appends a new-line character to the output. The terminating null character is not written.

Returns:

The puts function returns EOF if a write error occurs; otherwise it returns a non-negative value.

ungetc Function

Synopsis:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description:

The `ungetc` function pushes the character specified by `c` (converted to an `unsigned char`) back onto the input stream pointed to by `stream`. The pushed-back character will be returned by a subsequent read on that stream. A successful intervening call to a file positioning function (`fseek`, `fsetpos`, or `rewind`) with the same stream discards the pushed-back character for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is available. If the `ungetc` function is called more than once on the same stream without an intervening read or file positioning operation on that stream, the operation fails.

If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetc` function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding the pushed-back character is the same as it was before the character was pushed back. For a text stream, the value of its file position indicator after a successful call to the `ungetc` function is unspecified until the pushed-back character is read or discarded. For a binary stream, its file position indicator is decremented; if its value was zero before a call, it is indeterminate after the call.

Returns:

The `ungetc` function returns the character pushed back after conversion, or `EOF` if the operation fails.

Direct Input/Output Functions

The `fread` and `fwrite` functions are described below.

`fread` Function

Synopsis:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
      FILE *stream);
```

Description:

The `fread` function reads, into the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size`, from the stream pointed to by `stream`. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns:

The fread function returns the number of elements successfully read, which may be less than nmemb if a read error or end-of-file is encountered. If size or nmemb is zero, fread returns zero and the contents of the array and the state of the stream remain unchanged.

fwrite Function

Synopsis:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Description:

The fwrite function writes, from the array pointed to by ptr, up to nmemb elements whose size is specified by size, to the stream pointed to by stream. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Returns:

The fwrite function returns the number of elements successfully written, which will be less than nmemb only if a write error is encountered.

File Positioning Functions

The file positioning functions are described in the following subsections.

fgetpos Function

Synopsis:

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description:

The fgetpos function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored contains internal information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function.

Returns:

If successful, the fgetpos function returns zero. On failure, the fgetpos function returns nonzero and stores the (positive) error number value in errno.

fseek Function

Synopsis:

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

Description:

The fseek function sets the file position indicator for the stream pointed to by stream.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding offset to the position specified by whence. The specified position is relative to

- the beginning of the file if whence is SEEK_SET,
- the current value of the file position indicator if whence is SEEK_CUR, or
- end-of-file if whence is SEEK_END.

For a text stream, offset is either zero or a value returned by an earlier call to the ftell function on the same stream, and whence is SEEK_SET.

A successful call to the fseek function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fseek call, the next operation on an update stream may be either input or output.

Returns:

The fseek function returns nonzero only for a request that cannot be satisfied.

fsetpos Function

Synopsis:

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description:

The fsetpos function sets the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which must be a value obtained from an earlier call to the fgetpos function on the same stream.

A successful call to the fsetpos function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fsetpos call, the next operation on an update stream may be either input or output.

Returns:

If successful, the fsetpos function returns zero. On failure, the fsetpos function returns nonzero and stores the (positive) error number value in errno.

ftell Function

Synopsis:

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description:

The `ftell` function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains internal information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call. The difference between two such return values is not a meaningful measure of the number of characters written or read.

The position returned by `ftell` is only correct for positions within the first 2^{17} records and record positions within the first $2^{19} - 1$ bytes.

Returns:

If successful, the `ftell` function returns the current value of the file position indicator for the stream. On failure, the `ftell` function returns `-1L` and stores the (positive) error number value in `errno`.

rewind Function

Synopsis:

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description:

The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns:

The `rewind` function returns no value.

Error-Handling Functions

The error-handling functions are described in the following subsections.

clearerr Function

Synopsis:

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description:

The `clearerr` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

Returns:

The `clearerr` function returns no value.

feof Function

Synopsis:

```
#include <stdio.h>
int feof(FILE *stream);
```

Description:

The `feof` function tests the end-of-file indicator for the stream pointed to by `stream`.

Returns:

The `feof` function returns nonzero if and only if the end-of-file indicator is set for `stream`.

ferror Function

Synopsis:

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description:

The `ferror` function tests the error indicator for the stream pointed to by `stream`.

Returns:

The `ferror` function returns nonzero if and only if the error indicator is set for `stream`.

perror Function

Synopsis:

```
#include <stdio.h>
void perror(const char *s);
```

Description:

The perror function maps the error number in the integer expression errno to an error message. It writes a sequence of characters to the standard error stream thus: first (if s is not a null pointer and the character pointed to by s is not the null character), the string pointed to by s followed by a colon (:) and a space; then the error message string followed by a new-line character. The contents of the error message strings is the same as those returned by the strerror function with argument errno.

Returns:

The perror function returns no value.

Example:

The following C program

```
main() {
    FILE *fp;
    fp = fopen("**BOGUS**","r");
    perror("Example of perror()");
}
```

when run, results in the following output:

```
Example of perror():  **** FMN-M00113-2 File **BOGUS** does not exist in DP#SYS.X
```

Section 17

General Utility <stdlib.h> Functions

The header **<stdlib.h>** declares four types, several functions of general utility, and several macros.

The types declared are **size_t** and **wchar_t** (both described under Common Definitions, in Section 9),

  div_t

which is a structure type that is the type of the value returned by the **div** function, and

  ldiv_t

which is a structure type that is the type of the value returned by the **ldiv** function.

The macros defined include **NULL** (described under Common Definitions, in Section 9) and the following:

  EXIT_FAILURE

and

  EXIT_SUCCESS

which expand to integral expressions that may be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

  RAND_MAX

which expands to an integral constant expression, the value of which is the maximum value returned by the **rand** function; and

  MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), and whose value is never greater than **MB_LEN_MAX**.

String Conversion Functions

The string conversion functions are described in the following subsections.

The functions atof, atoi, and atol do not affect the value of the integer expression errno on an error.

atof Function

Synopsis:

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

Description:

The atof function converts the initial portion of the string pointed to by nptr to double representation. Except for not setting errno on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

Returns:

The atof function returns the converted value.

atoi Function

Synopsis:

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

Description:

The atoi function converts the initial portion of the string pointed to by nptr to int representation. Except for not setting errno on error, it is equivalent to

```
(int)strtol(nptr, (char **)NULL, 10)
```

Returns:

The atoi function returns the converted value.

atol Function

Synopsis:

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

Description:

The `atol` function converts the initial portion of the string pointed to by `nptr` to `long int` representation. Except for not setting `errno` on error, it is equivalent to

```
strtol(nptr, (char **)NULL, 10)
```

Returns:

The `atol` function returns the converted value.

strtod Function

Synopsis:

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

Description:

The `strtod` function converts the initial portion of the string pointed to by `nptr` to `double` representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function); a subject sequence resembling a floating-point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating-point number and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part as defined in Section 2, Lexical Elements, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules set forth in Floating Constants, in Section 2, except that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns:

The **strtod** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus HUGE_VAL is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the correct value would cause underflow, zero is returned and the value of the macro **ERANGE** is stored in **errno**.

strtol Function

Synopsis:

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description:

The **strtol** function converts the initial portion of the string pointed to by nptr to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function); a subject sequence resembling an integer represented in some radix determined by the value of **base**; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an integer and returns the result.

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described in Section 2, Lexical Elements, optionally preceded by a plus or minus sign, but not including an integer suffix. The value of **base** must be between 2 and 36; the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or A) through **z** (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules set forth in Integer Constants, in Section 2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns:

The **strtol** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MAX** or **LONG_MIN** is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**.

strtoul Function

Synopsis:

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr,
                          int base);
```

Description:

The **strtoul** function converts the initial portion of the string pointed to by **nptr** to **unsigned long int** representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function); a subject sequence resembling an unsigned integer represented in some radix determined by the value of **base**; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an unsigned integer and returns the result.

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described in Integer Constants, in Section 2, optionally preceded by a plus or minus sign, but not including an integer suffix. The value of **base** must be between 2 and 36; the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules set forth in Integer Constants, in Section 2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns:

The **strtoul** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **ULONG_MAX** is returned, and the value of the macro **ERANGE** is stored in **errno**.

Pseudo-Random Sequence Generation Functions

The **rand** and **srand** functions are described below.

rand Function

Synopsis:

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

Description:

The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX**.

Returns:

The **rand** function returns a pseudo-random integer.

srand Function

Synopsis:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description:

The `srand` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudo-random numbers is repeated. If `rand` is called before any calls to `srand` are made, the same sequence is generated as when `srand` is first called with a seed value of 1.

Returns:

The `srand` function returns no value.

Examples:

The following functions define a portable implementation of `rand` and `srand`:

```
static unsigned long int next = 1;

int rand(void)    /* RAND_MAX assumed to be 32767 */
{
        next = next * 1103515245 + 12345;
        return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
        next = seed;
}
```

Memory Management Functions

The ordering and location of the objects allocated by successive calls to the `calloc`, `malloc`, and `realloc` functions cannot be depended upon. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation yields a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the value returned is a null pointer. The value obtained when pointer refers to freed space is indeterminate.

The `calloc`, `free`, `malloc`, and `realloc` functions are described below.

`calloc` Function

Synopsis:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description:

The `calloc` function allocates space for an array of `nmemb` objects, each of whose size is `size`. The space is initialized to all bits zero.[1]

Returns:

The `calloc` function returns either a null pointer if the allocation cannot be made, or a pointer to the allocated space.

`free` Function

Synopsis:

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

Description:

The `free` function causes the space pointed to by `ptr` to be deallocated, that is, made available for further allocation. If `ptr` is a null pointer, no action occurs. Otherwise, the argument must match a pointer returned earlier by the `calloc`, `malloc`, or `realloc` function, and the space must not have been previously deallocated by a call to `free` or `realloc`.

Returns:

The `free` function returns no value.

`malloc` Function

Synopsis:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description:

The `malloc` function allocates space for an object whose size is specified by `size` and whose value is indeterminate.

Returns:

The `malloc` function returns either a null pointer if the allocation cannot be made, or a pointer to the allocated space.

[1] Note that this is not the same as the representation of floating-point zero or a null pointer.

realloc Function

Synopsis:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description:

The realloc function changes the size of the object pointed to by ptr to the size specified by size. The contents of the object are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If ptr is a null pointer, the realloc function behaves like the malloc function for the specified size. Otherwise, ptr must match a pointer returned earlier by the calloc, malloc, or realloc function, and the space must not have been previously deallocated by a call to free or realloc. If the space cannot be allocated, the object pointed to by ptr is unchanged. If size is zero and ptr is not a null pointer, the object it points to is freed.

Returns:

The realloc function returns either a null pointer if the allocation cannot be made, or a pointer to the possibly moved allocated space.

Communication with the Host Environment

The functions used in communication with the *CP-6* host environment are described in the following subsections.

abort Function

Synopsis:

```
#include <stdlib.h>
void abort(void);
```

Description:

In the absence of a SIGABRT signal handler, the abort function causes the program to terminate with abnormal status. If a SIGABRT handler is present, it can either terminate the program with abnormal status or longjmp to continue execution at a location specified by an earlier setjmp. Open output streams are flushed and all open streams are closed. The execution of the current program terminates.

Returns:

The abort function does not return to its caller.

atexit Function

Synopsis:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description:

The **atexit** function registers the function pointed to by **func**, to be called without arguments at normal program termination. Registered functions are called in the reverse order of their registration.

Implementation Limits:

CP-6 C supports the registration of 32 functions.

Returns:

The **atexit** function returns zero if the registration succeeds, nonzero if it fails.

exit Function

Synopsis:

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

Description:

The **exit** function causes normal program termination to occur. No more than one call to the **exit** function may be executed by a program.

First, all functions registered by the **atexit** function are called, in the reverse order of their registration.[2]

Next, all open output streams are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

Finally, control is returned to the *CP-6* operating system. If the value of **status** is zero or EXIT_SUCCESS, STEPCC is set to 0. If the value of **status** is EXIT_FAILURE, STEPCC is set to 4. If the status returned is an integer less than 512, STEPCC is set to that value; otherwise, **status** is treated as a *CP-6* error code.

Returns:

The **exit** function cannot return to its caller.

[2] Each function is called as many times as it was registered.

getenv Function

Synopsis:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Description:

The **getenv** function searches for an IBEX variable (see the *CP-6* Programmer Reference, CE40, for the IBEX LET command) that matches the string pointed to by **name**.

Returns:

The **getenv** function returns a pointer to a string containing the value of the variable. The string pointed to must not be modified by the program and will be overwritten by a subsequent call to the **getenv** function. If the specified **name** cannot be found, a null pointer is returned.

system Function

Synopsis:

```
#include <stdlib.h>
int system(const char *string);
```

Description:

The **system** function passes the string pointed to by **string** to the *CP-6* operating system for execution by the command processor, which is IBEX by default for time-sharing and batch users. A null pointer may be used for **string** to inquire whether a command processor exists.

Returns:

If the argument is a null pointer, the **system** function returns nonzero. If the argument is not a null pointer, the **system** function returns zero if the command executed successfully, or non-zero if an error has occurred.

Searching and Sorting Utilities

The **bsearch** and **qsort** functions are described below.

bsearch Function

Synopsis:

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Description:

The bsearch function searches an array of nmemb objects, the initial element of which is pointed to by base, for an element that matches the object pointed to by key. The size of each element of the array is specified by size.

The comparison function pointed to by compar is called with two arguments that point to the key object and to an array element, in that order. The function returns an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array element. The array consists of all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the key object, in that order.[3]

Returns:

The bsearch function returns a pointer to a matching element of the array, or a null pointer if no match is found.

qsort Function

Synopsis:

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description:

The qsort function sorts an array of nmemb objects, the initial element of which is pointed to by base. The size of each object is specified by size.

The contents of the array are sorted into ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared. The function returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Returns:

The qsort function returns no value.

[3] In practice, the entire array is sorted according to the comparison function.

Integer Arithmetic Functions

The abs, div, labs, and ldiv functions are described below.

abs Function

Synopsis:

```
#include <stdlib.h>
int abs(int j);
```

Description:

The abs function computes the absolute value of an integer j. If the result cannot be represented, an integer overflow occurs.[4]

Returns:

The abs function returns the absolute value.

div Function

Synopsis:

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Description:

The div function computes the quotient and remainder of the division of the numerator numer by the denominator denom. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The result quot * denom + rem equals numer.

Returns:

The div function returns a structure of type div_t, comprising both the quotient and the remainder. The structure contains the following members:

```
int quot;    /* quotient */
int rem;     /* remainder */
```

[4] The absolute value of the most negative number cannot be represented.

labs Function

Synopsis:

```
#include <stdlib.h>
long int labs(long int j);
```

Description:

The `labs` function is similar to the `abs` function, except that the argument and the returned value each have type `long int`.

ldiv Function

Synopsis:

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description:

The `ldiv` function is similar to the `div` function, except that the arguments and the members of the returned structure (which has type `ldiv_t`) all have type `long int`.

Multibyte Character Functions

The behavior of the multibyte character functions is affected by the `LC_CTYPE` category of the current locale. These functions are provided for compatibility with the ANSI standard since the *CP-6* system provides no multibyte characters.

