UNISYS

ClearPath Enterprise Servers

C Programming Reference Manual

Volume 1: Basic Implementation

ClearPath MCP 17.0

April 2015 8600 2268–206

Contents

Section 3. Declarations

Section 4. Type Conversions

Section 5. Expressions and Operators

Section 7. Functions

Section 8. The C Preprocessor

Section 9. Compiler Operations

Section 10. Compiler Control Options

Appendix A. Interface to the Library Facility

Figures

Tables

Examples

Section 1
Overview of the C Language

The C programming language is a general purpose programming language that features a variety of operators, modern program flow control, and modern data structures. The C language is not specialized to any particular area of application. It satisfies requirements of both high and low-level programming. Since the language has few restrictions, programmers are free to do just about any type of programming, hence the power and desirability of the C language.

The C language does not provide any input and output facilities, mathematical functions, string processing routines, and other services itself. They are, however, provided in an associated C library. Refer to Volume 2 for specific information on the library functions.

This section provides an introduction to the C language and defines commonly used program and language elements such as functions, types, identifiers, and constants.

Documentation Updates

This document contains all the information that was available at the time of publication. Changes identified after the release of this document are included in problem list entry (PLE) 19024997. To obtain a copy of the PLE, contact your Unisys representative or access the current PLE from the Unisys Product Support website:

http://www.support.unisys.com/all/ple/19024997

Note: *If you are not logged into the Product Support site, you will be asked to do so.*

What's New

New or Revised Information	Location
Removed all references to BIND_LARGE_PROGRAM.	Throughout document.
Updated LI_SUFFIX information on recently-set values in statements.	Section 10 Compiler Control Options

The Implementation of the C Language

The implementation of the C language is based on the *ANSI/ISO 9899-1990 (revision and redesignation of ANSI X3.159-1989) American National Standard for*

Programming Languages-C. The implementation also provides extensions, including cross-reference files, libraries, and compiler control options.

The Portable Operating System Interface (POSIX) Interface

The implementation contains some but not all of the POSIX headers and functions. The implementation is based on the *ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*. This standard is also referred to as the POSIX standard in this manual. The POSIX features discussed throughout the manual are identified.

Refer to the *C Programming Reference Manual, Volume 2* for more information about the POSIX interface.

The A Series X/Open Interface

X/Open is an unofficial standard based on the POSIX interface. The current implementation on A Series systems contains some headers and functions of X/Open, but not all of the X/Open headers and functions. This implementation of the X/Open interface is based on the *X/Open Portability Guide*, Issue 3, of the X/Open System Interface (XSI). The X/Open features discussed throughout the manual are clearly identified. Refer to Volume 2, "Headers and Functions," for more information on the X/Open interface.

The TransIT Open/OLTP for A Series Interface

TransIT Open/OLTP for A Series interface implements a client/server model. Clients invoke services but do not update the databases directly. Service providers can provide multiple services where a service is used to update a DMSII database. The client/server model applies to COMS online programs only because the implementation of services is provided by COMS.

Open/OLTP is based on the X/Open Distributed Transaction Processing (DTP) model, which is specified in standards developed by the X/Open Company, Ltd.

The following procedures describe the logic for a client:

1. Open the databases.
2. Start a global transaction.
3. Call Service 1 routine to debit savings account.
4. Call Service 2 routine to credit mutual fund account.
5. If services complete successfully, then commit global transaction. Otherwise, if services complete unsuccessfully, then rollback global transaction.
6. Close databases.

You can access Open/OLTP with C by incorporating the C header files included on the release media. The *TransIT Open/OLTP for A Series Programming Guide* explains the procedures for accessing Open/OLTP. This manual describes the following service functions related to the Open/OLTP interfaces:

- GET_BUFFER_DESIGNATOR
- GET_DESIGNATOR_USING_DESIGNATOR
- GET_ERRORTEXT_USING_NUMBER
- GET_INTEGER_USING_DESIGNATOR

For more information about creating applications using Open/OLTP, refer to the *TransIT Open/OLTP for A Series Programming Guide*.

Program Elements

A C program can contain the following program elements:

- Data types
- Variables and labels
- Functions
- Statements
- Declarations
- Preprocessor directives

These program elements are discussed in the following pages.

Data Types

A data type is a set of values and a set of operations that can be performed on those values. In the C language, each data type has its own set of values and its own set of operations. There are three groups of data types in the C language:

Data Type Group	Description
Basic data types	The basic data types consist of character, integer, and floating-point values.
Derived data types	The derived data types consist of combinations of basic data types. The derived data types include arrays, pointers, structures, unions, enumerations, and functions.
`Void` type	The `void` type is an empty set of values.

See Also

- Refer to Section 2, "Types," for more information on data types and data specifiers.
- Refer to Section 3, "Declarations," for information on type specifiers.

Statements

Statements perform actions and determine the flow of control. The following keywords indicate statements in the C language:

- `break`
- `continue`
- `do`
- `for`
- `goto`
- `if`
- `return`
- `switch`
- `while`

The following are several forms of statements:

Statement Form	Description
Labeled statement	A statement preceded by a named, case, or default label.
Compound statement	An optional sequence of declarations followed by a sequence of statements.
Expression statement	An expression followed by a semicolon.
Null statement	A statement that consists of a semicolon only and performs no operations.
Control statement	A statement that performs conditional executions when a particular condition is met.
Iteration statement	A statement that causes a loop body to be executed repeatedly depending on a controlling expression.
Jump statement	A statement that causes an unconditional jump to another statement.

See Also

Refer to Section 6, "Statements," for more information about statements.

Functions

In the C language, functions are similar to subroutines or procedures in other languages. Functions are separately defined sections of code that can be called by other functions. The following information is included when defining a function:

- The name of the function
- The type and number of formal arguments
- A storage class specifier
- The statements to be executed when the function is called
- The type of the value returned by the function, if any

See Also

- Refer to Section 2, "Types," for information on function types.
- Refer to Section 7, "Functions," for more information on function formats, argument passing, and return values.

Declarations

Declarations define variables. The following are four groups of declarators:

Declarator Group	Description
Simple declarator	The simple declarator defines variables with a data type of arithmetic, structure, union, and enumeration.
Pointer declarator	The pointer declarator declares variables of pointer types.
Array declarator	The array declarator declares objects of array types.
Function declarator	The function declarator declares objects of function types.

See Also

Refer to Section 3, "Declarations," for more information on declarators.

Preprocessor Directives

The preprocessor processes the source code of a C program before compilation begins. The C preprocessor is capable of macro substitution, conditional compilation, and inclusion of named files.

The preprocessor can be used to develop programs that are easier to read, develop, modify, and transport to another system. The preprocessor can also be used to customize the C language for a particular application.

The C preprocessor acts on directives that appear on special preprocessor control lines in the source file. There are two frequently used preprocessor directives: the

`#include` directive, which inserts text from another file, and the `#define` directive, which defines a preprocessor macro.

See Also

Refer to Section 8, "The C Preprocessor," for more information on the preprocessor directives.

Language Elements

The following language elements of C are discussed:

- Tokens
  - Identifiers
  - Constants
  - Operators
  - Keywords
  - String literals
  - Punctuators
- Character sets
- White space
- Comments

Tokens

The token is the smallest lexical element of the C language. The compiler collects the characters in a C source program into tokens. White space or comments separate adjacent keywords, identifiers, and constants. The C language contains six classes of tokens:

- Identifiers
- Constants
- Operators
- Keywords
- String literals
- Punctuators

When the C compiler scans the source lines, it forms each token from the longest possible sequence of characters going from left to right, using the syntax rules of the language.

Examples

If the following expression is written:

```
x+++++y
```

The compiler identifies the following sequence of five tokens:

```
x ++ ++ + y
```

This is not a valid expression even though the following expression is a valid expression:

```
x++ + ++y
```

A style that uses white space prevents ambiguity.

See Also

- Refer to "Identifiers" in this section for more information on identifiers.
- Refer to Section 5, "Expressions and Operators," for more information on the type, class, side effects, and order of evaluation of expressions.

Identifiers

An identifier is the name of one of the following program entities:

- Variable
- Tag member of a structure, union, or enumeration
- Function
- Member of a structure or union
- Enumeration
- Type
- Label
- Macro name

Allowable Characters

An identifier is a sequence of uppercase letters, lowercase letters, digits, and underscores. The identifier does not end until some other character is written. An identifier can fill an entire line.

The following rules apply for specifying identifiers:

- An identifier can contain only uppercase letters, lowercase letters, digits, and underscores.

- The first character of an identifier must be an uppercase letter, lowercase letter, or an underscore.
  - All identifiers beginning with an underscore are considered reserved.
  - Do not begin identifier names with an underscore, since they may conflict with one of the names internal to the C library.
- Except for a macro name, an identifier cannot be identical to a keyword.
- The first 250 characters of an internal identifier are significant.
- The first 250 characters of an external identifier are significant to the Binder.
- The C language is case sensitive. Therefore, the identifiers abc, Abc, ABc, abC, and ABC are all unique identifiers.
- Some other languages do not allow lowercase letters or translate lowercase letters to uppercase. If an identifier is to be used by another language, it should not contain lowercase letters.

Programming Conventions

The following conventions in the choice of identifiers are not part of the C language. They are, however, followed by many C programmers. The purpose of these conventions is to produce programs that are easier to understand and easier to move to a different computer system.

- Although the compiler is case sensitive, identifiers should differ in more than just their case. For example, it is generally considered poor programming style to have two identifiers called `number` and `Number`.
- Preprocessor macro names are usually all uppercase letters.

See Also

Refer to "Macro Definition and Replacement Examples" in Section 8 for more information.

Scope

Scope defines where in source text an identifier can be used and when it can be reused.

An identifier is "visible" (can be used) only within a region of a program called its scope. The scope of an identifier is the region of a program where its declaration or definition is in effect. There are four kinds of scope:

Type of Scope	Description
File scope	If the declaration of an identifier is outside all blocks, the identifier has file scope. File scope starts at the completion of the identifier's declarator and ends at the end of the file. File scope includes all files included by a preprocessor directive. All macro names have file scope.
Function scope	A label is the only identifier that has function scope. By using the `goto` statement, a label can be used anywhere in the function in which it is declared. Labels must be unique within a function.
Block scope	An identifier has block scope if its declaration appears within a block or in the list of parameter identifiers in a function definition. Block scope starts at the completion of the identifier's declarator and ends at the right brace (}) that closes the associated block. The scope of a parameter ends at the end of the function definition. If a declaration in an inner block contains the same identifier as one in an outer block, the outer block's declaration is hidden (not visible) until the right brace (}) closes the inner block. After the brace, the declaration of the identifier in the outer block is visible.
Function prototype scope	An identifier has function prototype scope if its declarator appears within the list of parameters in a function prototype. A function prototype is a declaration of a function that declares the types of its parameters. Function prototype scope starts at the completion of the parameter's declarator and ends at the end of the function prototype in a declaration.

See Also

- Refer to "Declarations," in Section 3 for more information the scope of declarations.
- Refer to "Function Call Operator" in Section 5 for more information on the function call operator.
- Refer to "Labeled Statements," "Compound Statements," and "Goto Statement" in Section 6 for more information on the scope and use of these statements.
- Refer to "Defining Functions" in Section 7 for more information on function definitions.
- Refer to "#include Directive" in Section 8 for more information the scope of the `#include` directive.

Linkages

Linkage is the process of allowing two identical identifiers declared in different scopes to refer to the same function or object. There are three kinds of linkage:

Type of Linkage	Description
External linkage	In the entire set of source files and libraries that constitute the program, every instance of a particular identifier with external linkage refers to the same function or object. If the declarations in two source files or in disjoint scopes within one source file differ, the result is undefined.
Internal linkage	In the same source file, each instance of an identifier with internal linkage refers to the same function or object.
No linkage	Identifiers with no linkage refer to certain identifiers. The following identifiers have no linkage: • An identifier that declares anything other than a function or an object. • An identifier declared as a function parameter. • An identifier that declares an object within a block without the keyword `extern`.