The `mblen`, `mbtowc`, and `wctomb` functions are described below.

mblen Function

Synopsis:

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description:

If `s` is not a null pointer, the `mblen` function determines the number of bytes constituting the multibyte character pointed to by `s`. The `mblen` function is equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

Returns:

If `s` is a null pointer, the `mblen` function returns a zero value, since multibyte character encodings do not have state-dependent encodings. If `s` is not a null pointer, the `mblen` function returns 0 (if `s` points to the null character), returns the number of bytes that constitute the multibyte character (if the next `n` or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

mbtowc Function

Synopsis:

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description:

If s is not a null pointer, the **mbtowc** function determines the number of bytes constituting the multibyte character pointed to by s. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and **pwc** is not a null pointer, the **mbtowc** function stores the code in the object pointed to by **pwc**. At most n bytes of the array pointed to by s will be examined.

Returns:

If s is a null pointer, the **mbtowc** function returns a zero value, since multibyte character encodings do not have state-dependent encodings. If s is not a null pointer, the **mbtowc** function returns 0 (if s points to the null character), returns the number of bytes that constitute the converted multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than n or the value of the **MB_CUR_MAX** macro.

wctomb Function

Synopsis:

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description:

The **wctomb** function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is **wchar** (including any change in shift state). It stores the multibyte character representation in the array object pointed to by s (if s is not a null pointer). At most, **MB_CUR_MAX** characters are stored.

Returns:

If s is a null pointer, the **wctomb** function returns a zero value, since multibyte character encodings do not have state-dependent encodings. If s is not a null pointer, the **wctomb** function returns -1 if the value of **wchar** does not correspond to a valid multibyte character, or returns the number of bytes that constitute the multibyte character corresponding to the value of **wchar**.

In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

Multibyte String Functions

The behavior of the multibyte string functions is affected by the LC_CTYPE category of the current locale.

The mbstowcs and wcstombs functions are described below.

mbstowcs Function

Synopsis:

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description:

The mbstowcs function converts a sequence of multibyte characters from the array pointed to by s into a sequence of corresponding codes and stores not more than n codes into the array pointed to by pwcs. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the mbtowc function.

No more than n elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behavior is undefined.

Returns:

If an invalid multibyte character is encountered, the mbstowcs function returns (size_t)-1. Otherwise, the mbstowcs function returns the number of array elements modified,[5]not including a terminating zero code, if any.

wcstombs Function

Synopsis:

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description:

The wcstombs function converts a sequence of codes that correspond to multibyte characters from the array pointed to by pwcs into a sequence of multibyte characters and stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb function.

No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behavior is undefined.

[5] The array will not be null- or zero-terminated if the value returned is n.

Returns:

If a code is encountered that does not correspond to a valid multibyte character, the `wcstombs` function returns `(size_t)-1`. Otherwise, the `wcstombs` function returns the number of bytes modified, not including a terminating null character, if any.[6]

[6] The array will not be null- or zero-terminated if the value returned is n.

Section 18

String Handling <string.h> Functions

String Function Conventions

The header <string.h> declares one type, functions (described in this section), and one macro useful for manipulating arrays of character type and other objects treated as arrays of character type. The type is size_t and the macro is NULL (both described under Common Definitions, in Section 9). Various methods are used for determining the lengths of the arrays, but in all cases a char * or void * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Copying Functions

The copying functions are described in the following subsections.

memcpy Function

Synopsis:

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

Description:

The memcpy function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Returns:

The memcpy function returns the value of s1.

memmove Function

Synopsis:

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description:

The memmove function copies n characters from the object pointed to by s2 into the object pointed to by s1. Copying takes place by first copying the n characters from the object pointed to by s2 into a temporary array of n characters that does not overlap the objects pointed to by s1 and s2; then the n characters from the temporary array are copied into the object pointed to by s1.

Returns:

The memmove function returns the value of s1.

strcpy Function

Synopsis:

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

Description:

The strcpy function copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Returns:

The strcpy function returns the value of s1.

strncpy Function

Synopsis:

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

Description:

The strncpy function copies not more than n characters (characters that follow a null character are not copied) from the array pointed to by s2 to the array pointed to by s1.[1]If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by s2 is a string that is shorter than n characters, null characters are appended to the copy in the array pointed to by s1 until n characters in all have been written.

Returns:

The strncpy function returns the value of s1.

Concatenation Functions

The strcat and strncat functions are used to append a string to the end of another string.

strcat Function

Synopsis:

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

Description:

The strcat function appends a copy of the string pointed to by s2 (including the terminating null character) to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. If copying takes place between objects that overlap, the behavior is undefined.

Returns:

The strcat function returns the value of s1.

[1] Thus, if there is no null character in the first n characters of the array pointed to by s2, the result will not be null-terminated.

strncat Function

Synopsis:

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

Description:

The strncat function appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result.[2]If copying takes place between objects that overlap, the behavior is undefined.

Returns:

The strncat function returns the value of s1.

Comparison Functions

The sign of a nonzero value returned by the comparison functions memcmp, strcmp, and strncmp is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the objects being compared.

All of the comparison functions are described in the following subsections.

memcmp Function

Synopsis:

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

Description:

The memcmp function compares the first n characters of the object pointed to by s1 to the first n characters of the object pointed to by s2.[3]

Returns:

The memcmp function returns an integer greater than, equal to, or less than zero, relative to the object pointed to by s1 being greater than, equal to, or less than the object pointed to by s2.

[2] Thus the maximum number of characters that can be placed in the array pointed to by s1 is strlen(s1)+n+1.

[3] The contents of "holes" used as padding for alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions may also cause problems in comparison.

`strcmp` Function

Synopsis:

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

Description:

The `strcmp` function compares the string pointed to by `s1` to the string pointed to by `s2`.

Returns:

The `strcmp` function returns an integer greater than, equal to, or less than zero, relative to the string pointed to by `s1` being greater than, equal to, or less than the string pointed to by `s2`.

`strcoll` Function

Synopsis:

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

Description:

The `strcoll` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Returns:

The `strcoll` function returns an integer greater than, equal to, or less than zero, relative to the string pointed to by `s1` being greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale. In *CP-6* C this is equivalent to the `strcmp` function.

`strncmp` Function

Synopsis:

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

Description:

The `strncmp` function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Returns:

The `strncmp` function returns an integer greater than, equal to, or less than zero, relative to the possibly null-terminated array pointed to by `s1` being greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

strxfrm Function

Synopsis:

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

Description:

The strxfrm function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the strcmp function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the strcoll function applied to the same two original strings. No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. If n is zero, s1 may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns:

The strxfrm function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, the contents of the array pointed to by s1 are indeterminate.

Examples:

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by s:

```
1 + strxfrm(NULL, s, 0)
```

Search Functions

The search functions are described in the following subsections.

memchr Function

Synopsis:

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

Description:

The memchr function locates the first occurrence of c (converted to an unsigned char) in the initial n characters (each interpreted as unsigned char) of the object pointed to by s.

Returns:

The memchr function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

strchr Function

Synopsis:

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

Description:

The **strchr** function locates the first occurrence of c (converted to a **char**) in the string pointed to by s. The terminating null character is considered to be part of the string.

Returns:

The **strchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

strcspn Function

Synopsis:

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

Description:

The **strcspn** function computes the length of the maximum initial segment of the string pointed to by s1, which consists entirely of characters *not* from the string pointed to by s2.

Returns:

The **strcspn** function returns the length of the segment.

strpbrk Function

Synopsis:

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

Description:

The **strpbrk** function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

Returns:

The **strpbrk** function returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

strrchr Function

Synopsis:

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

Description:

The strrchr function locates the last occurrence of c (converted to a char) in the string pointed to by s. The terminating null character is considered to be part of the string.

Returns:

The strrchr function returns a pointer to the character, or a null pointer if c does not occur in the string.

strspn Function

Synopsis:

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

Description:

The strspn function computes the length of the maximum initial segment of the string pointed to by s1, which consists entirely of characters from the string pointed to by s2.

Returns:

The strspn function returns the length of the segment.

strstr Function

Synopsis:

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

Description:

The strstr function locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2

Returns:

The strstr function returns a pointer to the located string, or a null pointer if the string is not found. If s2 points to a string with zero length, the function returns s1.

strtok Function

Synopsis:

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

Description:

A sequence of calls to the strtok function breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a character from the string pointed to by s2. The first call in the sequence has s1 as its first argument and is followed by calls with a null pointer as their first argument. The separator string pointed to by s2 may be different from call to call.

The first call in the sequence searches the string pointed to by s1 for the first character that is *not* contained in the current separator string pointed to by s2. If no such character is found, then there are no tokens in the string pointed to by s1 and the strtok function returns a null pointer. If such a character is found, it is the start of the first token.

The strtok function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The strtok function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Returns:

The strtok function returns a pointer to the first character of a token, or a null pointer if there is no token.

Examples:

```
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");         /* t points to the token "a" */
t = strtok(NULL, ",");        /* t points to the token "??b" */
t = strtok(NULL, "#,");       /* t points to the token "c" */
t = strtok(NULL, "?");        /* t is a null pointer */
```

Miscellaneous Functions

The memset, strerror, and strlen functions are described below.

memset Function

Synopsis:

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

Description:

The memset function copies the value of c (converted to an unsigned char) into each of the first n characters of the object pointed to by s.

Returns:

The memset function returns the value of s.

strerror Function

Synopsis:

```
#include <string.h>
char *strerror(int errnum);
```

Description:

The strerror function maps the error number in errnum to an error message string.

Returns:

The strerror function returns a pointer to the string, the contents of which is the message associated with the error code. The array pointed to must not be modified by the program, and will be overwritten by a subsequent call to the strerror function.

strlen Function

Synopsis:

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

Description:

The strlen function computes the length of the string pointed to by s.

Returns:

The strlen function returns the number of characters that precede the terminating null character.

Section 19

Date and Time `<time.h>` Functions

Components of Time

The header `<time.h>` defines two macros, four types, and several functions (described in this section) for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are not supported by *CP-6* C.

The macros defined are `NULL` (described under Common Definitions, in Section 9); and

```
CLK_TCK
```

which is the number per second of the value returned by the `clock` function.

The types declared are `size_t` (described under Common Definitions, in Section 9);

```
clock_t
```

and

```
time_t
```

which are arithmetic types capable of representing times; and

```
struct tm
```

which holds the components of a calendar time, called the *broken-down time.* The structure contains at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[1]

```
int tm_sec;    /* seconds after the minute — [0, 61] */
int tm_min;    /* minutes after the hour — [0, 59] */
int tm_hour;   /* hours since midnight — [0, 23] */
int tm_mday;   /* day of the month — [1, 31] */
```

[1] The range [0, 61] for `tm_sec` allows for as many as two leap seconds.

```
    int tm_mon;    /* months since January — [0, 11] */
    int tm_year;   /* years since 1900 */
    int tm_wday;   /* days since Sunday — [0, 6] */
    int tm_yday;   /* days since January 1 — [0, 365] */
    int tm_isdst;  /* Daylight Saving Time flag */
```

The value of tm_isdst is negative since the information is not available.

Time Manipulation Functions

The time manipulation functions are described in the following subsections.

clock Function

Synopsis:

```
#include <time.h>
clock_t clock(void);
```

Description:

The clock function determines the processor time used.

Returns:

The clock function returns the processor time used by the program since the beginning of program execution. To determine the time in seconds, the value returned by the clock function should be divided by the value of the macro CLK_TCK.[2]

difftime Function

Synopsis:

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Description:

The **difftime** function computes the difference between two calendar times: **time1 - time0**.

Returns:

The **difftime** function returns the difference expressed in seconds as a **double**.

[2] In order to measure the time spent in a program, the clock function is called at the start of the program; then its return value is subtracted from the value returned by subsequent calls.

mktime Function

Synopsis:

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description:

The `mktime` function converts the broken-down time, expressed as local time, in the structure pointed to by `timeptr` into a calendar time value with the same encoding as that of the values returned by the `time` function. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the `tm_wday` and `tm_yday` components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above. The final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

Returns:

The `mktime` function returns the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t)-1`.

Examples:

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/*...*/

time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 1;
time_str.tm_isdst  = -1;
if (mktime(&time_str) == -1)
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

time Function

Synopsis:

```
#include <time.h>
time_t time(time_t *timer);
```

Description:

The `time` function returns the current calendar time. The encoding of the value is a *CP-6* Universal Time Stamp (UTS) which is milliseconds since January 1, 1978.

Returns:

The `time` function returns the current calendar time. If `timer` is not a null pointer, the return value is also assigned to the object it points to.

Time Conversion Functions

Except for the `strftime` function, these functions return values in one of two static objects: a broken-down time structure and an array of `char`. Execution of any of the functions will overwrite the information returned in either of these objects by any of the other functions.

The time conversion functions are described in the following subsections.

asctime Function

Synopsis:

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Description:

The `asctime` function converts the broken-down time in the structure pointed to by `timeptr` into a string in the form

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

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
        static const char wday_name[7][3] = {
              "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
        };
        static const char mon_name[12][3] = {
              "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
          };
          static char result[26];
          sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
                wday_name[timeptr->tm_wday],
                mon_name[timeptr->tm_mon],
                timeptr->tm_mday, timeptr->tm_hour,
                timeptr->tm_min, timeptr->tm_sec,
                1900 + timeptr->tm_year);
          return result;
     }
```

Returns:

The **asctime** function returns a pointer to the string.

ctime Function

Synopsis:

```
#include <time.h>
char *ctime(const time_t *timer);
```

Description:

The **ctime** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to:

```
asctime(localtime(timer))
```

Returns:

The **ctime** function returns the pointer returned by the **asctime** function with that broken-down time as argument.

gmtime Function

Synopsis:

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Description:

The **gmtime** function returns a **NULL** pointer since Coordinated Universal Time (UTC) is not available.

Returns:

The **gmtime** function returns a **NULL** pointer since UTC is not available.

localtime Function

Synopsis:

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Description:

The `localtime` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time.

Returns:

The `localtime` function returns a pointer to that object.

strftime Function

Synopsis:

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

Description:

The `strftime` function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. The `format` string consists of zero or more conversion specifiers and ordinary characters. A conversion specifier consists of a `%` character followed by a character that determines the behavior of the conversion specifier. All ordinary characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the values contained in the structure pointed to by `timeptr`.

`%a` is replaced by the locale's abbreviated weekday name.

`%A` is replaced by the locale's full weekday name.

`%b` is replaced by the locale's abbreviated month name.

`%B` is replaced by the locale's full month name.

`%c` is replaced by the locale's appropriate date and time representation.

`%d` is replaced by the day of the month as a decimal number (`01 - 31`).

`%H` is replaced by the hour (24-hour clock) as a decimal number (`00 - 23`).

`%I` is replaced by the hour (12-hour clock) as a decimal number (`01 - 12`).

%j is replaced by the day of the year as a decimal number (001 - 366).

%m is replaced by the month as a decimal number (01 - 12).

%M is replaced by the minute as a decimal number (00 - 59).

%p is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.

%S is replaced by the second as a decimal number (00 - 61).

%U is replaced by the week number of the year (Sunday as the first day of week 1) as a decimal number (00 - 53).

%w is replaced by the weekday as a decimal number (0 - 6; Sunday represented by 0).

%W is replaced by the week number of the year (Monday as the first day of week 1) as a decimal number (00 - 53).

%x is replaced by the locale's appropriate date representation.

%X is replaced by the locale's appropriate time representation.

%y is replaced by the year without century as a decimal number (00 - 99).

%Y is replaced by the year with century as a decimal number.

%Z is replaced by the time zone name or abbreviation, or by no characters if no time zone is determinable.

%% is replaced by %.

If a conversion specifier is not one of the above, the behavior is undefined.

Returns:

If the total number of resulting characters including the terminating null character is not more than **maxsize**, the **strftime** function returns the number of characters placed into the array pointed to by **s** not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Section 20

CP-6 Library Extensions

The run-time library provides a number of additional functions not found in ANSI C. These library functions extend capabilities in these areas: File Access, Memory Management, and Communication with Host Environment.

File Access Extensions

The extensions to file access capabilities are the `touch`, `fwildfid`, `finform` functions.

`touch` Function

Synopsis:

```
#include <stdio.h>
int touch(const char *filename);
```

Description:

The `touch` function causes the *CP-6 Last modified time* for the specified file to be changed to the current time.

Returns:

The `touch` function returns zero if the operation succeeds, nonzero if it fails.

`fwildfid` Function

Synopsis:

```
#include <stdio.h>
fwildfid(char *wildstr, int num, int wildch);
```

Description:

The `fwildfid` function returns the names of files in a *CP-6* directory, one at a time.

The first argument is a pointer to the wild-card file name string. The second argument is the number of filenames found which match the wild-card criteria. The third argument is the character used to represent wild-carding which is "?" typically on *CP-6* systems.

This function returns *CP-6* file identifiers which match the wild-carded file name string argument. A full *CP-6* input fid is supported but only the filename portion may be wild-carded. The result of this function is a pointer to the next file identifier which matches the search criteria.

The **num** argument is used to determine when the wild-card string has changed. If **num** is equal to 0, then the first file matching the search criteria is returned. The remaining files which match the search criteria are returned by calling this function with **num** set not equal to 0, but otherwise with the same arguments.

Returns:

The result of this function is a pointer to a character string which will be over-written by the next call.

Examples:

```
!b example_fwildfid:c
/*   Sample fwildfid() demonstration program        */

#include <stdio.h>

main(int argn, char *argv[]) {
  int i;
  if (argn < 2) {   /* Should be at least one wild-card */
     printf("usage: !%s wild [wild...]\n", argv[0]);
     exit(0);
     }
  for (i=1 ; i<argn ; )  {    /* process each wild-card */
     char *wild = argv[i++]; /* wild-card                */
     char *fid;              /* result fid               */
     int   fcnt = 0;         /* number matching wild     */
     while (fid = fwildfid(wild, fcnt++, ’?’))
        printf("%s\n", fid);
     }
```

```
   18.000 }
   19.000
!cc example_fwildfid:c over *:o,*:ls
CC.B00 here at 12:48 Wed Apr  4 1990
!link *:o over *RU
LINK E02G00 here
*  :SHARED_C.:SYS (Shared Library) associated.
*  Library file :LIB_SYSTEM.:SYS used.
*  No linking errors.
*  Total program size = 3K.
!*RU a?
a:c.XXX
a:o:do.XXX
a:y.XXX
align:c.XXX
!l a?
a:c         a:o:do      a:y         align:c
..    4 files listed
!
```

`finform` Function

Synopsis:

```
#include <stdio.h>
#include <cp_6_subs.h>
#include <fileinfo.h>
int finform(FILE *fp, FILE_SET *info);
```

Description:

This function provides information about the specified stream such as the full *CP-6* file identifier, the mode of opening, the organization of the file, the number of lines per page, the output width and many more attributes.

The include file `<cp_6_subs.h>` can be used to obtain the values of manifest constants for the various fields. This include file defines all of the names defined in the PL-6 include file `CP_6_SUBS.:LIBRARY`. Many of the fields returned by the `finform` function are unions which means that their value is only meaningful when correctly accessed. For example, the field `info->file_org.ur.width` is meaningful only if the field `info->org` is equal to the value of the `define CP6_UR` found in `cp_6_subs.h`.

Table 20-1 shows the fields stored in `info` and their meaning.

Field Name	Meaning
`info->fun`	Function used on M$OPEN
`info->acs`	ACS used on M$OPEN
`info->org`	ORG used on M$OPEN
`info->asn`	ASN used on M$OPEN
`info->fid`	*CP-6* file identifier
`info->file_org.ur.width`	Output width
`info->file_org.ur.lines`	Lines per page
`info->file_org.indexed.keyl`	Indexed file key length
`info->file_org.indexed.keyx`	Indexed file key position
`info->file_org.fprg.fprg`	`fprg` name
`info->file_org.fprg.profile`	`profile` name
`info->file_org.se.nrecs`	Number of seam records
`info->file_org.se.spare`	Number of requested records
`info->file_asn.file.nrecs`	Number of records in file
`info->file_asn.uc.wlen`	Terminal window length
`info->file_asn.uc.wwid`	Terminal window width
`info->file_asn.uc.winline`	Starting line of window
`info->file_asn.uc.wincol`	Starting column of window

Table 20-1. FILE_SET *Fields*

The information supplied in the result returned by **finform** depends upon the "**asn**" of the file and upon the "**org**" of the file. A union is used to select the appropriate union structure to interpret the result.

Returns:

The **finform** function returns zero if the operation succeeds, nonzero if it failed.

Examples:

```
!b example_finform:c
EDIT E02G00 here
    1.000 /*  Sample finform() demonstration program  */
```

```
     
     #include <cp_6_subs.h>
     #include <fileinfo.h>
     #include <stdio.h>
     
     main()  {
       FILE         *f = fopen("LP", "w ur");
       unsigned    lpp = -1, width = -1;
       FILE_SET  finfo;
     
       if (!finform(f, &finfo) && finfo.org == CP6_UR)   {
          lpp   = finfo.file_org.ur.lines;
          width = finfo.file_org.ur.width;
          }
     
       printf("Width=%d,Lines=%d\n", width, lpp);
     }
     
!cc example_finform:c over *:o,*:ls
CC.B00 here at 12:31 Wed Apr  4 1990
!link *:o over *RU
LINK E02G00 here
*  :SHARED_C.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!*RU
Width=110,Lines=39
!
```

Memory Management Extensions

The extension to Memory Management is the **alloca** function.

alloca Function

Synopsis:

```
#include <stdlib.h>
void *alloca(size_t size);
```

Description:

The **alloca** function allocates space for an object whose size is specified by **size** in the current **auto** frame, whose allocation is automatically freed when the currently executing function **returns**. The value of the object is indeterminate.

Returns:

The **alloca** function returns a **NULL** pointer if the allocation cannot be made, or a pointer to the allocated space.

Virtual Memory Management Extensions

These functions are based upon the *Memory Management Functions* in Section 7. They provide access to a *CP-6* virtual data segment which allows the user to access much more memory than the 256k words available through the standard routines. With the exception of the **vmeminit** function, these functions work exactly like the same function without the leading "v". The **vmeminit** function must be executed before any of the other virtual memory functions are used.

The ordering and location of objects allocated by successive calls to **vcalloc**, **vmalloc**, and **vrealloc** cannot be depended upon. In particular, the memory allocated by successive executions of these functions may be in different *CP-6* data segments which makes pointer comparisons and pointer differences misleading other than for comparisons for equality or for the **NULL** pointer value. The pointer returned when the allocation succeeds is suitably aligned so that it may be assigned to a pointer of any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Every allocation yields a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest address) of the allocated space. If the space cannot be allocated, a **NULL** pointer is returned. The maximum allocation size of a single block of memory is 1048552 characters. If the space requested is zero, a **NULL** pointer is returned.

vcalloc Function

Synopsis:

```
#include <valloc.h>
void *vcalloc(size_t nmemb, size_t size);
```

Description:

The **vcalloc** function allocates space for an array of **nmemb** objects, each of whose size is **size** characters. The space is initialized to all bits zero.[1]

Returns:

The **vcalloc** function returns a **NULL** pointer if the allocation cannot be made, or a pointer to the allocated space.

[1] Note that this is not the same as the representation of floating-point zero or a NULL pointer.

vfree Function

Synopsis:

```
#include <valloc.h>
void vfree(void *ptr);
```

Description:

The vfree function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a NULL pointer, no action occurs. If the argument does not match a pointer earlier returned by the vcalloc, vmalloc or vrealloc functions, or if the space has been deallocated by a call to vfree or vrealloc function is indeterminate and likely to seriously damage the virtual heap.

Returns:

The vfree function returns no value.

vmalloc Function

Synopsis:

```
#include <valloc.h>
void *vmalloc(size_t size);
```

Description:

The vmalloc function allocates space for an object whose size is specified by size and whose value is indeterminate.

Returns:

The vmalloc function returns a NULL pointer if the allocation cannot be made, or a pointer to the allocated space.

vmeminit Function

Synopsis:

```
#include <valloc.h>
int vmeminit(size_t vspace, int vtype, size_t vbacking);
```

Description:

This function must be called before any of the other virtual memory management functions. Its purpose is to initialize the *CP-6* virtual data segment that will be used for allocation. The total amount of virtual memory available for allocation will be vspace pages of 4096 bytes. The minimum virtual memory backing is 3 pages of 4096 characters. The actual number of pages used to back the virtual segment is selected by the vtype argument whose value selects one of the following algorithms:

1) the backing memory will be `vbacking` physical pages;
2) the backing memory will be the total number of pages currently available to this user minus `vbacking` (subject to the minimum of 3 backing pages);
3) the backing memory will be the percentage of the available memory as indicated by the value of `vbacking`. If the `vbacking` value is 80 then 80 percent of available memory will be used to back the virtual data segment (subject to the minimum of 3 backing pages).

Returns:

The `vmeminit` function returns zero if successful.

`vmemscrub` Function

Synopsis:

```
#include <valloc.h>
void vmemscrub(void);
```

Description:

The `vmemscrub` function is used to release all memory blocks that are currently allocated in the virtual data segment. Pointers to these objects become obsolete and the associated memory will be re-allocated on future calls to `vmalloc`, `vrealloc` or `vcalloc`.

Returns:

The `vmemscrub` function returns no value.

`vrealloc` Function

Synopsis:

```
#include <valloc.h>
void *vrealloc(void *ptr, size_t size);
```

Description:

The `vrealloc` function changes the size of the object pointed to by `ptr` to the size specified by `size`. The contents of the object will be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If `ptr` is a `NULL` pointer, the `vrealloc` function behaves like the `vmalloc` function. Otherwise, if `ptr` does not match a pointer previously returned by the `vcalloc`, `vmalloc` or `vrealloc` function, or if the space has been deallocated by a previous call to the `vfree` or `vrealloc` function, the behavior is indeterminate. If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and `ptr` is not a `NULL` pointer, the object that it points to is freed.

Returns:

The `vrealloc` function returns either a `NULL` pointer or a pointer to the possibly moved allocated space.

Communication with Host Environment Extensions

The extensions to Communication with the Host Environment include the getopt, lsenv, sleep, ulimit, and uname functions.

getopt Function

Synopsis:

```
#include <stdlib.h>
int getopt(int argc, char *argv[], char *opstr);
extern char *optarg;
extern int   optind;
extern int   opterr;
```

Description:

The getopt function is a command line parser. It returns the next option letter in argv that matches a letter in opstr. opstr is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. The extern variable optarg is set to point to the start of the option argument on return from getopt.

getopt places the index of the next argument to be processed in optind. This variable must be initialized to 1 before the first call to to getopt.

When all options have been processed (i.e. up to the first non-option argument), getopt returns EOF. The special option "--" may be used to delimit the end of the options; in this case, EOF will be returned when "--" would be processed and the "--" will be skipped.

Returns:

The getopt function prints an error message on stderr and returns a question mark ('?') when it encounters an option letter not included in opstring. This error message may be disabled by setting opterr to zero. If the option letter was found in opstring then the getopt function returns the letter.

lsenv Function

Synopsis:

```
#include <stdlib.h>
char *lsenv(char *name);
```

Description:

The `lsenv` function searches for the next IBEX variable after the argument **name**. If there is none, a **NULL** pointer is returned. To get the first command variable name, a string of zero length is used.

Returns:

The **lsenv** function returns a pointer to the name of the next command variable. The string pointed to must not be modified by the program and will be overwritten by a subsequent call to the `lsenv` function. If there is no next command variable, a null pointer is returned.

Example:

The following example program prints the names of all of the IBEX variables that are currently defined.

```
!b example_lsenv:c
EDIT E02G00 here
    1.000 /*   Example of lsenv function      */
    2.000 #include <stdlib.h>
    3.000
    4.000 main()  {
    5.000   char *var = lsenv("");
    6.000
    7.000   if (var)
    8.000      do {
    9.000         printf("%s\n", var);
   10.000         } while (var = lsenv(var));
   11.000   else printf("No IBEX variables\n");
   12.000 }
   13.000
!cc example_lsenv:c over *:o,*:ls
CC.B00 here at 17:01 Mon Apr  9 1990
!link *:o over *RU (unsat=:LIB_C.:SYS)
LINK E02G00 here
*  :SHARED_C.:SYS (Shared Library) associated.
*  Library file :LIB_C.:SYS used.
*  No linking errors.
*  Total program size = 3K.
!*RU
LAST_LOGON_TIME
LOGON_FAILURE_COUNT
!
```

`sleep` Function

Synopsis:

```
#include <stdlib.h>
unsigned sleep(unsigned seconds);
```

Description:

The `sleep` function takes `seconds` seconds to complete execution. The actual number of seconds that `sleep` may be less than the requested because any signal will terminate the sleep. Also the suspended time may be longer due to the scheduling of other tasks on the system.

Returns:

The `sleep` function returns the number of seconds remaining to be slept. A nonzero value occurs when a signal caused termination of the sleep (`<CTL-Y>go` can also cause this to happen).

`ulimit` Function

Synopsis:

```
#include <stdlib.h>
unsigned ulimit(int cmd);
```

Description:

The `ulimit` function returns the maximum number of bytes available to be allocated by this process. The result value does not include space available through the `malloc` function that has been allocated but is not currently used. The argument `cmd` must have the value `3`.

Returns:

The `ulimit` function returns the number of bytes that are currently available for allocation from the *CP-6* system. `ulimit` reports an error condition by returning a value less than zero.

`uname` Function

Synopsis:

```
#include <uts_name.h>
int uname(struct utsname *name);
```

Description:

The `uname` function returns information identifying the current *CP-6* system in the structure pointed to by `name`. The `utsname` structure is defined in the include file `uts_name.h` and contains the following fields:

`sysname` is a string containing the site name.

`nodename` is a string containing the unique `SITE ID` assigned by *Bull.*

`version` is a string containing the version of the *CP-6* operating system.

`release` is a string containing the patch level of the system.

`machine` is a string containing the name of the cpu.

Returns:

The `uname` function returns a non-negative value upon successful completion; otherwise, it returns `-1`.

Appendix A

Language Syntax Summary

Lexical Grammar

Tokens

token:
  keyword
  identifier
  constant
  string-literal
  operator
  punctuator

preprocessing-token:
  header-name
  identifier
  pp-number
  character-constant
  string-literal
  operator
  punctuator
  each non-white-space character that cannot be one of the above

Keywords

keyword: one of

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

Identifiers

identifier:
  nondigit
  identifier nondigit
  identifier digit

nondigit: one of

```
_  a  b  c  d  e  f  g  h  i  j  k  l  m
   n  o  p  q  r  s  t  u  v  w  x  y  z
   A  B  C  D  E  F  G  H  I  J  K  L  M
   N  O  P  Q  R  S  T  U  V  W  X  Y  Z
$
```

digit: one of