An identifier of a function or object declared with file scope has internal linkage if its first declaration in the source file contains the keyword `static`. If the declaration does not contain the keyword `static`, the identifier has external linkage.

If the declaration of a function or object contains the keyword `extern`, the identifier has the same linkage as the previous declaration of the identifier with file scope. If the identifier was not previously declared with file scope, the identifier has external linkage.

If an identifier declared with external linkage is used in an expression, somewhere in the entire program there must be exactly one external definition for the identifier. The `extern` keyword in a file indicates that storage for the identifiers being declared is allocated in another file. In a multifile program, an external data definition without the `extern` specifier must appear in exactly one of those files. Any other files that contain an external definition for the identifier must include the `extern` keyword in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers in inner blocks declared with `extern` are considered to be that of any previous declaration of the identifier with file scope. Therefore, array size information can be provided on an inner declaration and not on an outer declaration if both declarations use `extern`. However, dimension information is relevant only in the inner block.

Example

The following example illustrates the accumulation of information about identifiers in inner blocks declared with extern:

```
extern x[];
{ extern x[5];
  sizeof(x);       /* legal - will get total size of array
                              in bytes */
}
sizeof(x);       /* illegal */
```

At the second sizeof operator, the dimension of the array is not known. The dimension is known only inside the inner block.

See Also

- Refer to Section 3, "Declarations," for more information on linkages to data declarations.
- Refer to Section 5, "Expressions and Operators," for more information on linkages to expressions.
- Refer to "Compound Statements" in Section 6 for more information on linkages to statements.

Name Spaces

In the C language, the same identifier can be declared for more than one program entity. When one identifier is associated with more than one program entity, the identifier is said to be "overloaded." The C compiler uses the syntactic context to determine which declaration to use. For example, an identifier might be both the name of a variable and a union tag. When used in a type specifier, the union tag association is used. If used in an expression, the variable is used.

When the name of an identifier is overloaded with several associations, each association has its own scope. The association can be visible or hidden by other declarations. For example, if an identifier is declared as both a variable and a union tag, an inner block can redefine the variable association without altering the union tag association.

Because of their associations, the C language considers some identifiers to have separate name spaces. There are six name spaces (or overloading classes) for identifiers.

Name Spaces	Description
Macros	Macro identifiers occupy their own name space. If an identifier is defined as a macro, the compiler treats it as a macro name in all contexts. Keywords can be used as macro names, although this is not recommended.
Keywords	Keywords occupy every other name space, except that of macros.

Name Spaces	Description
Labels	Identifiers used as statement labels occupy their own name space. The compiler can distinguish the name space because the identifier always appears as a name followed by a colon (:) in the definition of the label, or follows the keyword `goto` in the use of the label.
Structure, union, or enumerations tags	Structure, union, and enumeration tags occupy their own name space. The keywords `struct`, `union`, and `enum` always precede these tags so the compiler can distinguish their name space. Even though the keywords differentiate the tags, the C language contains only one name space for tags.
Structure or union members (component names)	Each structure or union has a separate name space for its members. This enables an identifier to be component names in any number of structures or unions at the same time. The members of structures or unions are always defined within the structure or union. A component name immediately follows the selection operations of the direct member selection operator (.) or the indirect member selection operator (->).
Ordinary identifiers	All other identifiers, called ordinary identifiers, fall into one name space. This name space includes variables, functions, `typedef` names, and enumeration constants.

Example

The following example is a legal use of overloading names:

```
struct s {int s;};
struct s s;
extern foo (int s);
main () {
        int i;
        goto s;
        i = s.s;
        s:;
        }
```

See Also

- Refer to “Enumeration Types,” “Structure Types,” and “Union Types” in Section 2 for more information on the use of name spaces.
- Refer to “Declarators” in Section 3 for more information on the use of name spaces for declarations.
- Refer to “Labeled Statements” in Section 6 for more information on the use of name spaces for labeled statements.
- Refer to Volume 2, “Headers and Functions,” for information on the use of name spaces in header files.

Syntax

The following syntax describes 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

A constant is the explicit representation of a value. In the C language, any literal number, single character (delimited by single quotes), or character string (delimited by double quotes) is a constant. There are five kinds of constants:

- Integer constants
- Floating-point constants
- Character constants
- Enumeration constants
- String constants

Every constant in a C program has a type. The form and value of the constant determines its type. The value of the constant must be representable in the storage appropriate for its type in the execution environment.

Integer Constants

An integer constant is a sequence of digits. A minus sign can precede the digits of an integer constant. However, the minus sign is a unary operator that is applied to the integer constant. It is not considered to be part of the constant.

Integer constants can be specified in decimal, octal, or hexadecimal notations.

Integer Constant	Description
Decimal	A decimal integer constant is a sequence of decimal digits (`0` through `9`). The first digit cannot be `0`.
Octal	An octal integer constant consists of a sequence of octal digits (`0` through `7`). The first digit in the sequence must be `0`.
Hexadecimal	A hexadecimal integer constant is a `0x` (or `0X`) followed by a sequence of decimal digits (`0` through `9`) and the letters `a` (or `A`) through `f` (or `F`).

The type of an integer constant is `int`. Add a suffix to any of these forms to control the type more closely. The suffix `L` or `l` indicates that the constant is type `long`. The suffix `U` or `u` indicates that the constant is type `unsigned`. The suffixes can be added in either order.

Examples

Integer Constant	Explanation
16	Decimal notation
020	Octal notation for decimal 16
0X10	Hexadecimal notation for decimal 16

Table 1-1 lists the integer constant limits for this C language implementation.

Table 1–1. Integer Constant Limits

Data Item	Value
short	1-2**39 to 2**39- 1
long	1-2**39 to 2**39 - 1
int	1-2**39 to 2**39 - 1
unsigned	0 to 2**39 - 1
signed integer	1-2**39 to 2**39 -1
unsigned integer	0 to 2**39 - 1
signed long	1-2**39 to 2**39 - 1
unsigned long	0 to 2**39 - 1

Any integer constant that cannot be represented in its type causes the compiler to issue an error message.

Syntax

The following syntax describes integer constants and their decimal, octal, and hexadecimal notations:

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
   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

Floating-Point Constants

A floating-point constant always uses decimal digits. The exponent is always a power of 10. The value of a floating-point constant is always nonnegative. A minus sign can precede a floating-point constant. If a minus sign precedes the constant, the minus sign is a unary operator that is applied to the floating-point constant. It is not considered part of the constant.

Examples

The following are all valid examples of floating-point constants:

```
0.              4e2          4.2e3
.0              3E-1         2.25e-4
2.0             125e-3       1.0E10
137.8536        67E4
```

Without a suffix, the type of a floating-point constant is `double`. If the letter `f` or `F` follows the floating-point constant, the type is `float`. If the letter `l` or `L` follows the floating-point constant, the type is `long double`.

If the size of the floating-point constant is too large or too small for the type to represent, the compiler issues an error message.

Table 1-2 lists the limits for floating-point constants in this C language implementation. For the `float` type, the hardware can have a digit precision of 11 digits. Rounding errors introduced by arithmetic operations make the eleventh digit inaccurate. The limits of a `double` are the same as either `float` or `long double`, depending on the value of the DBLTOSNGL compiler control option.

Table 1–2. Floating-Point Constant Limits

Data Item	Value
`float` digits of precision	10
`float` exponent range	8.75811540204E-47 to 4.31359146674E68
`long double` digits of precision	22
`long double` exponent range	1.93854585714E-29581 to 1.94882838205E29603

Syntax

The following syntax describes floating-point constants:

```
floating-point constant:
     fractional-constant exponent-partopt floating-suffixopt
     digit-sequence exponent-part floating-suffixopt

fractional-constant:
     digit-sequenceopt . digit sequence
     digit-sequence .

exponent-part :
     e signopt   digit-sequence
     E signopt   digit-sequence

sign : one of
     +   -

digit-sequence:
     digit
     digit-sequence digit

floating-suffix: one of
     f  l   F   L
```

Character Constants

A character constant is a sequence of one or more characters enclosed in single quotes (or apostrophes).

A wide character constant is a character constant prefixed by the letter L. A wide character constant has type `wchar_t`, an unsigned short integer defined in the `<stddef.h>` header.

With a few exceptions, the characters that can appear in a character constant are the characters in the source character set. The exceptions are the newline, backslash, and single quote characters. However, these characters can be entered by using escape sequences. Escape sequences can represent the single quote, double quote, question mark, backslash, and any arbitrary octal or hexadecimal value. Table 1-3 lists the escape sequences for these characters.

Table 1–3. Escape Sequences

Character	Escape Sequence
single quote (')	\'
double quote (")	\"
question mark (?)	\?

Table 1–3. Escape Sequences

Character	Escape Sequence
backslash (\ )	\\
octal integer	\ddd
hexadecimal integer	\xddd
bell (BEL)	\a
backspace (BS)	\b
horizontal tab (HT)	\t
vertical tab (VT)	\v
form feed (FF)	\f
carriage return (CR)	\r
newline (LF)	\n

To represent the double quote and the question mark, use either the characters themselves or the escape sequence. The escape sequence \' must be used to represent the single quote.

The escape character representations are only used within strings or character constants. They were created for that purpose only. The escape characters are independent of the host or target computer. They perform their function on any terminal provided that terminal is capable of performing this function.

The newline formatting character indicates the end of a line.

All characters can be specified by the *ddd* or *xddd* character sequences. The *ddd* character sequence represents from one to three octal (*ddd*) or one or more hexadecimal (*xddd*) digits. The digits represent the octal or hexadecimal value of the character in the EBCDIC character set. For example, represent the letter A in the EBCDIC character set by using the \301 or \xC1 escape sequences. 301 is the octal value and C1 is the hexadecimal value of the letter A in the EBCDIC character set.

An escape sequence with a backslash and lowercase letter other than a, b, f, n, r, t, or v is diagnosed as an error.

Refer to "Source Character Set" in this section for more information.

Examples

Character Constant	Explanation
'a'	Letter a
'\201'	Letter a (EBCDIC)
'\n'	Newline character
'\327'	Letter P (EBCDIC)
'\''	Apostrophe or single quote
'\0'	Null character
'\xD7'	Letter P (EBCDIC)

Character constants are type `int`. The value of the character constant is the numerical value of that character in the EBCDIC or ASCII character set, depending on the value of the STRINGS compiler control option (the default is EBCDIC). Refer to Appendix F, "The ASCII and EBCDIC Character Sets," for details on the EBCDIC character set.

The C compiler allows multicharacter constants like 'ABC'. The purpose of this is to enable programmers to create an integer value (not a string) from the characters. The compiler allows character constants up to four characters long. Because characters have different sizes on different computers, the use of this feature limits the portability of a program.

Syntax

The following syntax describes the character constants:

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

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

c-char:
   any character in the source character set, except the
       single-quote ('), backslash (\), or newline character
   escape-sequence

escape-sequence: one of
       \   escape-sequence-character
       \   octal-digit_opt octal-digit_opt octal-digit
       \x  hexadecimal-code

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

hexadecimal-code: one of
hexadecimal-code$_{opt}$ *hexadecimal-digit*

String Constants

A string constant is a sequence of zero or more characters enclosed in double quotes.

A wide string constant is a string constant prefixed by the letter L.

Like character constants, the printing characters include all the characters in the source character set, except the newline, double quote, and the backslash characters. Use the escape sequences for these characters in a string constant. For example, to include the double quote in a string constant, use the escape sequence \" wherever the double quote (") character is needed.

A string cannot cross a line boundary unless the line is ended with a character backslash (\). If a line is ended with a character backslash (\), the string continues at the beginning of the next line. Any spaces between the beginning of the next line and the first character are considered to be part of the string constant.

Strings that are adjacent tokens are concatenated.

A newline character (\n) can be included in a string constant. When the string constant is printed, the characters after the newline character are on the next line.

Examples

String Constant	Explanation
`"\""`	This represents a double quote.
`""`	This represents an empty string constant.

`"This string \`
`constant uses \`
`three lines."`
	This represents a string continuation across lines using line continuation.

`"This string"`
`"constant uses"`
`"three lines."`
	This represents a string continuation using implicit continuation of adjacent string tokens.

In the C language, a string constant is actually a character array. For example, the string constant `"ABCDEF"` is the same as the following character array:

```
| A | B | C | D | E | F | \0 |
  0   1   2   3   4   5    6
```

For every string constant of *n* characters, the compiler allocates a block of *n*+1 characters. The first *n* characters are initialized with the characters from the string constant. By convention, the null character (\0) ends a string in the C language. With string constants, the compiler automatically adds the null character to the end of the string. Even though a string constant is a constant, the compiler does not prevent storage of new values in the string constant during program execution.

The type of a string constant is an array of `char`.

The type of a wide string constant is an array of `wchar_t`, which is an unsigned short integer defined in the `<stddef.h>` header. The wide string constant `L"ABCDEF"` becomes an array of seven unsigned short integers of type `wchar_t` shown as follows:

```
{L'A', L'B', L'C', L'D', L'E', L'F', L'\0'}
```

Syntax

The following syntax describes string constants:

```
string-constant:
     "s-char-sequence_opt"
     L"s-char-sequence_opt"

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

s-char:
     any character in the source character set, except the
         double-quote ("), backslash (\), or newline character
     escape-sequence
```

Operators

An operator specifies an operation to be performed that yields a value (an evaluation). Operators act on operands. An expression contains one or more operators and operands that specify how to compute a value, or in the case of a `void` expression, how to generate side effects.

The one-character operators in the C language are the following:

```
!  %  ^  &  *  -  +  =|  >  <  >  /  ?  #  ,  .
```

The multicharacter operators in the C language are the following:

```
->   ++   --   <<   >>   <=   >=   ==   !=   &&   ||   ##

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

The other miscellaneous operators in the C language are the following:

```
( ) [ ]  { }  ,  ;  :
```

The following operators must always occur in pairs, possibly separated by expressions:

```
[ ]
( )
? :
```

The number sign (#) and the double number sign (##) operators occur in macro-defining preprocessor directives only.

A white space may not appear inside a multicharacter operator. For example, `x + = y` must be written in the following way:

```
x += y
```

See Also

- Refer to Section 5, "Expressions and Operators," for more information about expressions.
- Refer to "# Preprocessor Operator," and "## Preprocessor Operator" in Section 8 for more information about preprocessor directives.

Keywords

The C language reserves the following tokens for use as keywords (also called reserved words); use them only for their intended use as keywords.

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

Note: *Although the C language enables the use of a keyword as a preprocessor macro name, it is not recommended.*

Character Sets

This C language implementation defines two character sets:

Character Set	Description
Source	This is the character set in which C programs are written.
Target	This is the character set interpreted in the execution environment.

Source Character Set

Table 1-4 shows the source character set for this C language implementation.

Table 1–4. C Language Source Character Set

Classification of Source Characters	Literal Characters
52 uppercase and lowercase characters 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 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
10 decimal digits	0 1 2 3 4 5 6 7 8 9
32 graphics characters	& ampersand * asterisk @ at sign \ backslash ^ caret , comma $ currency symbol : colon " double quote = equal sign ! exclamation point ` grave accent > greater than - hyphen { left brace [left bracket (left parenthesis < less than sign % percent sign . period # number + plus sign ? question mark } right brace] right bracket) right parenthesis ; semicolon ' single quote / slash ~ tilde _ underscore | vertical bar
White space characters	Space character Horizontal tab Carriage return Vertical tab Form feed
Newline	This is represented in a source file by the end of a source record and all consecutive space characters at the end of the record.

Source Lines

In this C language implementation, a source line cannot contain more than 509 characters.

Terminating a line with a backslash continues the source line. After terminating a line with a backslash, continue a source line to the next line by starting at the very beginning of that next line.

Trigraph Character Sequences

The C compiler replaces each trigraph sequence listed in Table 1-5 with the specified single character in the source character set.

Table 1–5. Trigraph Character Sequences

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

The trigraph character sequences enable the input of characters that are not available on some input devices.

Digraph Character Sequences

The C compiler replaces each digraph sequence listed in Table 1-6 with the specified single character in the source character set.

Table 1–6. Digraph Character Sequences

Digraph Sequence	Source Character
%:	#
<:	[
:>	]
<%	{
%>	}

The digraph character sequences enable the input of characters that are not available on some input devices. Digraph sequences are tokens.

Target Character Set

The target character set is whatever can be represented in 8 bits. This includes the source character set (the 8-bit code of the EBCDIC character set values 0 through 255). Values up to 255 can be stored in a `char` through the use of an octal or hexadecimal escape sequence or a multicharacter character constant.

The target character set can be changed from EBCDIC to 8-bit ASCII. All source characters in character and string constants are translated from EBCDIC to their corresponding position in ASCII. The sign attribute of the plain `char` type (also called `char` type) can also be changed from unsigned to signed.

See Also

Refer to "Character Types" in Section 2 for more information on `char` types.

White Space

The C language requires white space to separate adjacent keywords, identifiers, and constants. Otherwise, the C compiler ignores white space, except when it appears in a character or string constant. White space is normally used to separate tokens and to make programs easier to read. The amount of white space between tokens is not significant.

Legal white space for non preprocessor lines can be generated by any of the following:

- Horizontal tab
- Space
- Newline
- Carriage return
- Vertical tab
- Form feed
- Comments

Legal white space for preprocessor lines can be generated by any of the following:

- Horizontal tab
- Space
- Comments

Note: Most other A Series software products do not correctly handle horizontal tab, carriage return, vertical tab, and form feed commands when used as white space. These should be avoided whenever possible.

Comments

In the C language, the /* character sequence begins a comment and the */ character sequence ends a comment. A comment must end before the end of the logical file in which it begins. A Series C allows C++ style comments, where the // character sequence begins the comment and the logical end of the line ends the comment.

The C compiler treats the comment as white space.

Note: *Comments cannot be nested.*

Examples

The text between the /* character sequence and the */ character sequence illustrates a comment in the C language.

```
/* This is a comment in the C language. */

// This is a C++ style comment.

#define LOWER 0      /* lower limit of table */
#define UPPER 500    /* upper limit */

#include <stdio.h>
main ( )             /* Fahrenheit - Celsius Table */
{
  int  fahren;
  for (fahren = 0; fahren <= 300; fahren=fahren + 20)
   printf ("%5d %6.1f\n",fahren,(5.0/9.0) * (fahren - 32));
}
```

The C preprocessor replaces comments with white space. The preprocessor ignores the text between the comment delimiters and does not examine that text for commands or macro calls.

Section 2
Types

A data type is a set of values and a set of operations. For example, the integer data type accepts integers only within a specific range. In this C language implementation, the integer range is 1-2**39 to 2**39-1. The operations that can be performed on integer data are integer addition, integer subtraction, integer division, integer multiplication, and so forth.

Summary of Data Types and Data Type Specifiers

The C language divides data types into three groups

- Basic (or fundamental) data types
- Derived data types
- `void` type

Tables 2–1 summarizes the basic data types and the type specifiers that correspond to each data type.

Table 2–1. Summary of Basic Data Types and Data Type Specifiers

Data Type	Type Specifier	Description
Character	`char` `unsigned char`	Represents an unsigned whole number in 8 bits (1 byte).
	`signed char`	Represents a signed whole number in 8 bits (1 byte).
Floating-point	`float`	Represents a real number in 48 bits (1 word).
	`double`	Represents a real number in 48 bits (1 word) or a real number in 96 bits (2 words), depending on the value of the DBTOSNGL compiler control option. The default is `float`.
	`long double`	Represents a real number in 96 bits (2 words).

Table 2–1. Summary of Basic Data Types and Data Type Specifiers

Data Type	Type Specifier	Description

Integer	`int`
	`signed`
	`signed int`
	`short int`
	`signed short int`
	`long int`
	`signed long int`
		Represents a signed whole number in 48 bits (1 word).

	`unsigned`
	`unsigned int`
	`unsigned short int`
	`unsigned long int`
		Represents an unsigned whole number in 48 bits (1 word).

Table 2–2 summarizes the derived data types and the data type specifiers that correspond to each data type.

Table 2–2. Summary of Derived Data Types and Data Type Specifiers

Data Type	Type Specifier	Description
Array		An array is a set of contiguously allocated members of any one type of object.
Enumeration	`enum`	An enumeration is a set of named integer constants.
Pointer		Pointers point to functions, to objects of any type, and to void.
Structure	`struct`	A structure is a set of sequentially allocated named members of various types of objects.
Union	`union`	A union is a set of overlapping named members of various types of objects.

The `void` type specifies an empty set of values. No object can have type `void`.

Basic Data Types

The basic data types are

- Character
- Floating-point
- Integer

Character Types

Character types are used to contain a single character. The type specifiers for the character types are

- `char` (also called plain `char`)
- `unsigned char`
- `signed char`

In the C language, the character type is considered an integral type, but not an integer type. An integral type consists of character, integer, and enumeration types. The values associated with a `type char` variable are integers and can be used in integer expressions.

An object declared as a character (`char`) is large enough to store any member of the source character set. The values of the characters in the source character set are always guaranteed to have nonnegative values. These values range from 0 to 255. The C compiler does not sign-extend `char` and `unsigned char` types. These character types are widened to an integer by zero-filling. However, a `signed char` character type is sign-extended.

The following declarations declare character data:

```
char char_var = 'W';
char ch1 = '\0';
char ch2 = '\xFF';
```

The C language stores a character value as the integer that corresponds to the internal representation (encoding) of the character, except for negative values stored in a `signed char`. These values are stored as if they were subtracted from 256 (as an eight-bit two's complement value). For example, -1 is stored as 255, -128 is stored as 128.

By default, a negative value is stored as is in an `unsigned char`. In order to have `unsigned chars` stored as an eight-bit two's complement value, the compiler control option PORT(CHAR2) should be set.

The sign of a plain `char` is important. As stated earlier, the C compiler does not sign-extend a plain `char`. A plain `char` is widened to an integer by zero-filling. Traditionally, C programming uses `char` to contain negative values, typically in I/O functions where -1 means end-of-file. This is not a portable method using plain `chars` and will not work in

this C language implementation. Functions that return -1 or objects that contain a value of -1 must be declared as `signed` in order to be portable.

By default, a plain `char` is equivalent to an `unsigned char`. The compiler control option PORT(SIGNEDCHAR) may be used to make plain `char` equivalent to `signed char`.

See Also

- "Character Constants" and "Source Character Set" in Section 1.
- "SIGNEDCHAR" and "CHAR2" suboptions under "PORT Option" in Section 10.

Floating-Point Types

The floating-point types are used to contain real numbers. The C language contains a single-precision floating-point type and a double-precision floating-point type. The two type specifiers for single-precision floating-point types are `float` and `double`.

The `long double` type specifier is for double-precision floating-point types.

Examples

The following examples use the floating-point types:

```
double float_var = 3.31379e-16;
float value1, value2;
float absolute_value ( );  /* absolute_value is a function
                           that returns a value of type float */
double exp = 1E20;
```

Table 2–3 lists the size and range of the various types of floating-point types.

Table 2–3. Size and Range of Floating-Point Types

Type	Size	Range
float double	48 bits	8.75811540204E-47 to 4.31359146674E68
long double	96 bits	1.93854585714E-29581 to 1.94882838205E29603

The A Series hardware provides a range and precision for single-precision floating point types sufficient for most use of `double` type. If a C program requires the range and precision of double precision for `double` type, the compiler control option DBLTOSNGL (DouBLe TO SiNGLe) can be reset.

See Also

- "Floating-Point Constants" in Section 1.
- "Floating-Point to Floating-Point" in Section 4 (for information on floating-point conversions).

- "DBLTOSNGL Option" in Section 10.

Integer Types

The C language provides a wide variety of integer types. The variety reflects the different sizes and signed or unsigned values. Integer types can be divided into the following three classes:

- Plain integer types
- Signed integer types
- Unsigned integer types

Plain Integer Types

The type specifiers for plain integer types are

- `short`
- `short int`
- `int`
- `long`
- `long int`

Plain integers are implemented as signed integers.

Examples

The following declarations declare plain integers:

```
int index = 0;
long value;
short i, j;
```

Signed Integer Types

The type specifiers for the signed integer types are:

- `signed`
- `signed short int`
- `signed short`
- `signed int`
- `signed long int`
- `signed long`

Examples

The following declarations declare signed integers:

```
signed int index = 0;
signed long int convert_number;
signed short i,j;
```

Table 2–4 lists the size and range of plain and signed integer types.

Table 2–4. Size and Range of Plain and Signed Integer Types

Type	Size	Range
short signed short signed short int	48 bits	1-2**39 to 2**39 -1
int signed signed int	48 bits	1-2**39 to 2**39 -1
long signed long signed long int	48 bits	1-2**39 to 2**39 -1

See Also

- "Integer Constants" in Section 1.
- "Integral to Integral" in Section 4 (for information on integer type conversions).

Unsigned Integer Types

The type specifiers for the unsigned integer types are

- `unsigned short int`
- `unsigned short`
- `unsigned int`
- `unsigned`
- `unsigned long int`
- `unsigned long`

Examples

The following declarations declare unsigned integers:

```
unsigned int x,y,z;
unsigned short v;
```

Table 2–5 lists the size and range of the various types of unsigned integers.

Table 2–5. Size and Range of Unsigned Integer Types

Type	Size	Range
unsigned short int unsigned short	48 bits	0 to 2**39 - 1
unsigned unsigned int	48 bits	0 to 2**39 - 1
unsigned long int unsigned long	48 bits	0 to 2**39 - 1

See Also

- "Integer Constants" in Section 1.
- "Integral to Integral" in Section 4 (for information on integer type conversions).

Range of C Variables

Table 2–6 summarizes the range of character, floating-point, and integer data types for C variables.

Table 2–6. Range of Data Types

Type	Bits	sizeof	Range
char	8	1	0 to 255
signed char	8	1	-128 to 127
unsigned char	8	1	0 to 255
short int	48	6	1-2**39 to 2**39 -1
signed short	48	6	1-2**39 to 2**39 -1
unsigned short	48	6	0 to 2**39 -1
int	48	6	1-2**39 to 2**39 -1
signed int	48	6	1-2**39 to 2**39 -1
unsigned int	48	6	0 to 2**39 -1
long int	48	6	1-2**39 to 2**39 -1
signed long	48	6	1-2**39 to 2**39 -1
unsigned long	48	6	0 to 2**39 -1

Table 2–6. Range of Data Types

Type	Bits	sizeof	Range
float	48	6	8.75811540204E-47 to 4.31359146674E68
double	48	6	8.75811540204E-47 to 4.31359146674E68
long double	96	12	1.93854585714E-29581 to 1.94882838205E29603

Derived Data Types

A derived data type is a type that is built or constructed from one or more of the basic data types. There are six derived data types.

Derived Data Types	Description
Array	Arrays consist of a set of contiguously-allocated members of any one type of object.
Enumeration	Enumerations comprise a set of named integer constants.
Pointer	Pointers point to functions, to objects of any type, and to `void` type.
Structure	Structures consist of a set of sequentially-allocated named members of various data types.
Union	Unions comprise a set of overlapping named members of various data types.

Array Types

An array is an aggregate object consisting of objects that are called elements. All elements of an array have the same type.

The format of a simple array definition is written as follows:

```
type name[n]
```

type This is the data type of the array. The *type* can be any C type, except `void` or a function type.

name This is an identifier that is being defined as the name of an array.

n This is the number of elements in the array.

In the C language, arrays are 0-origin. For example, `A[3]` consists of the elements `A[0]`, `A[1]`, and `A[2]`.

Syntax

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

type-specifier:
    char
    const
    double
    enum-specifier
    __far
    float
    int
    long
    __near
    short
    signed
    struct-or-union-specifier
    typedef-name
    unsigned
    void
    volatile

type-specifier-list:
    type-specifier
    type-specifier-list type-specifier

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*$_{opt}$)

See Also

- "Array Declarators," "Initializing Arrays," and "Type Equivalence" in Section 3 (for more information on array types).
- "Array to Pointer" in Section 4 (for information on array conversions).
- "Array Subscripting Operator" in Section 5.

Multidimensional Arrays

A multidimensional array is an array of arrays. The C compiler enables as many dimensions as there is room in the internal data structures of the compiler (typically over 100). To be strictly compatible with the ANSI C standard, six dimensions is the limit.

The format of a multidimensional array is written as follows:

type name $[n_1]$ $[n_2]$ $[n_3]$ . . . $[n_n]$

type	This is the data type of the array. The *type* can be any C type, except `void` or a function type. All elements of the array have the same type.
name	This is an identifier that is being defined as the name of an array.
n_n	This is the number of elements in that dimension of the array.

Multidimensional arrays are stored so that the last subscript varies most rapidly. For example, the elements of the array `int x[2][3]` are stored in increasing addresses as follows:

```
X[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2]
```

See Also

- "Array Declarators" and "Initializing Arrays" in Section 3 (for more information on array types).
- "Array Subscripting Operator" in Section 5.

Array Bounds

Any time the compiler allocates storage to an array, it must know the size of the array. However, there are cases when the compiler does not need to know all of the bounds of the array. It is possible to omit the bounds of the first dimension of an array in the following cases:

- When declaring an array defined in another module (an external array) or elsewhere in the same module
- When declaring an array that is a formal parameter to a function

Examples

The following example illustrates the declaration of an external, one-dimensional array whose bounds are not specified:

In File F1	In File F2
`extern int a[];`	`int a[5];`

The following example illustrates an acceptable declaration of a multidimensional array:

```
extern int array[] [3] [5];
```

See Also

- Appendix B, "Internal Compiler Limits," for information on the maximum size of arrays.
- "Array Declarators" in Section 3 for more information on arrays.

- "Array Subscripting Operator" in Section 5 for more information on arrays.

Arrays and Pointers

In the C language, there is a close correspondence between an "array of type" and "pointer to type." Whenever an array appears in an expression except in a `sizeof` or `offsetof` argument, the type is converted from "array of type" to "pointer to type," and the value of the array is converted to a pointer to the first (index zero) element of the array. For example:

```
int x[4], *xp;
xp = x + 2;    /* xp points to the third element, or x[2] */
```

Array subscripting is defined in terms of pointer arithmetic.

Examples

The expression `x[i]` is the same as the following expression:

```
*(x + i)
```

Likewise, the expression `*(x + i)` is the same as the following expression:

```
*(&x[0] + i)
```

See Also

- "Array Declarators" and "Pointer Declarators" in Section 3 for more information on arrays and pointers.
- "Address Operator," "Array Subscripting Operator," and "Indirection Operator" in Section 5 for more information on arrays and pointers.

Enumeration Types

In the C language, an enumeration type is a set of values represented by identifiers. The identifiers are also called enumeration constants.

An object defined as an enumeration type behaves like an `int type` in an expression.

The size of an enumeration type is equal to the size of the `int` type.

The identifiers in an enumerator list are declared as constants that have type `int.` They can appear wherever an `int` is permitted.

The C language implements enumeration types by associating integer values with the enumeration constants. The assignment and comparison of values of enumeration types are implemented as integer assignment and comparison. Integer values are associated with enumeration constants by the following method:

1. If the first enumerator has no equal sign (=), its enumeration constant has value zero (0).

2. An enumerator with an equal sign (=) gives the associated enumeration constant the value of the constant expression.

3. A subsequent enumerator without an equal sign (=) gives its enumeration constant a value of one more than the value of the previous enumeration constant.

The integers chosen determine the equality and ordering relationships among values of the enumeration type.

The scope of an enumeration constant begins after its declaration and ends with the scope of the enumeration of which it is a member. The identifiers of enumeration constants in the same scope must all be distinct from each other and from other identifiers in the ordinary name space (names of variables, functions, `typedefs`, and other enumeration constants).

The tag names a particular enumeration and allows it to be referenced after its definition. The enumeration tag is in the same name space as structure and union tags. The scope of the tag is the same as a variable declared at the same location in the source program.

Example

The following example illustrates the enumeration type specifiers:

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

This enumeration 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 }.

Syntax

The following syntax describes the enumeration type specifier:

```
enum-specifier :
   enum identifier_opt {enumerator-list}
   enum identifier

enumerator-list :
   enumerator
   enumerator-list, enumerator

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

enumeration-constant:
   identifier
```

See Also

- "Identifiers" and "Names Spaces" in Section 1 for more information on enumeration types.
- "Type Equivalence" in Section 3 for more information.
- Section 4, "Type Conversions," for information on enumeration conversions.
- Appendix G, "Syntax Summary," for information on the syntax of *enumeration-constant* and *constant-expression*.

Pointer Types

A pointer is an entity that refers to some region of storage or to a function.

*type *name*

type	This is any type.
*	The asterisk indicates that the variable name is of type pointer.
name	This is an identifier that is being defined as the name of the pointer.

Pointers to objects always compare as equal when pointing to the same object and always compare as unequal when pointing to different objects. The same is true for pointers to functions. However, a pointer to an object can compare as equal to a pointer to a function even though the object and the function are not the same.

Examples

The following examples illustrate the declaration of pointer types:

```
int *x;         /* x is a pointer to an object of type int    */
char *ltr;      /* ltr is a pointer to an object of type char */
```

The pointer value 0 or NULL is associated with all pointer types. The file <stddef.h> contains the definition for the preprocessor macro name NULL. A pointer whose value is NULL is guaranteed not to point to any object or function.

Syntax

pointer
   **type-specifier-list*$_{opt}$
   **type-specifier-list*$_{opt}$ *pointer*

type-specifier:
       char
       const
       double
       enum-specifier
       __far
       float
       int

```
        long
        __near
        short
        signed
        struct-or-union-specifier
        typedef-name
        unsigned
        void
        volatile

type-specifier-list:
        type-specifier
        type-specifier-list type-specifier
```

See Also

- "Pointer Declarators" in Section 3 for information on declaring variables of pointer types.
- Section 4, "Type Conversions," for information on pointer conversions.
- "Indirection Operator" in Section 5 for more information on the unary asterisk (*) operator.

Pointer Arithmetic

The constraints of the arithmetic operators are described in Section 5. Some of the operations are as follows:

Arithmetic Operations on Pointers	Description
Comparing two pointers	The following conditions can be tested: • less than (<) • less than or equal (<=) • greater than (>) • greater than or equal (>=) • equality (==) • not equal (!=)
Adding integers to pointers or subtracting integers from pointers	• The expression *p + n* points to the *n*th object beyond the object currently pointed to by *p*. • The expression *p-n* points to the *n*th object before the object currently pointed to by *p*.
Subtracting pointers	The expression *p-q* is the number of objects between *p* and *q*, where the object pointed to by *p* is included, but not the object pointed to by *q*.

See Also

"Addition Operator," "Assignment Operators," "Equality Operators," and "Relational Operators" in Section 5.

Structure Types

Structures are collections of objects, possibly of different types, that are grouped together for ease of manipulation. Structures correspond to records in other languages. Individual objects are distinguished from one another by unique member names. The direct member selection operator (.) or the indirect member selection operator (->) selects a specific member. Member names are local to structures.

Each structure type definition defines a structure type. If the structure type definition contains a tag, the tag is associated with the type. The tag can be used to reference that structure type. Tags must be defined before use. The scope of the structure tag extends to the end of the block in which the structure is defined.

Examples

Given the following structure type definition:

```
struct test { int exp; long real; } x, y;
```

The following structure type definition is the same as the previous structure type definition:

```
struct test { int exp; long real; };
struct test x, y;
```

The structure tag in the previous structure type definitions is `test.`

The size of an object of a structure type is the amount of storage needed to represent all members in the structure. This includes any unnamed holes between or after members within a structure.

Syntax

The following syntax describes the structure type specifier:

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

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

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

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

```
struct-declarator :
     declarator
     declarator_opt : constant-expression
```

See Also

- "Names Spaces" and "Scope" in Section 1.
- "Initializing Structures" in Section 3 for information on initializing a data object of type `struct`.
- "Structure to Structure" in Section 4 for information on structure conversions.
- "Direct Member Selection Operator," and "Indirect Member Selection Operator" in Section 5 for more information on structures.
- Appendix G, "Syntax Summary," for information on the syntax of *type-specifier-list* and *declarator*.

Structure Members

The member of a structure can be any type, except function and `void`. A structure cannot contain an instance of itself, but it can contain a pointer to an instance of itself.

Member names within a structure must be unique. However, a member name can be the same as a variable, function, type name, or member name in another structure.

The C compiler assigns members in increasing memory addresses in a strict left-to-right order. The first member starts at the beginning address of the structure.

Because members within a structure may have different types, they may have different storage allocation requirements. Holes can appear between two consecutive members of a structure to allow for the proper alignment of the members in storage.

See Also

"Name Spaces" and "Scope" in Section 1.

Bit Fields

The C compiler allows a variation of a structure member called bit fields. A bit field is a set of contiguous bits within a word, which is treated as an integer. With a bit field, a variable can be created that occupies one or more bits of storage. The purpose of bit fields is to allow members to be packed as tightly as possible in a structure.

A bit field object can be type `int` (`short`, `long`, `signed`, or `unsigned`) or type `char` (`signed` or `unsigned`). Plain integers and plain characters act as their `unsigned` types. ANSI C specifies that a bit field must have type `int` (`signed`, `unsigned`, or plain). If the compiler control option ANSI is set, an error is generated for a bit field that is declared with any other type.

By default, plain `int` bit fields are unsigned. You can use the compiler control option PORT(SIGNEDFIELD) to make plain `int` bit fields signed. Values are stored into signed bit fields in two's complement using as many bits as specified.

A bit field is specified by following the member declarator with a colon and a constant integer expression that indicates the width of the field in bits. For example, the following structure type definition defines a 4-bit field for x and a 6-bit field for y:

```
struct T { unsigned x:4; unsigned y:6;};
```

Bit fields are packed into adjacent bits within a machine word. A field that does not fit into the space remaining in a word is put into the next word. No field can be wider than the normal width of the declared type, or 39 bits, whichever is smaller.

A bit field with no declarator can be used as padding to conform to externally imposed layouts. As a special case of this, a bit field with a width of 0 (zero) causes any further bit fields to be packed in a different word than any previous bit field. For example, the following structure type definition causes bit field y to be put into a different word than x even though both x and y would normally fit into one word:

```
struct T {unsigned x:4; unsigned:0; unsigned y:12;};
```

Note: *Because bit fields are allocated differently on different machines, use of this feature limits the portability of a program.*

Union Types

Unions are similar to structures. Unions are defined to have a number of members that all have zero displacement from the beginning of the union. Therefore, the members of a union share storage. Unlike structures, a union can contain the value of only one of its members at a time. A union can be thought of as a structure whose members overlap and whose size is sufficient to contain the largest of its members.

Each union type definition defines a union type. If the union type definition contains a tag, the tag is associated with the type. The tag can be used to reference that union type.

The members of a union can be any type, except function and `void`. A union cannot contain an instance of itself, but it can contain a pointer to an instance of itself. Member names within a union must be unique. However, a member name can be the same as a variable, function, type name or a member name in another union. Member names are local to the union. Member names are selected by the direct member selection operator
(.) or the indirect member selection operator (->).

Example

The following example illustrates the union type specifier:

```
union arithval             /* this union      */
  {   int ival;            /* can hold an int */
      double dval;         /* or double       */
  };
```

Each member of the union is allocated storage starting at the beginning of the union. An object of a union type begins on a storage alignment boundary appropriate for all contained members.

The size of an object of a union type is the amount of storage needed to present the largest member of that type. This includes any padding or unnamed holes that may be needed after a member to raise the length to the appropriate alignment boundary.

Syntax

The following syntax describes the union type specifier:

```
union-specifier :
     union identifier_opt { struct-declaration-list }
     union identifier
```

See Also

- "Names Spaces" and "Scope" in Section 1.
- "Initializing Unions" and "Type Equivalence" in Section 3 for more information on union types.
- "Union to Union" in Section 4 for information on union conversions.
- "Direct Member Selection Operator," and "Indirect Member Selection Operator" in Section 5 for more information on unions.
- "Structure Types" in this section for information *struct-declaration-list* syntax.

Classification of C Data Types

C data types are related in the following ways:

- The character and integer types (of all sizes), both `signed` and `unsigned`, and enumerations are collectively called *integral* types.
- The single-precision and double-precision floating-point values are collectively called *floating-point* types.
- Integral and floating-point types are collectively called *arithmetic* types.
- Arithmetic types and pointers are collectively called *scalar* types. A single value or expression initializes a scalar data type.
- Arrays and structures are collectively called *aggregate* types. A list or series of values or initializes an aggregate data type.

Figure 2–1 illustrates this classification of C data types.

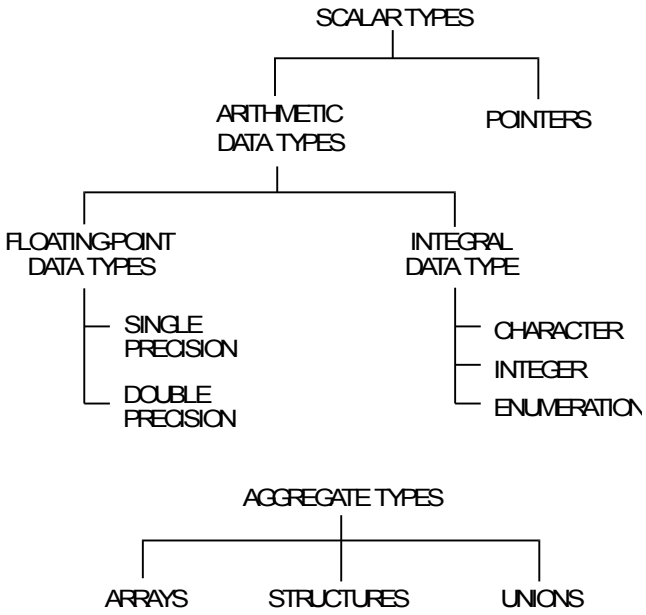


Figure 2–1. C Data Types

See Also

"Declarators" in Section 3 for more information on declaring data. Data types and data declarations are closely related topics.

Function Types

A function type specifies the type of the result and optionally the types of the arguments of a function. The result type can be `void` or any data type, except "array of type." In other words, a function cannot return an array or another function. A function can, however, return a pointer to an array or a function.

The following two methods are used to introduce function types:

- Function definition
- Function declaration

Function Definition

A function definition does the following:

- Creates a function name
- Defines its arguments

- Defines its return type
- Supplies the body of the function

For example, the following is a function definition:

```
int max(int a, int b)
{
  if (a > b)
     return a;
  else
     return b;
}
```

See Also

"Defining Functions" in Section 7 for a description of function definitions.

Function Declaration

A function declaration introduces an external reference to a function defined some place else. The following syntax is an example of a function declaration:

```
int max(int a, int b);
```

Naming a Function

The following operations are typically applied to the name of a function in an expression:

Operations Applied to Name of a Function	Description
Call the function.	Add to the name one of the following: • A list of arguments in parentheses and separated by commas. • Only the parentheses if the function was defined or declared as having `void` or no parameters.
Use the function name as an actual argument to another function.	In this case, a pointer to the function is passed.
Assign the function to a variable of the appropriate pointer to function type.	In this case, a pointer to the function is assigned.

See Also

- "Function Declarators" in Section 3 for information on declaring objects of function types.
- "Type Conversions" in Section 4 for information on function conversions.
- "Functions" in Section 7 for more information on functions.

Void Type

The `void` type has no representation. An object cannot have type `void`. The `void` type can be used between the parentheses in a function declaration or definition to represent the fact that the function has no parameters. The `void` type is used as the return type for a function that does not return a value. Another valid type based on `void` is a pointer to `void`. The `void` type can also be used in a cast expression when explicitly discarding a value.

See Also

"Discarded Values" in Section 5.

Section 3
Declarations

A declaration specifies the visibility, lifetime, and type of identifiers for a program. The identifiers that can be declared in C are

- Variables
- Functions
- Types (`typedef` names)
- Structure, union, and enumeration tags
- Structure and union members
- Enumeration constants
- Statement labels
- Preprocessor macros

This section describes the storage class specifiers, type specifiers, declarators, initializers, and other aspects of writing declarations.

Declaration Syntax

The syntax for a declaration is

```
declaration :
    declaration-specifiers init-declarator-list(opt) ;

declaration-specifiers:
    storage-class-specifiers declaration-specifiers(opt)
    type-specifier declaration-specifiers(opt)

    fct-specifier declaration-specifiers(opt)

init-declarator-list :
    init-declarator
    init-declarator-list , init-declarator
```