```
0  1  2  3  4  5  6  7  8  9
```

Constants

constant:
  floating-constant
  integer-constant
  enumeration-constant
  character-constant

floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ `.` *digit-sequence*
  digit-sequence `.`

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  `f l F L`

integer-constant:
  decimal-constant integer-suffix$_{opt}$
  octal-constant integer-suffix$_{opt}$
  hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
  nonzero-digit
  decimal-constant digit

octal-constant:
  `0`
  octal-constant octal-digit

hexadecimal-constant:
  `0x` *hexadecimal-digit*
  `0X` *hexadecimal-digit*
  hexadecimal-constant hexadecimal-digit

nonzero-digit: one of
  `1 2 3 4 5 6 7 8 9`

octal-digit: one of
  `0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
  `0 1 2 3 4 5 6 7 8 9`
  `a b c d e f`
  `A B C D E F`

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
  `u U`

long-suffix: one of
  `l L`
enumeration-constant:
  identifier
character-constant:
  `'`*c-char-sequence* `'`
  `L'`*c-char-sequence* `'`

c-char-sequence:
  c-char

c-char-sequence c-char

c-char:
any member of the source character set except
the single quote ('), backslash (\\), or new-line character
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\'  \"  \?  \\`
`\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
\\ *octal-digit*
\\ *octal-digit octal-digit*
\\ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

String Literals

string-literal:
"*s-char-sequence*$_{opt}$"
L"*s-char-sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ("), backslash (\\), or new-line character
escape-sequence

Operators

operator: one of

```
[   ]   (   )   .   ->
++   --   &   *   +   -   ~   !   sizeof
/   %   <<   >>   <   >   <=   >=   ==   !=   ^   |   &&   ||
?   :
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
,   #   ##
```

Punctuators

punctuator: one of

```
[   ]   (   )   {   }   *   ,   :   =   ;   ...   #
```

Header Names

header-name:
    <*h-char-sequence*>
    "*q-char-sequence*"

h-char-sequence:
    h-char
    h-char-sequence h-char

h-char:
    any member of the source character set except
        the new-line character and >

q-char-sequence:
    q-char
    q-char-sequence q-char

q-char:
    any member of the source character set except
        the new-line character and "

Preprocessing Numbers

pp-number:
- *digit*
- . *digit*
- *pp-number digit*
- *pp-number nondigit*
- *pp-number* e *sign*
- *pp-number* E *sign*
- *pp-number* .

Phrase Structure Grammar

Expressions

primary-expression:
- *identifier*
- *constant*
- *string-literal*
- (*expression*)

postfix-expression:
- *primary-expression*
- *postfix-expression* [*expression*]
- *postfix-expression* (*argument-expression-list*$_{opt}$)
- *postfix-expression* . *identifier*
- *postfix-expression* -> *identifier*
- *postfix-expression* ++
- *postfix-expression* --

argument-expression-list:
- *assignment-expression*
- *argument-expression-list* , *assignment-expression*

unary-expression:
- *postfix-expression*
- ++ *unary-expression*
- -- *unary-expression*
- *unary-operator cast-expression*
- sizeof *unary-expression*
- sizeof (*type-name*)

unary-operator: one of

`& * + - ~ !`

cast-expression:
- *unary-expression*
- (*type-name*) *cast-expression*

multiplicative-expression:
    cast-expression
    multiplicative-expression * *cast-expression*
    multiplicative-expression / *cast-expression*
    multiplicative-expression % *cast-expression*
additive-expression:
    multiplicative-expression
    additive-expression + *multiplicative-expression*
    additive-expression - *multiplicative-expression*
shift-expression:
    additive-expression
    shift-expression >> *additive-expression*
    shift-expression << *additive-expression*
relational-expression:
    shift-expression
    relational-expression < *shift-expression*
    relational-expression > *shift-expression*
    relational-expression <= *shift-expression*
    relational-expression >= *shift-expression*
equality-expression:
    relational-expression
    equality-expression == *relational-expression*
    equality-expression != *relational-expression*
AND-expression:
    equality-expression
    AND-expression & *equality-expression*
exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ *AND-expression*
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | *exclusive-OR-expression*
logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && *inclusive-OR-expression*
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || *logical-AND-expression*
conditional-expression:
    logical-OR-expression
    logical-OR-expression ? *expression* : *conditional-expression*
assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
assignment-operator: one of
    = *= /= %= += -= <<= >>= &= ^= |=

expression:
    assignment-expression
    expression , *assignment-expression*
constant-expression:
    conditional-expression

Declarations

declaration:
    declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
    init-declarator
    init-declarator-list , *init-declarator*

init-declarator:
    declarator
    declarator = *initializer*
storage-class-specifier:
    `typedef`
    `extern`
    `static`
    `auto`
    `register`
type-specifier:
    `void`
    `char`
    `short`
    `int`
    `long`
    `float`
    `double`
    `signed`
    `unsigned`
    struct-or-union-specifier
    enum-specifier
    typedef-name
struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
    struct-or-union identifier

struct-or-union:
  `struct`
  `union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

struct-declaration:
  specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{opt}$
  type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
  struct-declarator
  struct-declarator-list **,** *struct-declarator*

struct-declarator:
  declarator
  declarator$_{opt}$: *constant-expression*
enum-specifier:
  enum *identifier*$_{opt}$ { *enumerator-list* }
  enum *identifier*

enumerator-list:
  enumerator
  enumerator-list **,** *enumerator*

enumerator:
  enumeration-constant
  enumeration-constant = *constant-expression*
type-qualifier:
  `const`
  `volatile`
declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  (*declarator*)
  direct-declarator [*constant-expression*$_{opt}$]
  direct-declarator (*parameter-type-list*)

direct-declarator (*identifier-list*$_{opt}$)

pointer:
* * *type-qualifier-list*$_{opt}$
* * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list
parameter-list , . . .

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list , *identifier*
type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)
typedef-name:
identifier
initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* , }

initializer-list:
initializer
initializer-list , *initializer*

Statements

statement:
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement

labeled-statement:
  identifier : *statement*
  case *constant-expression* : *statement*
  default : *statement*

compound-statement:
  { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
  declaration
  declaration-list declaration

statement-list:
  statement
  statement-list statement

expression-statement:
  expression$_{opt}$;

selection-statement:
  if (*expression*) *statement*
  if (*expression*) *statement* **else** *statement*
  switch (*expression*) *statement*

iteration-statement:
  while (*expression*) *statement*
  do *statement* **while** (*expression*) ;
  for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

jump-statement:
  goto *identifier* ;
  continue ;
  break ;
  return *expression*$_{opt}$;

External Definitions

object-unit:
  external-declaration
  object-unit external-declaration

external-declaration:
  function-definition
  declaration

function-definition:
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

Preprocessing Directives

preprocessing-file:
  group$_{opt}$

group:
  group-part
  group group-part

group-part:
  pp-tokens$_{opt}$ *new-line*
  if-section
  control-line

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
  # if *constant-expression new-line group*$_{opt}$
  # ifdef *identifier new-line group*$_{opt}$
  # ifndef *identifier new-line group*$_{opt}$

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # elif *constant-expression new-line group*$_{opt}$

else-group:
  # else *new-line group*$_{opt}$

endif-line:
  # endif *new-line*

control-line:

```
# include pp-tokens new-line
# define  identifier replacement-list new-line
# define  identifier lparen identifier-list_opt ) replacement-list new-line
# undef   identifier new-line
# line    pp-tokens new-line
# error   pp-tokens_opt new-line
# pragma  pp-tokens_opt new-line
#         new-line
```

lparen:
the left parenthesis character without preceding white space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

Appendix B

Library Summary

This appendix summarizes the *CP-6* C library macros, types, and functions.

Errors `<errno.h>`

```
EDOM
ERANGE
errno
```

Common Definitions `<stddef.h>`

```
NULL
offsetof(type, member-designator)
ptrdiff_t
size_t
wchar_t
```

Diagnostics `<assert.h>`

```
NDEBUG
void assert(int expression);
```

Character Handling `<ctype.h>`

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

Localization <locale.h>

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
NULL
struct lconv
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);
```

Mathematics <math.h>

```
HUGE_VAL
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
double cosh(double x);
double sinh(double x);
double tanh(double x);
double exp(double x);
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
double pow(double x, double y);
double sqrt(double x);
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

Non-Local Jumps <setjmp.h>

```
jmp_buf
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Signal Handling <signal.h>

```
sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGALRM
SIGFPE
SIGHUP
SIGILL
SIGINT
SIGSEGV
SIGTERM
SIGUSR1
SIGUSR2
void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```

Variable Arguments <stdarg.h>

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

Input/Output <stdio.h>

```
_IOFBF
_IOLBF
_IONBF
BUFSIZ
EOF
FILE
FILENAME_MAX
FOPEN_MAX
fpos_t
L_tmpnam
NULL
SEEK_CUR
SEEK_END
SEEK_SET
size_t
stderr
stdin
stdout
```

```
TMP_MAX
int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode,
      FILE *stream);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int scanf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
int touch(const char *filename);
char *fwildfid(const char *filename, int num, int wildch);
int finform(FILE *stream, FILE_SET *info);
```

General Utilities <stdlib.h>

```
EXIT_FAILURE
EXIT_SUCCESS
MB_CUR_MAX
NULL
RAND_MAX
div_t
ldiv_t
size_t
wchar_t
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr,
                          int base);
int rand(void);
void srand(unsigned int seed);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
void *alloca(size_t size);
int getopt(int argc, char *argv[], char *opstr);
char *lsenv(char *name);
unsigned ulimit(int cmd);
unsigned sleep(int seconds);
```

String Handling <string.h>

```
NULL
size_t
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *s1, const char *s2, size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

Date and Time <time.h>

```
CLK_TCK
NULL
clock_t
time_t
size_t
struct tm
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

System Information <uts_name.h>

```
struct uts_name
int uname(struct utsname *name);
```

Virtual Memory Management <valloc.h>

```
void *vcalloc(size_t nmemb, size_t size);
void vfree(void *ptr);
void *vmalloc(size_t size);
void vmeminit(size_t vspace, int vtype, size_t vbacking);
void vmemscrub(void);
void *vrealloc(void *ptr, size_t size);
```

Appendix C

Debugging C Programs

The purpose of this section is to give an overview of the *CP-6* debugging system, DELTA, for use as an aid in debugging C programs. For details regarding items in this section the DELTA manual or HELP facility should be consulted.

DELTA is a self-contained external debugging system which does not require specially compiled versions of a program for debugging. The same program which will be used for production can be debugged with DELTA; the use of DELTA does not require the program to be any larger or slower.

Running a program under DELTA allows the user to control and observe its execution. DELTA makes it possible to stop the program at any point (procedure breakpoints), trace the program flow (all transfers, or just subroutine and function calls) and allows the user to examine the values of program variables anytime, by name. The user can also change the value of a variable and test the value of a variable using the result to conditionally specify the actions of DELTA. DELTA also allows the user to step through the program one statement at a time to help pinpoint a problem. DELTA interfaces with the monitor's fault handler, catching any trap or fault caused by the program, reporting the fault, and allowing the user to examine and/or change the value of program variables and perhaps continue execution of the program after a fault occurs.

DELTA Overview

This section is presented in two parts. The first part is an overview of DELTA as it applies to the C programmer. The second part contains a summary of the available DELTA commands.

Invoking DELTA

DELTA may be entered at three times during the life of a program: as it starts execution, during execution and after a program aborts.

Most debugging sessions are begun by starting the program to be debugged under DELTA. This is accomplished by using the IBEX u command:

```
!u
!fid.
```

This command brings a run unit named *fid* into memory and prepares it for execution. DELTA is entered with the user program ready to run. DELTA prints the current value of the instruction counter, the program start address.

Example:

```
!u
!c_rununit.
DELTA xxx HERE IC = main:0 [INITIALIZE]
>
```

This method can be used on-line and in batch. The user may now use any of the DELTA commands, or just say go to begin execution. IC is the Instruction Counter and indicates the address that is about to be executed.

Example:

```
!u
!mycomp. me on oufile,lp (ls,ou)
DELTA xxx HERE IC = main:0 [INITIALIZE]
>
```

DELTA may be associated with a running program at any time. This is useful when a program appears to be looping or is in a bad or unexpected state, and the user wants to interrupt execution and see which statement is in execution, or look at program variables. To associate DELTA after the program has been started type <CTRL><Y>. IBEX prompts with a double bang (!!). Type DELTA to associate DELTA with the interrupted program.

Example:

```
!MYPROG.
```

Program does not prompt as expected, loop is suspected. User enters <CTRL><Y>.

```
!!DELTA
DELTA xxx HERE IC = initvals:54,,.3 [ASSIGNMENT]
>
```

This method is only available on-line.

When a user program aborts, IBEX holds the image of the run unit in memory. The user can associate DELTA with the image by typing DELTA immediately after the abort message.

Example:

```
!*RU
SECURITY 2 FAULT @ strlen_+.224/ SCM (AR,RL,0),(0,0,0,.3),(0,0,0),.0
**** C run-time error
**** Exceptional condition "Hardware_Detected_Fault" occurred
     at location
strlen_+.224/ SCM (AR,RL,0),(0,0,0,.3),(0,0,0),.0
**** HFA-M00521-6 Security 2 fault
     Traceback follows:
XBI_RAISE_SIGNAL+.23/ TSX1 XBI_DEFAULT_SIGNAL_HANDLER (+.1250)
XBI_PRINT+.1634/ TSX1 strlen_+.3( XBI$STRLEN_ ) (+.1242)
printf_+.33/ TSX1 XBI_PRINT (+.56)
main:4,,.14 [CALL] / TSX1 printf
main:0,,.1 [INITIALIZE] / TSX0 __XBI_CSTARTUP
Bottom frame
  M$ERR issued by user.
!DELTA
DELTA E02F00 here IC = strlen_+.224
>
```

DELTA reports the IC value at the time of the fault. Any DELTA command can be issued, and the program can be continued by the go command.

If DELTA is already associated, depressing the break key will cause the running program to be interrupted. DELTA will report the current IC position and prompt for input.

DELTA uses the prompt character > when it is waiting for the user to input a command at the terminal. All commands should be ended with the <RETURN> key on the keyboard. Do not use <LINEFEED>.

By default, DELTA uses a special End-of-Message (EOM) character set (see Table C-8). These are characters which terminate the input mode and activate DELTA. The complete list of DELTA EOM characters are: |, <LINEFEED>, <TAB>, <RETURN>, [, {,], }, /, and =. Most of the special EOM characters are used only when debugging at the assembly or machine language level. However,] and }, which have the same effect, are a shorthand for the `step` command and are very convenient for stepping through a program. In order to enter any of these characters without activation to DELTA, for instance in a character string constant for the `let` command, DELTA's special activation set must be turned off. This is done with the `kill eom` command. The `eom` command turns the special activation set back on.

DELTA Summary of Commands

Tables C-1 through C-6 display the repertoire of DELTA commands. During a debugging session, the user controls the execution of the run unit. This control is achieved primarily with the `at`, `on`, and **when** commands. These three commands are referred to collectively as *Breakpoint Commands.* Since these commands allow DELTA to assume control of the run unit, the remainder of the DELTA commands may be considered as complements to them.

The commands issued to DELTA are acted upon at different times. Some are executed immediately; others are stored for subsequent execution based upon some specific occurrence, and some are executed in conjunction with those which have been stored. DELTA commands can assume one or more of the following attributes:

1. Stored

2. Attached

3. Immediate

4. Toggled

DELTA Stored Commands

Stored commands instruct DELTA to perform an action at some later time based upon the arrival of the Instruction Counter at a specific location or upon the occurrence of some specific event (the `at`, `on`, and **when** commands primarily). When issued, these commands are assigned an identification number (id) either by the user or by DELTA if the user does not; this is included in DELTA's report to the user each time the command is activated. Stored commands may have other commands attached to them which are to be executed whenever the stored command is activated.

General Syntax:

A stored command and its attachments are issued in the form:

[id] [stored_command][; attachment[; attachment]...]

The command line ends with a carriage return or end-of-record.

Example:

```
10 at 200;display netpay,grosspay,deductions  <RET>
```

Notes:

1. The stored command "at 200" was assigned an id of "10" by the user. If a stored command with the same id already existed, it would be replaced.
2. The single "display" attachment will display three variables within the user's run unit: netpay, grosspay, and deductions.
3. The semicolon (;) is used to separate attachments.

Note: Stored commands can be continued onto a new line if the previous line ends with a semicolon and the new line begins with an attachment.

In the above example, the programmer directs DELTA to set a breakpoint at statement 200 in the current procedure. When the Instruction Counter reaches this location, DELTA will report the breakpoint at the user's terminal and then display the requested variables. If the user does not wish to interact with DELTA following the display, the command can be written:

```
10 at 200;display netpay,grosspay,deductions;go
```

With the command written in this form, DELTA will not stop after performing the display but will cause the run unit to resume execution as though the breakpoint has not occurred.

DELTA Conditional Execution

All stored commands may be formed to specify varying levels of conditional execution by using the if conditional. For example:

```
at payroll:22 if netpay eq 0;display deduct;go
```

In this example, if specifies conditional execution. When line 22 of the subroutine payroll is reached, netpay will be examined to determine if its value is zero. If its value is not zero, DELTA will not report to the user nor will any of the attachments be executed. The program will continue as if no breakpoint has been specified.

Format:

`if` *var logical_operator cons*

Parameters:

var is any variable reference. Variable references are described in detail under Display and Modification of Program Variables in this section.

logical_operator is any of:

Operator	Meaning
= eq	Equal
>< <> ne	Not equal
< lt	Less than
> gt	Greater than
<= =< le	Less than or equal
>= => ge	Greater than or equal

cons is any constant literal: an integer, bit, pointer or character string. The format for constants is completely specified in the description of the `let` command under Display and Modification of Program Variables.

NOTE: Remember that the character = cannot be used unless the `kill eom` command has been issued. Instead, the mnemonic version (e.g., `eq`, `ge`) may be used.

The value described by var is compared with the constant value. If the logical relation is true, the stored command is reported and its attachments executed (if the condition was on a stored command), or the attachment is executed (if the condition was on an attachment).

Specification of an `if` condition is allowed on almost all DELTA commands. Specific instances where the `if` is not allowed are noted in the description of the commands.

Taking into account the `if` condition, the complete specification of stored commands is:

*[id] **stored_command** [*`if` *var logical_operator cons]*

*[;**attachment** [*`if` *var logical_operator cons]*

*[;**attachment** [*`if` *var logical_operator cons]] ...]*

DELTA Attached Commands

Most DELTA commands may be issued as attachments to any stored commands. The exceptions are **step**, **xeq**, and all stored commands.

DELTA Immediate Commands

A command (other than a stored command) assumes the immediate attribute whenever it is issued in-line. In the interactive mode this occurs whenever DELTA has issued its prompt character (>). In the batch mode this occurs whenever DELTA reads a command.

Example:

```
>display netpay
```

In this context, **display** is being used in an immediate way.

Displaying Variables

Since DELTA does not understand the C language syntax, the C compiler generates schema compatible with a PL1-like language. The result is that variables declared as pointers display as pointers under DELTA, and to obtain the value they point at, the user must "de-reference" them with PL1 syntax. For example, to display the value that the integer pointer variable **intp** points at, the user would enter:

```
>display intp->0\s
intp->0 = 123
```

For pointers to C **struct** variables, the **struct** name is used (with a minor change) and must appear before the field name to display. The structure name is renamed slightly so that there will not be a name clash between the name of the structure and that of another variable. The renamed structure begins with the string "**s_**" and ends with the single character "**#**". For example, if the following declaration is found in a C source file:

```
struct   example  {
   struct example *next;
   int            ex_val;
   char           ex_name[8];
   }  *exp;
```

the following commands may be entered in DELTA to display the value of the **exp** variable:

```
>display exp->s_example#.next
exp->s_example#.next = .31274-0-0,$LS6
>display exp->s_example#.ex_val
exp->s_example#.ex_val = 8
>display exp->s_example#.ex_name(3:5)
exp->s_example#.ex_name(3) = 'd'
exp->s_example#.ex_name(4) = 'a'
exp->s_example#.ex_name(5) = 'y'
```

Static Functions

When functions are declared to be `static`, the C compiler generates a function name based upon the first global definition it finds in the file. (First it searches through all function names; then if no global functions are defined, it searches through all data names.) The resulting name of the static function consists of the static function name, followed by the "#" character followed by the name of the global definition. For example, if a file contains a definition of the global function `main`, and a static function called `test`, then under DELTA the static function's name would be `test#main`.

DELTA Toggle Commands

Certain of the commands in the housekeeping category set toggles within DELTA. These toggles may be reset by the **kill** command or their status determined by the **show** command.

DELTA Housekeeping/Miscellaneous Commands

The commands discussed under this heading are those which influence the behavior of the DELTA processor itself. Their purpose is to provide the greatest possible flexibility in specifying the manner in which the user wishes to communicate with DELTA, how DELTA is to communicate with the user, how it is to interact with the run unit, and how it is to deal with both predictable and unpredictable events which occur during the execution of the run unit.

All of the commands in this category affect, in some manner, the way in which DELTA behaves in a given situation. Some set toggle switches which DELTA examines to determine whether or not a given activity is enabled or disabled. Others override certain default assumptions which are automatically established when DELTA is invoked. The default toggle settings and the default assumptions are those which would normally be specified.

DELTA Commands

Tables C-1 through C-6 contain an alphabetized list of the DELTA commands by category. Table C-7 contains the keywords used with the `kill` and `show` commands. Table C-8 contains the EOM characters and sub-commands. Table C-9 contains the format specifiers of the `format` command.

Command	Function
`AC[TIVE]|IN[ACTIVE]`	Activates or deactivates a single or a range of stored commands.
`A[LTERNATE] V[ARIABLES]`	Specifies alternate debug schema to be searched when an unqualified variable reference is not satisfied by searching the current schema.
`BY[PASS]`	Bypasses assembler program units during stepping. This command sets a toggle within DELTA.
`C[OPY]`	Causes DELTA output to be copied on the user terminal when the specified destination for output is other than the user terminal. This command sets a toggle within DELTA.
`DE[FINE]`	Associates a value or location with a symbol.
`DO`	Executes the attachments to a stored command or a group of commands identified by the `SAVE` command.
`EC[HO]`	Causes input to be echoed to an output device when DELTA input is from a device other than an on-line terminal. This command sets a toggle within DELTA.
`EO[M]`	Set or reset special activation (end of message) character set. This command sets a toggle within DELTA.
`FO[RMAT]`	Specifies default format for `MODIFY` and `EVALUATE` display output.
`KE[EP]|TRAP|IG[NORE]`	Direct DELTA's handling of asynchronous events and other exceptional conditions.

Table C-1. Housekeeping Commands

Command	Function
K[ILL]	Deactivates a toggle or removes a stored command or a range of stored commands.
O[N] A[BORT]	Specifies activities to occur upon abort.
O[N] E[XIT]	Specifies activities to occur upon normal exit.
OU[TPUT]	Specify destination for DELTA output.
PRO[MPT]	Sets the DELTA prompt character (default is >).
RA[NGE]	Specify range of offsets from defined symbol to be used for position reporting.
R[EAD]	Causes DELTA to read other than the normal input stream.
REP[ORT]	Directs DELTA's formatting of position reporting.
SA[VE]	Stores and remembers a single or a range of stored commands.
SC[HEMA]	Activates or deactivates schema usage or sets *current* schema. This command sets a toggle within DELTA.
SH[OW]	Displays the status of toggled options, keyword options or a single or range of stored commands and attachments.
SI[LENT]|UN[SILENT]	Activates or deactivates the reporting of a single or a range of stored commands.
SY[NTAX]	Allows specification of FORTRAN, COBOL, or C input syntax.
UP[DATE]	Updates stored commands or attachments of stored commands.
U[SE] N[ODE]	Activates schema(s) associated with a specific overlay node.

Table C-1. Housekeeping Commands (part 2)

Command	Function
ALI[B]	Specifies return/altreturn from M$ALIB call to DELTA.
A[T]	Sets an instruction breakpoint.
B[REAK]	Passes control to user interrupt routine.
EX[IT]	Exits from a run unit invoked by M$LINK and returns to the linking program, or continues an M$LDTRC or M$SAVE.
G[O]	Proceeds with program execution.
G[O] S[TEP]	Goes to a specified location and executes one step.
G[O] T[RAP]	Passes control to user's event handling routine when DELTA has been entered for an exceptional or asynchronous event.
G[O] T[RAP] ST[EP]	Same as GOTRAP except that one step is executed.
O[N] C[ALL]	Sets breakpoints on a specific procedure call.
O[N] [X] C[ALLS]	Sets breakpoints on all procedure calls. If X is specified, sets breakpoints only on external procedure calls.
O[N] N[ODE]	Sets a breakpoint on a specific overlay.
O[N] N[ODES]	Sets breakpoints on all overlays.
S[TEP]	Steps by statement or instruction.
UNSH[ARE]	Unshares an autoshared program and/or library so the user can have execution control.
W[HEN]	Sets a data breakpoint.

Table C-2. Execution Control Commands

Command	Function
XC[ON]	Passes control to the user's exit control procedure simulating an exit condition.

Table C-2. Execution Control Commands (part 2)

Command	Function
H[ISTORY]	Displays contents of history buffer (filled by TRACE).
PL[UGH]	(Acronym for "Procedure List Used to Get Here"). Traces back through the automatic stack and lists the return addresses leading to the arrival at the current procedure point.
T[RACE] T[RANSFERS]	Traces all transfer instructions.
T[RACE][X]C[ALLS]	Traces entry to all procedures. If X is specified, trace entry to external procedures only.

Table C-3. Execution Tracing Commands

Command	Function
D[ISPLAY]	Displays the value of a variable or the contents of an address.
DU[MP]	Dumps a specified range of memory in octal or hexadecimal format. Optionally allows no suppression of duplicate lines. Optionally provides ASCII translation.
E[VALUATE]	Evaluates an expression and reports its value in a specified format. Reports the address of a program entity by segment and offset.
F[IND]	Searches memory under mask and optionally substitutes under mask.
L[ET]	Changes the value of a variable or the contents of an address.

Table C-4. Memory Display and Modification Commands

Command	Function
M[ODIFY]	Displays the contents of an address and optionally replaces it with new contents.
PMD	Dumps specified portions of a program which terminates abnormally.
STO[RE]	Modifies a range of memory. Optionally performs the modification under mask.

Table C-4. Memory Display and Modification Commands (part 2)

Command	Function
AN[LZ]	Associates the schemas for the *CP-6* Monitor, or associates the schema from the specified file, and sets DELTA's domain of reference to that of the running monitor, a specified system dump file, or the running program.
RU[M]	Invokes the Run Unit Modification mode, optionally specifying the number of buffers to use for faster I/O (up to 10, default is 5).

Table C-5. Mode Control Commands

Command	Function
EN[D] or Q[UIT]	Unconditionally exits to the command processor.
HELP	Provides HELP via the HELP facility.
LI[ST]	Lists changes made during Run Unit Modification.
PROT[ECT]	Sets Protect mode (disallows LET, MODIFY store).
SAD	Special Access Descriptor allows addressing through a Monitor descriptor for privileged users.

Table C-6. Miscellaneous Commands

Command	Function
UNF[ID]	Performs M$UNFID on specified DCB.
X[EQ]	Executes an assembler instruction.

Table C-6. Miscellaneous Commands (part 2)

Keyword	Meaning with Kill	Meaning with Show
AL[L]	Remove all stored commands.	Display status of all stored commands, toggle options, and modes.
A[LTERNATE] V[ARIABLES]	Discontinue use of alternate variables.	Show status of toggle and schema name if any.
AN[LZ]	Return to debug mode.	Show status of toggle.
A[TS]	Remove all AT breakpoints and their attachments.	Display all AT breakpoints and their attachments.
B[YPASS]	Do not bypass step reporting in assembler modules.	Show status of toggle.
C[OPY]	Discontinue COPYing.	Show status of toggle.
D[EF]	Remove a specific named DEFINED symbol.	Not applicable.
DEFS	Remove all DEFINED symbols.	Display all defined symbols.
DEL[TA]	Causes DELTA to be disassociated from the current run unit being debugged.	Not applicable.
EC[HO]	Discontinue ECHOing.	Show status of toggle.
E[OM]	Deactivate the EOM character set.	Show status of toggle.
F[ORMAT]	Set display formats for MODIFY and EVAL back to initial defaults.	Display current default formats for MODIFY and EVAL.

Table C-7. Keywords Used with KILL and SHOW

Keyword	Meaning with Kill	Meaning with Show
I[GNORE]	Not applicable.	Display which exceptional condition groups and/or names are being ignored.
K[EEP]	Not applicable.	Display which exceptional conditions will be intercepted and reported by DELTA.
O[N] A[BORT]	Remove ON ABORT breakpoint and its attachments.	Display ON ABORT breakpoint and its attachments.
O[N] C[ALLS]	Remove all ON CALL(S) commands and their attachments.	Display all ON CALL(S) commands and their attachments.
O[N] E[XIT]	Remove ON EXIT breakpoint and its attachments.	Display ON EXIT breakpoint and its attachments.
O[N] N[ODES]	Remove all ON NODE(S) commands and their attachments.	Display all ON NODE(S) commands and their attachments.
P[ROTECT]	Discontinue PROTECT mode.	Display PROTECT mode.
R[ANGE]	Not applicable.	Display value of range specification for relation position reporting.
RE[PORT]	Not applicable.	Display reporting mode for position reporting.
RU[M]	Return to debug mode.	Show status of toggle.
SAD	Not applicable.	Display special access descriptor number.
SA[VES]	Remove all SAVE commands and their attachments.	Display all SAVE commands and their attachments.

Table C-7. Keywords Used with KILL and SHOW (part 2)

Keyword	Meaning with Kill	Meaning with Show
SC[HEMA]	Discontinue schema usage.	Display the position which defines the current schema, as set by the instruction counter or the SCHEMA command.
S[TEP]	Default to step by statement.	Show status of STEP mode.
SY[NTAX]	Not applicable.	Display which input syntax DELTA is currently accepting.
T[RACE]	Discontinue tracing.	Display all TRACE commands.
TRAP	Not applicable.	Display which exceptional conditions will be passed to the trap handler in the target run unit.
W[HENS]	Remove all WHEN breakpoints and their attachments.	Display all WHEN breakpoints and their attachments.
FEP Keywords:		
DEL[TA] fprg-res	Causes DELTA to be disassociated from the specified FEP program.	Not applicable.
FP[RGS]	Not Applicable.	Displays the state of all existing FEP programs being debugged.
fprg-res	Not Applicable.	Display the state of the specified FEP program.

Table C-7. Keywords Used with KILL and SHOW (part 3)

Character	Sub-Command	Action Indicated
`Linefeed`	`N[EXT]`	Open next memory cell for modify. Cell will be opened and displayed as if it has been addressed directly with a `MODIFY` command.
`Up Arrow`	`P[REV]`	Open previous cell for modify. The cell preceding the current cell will be opened and displayed as if it has been addressed directly with a `MODIFY` command.
`Left Bracket` `Left Brace`	`O[PEN]`	Re-open and re-display the last cell addressed by a `MODIFY` command. DELTA remembers the address of the last cell (if any) referenced in a `MODIFY` command.
`Tab Character`	`Asterisk (*)`	Indirect addressing. Display and open for modification the cell specified by the contents of the currently open cell. The interpretation of the address contained within the cell is dependent upon the format of the current display. If the current display is in pointer format, then the "segid" portion of the pointer is used to determine the appropriate segment, and the word offset portion of the pointer determines the offset within the segment. If the current display is in relative format, then the right half (the least significant eighteen bits) of the currently open cell is assumed to specify an address in the instruction segment. If the current display is in other than pointer format, then the left half (the most significant eighteen bits) of the currently open cell is assumed to specify an address in the instruction segment.

Table C-8. EOM Characters and Sub-Commands

Character	Sub-Command	Action Indicated
[\f]/	None	Same as * except do not open the cell for modification. Optionally a format specifier may be used.
None	*L[\f]	Treat left 18 bits as an address regardless of displayed format. Optionally a format specifier may be used.
None	*R[\f]	Same as *L except that the right half (the least significant eighteen bits) of the currently open cell is assumed to specify an address in the instruction segment. Optionally a format specifier may be used.
None	*P[\f]	Treat contents of the currently open cell as a pointer regardless of the format in which it was displayed. Take same action as for the tab EOM character or the * sub-command. Optionally a format specifier may be used.
FEP EOM Characters:		
None	*L[\f]	Invalid for FEP programs.
None	*R[\f]	Invalid for FEP programs.
None	*P[\f]	Treats the contents of the currently open cell (2 words) as a pointer. Take the same action as for the tab EOM character or the * sub-command. Optionally a format specifier may be used.
None	*SP[\f]	Treats the contents of the currently open cell (1 word) as a 16 bit address. Take the same action as for the tab EOM character or the * sub-command. Optionally a format specifier may be used.

Table C-8. EOM Characters and Sub-Commands (part 2)

Character Sub-Command	Action Indicated
`Tab Character  Asterisk (*)`	Indirect Addressing. Display and open for modification the cell(s) starting at the address specified by the contents of the currently open cell(s). The interpretation of the address contained within the cell(s) is dependent upon the format of the current display. If the current display is in instruction format, pointer format or relative format, the least significant 20 bits of the currently open cell (2 words) is assumed to specify an address. If the current display is in octal, unsigned integer, signed integer, bit, or hex format, the contents of the currently open cell (1 word) is assumed to specify an address. Other formats will be errored.

Table C-8. EOM Characters and Sub-Commands (part 3)

Specifier	Meaning
`A[R]`	Display left 24 bits of a word as word-char-bit. *Example:* `.35-2-5`
`B[IT]`	Display in binary format. *Example:* `'010110100'B`
`C[HAR]`	Character. *Example:* `'ABCD'`
`D[ESCR]`	Descriptor. *Example:* `.46000,BD=.75777-3, FL=.643,WSR=7,TY=0`
`EB[CDIC]`	EBCDIC Character. *Example:* `'694E'`
`E[PTR]`	Displays left half of word as `ENTDEF+.offset[:stmnt#]` *Example:* `PROGB+.374 :27`

Table C-9. Format Specifiers

Specifier	Meaning
F[LOAT]	Floating point binary. Single precision for 36-bit items, double precision for 72-bit items. *Example:* `5.789604E+76`
I[NSTR]	Assembly language instruction. *Example:* `LDQ .1,DL`
J[DE]	JIT Dot ERR. Displays error message for the value stored in JIT.ERR. *Example:* `FMN-M00113-0` `File does not exist`
O[CTAL]	Octal digits with leading zeroes suppressed. *Example:* `.1024`
P[TR]	Pointer. word-char-bit, segid. *Example:* `.35-2-7,$LSO`
R[EL]	Relative. `Primary ENTDEF+ offset[,:stmnt#, substmnt, offset]` or `SYMDEF + offset.` *Example:* `PROGA+.6 :12,,.1(LOOP)`
REM[EMBER]	Remember. *Example:* `TEST:6(LABEL)[ASSIGNMENT](+.4)`
S[BIN]	Signed binary (decimal). *Example:* `-357`
T[IME]	Convert UTS to display format. *Example:* `13:52:36.82 06/25/79`
U[BIN]	Unsigned binary (decimal). *Example:* `357`
V[ECTOR]	Vector. *Example:* `.6245-0-0,$LSO,BD=.14-2, FL=.777,TY=NORMAL SHRINK`
X	Hexadecimal. *Example:* `'F100CS40D'X`
X1	Pseudo-hexadecimal. Leading bit of each byte ignored. *Example:* `'F0F8F6F4'X`

Table C-9. Format Specifiers (part 2)

Specifier	Meaning
`Z[ERO]`	Displays a word value in octal with leading zeroes. *Example:* `.000000001024`

Table C-9. Format Specifiers (part 3)

Appendix D

Interfacing PL-6 and Assembler Routines to C

Data Types

Unary conversions applied to function arguments greatly reduce the number of data types that must be handled by run-time library routines. ANSI C retains the same unary conversions as defined in Kernighan and Ritchie for compatibility. Use of function prototypes, as defined in ANSI C, has minor effects on the unary conversions done to function arguments.

The following table summarizes the correspondence between C data types and PL-6 data types. The type received by the PL-6 subroutine depends upon the C argument type and whether or not the C function definition of the PL-6 subroutine has a prototype.

Description	C Type	Prototype PL-6 Type	PL-6 Type
signed integer	`short`	`SBIN(36)`	`SBIN(36)`
	`int`	`SBIN(36)`	`SBIN(36)`
	`long`	`SBIN(36)`	`SBIN(36)`
unsigned integer	`unsigned short`	`UBIN(36)`	`UBIN(36)`
	`unsigned`	`UBIN(36)`	`UBIN(36)`
	`unsigned long`	`UBIN(36)`	`UBIN(36)`
character	`char`	`CHAR(1)`	`SBIN(36)`
floating point	`float`	`BIT(36) ALIGNED`	`BIT(72) DALIGNED`
	`double`	`BIT(72) DALIGNED`	`BIT(72) DALIGNED`
pointer	*type* `*`	`PTR`	`PTR`
array	*name*`[]`	`PTR`	`PTR`
enumeration	`enum`	`SBIN(36)`	`SBIN(36)`
function	*name*`()`	`EPTR`	`EPTR`

Table D-1. C Data Type Correspondence

C Calling Sequence

CP-6 C makes use of the standard *CP-6* calling sequence, with the benefit that it allows modules written in other languages to be linked with those written in C.

The standard calling sequence normally expects the calling routine to pass a list of pointers to the actual parameters to the called routine. This is an efficient way to handle parameter passing in languages that use *call by reference* argument passing. Languages like Fortran and PL-6 fall into this category. The C language on the other hand uses *call by value* argument passing exclusively.

To reduce procedure call overhead, the standard calling sequence is modified slightly for C. The called routine passes a block of memory containing the values of the actual parameters, instead of a list of pointers to the actual parameters. C uses the same setup routines that other compilers use; the setup routines copy a block of parameter values into the local stack frame instead of a list of pointers.

PL-6 Receiving Sequence

PL-6 routines expect to receive pointers to the argument values, but this is not the way C passes arguments. There are two ways in which a PL-6 routine can receive the arguments from a C function:

1. The first method may be used if all actual parameters are pointers or are word-sized.
   - If an actual parameter is a pointer then the PL-6 routine can declare the parameter to be the type of object that the pointer points at. For example, if the actual parameter is of type `(int *)` then the formal parameter should be declared to be of type `SBIN`.
   - If an actual parameter is not a pointer, the formal parameter may be declared to be of any type because the only operation that can be done on the formal parameter is to take its address. The reason for this is that the location in the stack frame that PL-6 believes contains a pointer to the actual parameter really contains the value of the actual parameter. The value of the actual parameter can be made accessible however, by declaring a local variable of the right type, `REDEF`'ing it as a pointer and assigning the `ADDR` of the formal parameter to the pointer variant. This is more easily understood by examining Figure D-1.
2. The second method can be used in all cases. It involves
   - Declaring the PL-6 routine to have no parameters.
   - `MATERIALIZE`ing `$PRO` which points to the argument list.
   - Copying the argument list into local memory.

C main program