```
init-declarator :
    declarator
    declarator = initializer
```

Except for statement labels and macro definitions, all declarations use this syntax. The declarators in the *init-declarator-list* contain the identifier names being declared.

See Also

- Refer to "Declarators" in this section for more information.
- Refer to "Labeled Statements" in Section 6 for information about statement labels.
- Refer to "Macro Definition and Replacement Examples" in Section 8 for information about macros.
- Refer to Appendix G, "Syntax Summary," for information on the syntax of *storage-class-specifiers*, *type-specifier*, *fct-specifier*, *declarator*, and *initializer*.

Storage Class Specifiers

The term *storage class* refers to the way the compiler allocates storage to variables. The storage class specifier determines the lifetime of an identifier. The following are valid storage class specifiers in the C language:

Storage Class Specifiers	Description
`asm`	The `asm` storage class specifier indicates that a function is to be exported from a library. It is available only if the ANSI compiler control option is not set. Refer to Appendix A, "Interface to the Library Facility," for more information. ***Note:*** *The `asm` storage class specifier is an A Series extension and is not portable.*
`auto`	`The auto` storage class specifier is short for automatic. `The auto` storage class specifier indicates that a variable is local. Local variables are allocated storage at the beginning of a block. Storage is released at the end of the block. The `auto` storage class specifier can only be used with a declaration inside a block.
`extern`	A declaration with the `extern` storage class specifier indicates that somewhere in the set of source files that constitute the entire program, there exists an external definition for the given identifier.

Storage Class Specifiers	Description
`inline`	The `inline` specifier requests that the function body should be substituted for the function call. It is only a hint to the compiler; it may be ignored and it does not affect the meaning of the program. It may appear with any other storage class specifier. Typically, the `inline` specifier is used for very small functions that are frequently executed. The substituted code is usually as efficient as macro expansion; sometimes extra variables are added to preserve the meaning of the program. ***Note:*** *The `inline` storage specifier can be used only when a function is being defined (a function body is supplied).*
`register`	A declaration with the `register` storage class specifier has the same lifetime as an `auto` declaration. The address of the variable is never taken. The `register` storage class specifier suggests that the variable will be heavily used and should be in a storage location where the instruction processor can access it quickly. For example, it could be stored in the hardware stack.
`static`	The `static` storage class specifier causes the object to exist for the life of the program. The `static` storage class specifier can appear on declarations of functions, local variables, and variables outside of functions. On functions, the `static` storage class specifier indicates that the function name is not to be exported. On data variables outside of functions, the `static` storage class specifier indicates a defining declaration that is not exported. On local variables, the `static` storage class specifier indicates the value is preserved across different calls on the function.
`typedef`	The `typedef` storage class specifier is the mechanism that the C language provides for creating data type names. If the storage class is `typedef`, the declaration is a type definition. The identifier becomes a synonym for that type. Refer to "Typedef Names" in this section for a description of the `typedef` mechanism. ***Note:*** *The `typedef` specifier is listed under storage class specifiers for syntactic convenience only.*

Table 3–1 summarizes the storage class specifier rules for variables and functions.

Table 3–1. Summary of Storage Class Specifier Rules

Storage Class Specifier	If the declaration is:	Then the lifetime of the storage class specifier is:
`auto`	Inside a block or function	The lifetime of the block or function execution
`extern`	Outside a function	The lifetime of the program

Table 3–1. Summary of Storage Class Specifier Rules

Storage Class Specifier	If the declaration is:	Then the lifetime of the storage class specifier is:
	Inside a block or function	
`register`	Inside a block or function	The lifetime of the block or the function execution
`static`	Outside a function	The lifetime of the program
	Inside a block or function	
`none`	Outside a function	The lifetime of program
	Inside a block or function	The lifetime of the block or function execution

Storage Class Specifier Defaults

A declaration can contain only one storage class specifier. If a storage class specifier is not supplied on a declaration, the compiler assigns a specifier based on the context of the declaration.

- If the declaration occurs outside of a function or is a function definition, the compiler assumes the declaration is externalized and has allocated storage for that object.
- An argument declaration can take only the `register` storage class specifier. If the storage class specifier is omitted, the compiler assumes no `register` is wanted.
- For declarations at the beginning of a block, the compiler assumes `extern` for functions and `auto` for everything else.

C language programmers usually omit the `auto` storage class specifier, which is acceptable.

Example

The following program segment indicates the use of storage class specifiers on declarations:

```
extern int a;              /* externally defined   */
int b;                     /* externally known     */
static int function ()     /* not externally known */
{     auto int c;          /* automatic */
      int d;               /* automatic */
      static int e;        /* static    */
}
```

Type Specifiers and Qualifiers

A type specifier provides some additional information about the data type of the identifier being declared. The data type specified in a declaration or a definition can be one of the following type specifiers:

```
char            long              union tag
double          short             unsigned
enum tag        signed            void
float           struct tag
int             typedef-name
```

Note: If a declaration does not contain a type specifier, the type is assumed to be *int*.

A type specifier can be qualified to indicate special properties of the objects being declared. These type qualifiers are the following:

```
const
__far
__near
volatile
```

See Also

Refer to Section 2, ''Types,'' for more specific information on data types and their corresponding type specifiers.

const Type Qualifier

The properties associated with the `const` type qualifier are meaningful only for expressions that are lvalues. Refer to "Lvalue," "Rvalue," and "Function Locator" in Section 5 for more information.

An expression that designates an object that has a type declared with the `const` type qualifier cannot be a modifiable lvalue. Use the `const` type to declare data objects whose values are not intended to be altered during the execution of the program. If an attempt is made to alter such an object by means of an lvalue that has a type not declared with the `const` type qualifier, the behavior is undefined.

The address of a non-`const` object can be assigned to a pointer to a type with the `const` attribute. The value, however, cannot be modified by means of the pointer to `const`.

Except by an explicit cast, the address of a `const` object cannot be assigned to a pointer to a type without the `const` attribute. If an attempt is made to change the value by means of the pointer to a non-`const` attribute, the behavior is undefined.

Examples

The following pointer declarations demonstrate the difference between a modifiable pointer to a constant value and a constant pointer to a modifiable value:

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

The contents of the `int` pointed to by `ptr_to_constant` must not be modified, but `ptr_to_constant` itself can be changed to point to another `const int`. Likewise, the contents of the `int` pointed to by `constant_ptr` can be modified, but `constant_ptr` itself must always point to the same location.

The declaration of the constant pointer `constant_ptr` can be clarified by using a `typedef` for the type pointer to `int`.

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

If an aggregate object type is declared with the `const` type qualifier, the type of each member of the aggregate is implicitly declared with that specifier.

See Also

- Refer to "Reading and Writing Complex Declarators" in this section for more information on how to read declarators.
- Refer to "Lvalue," "Cast Operator," and "Constant Expressions" in Section 5 for more information.

__far and __near Type Qualifiers

The `__far` and `__near` type qualifiers are used for data and data pointer declarations to do the following:

- Increase the amount of memory that can be addressed
- Declare data in a non-default memory segment
- Address memory in non-C code
- Address memory in another C program or library

The FARHEAP compiler control option must be set to use the `__far` and `__near` type qualifiers.

See Also

Refer to Section 10, "Compiler Control Options," for more information on the FARHEAP compiler control option.

__far and __near Data

Data that is allocated in the first memory segment is called near data, and data that is allocated in any other memory segment is called far data.

In the TINY and SMALL memory models, data is allocated near, by default. In the LARGE and HUGE memory models, data is allocated far, by default. If the FARHEAP

compiler control option is set, near and far data can be explicitly defined in any memory model.

Far and near data can be defined by using the __far and __near type qualifiers in data declarations. If the standard library header <alloc.h> is included, the macros _far and _near (using a single leading underscore character) expand to the type qualifiers __far and __near (using double leading underscores). The rules for type declarators using the __far and __near type qualifiers are the same as those for the const and volatile type qualifiers.

See Also

- Refer to Section 10, "Compiler Control Options," for more information on the FARHEAP option and the MEMORY_MODEL option.
- Refer to Volume 2, "Headers and Functions," for information on the header `<alloc.h>`.

Examples

The following examples illustrate the use of the `__far` and `__near` data type qualifiers:

Far and Near Data Types	Description
`__near char nc;`	This defines data in near memory as type char.
`char __far fc;`	This defines data in far memory as type char.
`int __near na[100];`	This defines an array of data type int in near memory.
`__far int fa[100];`	This defines an array of data type int in far memory.

__far and __near Pointers

Pointers that contain only an index into the first memory segment are called near pointers. Pointers that contain a segment number and an index into the memory segment are called far pointers.

In the TINY and SMALL memory models, pointers are near pointers by default. In the LARGE and HUGE memory models, pointers are far pointers by default. If the FARHEAP compiler control option is set, far and near pointers can be explicitly defined in any memory model. Only the first memory segment can be accessed from the TINY or SMALL memory models unless far pointers are used. Near pointers provide fast addressing for the first memory segment from the LARGE and HUGE models.

Far and near pointers can be defined by using the __far and __near type qualifiers in pointer declarations. If the standard library header <alloc.h> is included, the macros _far and _near (using a single leading underscore character) expand to the type qualifiers __far and __near (using double leading underscores). The rules for type declarators using the __far and __near type qualifiers are the same as those used with the const and volatile type qualifiers.

The compatibility rules for assignment, parameter passing, and so on for pointer declarations using the __far or __near type qualifiers are the same as for pointers

without these type qualifiers with one exception: Pointers to near data can be assigned or passed to pointers to far data so long as the same number of levels of pointers occur in each declaration and the data types are assignment compatible. This rule is enforced across any number of levels of indirection.

See Also

- Refer to Section 10, "Compiler Control Options," for more information on the FARHEAP option and the MEMORY_MODEL option.
- Refer to Volume 2, "Headers and Functions" for information on the header `<alloc.h>`.

Examples

The following examples illustrate the use of the `__far` and `__near` pointer type qualifiers:

Far and Near Pointers	Description
`__near char *np;`	This defines a pointer to a type `char` in near memory.
`char __far *fp;`	This defines a pointer to a type `char` in far memory.
`__near char *np[100];`	This defines an array of pointers to type `char` in near memory.
`char __far *fp[100];`	This defines an array of pointers to type `char` in far memory.
`char *__far afp[100]`	This defines an array of pointers in far memory to type `char`.
`__far int *__near fnp;`	This defines a pointer in near memory to type `int` in far memory.
`int __near *__far ffp;`	This defines a pointer in far memory to type `int` in near memory.

volatile Type Qualifier

Use the `volatile` type qualifier to declare data objects whose stored value may change in ways the compiler cannot anticipate. Most applications do not need `volatile` types. Programmers use them to declare data objects in storage that are shared among multiple processes. A `volatile` declaration can be used to describe an object accessed by an asynchronously interrupting function. Actions on objects declared as `volatile` are not optimized or reordered, except as permitted by the rules for evaluating expressions.

The address of a non-`volatile` object can be assigned to a pointer to a type with the `volatile` attribute. Then the rules for `volatile` objects must be obeyed when the object is referred to by means of a pointer to `volatile`.

Except by an explicit cast, the address of a `volatile` object cannot be assigned to a pointer to type without the `volatile` attribute. If the value is referred to by means of the pointer to non-`volatile`, the behavior is undefined.

If an aggregate object type is declared with the `volatile` type qualifier, the type of each member of the aggregate is implicitly declared with that qualifier.

Example

The following declared object can be modified by an external process; but it cannot be assigned to, incremented, or decremented explicitly by the program containing this declaration:

```
extern volatile const real_time_clock;
```

See Also

Refer to "Cast Operator" and "Side Effects" in Section 5 for more information.

Declarators

Declarators introduce the name being declared and provide additional type information. The types of declarators are listed as follows and are explained in this section:

- Simple
- Pointer
- Array
- Function

Simple Declarators

The simple declarators define variables with a data type of arithmetic, structure, union, and enumeration. A simple declarator is found in a declaration where the type qualifier provides all the type information. The simplest declarator that can be written is a type specifier such as `int` and a single identifier such as `count`.

In the following declaration, *`type`* is the type specifier such as `int` and *`identifier`* is the declarator such as `count`. This declaration specifies that the variable named by *`identifier`* has type indicated by *`type`*:

```
type identifier;
```

Examples

The following examples illustrate simple declarators:

Declaration	Description
`int i,j;`	The `i` and `j` variables are integer variables.
`float value1;`	The `value1` variable is a floating-point variable.

`struct date`
`{`
`int month;`
`int day;`
`int year;`
`} todays_date;`
	The `todays_date` variable is a structure of three members.

See Also

Refer to Section 2, "Types," for more information on data types and data specifiers.

Pointer Declarators

Pointer declarators declare variables of pointer types. Declare a pointer by writing any type qualifier, followed by an asterisk (*), followed by an identifier.

In the following declaration, the variable named by *identifier* is of type "pointer to *type*":