```
main()
{
 int i, j;
 i = 1;
 j = 2;
 ADDP1TOP2(i, &j);
}
```

PL-6 routine

```
ADDP1TOP2: proc(P_i, j);

dcl P_i                      sbin; /* type is irrelevant here */
dcl j                        sbin;

dcl i                        sbin;
dcl i_as_ptr    redef i      ptr;

     i_as_ptr = addr(P_i);  /* i now has value passed from C */
     j = j + i;

end ADDP1TOP2;
```

Figure D-1. PL-6 Routine Example

The previous example illustrates the use of the first method to implement a function which has a word-sized parameter and pointer parameter.

C main program

```
main()
{
  int i, j;
  i = 1;
  j = 2;
  ADDP1TOP2(i, &j);
}
```

PL-6 routine

Figure D-2. PL-6 Materialize Example

```
ADDP1TOP2: proc materialize($PRO in parameter$);

dcl  parameter$               ptr;

dcl  1   p                    based(parameter$),
         2  i                 sbin,
         2  j                 ptr;

dcl  1   arg,
         2  i                 sbin,
         2  j                 ptr; /* pointer to int */

dcl  based_int                sbin based;

     arg = p; /* copy value of actual parameters to local memory */
     arg.j -> based_int = arg.j -> based_int + arg.i;

end ADDP1TOP2;
```

Figure D-2. PL-6 Materialize Example (part 2)

Figure D-2 illustrates the use of MATERIALIZE to interface PL-6 routine to C. Strictly speaking, the "arg" structure is redundant as the based "p" structure could be used to access the parameters. The advantage of copying the parameters to a local variable is that PL-6 is able to generate faster code for the routine, particularly if parameters are accessed more than once.

Double Word Aligned Parameters

double parameters (and float parameters when a function prototype is not used) result in a double word value being passed. The DPS 8 hardware requires that double word variables be double word aligned; that is, they must begin on even word boundaries. Argument lists constructed by C are double word aligned when necessary.

When writing assembler routines, note that $PRO points to an even word location if there is a double value being passed. There may be filler words in the parameter block to ensure that double values are correctly aligned.

C main program

```
main()
{
  double a, b;
  a = 4.0;
  b = 2.0;
  COPY_DOUBLE(&a, b);
}
```

PL-6 routine

```
COPY_DOUBLE: proc materialize($PRO in parameter$);

dcl  parameter$                ptr;

dcl  1  p                      daligned based(parameter$),
        2  a                   ptr,
        2  *                   sbin, /* Filler to bound b correctly */
        2  b                   bit(72) daligned;

dcl  1  arg                    daligned,
        2  a                   ptr,
        2  *                   sbin, /* Filler...   */
        2  b                   bit(72) daligned;

dcl  based_double              bit(72) daligned based;


     arg = p;           /* copy parameters into local stack frame */
     arg.a -> based_double = arg.b;

end COPY_DOUBLE;
```

Figure D-3. Double Word Alignment Example

The previous example illustrates the PL-6 header code that has to be used to pass a double type parameter.

C main program

```
double COPY_VALUE();
main()
{
  double a, b;
  a = 2.0;
  b = 4.0;
  b = COPY_VALUE(a);
  if (b != a)
     printf ("**ERROR**\n");
}
```

BMAP routine

```
           ENTDEF        COPY_VALUE
* Return double argument
           USE           COPY_VALUE,1
COPY_VALUE EQU           *
           DFLD          0,,PRO
           TRA           1,X1
           END
```

Figure D-4. BMAP Example

The previous example illustrates a BMAP routine that accepts a single double parameter.

Returning Function Results

CP-6 C expects float and double type function results to be returned in the EAQ register and expects all other function result types returned in the Q register. This is consistent with other *CP-6* language processors.

Since user-defined functions are not available in PL-6, two special routines are provided by the C library to permit PL-6 subroutines to appear to be functions. These two routines, XB_RETURN_ORDINAL_RESULT or XB_RETURN_REAL_RESULT return from the PL-6 subroutine with the argument as the result.

These special routines assume that they are called from a routine that has a stack frame. They do not work if the PL-6 routine has the NOAUTO attribute.

C main program

```
main()
{
  int i = 4, j = 5;
  j = NOOP(i);
  if (j != 4)
     printf ("**ERROR**\n");
}
```

PL-6 Function

```
NOOP: proc materialize ($PRO in parameter$);

dcl  parameter$                      ptr;
dcl  based_int                       sbin based;
dcl  i                               sbin;

dcl  XB_RETURN_ORDINAL_RESULT        entry(1);

     i = parameter$ -> based_int;
     call XB_RETURN_ORDINAL_RESULT(i);    /* Return parameter */

end NOOP;
```

Figure D-5. PL-6 Function Example

The previous example demonstrates the use of XB_RETURN_ORDINAL_RESULT.

Object Unit Names

PL-6 and BMAP entry names are always in uppercase regardless of how they were entered in the source file. Identifiers are case sensitive in C; all C run-time library functions have lowercase names.

PL-6 or BMAP object files can be changed to have lowercase ENTDEF names by using the X account tool FALCON. For example:

```
!FALCON.X  object_file
```

Writing I/O Routines in PL-6

The include file `xb_stdio_i6` found in the `:LIBRARY` account contains macros for the C `FILE` structure and other I/O related structures.

Useful Entries in the C Run-time Library

The following entry points in the C run-time library are of use to programmers interfacing PL-6 routines to C.

XBI_SET_ERRNO Subroutine

Synopsis:

```
%include CP_6;
%VLP_ERRCODE (FTPN=supplied_error);
CALL XBI_SET_ERRNO (supplied_error);
```

Description:

The XBI_SET_ERRNO routine places an error code into the global **errno** variable that is visible to C functions. The error code comes from **supplied_error**, or if the routine is called with no arguments, from the **altreturn** frame of the TCB.

XBI_SET_STREAM_ERRNO Subroutine

Synopsis:

```
%include CP_6;
%include xb_stdio_i6;
%File_Header (NAME=F);
%VLP_ERRCODE (FPTN=supplied_error);
CALL XBI_SET_STREAM_ERRNO (F, supplied_error);
```

Description:

The XBI_SET_STREAM_ERRNO routine places an error code into the global **errno** variable that is visible to C functions. The error code comes from **supplied_error**, or if the routine is called with no arguments, from the **altreturn** frame of the TCB. The DCB number of the stream on which the error occurred is extracted from F.

XBI_GET_ERRNO Subroutine

Synopsis:

```
%include CP_6;
%VLP_ERRCODE (FPTN=error);
CALL XBI_GET_ERRNO (error);
```

Description:

The XBI_GET_ERRNO routine returns the current value of **errno** converted to a *CP-6* error code.

XBI_GET_CP6_DCBNUMBER Subroutine

Synopsis:

```
dcl dd_number sbin;
CALL XBI_GET_CP6_DCBNUMBER (dcb_number);
```

Description:

The XBI_GET_CP6_DCBNUMBER routine returns the DCB number associated with **errno**. If the most recent error was not an I/O error, the value zero will be returned.

XBI_CLOSE_DCBS Subroutine

Synopsis:

```
CALL XBI_CLOSE_DCBS;
```

Description:

The XBI_CLOSE_DCBS routine closes all currently open streams, except **stderr**.

XB$INIT_CLIB Subroutine

Synopsis:

```
CALL XB$INIT_CLIB ()
```

Description:

The XB$INIT_CLIB routine initializes the C run-time library for programs that do not have a C main program, but want to use C I/O.

Appendix E

Porting C Programs to CP-6 Systems

This appendix outlines the major pitfalls encountered when porting C applications to *CP-6* systems. Programs which claim to be portable or have been ported to a number of other systems can still run into problems because:

1. The operating systems to which it has been ported are all the same.

   This is the most common problem. For example, programs written for and used exclusively in a UNIX environment can be quite portable to other UNIX systems, but when ported to a non-UNIX system it is found that they use features of UNIX not generally available under other operating systems.

   Of course, *CP-6* C has been written with porting these applications in mind; however, there are cases that cannot be duplicated as the program might expect.

2. The compilers that have been used create the same environment.

   This is similar to the first point but slightly different. An example of this would be assuming that `shorts` are 2 `chars` and `ints` are 4 `chars`. This works on many (if not most) systems but not on *CP-6* where `int`, `short` and `long` are all represented with the same size.

3. The architectures are all very similar.

   This addresses the areas where the biggest problem might exist in porting programs to *CP-6* systems. On *CP-6*, the bit pattern used to represent a floating point 0.0 and the bit pattern to represent the `NULL` pointer are not the same as the bit pattern for an `int` value of 0. Many programs make this non-portable mistake and claim portability.

The rest of this section attempts to address the issues involved in porting applications to *CP-6* systems.

Implementation-defined Behavior

Even when an application compiles with a standard conforming C compiler, its portability is not guaranteed. This is because it may be relying upon the *implementation-defined* behavior of the compiler and system upon which it is running. This section points out the implementation-defined elements of the C language which can cause problems when porting applications to *CP-6* systems.

The Environment

- The `main` function in some implementations of C takes an extra argument which is not available with *CP-6* C. This argument is called the environment pointer and contains pointers to character strings which define the environment variables.

  On *CP-6*, these variables are available through the **getenv** and `lsenv` functions (**getenv** is also available on other standard conforming systems).
- Terminals and their full use varies from system to system. On *CP-6* systems, the terminal can be open in `x364` mode which makes any terminal appear to be like a `VT 100` terminal. On other systems there are different techniques used to obtain terminal independence.

Use of Identifiers

Programs that consistently name their identifiers or limit their length to 64 characters will not have problems on *CP-6* systems. Otherwise, the following problems may be encountered:

- Every ANSI standard conforming C implementation provides at least 31 significant characters in a non-external identifier. *CP-6* C provides 64 significant characters. This may cause problems with older programs where the minimum was not as large.
- Every ANSI standard conforming C implementation provides at least 6 significant characters in an external identifier. *CP-6* C provides 64 significant characters. Again this can cause problems if an application was written knowing that only the first *n* characters mattered and therefore did not name the external items consistently.
- Some C implementations do not distinguish differences between upper and lower case letters in external identifiers. *CP-6* C does provide this distinction so programs that have been written which do not honor the distinction will not work correctly.

Character Set

- If programs depend upon displaying characters that are not in the 7-bit ASCII character set, their portability to *CP-6* systems is limited. Additionally, if the program depends upon the actions of a specific terminal and control characters, it may not be portable (although *CP-6* C does permit special characters to be sent to a terminal).
- *CP-6* C does not provide unique display for multibyte characters. All characters in *CP-6* C are 7-bit ASCII.
- *CP-6* C uses 9 bits to represent the **char** data type. This is unlike most systems which use 8 bits. Programs which depend upon this (such as assuming a value being truncated to 8 bits when stored in a **char**) must be modified.

- When more than one character is found in an integer character constant (such as `'abcd'`) the actual value varies from system to system and even the ordering of the individual characters within the constant may vary from system to system. In particular, on *CP-6* systems since 9-bit `chars` are used, the values will be much larger than on systems with 8-bit `chars`.
- Finally, on *CP-6* C, the default `char` data type is an unsigned value whose range is 0 through 511. On other systems, the `char` data type may be signed and 8 bits so the range on those systems is `-128` through `128`. The default `char` data type may be changed to `signed` using the command line option `char=signed`.

Integral Data Types

Implementations of C differ in the size and actual bit representations of the various integral data types. There are also a number of operators where the result depends upon the machine architectures.

- Programs may rely upon the representations and range of values of the various integral types (`char`, `short`, `int`, and `long`). Many implementations provide distinct representations for each of these basic sizes. On *CP-6*, only `char` has a different representation from the other integral types. Programs which depend upon a `short` being 2 `chars` or holding a more restricted range of values than `int` will need to be modified.
- The right shift of a negative signed integral value differs from implementation to implementation. On some systems, the shift preserves the sign and on others, it does not. *CP-6* C does preserve the sign.
- Bitwise operations on negative integers are not portable. This is because the result of the operation depends upon the size (in bits) of the value.
- The remainder of the integer division operator `%` may differ for negative arguments. *CP-6* provides a negative remainder when the dividend is negative; otherwise, the remainder is positive.
- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length differs when the final value cannot be represented. The *CP-6* system truncates the most significant bits and attempts to preserve the actual bit pattern being converted.

Array and Pointer Data Types

CP-6 C is very sensitive to the correct usage of pointers. In particular:

- Illegal pointers very quickly generate signals indicating incorrect usage. Pointers are not simply integers, they contain the following information:

1. A 12-bit segment identifier. This field is in the least significant 12 bits if the pointer is viewed as an `int`.

   CP-6 C programs always have three distinct segments containing user data. The instruction segment contains the text of the functions and all static data. The `auto` segment contains the local (non-static) variables for all currently executing functions. Finally the data returned by the `malloc` and `realloc` functions is in the third segment.
2. A 20-bit byte offset within the segment. This field is in the most significant 20 bits if the pointer is viewed as an `int`.
3. A 4-bit bit offset. This is always zero for pointers created by C, but it enables a pointer to point down to the bit level.

It should be obvious that treating a pointer as an `int` using casts or within a `union` is not portable. If an application being ported to the *CP-6* system does this, it will need to be modified.

- On *CP-6* systems, a special segment is reserved for the `NULL` pointer. The representation of this pointer (as an `int`) is not zero. This can can cause problems with porting programs when, for example, the program uses the `calloc` function to allocate a structure and then assumes that the pointers in that structure will be `NULL`. Of course, the compiler does correctly cast the integer value zero to a `NULL` pointer and it also initializes static memory to `NULL` for static pointers.
- Casting between a pointer and an integer changes the bit pattern. As indicated by the previous point, a `NULL` pointer is not zero; therefore, when a cast from an integral type to pointer occurs (or pointer to integral type), the compiler actually modifies the bit pattern. The compiler reports a warning in most of these cases so that they may be located.
- The integral data type required to hold the maximum size of an array can vary between systems. It is always best to use the type `size_t` defined in the header file `<stddef.h>`. *CP-6* C is not very sensitive to this as any of the integral types other than `char` are capable of holding these values.
- The integral data type required to hold the maximum difference between two pointers to elements of the same array can vary. It is always best to use the type `ptrdiff_t` defined in the header file `<stddef.h>`. *CP-6* C is not sensitive to this as any of the integral types other than `char` are capable of holding these values.
- When pointers that are not part of the same array object are subtracted, the answer is not meaningful if the pointers point to objects in different segments.

  This is also true when comparing pointers. If they are not in the same segment then comparisons for anything other than equality are not meaningful.
- If a function that is expected to return a result uses a `return` statement without a value, then the actual value returned is indeterminate.

Floating Point Data Types

The floating point data type is the area of greatest divergence between systems. The areas of divergence include the range of values that may be represented (indicated in the `<float.h>` header file), whether truncation or rounding occurs, and the direction of truncation and rounding.

The most likely problem to occur porting programs which perform floating point arithmetic is that some programs expect that the bit pattern for an integer value of zero is the same as the bit pattern for a floating point zero. This is not true on *CP-6* systems. The solution to this is similar to the solution for programs which make the same assumption about `NULL` pointers.

CP-6 C floating point representation is difficult for even a smart program to discern because it uses hexadecimal normalization with a result that is 1 bit short in the last hexadecimal digit.

Structure and Union Data Types

- When a value is stored in a `union` using one member and later is accessed using another member, there can be portability problems.

  Depending upon the actual member types, the problems can range from the expected field sizes being incorrect to expecting a `NULL` pointer to be equal to an `int` value of zero.

- C compilers are different in the manner in which they position members within a structure. On *CP-6* systems, members of type `char` or structures which contain only members of type `char` immediately follow each other in memory. Members of the other integral types (including `enum`) as well as members of the `float` type are always aligned on the next word boundary. Finally, members containing `double` or `long double` are aligned on the next double word boundary. This means that there may be unused `char`s preceding a member which requires alignment.

  Programs which depend upon the alignment requirements of other compilers may need modification. This would normally not be a problem unless binary data written by one compiler needed to be read by another compiler.

- Bit-fields within a member may cause problems. For example, *CP-6* C can allocate up to 36 bits within each `int`. In addition, the actual bit-fields are allocated from left to right within a word (that is, the most significant positions are filled first).

- Bit-fields in *CP-6* C cannot straddle a word boundary. Some implementations permit this. This is normally not a problem.

Function Execution

- Functions may depend upon the order in which side effects occur within expressions. This is dependent upon the actual compiler (and possibly the version of the compiler) used to create the application.
- Side effects include the actual timing of assignments particularly when a variable is post incremented in an expression and used elsewhere in the same expression.

  Other problems in this area may include the order in which arguments to functions are evaluated if they contain side effects or call other functions. *CP-6* C evaluates the function's arguments from left to right within the argument list and then evaluates the expression which represents the function to be called.
- When a label is defined in a C function, its scope is function-wide. This is not true for all old compilers so there will be problems if an application defines a label more than once within a function body.
- When a function is called, and the definition and the reference to the function both had prototypes, the system automatically checks for the correct number of arguments at link time. This does not guarantee that the types of the arguments are correct but it is a good first order check. Most of the library functions will cause linker errors to be reported if they are called using their standard header include file at compile time.

  When a function is called, and the definition or references was not a prototype, there is no argument checking by the linker. If the function reference has fewer arguments than the function expects, the unsupplied arguments will have the value `06014` if they are integral values; pointers will have the `NULL` value; floating-point variables will have very small (but non-zero) values.

  If the expected types of the arguments do not agree between the calling and called function, there will be problems which need to be addressed if the types are non-integral.

The Preprocessor

When porting programs to *CP-6* systems, problems may be found in the following areas:

- All of the header files that an application expects to find may not be available on *CP-6* systems. The application may rely upon header files which are not a part of standard C or which are not available in *CP-6* C.

  There is no simple method of dealing with this problem. Often such applications are doing things which are done differently on *CP-6* systems. In porting such applications, determine why the missing header file is being used and then try to use a method provided by the C standard to perform it, or find the way it is done on *CP-6* systems.

- The method of locating include files may be different. Since what constitutes a legal file name varies from system to system, this may be a simple task or it may be more complex. Note that by default, *CP-6* C does not honor the distinction between include files mentioned within angle brackets (`<fid.h>`) or quoted (`"fid.h"`).

  CP-6 C does provide a method of translating file names mentioned in include directives into legal *CP-6* file identifiers. This is used to minimize the number of actual source changes necessary to port an application.
- The **`#pragma`** directive on *CP-6* C provides listing control. When an unrecognized pragma is encountered, it is ignored.

The C Library

CP-6 C provides more functions and capabilities than the ANSI standard requires; however, there is a large number of system-specific functions that C programs often use. In order to port these programs, it is necessary to either modify the program to use standard features, modify the program to not use the unsupported feature, or to write a *CP-6* version of this capability.

- The values stored in **`errno`** when an error is reported by the library vary from system to system. On *CP-6* C, the value is a slightly modified *CP-6* error code. Some systems use a small integer to represent these codes; on *CP-6* systems they are always very large integers.

  This also precludes the use of the value in **`errno`** as an index into a table of system error messages (as is done on some systems). Use the **`perror`** or **`strerror`** function to obtain the actual text of error messages.
- Some systems supply signals that are not available on *CP-6* C. Use of these signals indicates use of an unsupported system feature which must be modified.
- Signals have default states in every implementation which may not be identical to the default for another implementation. Also, signals may be disabled as the signal handler for a given signal is entered. Normal use of the standard signals is extremely portable; however, problems may be encountered in the above system-dependent areas.
- The rules for constructing file names vary from system to system. Even for programs which are maximally portable, constructing file names may involve modifying the source. On *CP-6* C, file names are split into four pieces:
  1. Packset name. The packset name (not normally used) identifies the physical device where the file resides. This is normally only used when the file resides on a device that is not publicly mounted. The packset name looks like `DP#sn0000` where `sn0000` indicates the name of the packset.
  2. File name. The file name is separated from the packset name by a slash. The slash is optional if the packset name is not present. The file name consists of 1 to 31 characters from the set **`a-z`**, **`A-Z`**, **`0-9`** and **`_$:[\]^{}|~'`**

3. Account name. If the account name is present, it consists of 1 to 8 characters from the 7-bit ASCII set, SP through ~. The file name and account are separated by a single "." character.
4. Password. The optional password, if present, consists of any bit pattern, preceded by a "." character which separates it from the account (or two "." characters if the account is not present).

- The use of the `system` function is very system dependent. On *CP-6* systems, the command is interpreted by the IBEX processor which may in turn start up another program to complete the command. The exact syntax of commands varies from system to system; therefore, the text used as the command will need to be modified when porting an application to *CP-6* systems.
- The full list of current environment names may be determined by using the `lsenv` function. Additionally, the current environment names may be modified using the `system` function which contains an IBEX `let` command. Some programs may expect certain environment variables to exist which are created, by convention, on other systems. In this case the application may need modification.
- The ANSI standard permits some identifiers to be macros or actual identifiers. Examples include `errno`, `setjmp`, `va_end`, `va_start`, and `va_arg`. Portable applications should not declare library functions; the definitions in the system header file should be used.
- Portable applications should also not refer to the value stored by the `fgetpos` function or the actual value returned by the `ftell` and `time` functions.
- The value returned when a mathematical function is reporting a domain error varies from system to system. *CP-6* C attempts to return a value that is meaningful. Portable applications should never depend upon the value returned when an error occurs. This is not normally a problem when porting applications to *CP-6* systems.
- When an underflow occurs in the mathematical library functions, the situation is not always reported by all systems. *CP-6* C does not report this situation unless it leads to an error. Portable applications should not depend on underflow being reported in any way. This is not normally a problem when porting applications to *CP-6* systems.
- The `fmod` function may return zero or report a domain error when the second argument is zero. In *CP-6* C, a zero is returned. Applications which depend upon a domain error being reported may have to be modified. This is not normally a problem when porting applications to *CP-6* systems.
- The actual data in a file varies from system to system when a new-line character has not been written as the last character of a text stream. *CP-6* C adds a terminating new-line character. Other systems may not add a new line. Portable applications should ensure that a new-line character is written. This is not normally a problem when porting applications to *CP-6* systems.
- Space characters immediately preceding a new-line character in a text stream may not be physically written on some systems. On *CP-6* systems the space characters are written. Applications that depend upon space characters before the new-line character may need to be modified. This is not a normally a problem when porting applications to *CP-6* systems.

- Some systems pad the end of a binary stream with null characters. *CP-6* C does not pad. Applications that depend upon padding may need to be modified. This is not normally a problem when porting applications to *CP-6* systems.
- When a file is opened in append mode, its actual position for reading varies from system to system. On *CP-6* systems, the stream is positioned at the end of file which means that an attempt to read will immediately receive an `EOF` indication. In order to read the file on *CP-6* systems it is necessary to use the `rewind` function before reading.
- The contents of a file varies from system to system when a write occurs on a text stream with the file not positioned at the end. *CP-6* C only permits text streams to be written at the end-of-file position ("`a+`" or "`w`" mode). Requesting any other mode forces the file to be opened in binary mode ("`w+`" or "`r+`" modes). Applications that try to modify text files in this way will probably require modification to create binary files.
- Programs that rely upon the system-dependent meaning of the file buffering mechanisms other than *fully buffered* may need to be modified. *CP-6* C implements fully buffered output for all files irrespective of the buffering requested. Programs which use the `setvbuf` function (or the `setbuf` function with a `NULL` pointer as its second argument) may exhibit this problem. In particular, the unbuffered mode to terminals may not cause immediate writes of characters to terminals. Applications that depend upon the meaning of these modes on other systems may need to be modified.
- Zero-length files may not be created on some systems. On *CP-6* systems zero-length files are simply files to which no data was written. Portable applications should not depend upon empty files not being created. This is not normally a problem when porting applications to *CP-6* systems.
- On some systems a file may not be opened more than once. On *CP-6* systems a file may be opened for input many times. This is not normally a problem when porting programs to *CP-6* systems.
- On some systems, the `remove` function may not delete a file that is currently opened by the current or another user. On *CP-6* systems, the `remove` function may be used to delete a file even while it is being read by the current user or other users. It is made immediately inaccessible for new readers and physically deleted when all the current input readers have closed the file. This is not normally a problem when porting programs to *CP-6* systems.
- Some systems do not report an error when the `rename` function attempts to rename a file to a file name that already exists. On *CP-6* systems, an error is reported in this situation. Portable programs should not depend upon the existing file being deleted and no error being reported.
- The meaning of the `%p` conversion specification of the `printf` function varies from system to system. In *CP-6* C, this conversion specification requests a word in unsigned octal. Portable programs should avoid using this conversion specification.
- The meaning of the `%p` conversion specification of the `scanf` function varies from system to system. In *CP-6* C, this conversion specification requests the octal input be converted into a word value. This is not normally a problem when porting applications to *CP-6* systems.

- The meaning of the - character in the scanlist of the `%[ ]` specification of the `scanf` function varies from system to system. On *CP-6* systems it is treated as that specific character. On other systems it may also mean a range of characters when preceded and followed by another character. For maximal portability it is always best to put the - character at the beginning of the list of characters.
- The treatment of a zero passed to the `calloc`, `malloc` and `realloc` functions varies from system to system. *CP-6* C politely returns a `NULL` pointer but other systems may not be as forgiving. For maximal portability, these functions should not be called requesting a block of memory of zero bytes. This is not normally a problem porting programs to *CP-6* systems.
- The final status of files varies from system to system when the `abort` function is called. Some implementations may not close the files. On *CP-6* systems, all open files are flushed and closed.

Common Extensions to ANSI C

Many C implementations contain extensions to the standard. Often an extension is implemented in many compilers which then makes the distinction of portable applications difficult to determine when they use one of these common extensions. This section will attempt to point out the more common extensions and indicate which ones are implemented in *CP-6* C.

Specialized Identifiers

In *CP-6* C, the **$** character may be used in identifiers. Other implementations may go so far as to allow the ***national use characters*** from the 7-bit ASCII character set to be used in identifiers.

Scopes of Identifiers

Function declarations and variables declared with the keyword **`extern`** have file scope independent of the actual scope at which the variable was declared. This means that once an **`extern`** variable is declared (even within a function), its definition is remembered for the rest of the compilation.

Writable String Literals

By default, string literals in *CP-6* systems are not modifiable. Additionally, there is only one copy of a particular literal made even if it appears multiple times within a file.

When the command line option **`strings=write`** is given, then the compiler puts string literals in static writeable memory and makes new copies of the literal for every reference.

Other Arithmetic Types

Some compilers provide additional types such as `long long int` which is not available in *CP-6* C. *CP-6* C provides only the standard C arithmetic data types.

Function Pointer Casts

A pointer to an object or to `void` may be cast to a pointer to a function in *CP-6* systems as well as the inverse; that is, a pointer to a function may be cast to a pointer to `void` or an object. This is very system dependent and programs which do this will need to be modified.

Non-`int` Bit-field Types

Any integral type may be used as a bit-field type in *CP-6* C. Because only the `char` bit-field type is a different size, bit-fields may be declared as a part of a 36-bit word or as part of a 9-bit character.

The `fortran` and `asm` Keywords

The `fortran` keyword is not necessary on *CP-6* systems (since C uses the same calling sequence as all other *CP-6* compilers). Programs that use this keyword should be modified; or a `#define fortran` command should be used to remove it.

The `asm` keyword is not available in *CP-6* C. This keyword is used to insert assembler code into the program and is by definition therefore not portable. Any program which uses this keyword will need to be modified.

Multiple External Definitions

A variable whose type is `extern` may contain only one definition in the C program. Some systems permit multiple definitions. In order to port a program like this, all but one of the definitions should have the `extern` keyword inserted and any initializers removed.

Empty Macro Arguments

Some C preprocessors permit an argument to consist of no tokens. *CP-6* C requires that there be at least one token as an argument. A message of the form: "`(warning) argument mismatch, name`" is reported in this case. *CP-6* C does not expand such a macro use.

Predefined Macro Names

This is perhaps the most common extension. Most C compilers permit macro names to be defined on the command line, before the program has started compilation. *CP-6* C also permits this, although the exact way that this is done does vary from system to system (see the command line option `define`).

Also a number of preprocessor variables are automatically defined for every compilation (unless an `ndefine` command line option is provided). These variables include `TS_CP6`, `TM_L66` and `_CP6_`.

Extra Arguments for Signal Handlers

Some systems call the signal handlers with arguments in addition to the signal number. *CP-6* C does not do this and therefore any code which depends upon this behavior will have to be modified.

Additional Stream Types and File-opening Modes

The actual mapping of files to stream types varies in different C implementations and systems. *CP-6* files are record-oriented whereas many other systems have file systems where the files are simply byte streams. It may be necessary to modify applications which expect to read files as byte streams to use binary files.

CP-6 also supplies a number of file opening modes which are not portable. On *CP-6* systems, if ported code uses special mode arguments to open files, the program will most likely not function properly. The mode arguments will have to be modified to work correctly on *CP-6* systems.

Appendix F

Environmental Limits

The environmental limits for the *CP-6* C compiler are presented in the following subsections.

Translation Limits

The compiler can translate and execute programs that contain instances of every one of the following limits.

- Nesting levels of compound statements, iteration control structures, and selection control structures are limited by available memory.
- There may be 10 nesting levels of conditional inclusion.
- Pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration are limited by available memory.
- Nesting levels of parenthesized declarators within a full declarator are limited by available memory.
- Nesting levels of parenthesized expressions within a full expression are limited by available memory.
- There may be 64 significant initial characters in an internal identifier or a macro name.
- There may be 64 significant initial characters in an external identifier.
- The number of external identifiers in one object unit is limited by available memory.
- The number of identifiers with block scope declared in one block is limited by available memory.
- The number of macro identifiers simultaneously defined in one object unit is limited by available memory.
- The number of parameters in one function definition is limited by available memory.
- The number of arguments in one function call is limited by available memory.
- There may be 200 parameters in one macro definition.
- There may be 200 arguments in one macro invocation.
- There may be 509 characters in a logical source line.

- The number of characters in a character string literal or wide string literal (after concatenation) is limited by available memory.
- There may be 1,048,576 bytes in an object.
- There may be 10 nesting levels for `#included` files.
- The number of `case` labels for a `switch` statement (excluding those for any nested `switch` statements) is limited by available memory.
- The number of members in a single structure or union is limited by available memory.
- The number of enumeration constants in a single enumeration is limited by available memory.
- The number of levels of nested structure or union definitions in a single struct-declaration-list is limited by available memory.

Numerical Limits

The numerical limits, as specified in the headers `<limits.h>` and `<float.h>`, are explained below.

Sizes of Integral Types `<limits.h>`

The values given below are constant expressions suitable for use in `#if` preprocessing directives. Moreover, except for `CHAR_BIT` and `MB_LEN_MAX`, the following expressions have the same type as would an expression that is an object of the corresponding type converted according to the integral promotions. Their values are equal or greater in magnitude (absolute value) to those shown, with the same sign.

- Number of bits for smallest object that is not a bit-field (byte):