```
type *identifier;
```

A pointer can be declared to point to a structure when the structure is not declared in the same compilation unit. However, retrieving a value through such a pointer is illegal.

Examples

The following examples illustrate the use of pointer declarators:

Declaration	Description
`int *int_ptr;`	The `int_ptr` variable is a pointer to an integer.
`int *array_ptr[ ];`	The `array_ptr` variable is an array of pointers to integers.
`int *function_ptr();`	`function_ptr` is a function returning a pointer to an integer.

See Also

- Refer to "Reading and Writing Complex Declarators" in this section for information on reading declarators.
- Refer to "Pointer Types" in Section 2 for more information on pointers.

Array Declarators

Array declarators declare objects of array types. Write an array declarator as a type and identifier followed by an array size in brackets ([ ]). The constant expression that specifies the size of an array must have an integral type and a value greater than zero. The constant expression must be present, except that the first size may be omitted when

- An array is being declared as a formal argument of a function
- The array declaration has storage-class specifier `extern` and the definition that actually allocates storage is given elsewhere
- The declarator is followed by initialization; the number of initializers supplied determines the size

In the following declaration, the variable named by *identifier* is of type "array of *type*":

```
type identifier[constant-expression];
```

In the C language, arrays are zero-origin, which means the numbering of the elements in the array starts at zero. For example, a program contains the following declaration:

```
int number [3];
```

The three integer elements in the array are `number [0]`, `number [1]`, and `number [2]`.

When several bracketed *expressions* are adjacent, a multidimensional array (also called "arrays of arrays") is declared. For example, the following declarations declare "arrays of arrays":

```
int number [10][10];
       /*  This declaration is the same as int (number [10])[10];  */
int matrix [3][7];
```

Examples

The following examples illustrate the use of arrays:

Declaration	Description
`int array [5];`	The `array` declarator is an array of integers.
`int *array-ptr [19];`	The `array-ptr` declarator is an array of pointers to an integer.
`int (*ptr_array)[19];`	The `ptr_array` declarator is a pointer to an array of integers.
`int array2d[7][5];`	The `array2d` declarator is an array of arrays of integers.

See Also

- Refer to "Constant Expressions" in Section 5 for more information.
- Refer to "Defining Functions" in Section 7 for more information.
- Refer to Appendix B, "Internal Compiler Limits," for information on the maximum size of arrays.

Function Declarators

Function declarators declare function types. Write the function declarator as an identifier followed by an argument list enclosed in parentheses.

In the following declaration, the function named by *`identifier`* is of type "the function returning *`type`*":

```
type identifier ();
```

All declarations of a particular function must agree in the type returned.

There can be a declaration for a function that returns a pointer to a structure when the structure is not defined in the same compilation unit. However, retrieving a value through such a pointer is illegal.

Two kinds of argument lists can be written, as follows:

Argument List	Description
Old style	This method has been the method traditionally used in C. Using this method, the default promotions of arguments are performed. The number of arguments must match the number declared in the function definition.
Function prototype	The function prototype method causes the compiler to check the types and number of arguments on subsequent function calls. Actual arguments are promoted according to what is specified in the function prototype. Calling a function declared with a prototype is faster than calling an old style function.

See Also

Refer to "Old Style Format" and "Function Prototype Format" in Section 7 for more information on defining functions.

Old Style

In an old style definition, an empty argument list means that the function has no arguments. Otherwise, write the argument identifiers alone within the argument list. Declare any arguments with argument level declarations immediately following the declarator. The function body, enclosed in braces, terminates the argument level declarations. Each argument can be declared at most once. Arguments can be declared only at the argument level. If an argument is not declared, the compiler implicitly declares it to have type `int`.

See Also

Refer to "Old Style Format" and "Function Prototype Format" in Section 7 for more information.

Function Prototype

For a function with a fixed number of arguments, write a list of declarations, one for each argument within the argument list. The types of the function arguments must be provided in the argument list. The argument identifiers of the arguments may be provided. Separate the declarations with commas. Each declaration can have only one declarator. The compiler checks the types and number of arguments on a function call in scope of a prototype. If the argument identifiers are supplied, their scope ends with the end of the argument list or at the end of the body of the function if it is being defined. They need not match identifiers used in redeclarations of the function.

Write a function prototype definition for a function `f` with no arguments as `f(void)`.

To call a function with a variable length argument list, write a function prototype whose argument list ends with a comma followed by an ellipsis. For example:

```
int printf(char *fmt, ...);
```

At least one argument must be specified in the function prototype. On a function call, the compiler checks the types and number of all the arguments that were specified in the function prototype. A function with more arguments than specified in the function

prototype may be called, but not a function with fewer arguments than specified. All function calls must be in the scope of such a function prototype.

If a function is declared again, one of the declarators may be a function prototype and the other an old style declaration. If both provide a function prototype, the number of arguments and their types must be the same.

The definition and every prototype of a function must agree in the number and type of the arguments and in the use of the ellipsis terminator. The argument type checking includes agreement on the number of dimensions for arrays and on the bounds for each dimension, including the first if it is specified.

See Also

- Refer to "Scope" in Section 1 for more information.
- Refer to Section 7, "Functions," for more information on defining functions.

Mixing Formats

Mixing old style declarations and prototype declarations for the same function is not recommended. If you must mix formats, the following restrictions apply when mixing old style and prototype declarations for the same function:

Restrictions for Declaring a Function When Mixing Formats

If you declare . . .	And you define . . .	Then you must . . .
A function using the old style format	A function using the function prototype format	Ensure that the types of the actual arguments after the default argument promotions agree with the types of the formal arguments in the function definition.
A function using the function prototype format	A function using the old style format	Ensure that the function prototype declares the formal arguments to be the same types as used by the default argument promotions, after the promotions occur.

Except in the cases described in the preceding table, a function prototype must be in the scope of the function call if the function is defined using the function prototype format. Otherwise, the behavior is undefined.

All declarations of a particular function must declare the identical return type. Each argument type list must agree in the number and types of the arguments and in the use of the ellipsis terminator. The argument type checking includes agreement on the number of dimensions of arrays and on the bounds for each dimension except the first.

A function definition can serve as a declaration if, and only if, the definition appears in the source file before any calls to that function.

See Also

- Refer to "Scope" and "Linkages" in Section 1 for more information.
- Refer to "Function Types" in Section 2 for more information on defining functions.
- Refer to "Defining Functions," and "Return Values" in Section 7 for more information on defining functions.

Examples

The following examples illustrate old style and prototype declarations and definitions of functions:

Declarations	Description
`int fri();`	This is the old style method with no definition.
`int fri(int, double);`	This is the function prototype method with no definition.
`int fri(int a, double d);`	This is the function prototype method with no definition.

`int fri(x, y)`
`  int x;`
`  double y;`
`  {`
`  .`
`  .`
`  .`
`  }`
	This is the old style method with a definition.

`long frlo(long lo)`
`  {`
`  .`
`  .`
`  .`
`  }`
	This is the function prototype method with a definition

`static int f(void), *fip(),` `  (*pfi)();`	• A function `f` with no arguments returning an `int`; this is a prototype declaration. • A function `fip` (with no argument type information) returns a pointer to an `int`; this is an old style declaration. • A pointer `pfi` to a function (with no argument type information) returning an `int`; this is an old style declaration. The identifiers of the functions `f` and `fip` are declared as having file scope (not program scope) and internal linkage. The pointer `pfi` is declared as having file scope and internal linkage if the declaration is ioutside any function; and block scope and no linkage if the declaration is inside a function

Declarations	Description
`extern int (*apfi[ ])(int *x, int *y);`	This declares an array `apfi` of 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`.
`int (*fpfi(int (*)(long), int))(int, ...);`	This declares a function `fpfi` that returns a pointer to a function returning an `int`. The function `fpfi` has two arguments: • A pointer to a function returning an `int` and one argument of type `long` • An `int` The pointer returned by `fpfi` points to a function that has at least one argument, which has type `int`.

See Also

- Refer to "Reading and Writing Complex Declarators" in this section for more information on reading declarators.
- Refer to "Scope" and "Linkages" in Section 1 for more information.
- Refer to Section 7, "Functions," for information on defining functions.

Reading and Writing Complex Declarators

The only restrictions the C language places on declarators is that the resulting type must be a legal type in C. The only illegal types are

- Any type involving `void`, except "function returning `void`ö and "pointer to `void`."

  Refer to "Void Type" in Section 2 for more information.

- Arrays that contain functions.

  Arrays can contain pointers to functions, but not functions themselves.

- Functions returning arrays.

  Functions can return pointers to arrays, but not arrays themselves.

- Functions returning functions.

  Functions can return pointers to other functions, but not the functions themselves.

Rules for Reading and Writing Valid Declarators

The following are some simple rules to help read and write declarators correctly:

Rule	Example	Description
Read declarators from right to left.	`type*funcptr()`	This is of type "function returning pointer to *type*"
	`type**aryptrptr[10]`	This is of type "array of pointer to pointer to *type*."
Parentheses specify a different order. Read the contents of the innermost parentheses first.	`type(*ptrfunc)()`	This is of type "pointer to function returning *type*."

The maximum number of declarators that can modify a basic type depends on the size of the compiler's internal data structures, typically over 100. For example, the following declarator indicates that `p` is a pointer to a pointer to a pointer to a pointer that points to a `char`:

```
char ****p
```

Any combination of pointer, array, and function declarators can be nested up to the given limit.

Examples

The following are examples of complex declarations.

Declaration	Description
`int (*alert(int sig, int func))( );`	This is a function with two arguments returning a pointer to a function that returns an integer.
`int *(*ptraryptr)[10];`	This is a pointer to an array of pointers to int.
`int *(*(*arr[3][4])[5])();`	The `arr` variable is a two-dimensional array of pointers to an array of pointers to functions returning pointers to integers.

Declarator Syntax

The syntax for declarators is as follows:

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-specifier-list opt
     *type-specifier-list opt pointer

type-specifier-list :
       type-specifier
       type-specifier-list type-specifier

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

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

parameter-declaration :
       declaration-specifiers declarator
       type-name

identifier-list :
       identifier
       identifier-list , identifier
```

Typedef Names

The `typedef` keyword defines a name as a synonym for a data type. Define the new name of a type by starting the definition with the keyword `typedef` and following it with a declaration. For example:

```
typedef int integer;
```

The identifier `integer` is defined as a synonym for `int` and can be used wherever a type specifier is permitted.

The keyword `typedef` does not define a new type, but it does define a new name for an existing basic or derived data type. The compiler treats variables declared with the `typedef` name exactly as if they were declared to be of the type associated with the `typedef` name.

A `typedef` name shares the same name space as other identifiers declared in ordinary declarators. It can be declared again in an inner block, but the type specifiers cannot be omitted in the inner declaration.

Examples

The following are examples of using the keyword `typedef` to define a name as a synonym for a data type:

Typedef	Description

Given the following typedef declarations:
`typedef int MILES, KLICKSP();`
`typedef struct { double re, im; } complex;`
The following are all valid declarations:
`MILES distance;`
`extern KLICKSP *metricp;`
`complex z, *zp;`
	The type of `distance` is `int`. The type of `metricp` is "pointer to function with no argument-type information returning `int`." The type of `z` is the specified structure; `zp` is a pointer to such a structure. The object `distance` is considered to have exactly the same type as any other `int` object.

`typedef struct s1 { int x; } t1, *tp1;` `typedef struct s2 { int x; } t2, *tp2;`	The type `t1` and the type pointed to by `tp1` are equivalent to each other and to the type `struct s1`, but different from the types `struct s2` and `t2` and the type pointed to by `tp2` and from type `int`.

`typedef float length;`
`{`
`  static int length;`
`}`
	The name `length` is declared first as a type and then as an `int` variable. If the `int` type specifier were omitted from the second declaration, it would be taken to be a declaration with type `length` and no declarators.

`typedef int intarray [10];` `intarray ia;`	The type `intarray` is an integer array with 10 elements. The variable ia has type `intarray`.

A `typedef` cannot be used to define a function as in the following example:

```
typedef int funct(int a);
extern funct extft;              /* not a definition - allowed */

funct  intfn
            { . . . }            /* error - not allowed */

int intfn(int x)
            { . . . }            /* allowed */
```

Type Equivalence

The statement "two objects must have the same type" refers to two types that are the same if they have the same ordered set of type specifiers and abstract declarators, either directly or indirectly by a `typedef`.