  ```
  CHAR_BIT                              9
  ```

- Minimum value for an object of type `signed char`:

  ```
  SCHAR_MIN                          -256
  ```

- Maximum value for an object of type `signed char`:

  ```
  SCHAR_MAX                           255
  ```

- Maximum value for an object of type `unsigned char`:

  ```
  UCHAR_MAX                           511
  ```

- Minimum value for an object of type `char`:

  ```
  CHAR_MIN                              0
  ```

- Maximum value for an object of type **char**:

CHAR_MAX	511

- Maximum number of bytes in a multibyte character, for any supported locale:

MB_LEN_MAX	1

- Minimum value for an object of type **short int**:

SHRT_MIN	-34359738368

- Maximum value for an object of type **short int**:

SHRT_MAX	+34359738367

- Maximum value for an object of type **unsigned short int**:

USHRT_MAX	68719476736

- Minimum value for an object of type **int**:

INT_MIN	-34359738368

- Maximum value for an object of type **int**:

INT_MAX	+34359738367

- Maximum value for an object of type **unsigned int**:

UINT_MAX	68719476736

- Minimum value for an object of type **long int**:

LONG_MIN	-34359738368

- Maximum value for an object of type **long int**:

LONG_MAX	+34359738367

- Maximum value for an object of type **unsigned long int**:

ULONG_MAX	68719476736

Characteristics of Floating Types `<float.h>`

The following parameters are used to define the model for each floating-point type:

s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum e_{min} and a maximum e_{max})
p	precision (the number of base-*b* digits in the significand)
f_k	non-negative integers less than *b* (the significand digits)

A normalized floating-point number $x(f_1 > 0$ if $x \neq 0)$ is defined by the following model:

$$x = s \times b_e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{min} \leq e \leq e_{max}$$

All except FLT_RADIX and FLT_ROUNDS have separate names for all three floating-point types. The floating-point model representation is provided for all values except FLT_ROUNDS.

The rounding mode for floating-point addition is characterized by the value of FLT_ROUNDS as follows:

-1	indeterminable
0	toward zero
1	to nearest
2	toward positive infinity
3	toward negative infinity

The *CP-6* C compiler has a value of 0 (rounds toward zero).

The characteristics of the floating-point system are as follows:

- Radix of exponent representation, *b*:

```
FLT_RADIX                         16
```

- Number of base-FLT_RADIX digits in the floating-point significand, *p*:

```
FLT_MANT_DIG                       6 /* .75 */
DBL_MANT_DIG                      15 /* .75 */
LDBL_MANT_DIG                     15 /* .75 */
```

- Number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits,

  $\lfloor (p-1) \times \log_{10^b} \rfloor + \begin{cases} 1 \text{ if } b \text{ is a power of } 10 \\ 0 \text{ otherwise} \end{cases}$:

FLT_DIG	8
DBL_DIG	20
LDBL_DIG	20

- Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, e_{min}:

FLT_MIN_EXP	-129
DBL_MIN_EXP	-129
LDBL_MIN_EXP	-129

- Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{min}-1} \rceil$:

FLT_MIN_10_EXP	-155
DBL_MIN_10_EXP	-155
LDBL_MIN_10_EXP	-155

- Maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, e_{max}:

FLT_MAX_EXP	+126
DBL_MAX_EXP	+126
LDBL_MAX_EXP	+126

- Maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{max}} \rfloor$:

FLT_MAX_10_EXP	+152
DBL_MAX_10_EXP	+152
LDBL_MAX_10_EXP	+152

- Maximum representable finite floating-point number, $(1 - b^{-p}) \times b^{e_{max}}$:

FLT_MAX	8.37988E152
DBL_MAX	8.3798799562141231863E152
LDBL_MAX	8.3798799562141231863E152

- The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating point type, b^{1-p}:

FLT_EPSILON	1.1920928E-7
DBL_EPSILON	1.7347234759768070944E-18
LDBL_EPSILON	1.7347234759768070944E-18

- Minimum normalized positive floating-point number, $b^{e_{min}-1}$:

FLT_MIN	4.661463E-156
DBL_MIN	4.66146295700012921 46E-156
LDBL_MIN	4.6614629570001292146E-156

Summary of Floating-Point Representation

The following summarizes the floating-point representation and the appropriate values in a `<float.h>` header for type `float` :

$$x_f = s \times 2^e \times \sum_{k=1}^{6} f_k \times 16^{-k}, \quad -128 \le e \le +127$$

$$x_d = s \times 2^e \times \sum_{k=1}^{15} f_k \times 16^{-k}, \quad -128 \le e \le +127$$

FLT_RADIX	16
FLT_MANT_DIG	6
FLT_EPSILON	1.1920928E-7
FLT_DIG	8
FLT_MIN_EXP	-129
FLT_MIN	4.661463E-156
FLT_MIN_10_EXP	-155
FLT_MAX_EXP	+126
FLT_MAX	8.37988E152
FLT_MAX_10_EXP	+152
DBL_MANT_DIG	15
DBL_EPSILON	1.7347234759768070944E-18
DBL_DIG	20
DBL_MIN_EXP	-129
DBL_MIN	4.6614629570001292146E-156
DBL_MIN_10_EXP	-155
DBL_MAX_EXP	+126
DBL_MAX	8.3798799562141231863E152
DBL_MAX_10_EXP	+152
LDBL_MANT_DIG	15
LDBL_DIG	20
LDBL_MIN_EXP	-129
LDBL_MIN_10_EXP	-155
LDBL_MAX_EXP	+126
LDBL_MAX_10_EXP	+152
LDBL_MAX	8.3798799562141231863E152
LDBL_EPSILON	1.7347234759768070944E-18
LDBL_MIN	4.6614629570001292146E-156

Index

CUT ALONG LINE

Technical Publications Remarks Form

TITLE	CP-6 C Language Reference	ORDER NO.	HA17-00
		DATED	JUNE 1990

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

PLEASE FILL IN COMPLETE ADDRESS BELOW.

FROM: NAME ______________________ DATE __________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

ASE FOLD AND TAPE -
TE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.39531 WALTHAM, MA

POSTAGE WILL BE PAID BY ADDRESSEE

BULL HN Information Systems Inc.
ATTN: Publications - MA02-305C
Technology Park
Billerica, MA 01821-9904

Bull

Bull HN Information Systems Inc.

Corporate Headquarters: Technology Park, Billerica, MA 01821-4199
Mexico: Hamburgo No. 64, Col. Juarez Delegacion Cuauhtemoc, 06600 Mexico, D.F.
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH, England **Italy:** 32 Via Pirelli, 20124 Milan
Canada: 155 Gordon Baker Rd., North York, Ontario, M2H 3P9 **New Zealand:** 14/16 Liverpool Street, Auckland 1
Asia: 4/F, Shui on Centre, 6-8 Harbour Rd., Wanchai, Hong Kong **Australia:** 124 Walker Street, North Sydney, N.S.W. 2060

Printed in U.S.A.

HA17-00