- Pointers

Two pointers, "pointer to *type1*" and "pointer to *type2*," are the same if *type1* and *type 2* are the same.

- Functions

Two functions, "function returning *type1*" and "function returning *type2*," are the same if *type1* and *type2* are the same and their argument information is the same.

- Arrays

Two arrays, "*x*-element array of *type1*" and "*y*-element array of *type2*," are the same if *type1* and *type2* are the same and both arrays have the same number of elements (*x* = *y*).

  – In those situations where the size of the array can be omitted, the size does not participate in determining whether the two arrays are the same. If an array is multidimensional and if the size of the first dimension can be omitted, then only the first dimension does not participate. All other dimensions must match. The following two external definitions declare the variable array `alpha` and `alphaII`. They are considered to be the same type.

```
extern int alphaII[7]
extern int alpha[];
```

  – In situations where the value of type "*x*-element array of *type1*" is converted to a value of type "pointer to *type1*," the rules for pointer equivalence apply (the size of the array does not matter).

- Enumerations, structures, and unions

Two enumerations, structures, or unions are different if they have no names or different names (taking tags and `typedef` synonyms into account), even if their members are identical.

- Typedef names

Typedef names, by definition, are synonyms for types, not new types. Therefore, in the following example, `func` is the same as the type `int *()`:

```
typedef int *func();
```

See Also

Refer to "Array Types," "Enumeration Types," "Function Types," "Pointer Types," "Structure Types," and "Union Types" in Section 2 for more information.

Type Names

Write a type name in contexts where a type is specified without declaring an object of that type. Such contexts occur for the following tasks:

- Specifying a type conversion explicitly by means of a cast
- Declaring a type as the argument of `sizeof`
- Declaring the type of a formal argument in a function declaration

A type name is syntactically a declaration for an object of that type that omits the identifier of the object.

An abstract declarator resembles a regular declarator, except the identifier is omitted.

Note: *Empty parentheses in a type name are interpreted as a function with no argument type information rather than an redundant parentheses around the omitted identifier. For example,* `char*()` *is interpreted as a function returning a pointer to* `char` *rather than as a* `char` *pointer.*

Examples

The following constructions name the corresponding types:

Construction	Type
`int`	This is type `int`.
`int *`	This is a pointer to `int`.
`int *[3]`	This is an array of three pointers to `int`.
`int (*)[3]`	This is a pointer to an array of three `int`s.
`int *()`	This is a function with no argument-type information returning a pointer to `int`.
`int (*)(void)`	This is a pointer to a function that has no arguments and returns an `int`.
`int (*const[ ])(unsigned int, ...)`	This is an array of an unspecified number of constant pointers to functions, each with one argument that has type `unsigned int` and an unspecified number of other arguments, returning an `int`.

Syntax

The following syntax describes a type name:

type-name:
type-specifier-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*$_{opt}$)

See Also

- Refer to “Cast Operator” in Section 5 for more information.
- Refer to “Defining Functions” in Section 7 for more information.

Initializers

A declaration can specify an initial value for the identifier being declared. This initial value is called the initializer. The following is the basic format of an initializer:

```
declarator = value
```

or

```
declarator = { list-of-values }
```

The values permitted on an initializer in a particular declaration depend on

- The type and storage class of the variable
- Where the declaration appears, such as outside of a function or at the head of a block

In general, if the variable is declared `static`, the initializer must be a constant expression. Since a `static` variable is allocated storage before the beginning of program execution, the initialization also occurs before the C program executes.

The initializer for an automatic variable need not be a constant expression. Automatic variables are always declared at the head of a block. When entering the block, the compiler produces code that evaluates the expression and assigns the result to the variable.

If the storage class for the declarator is . . .	**Then . . .**
`static` (or there is no storage class specifier)	Any constant expression can initialize the variable.
`auto` or `register`	Any expression (not necessarily a constant expression) can initialize the variable.

If a declaration does not contain an initializer, static and external variables are initialized to zero by default. Automatic variables are not initialized by default. If automatic or register variables are not initialized, the contents are undefined.

The declaration of a function's formal arguments cannot have initializers.

Since some data types have special requirements, the following sections explain the initializers for each type of data. The C language does not allow initialization of function types and `void` type.

Syntax

The syntax of the initializers is

```
initializer :
      assignment-expression
```

```
        { initializer-list }
        { initializer-list , }

initializer-list :
        initializer
        initializer-list ,  initializer

init-declarator:
        declarator
        declarator = initializer
```

See Also

- Refer to "Constant Expressions" in Section 5 for more information.
- Refer to "Defining Functions" in Section 7 for more information.
- Refer to Appendix G, "Syntax Summary," for information on the syntax of *assignment-expression*.

Initializing Arithmetic Data

The initial value of the declarator is the value of the expression. The same conversions as done for assignment are performed.

The following is the basic format of an initializer for an arithmetic variable:

```
declaration-specifiers declarator = expression;
```

or

```
declaration-specifiers declarator = {expression}
```

Examples

The following examples illustrate the initialization of arithmetic variables:

```
static int number = 100;

#define MAX 50
int table = MAX+1;

float value = 7.5*3.6;
```

See Also

- Refer to "Integer Types" in Section 2 for more information.
- Refer to "Constant Expressions" in Section 5 for more information.

Initializing Pointers

The following elements form constant expressions used as initializers of type "pointer to type." The initializer for a pointer type must evaluate to an address, an address plus (or minus) an integer constant, or the value zero (0).

Initializing a Pointer to NULL

The pointer constant zero (0) is the null value for any pointer type.

```
#define NULL  0
int *value_ptr = NULL;
```

The macro definition for NULL can be found in `<stddef.h>`.

Initializing a Pointer to a Constant

Specifying a constant initializes the pointer to the value of the constant.

```
long *p = (long*)385;
```

Initializing a Pointer to an Address

The & (address of) operator initializes the pointer to the address of the subsequent object.

```
static int digit;
int *ptr = &digit;
```

Initializing a Pointer to the Address of a Function

Specifying the name of a static or external function initializes the pointer to the address of the function.

```
extern long function();
long (*func_ptr)() = function;
```

Initializing a Pointer to an Array

Specifying the name of an array initializes the pointer to the beginning of the array.

```
static int array[10];
static int *constant_ptr = array;
```

Initializing a Pointer *into* an Array

Specifying the name of an array plus an offset *n* initializes the pointer to the *n*th+1 element of the array.

```
static int array[10]
int *iptr = array + 4;
```

In this example, the pointer `iptr` points to the fifth element of the array (`array[4]`).

Initializing a Pointer to a Character String

Specifying a character string initializes the pointer to the beginning of the character string.

```
char *error_msg = "Error message #1";
```

Pointer *into* a Character String

Specifying a character string plus an offset *n* initializes the pointer to the *n*th+1 element of the array.

```
char *cptr = "STRING" + 4;
```

In this example, the pointer `cptr` points to the character "`N`" in the string "`STRING`."

See Also

- Refer to "String Constants" in Section 1 for more information strings.
- Refer to "Pointer Types" in Section 2 for more information pointers.
- Refer to "Address Operator," "Constant Expressions," and "Cast Operator" in Section 5 for more information.
- Refer to "Defining Functions" in Section 7 for more information.

Initializing Arrays

The following is the basic format for the initialization of an array:

declaration-specifiers declarator[n] = { I_0, . . . I_{n-3}, I_{n-2}, I_{n-1} }

The I_j variables (for j = 0, 1, . . . n-1) are each initializers. The initializer I_j initializes the element j in the array. All initializers of the array must have a type that is assignment compatible with the array element type of the array. For example, the following array is a single dimension array with five elements. The zero element of this array is initialized to the integer 12:

```
int dates[5] = { 12, 14, 16, 18, 20 };
```

Initializing One-dimensional Arrays

The number of initializers must be less than or equal to the number of array elements. If the array has more elements than initializers, the remaining elements are initialized to zero.

The following initialization:

```
int count[7]     = { 0, 1, 2, 3, 4 };
```

is the same as:

```
int count[7]     = { 0, 1, 2, 3, 4, 0, 0 };
```

The following initialization is invalid because it has too many initializers:

```
int count[5] = { 0, 1, 2, 3, 4, 5 }
```

Initializing Two-dimensional Arrays

The C language is row-major, so the last subscript (column) varies the most.

As non-specified elements are initialized to zero, the following initialization:

```
int matrix[3][4] = { { 1, 2, 3 },
                     { 1, 2 },
                     { 1 }          };
```

is the same as:

```
int matrix[3][4] = { { 1, 2, 3, 0},
                     { 1, 2, 0, 0},
                     { 1, 0, 0, 0}  };
```

Initializing Multidimensional Arrays

Multidimensional arrays are also initialized by row, the last subscript varying the most:

```
int matrix[3][3][3] =
    { { { 1, 1, 1}, { 1, 1, 2}, { 1, 1, 3} },
      { { 1, 2, 1}, { 1, 2, 2}, { 1, 2, 3} },
      { { 1, 3, 1}, { 1, 3, 2}, { 1, 3, 3} },

      { { 2, 1, 1}, { 2, 1, 2}, { 2, 1, 3} },
      { { 2, 2, 1}, { 2, 2, 2}, { 2, 2, 3} },
      { { 2, 3, 1}, { 2, 3, 2}, { 2, 3, 3} },

      { { 3, 1, 1}, { 3, 1, 2}, { 3, 1, 3} },
      { { 3, 2, 1}, { 3, 2, 2}, { 3, 2, 3} },
      { { 3, 3, 1}, { 3, 3, 2}, { 3, 3, 3} } };
```

Declaring Array Dimensions Using Initializers

If the number of elements in the array are not specified, the compiler determines the size of the array from the number of initializers.

The following declaration:

```
int digits[] = {0, 1};
```

is the same as this declaration:

```
int digits[2] = {0, 1};
```

Initializing Character Strings

Use string literals to initialize variables of type "array of `char`." Each character of the literal corresponds to one element of the array. The first character of the string is the first element of the array, the second character, the second element, and so forth. Remember that a string literal has at the end an implicit null character, which is stored if there is room for it.

Initializing an array with an unspecified dimension causes the dimension to be implied by the number of elements in the initializer. If the initializer is a string literal, the implicit terminating null character is included in the number of elements.

The following three declarations:

```
char name[5]  = "WXYZ";
char WXYZ[4]  = "WXYZ";
char string[] = "SMITH";
```

are the same as these three declarations:

```
char name[5]   = { 'W', 'X', 'Y', 'Z', '\0' };
char WXYZ[4]   = { 'W', 'X', 'Y', 'Z' };
char string[7] = { 'S', 'M', 'I', 'T', 'H', '\0' };
```

Initializing Lists of Strings

A list of strings can initialize an array of character pointers.

```
char *strings[] = { "str1", "str2", "str3", "str4" };
```

Initializing Arrays of Structures

Arrays of structures can also be initialized in the same way as one-dimensional arrays.

```
struct { int base; int exp;} base[4] = { { 1, 1},
                                         { 2, 4},
                                         { 4, 16},
                                         { 8, 64} };
```

See Also

- Refer to “String Constants” in Section 1 for more information.
- Refer to “Array Types” in Section 2 for more information.

Initializing Structures

Static structures and external structures can be initialized. The member initializers must be legal initializers for static or external variables of the member types. An automatic structure can also be initialized with an expression of compatible structure type.

Specifying All Arguments

To initialize a data object of type `struct`, write one data item for each structure member.

```
struct employee {
            char name[20];
            int number;
            int salary;
            int start_date;
            } new_hire = { "John L. Smith", 33425, 26000, 860301 };
```

Specifying Less Than All Arguments

If the structure has fewer initializers than it has members, the remaining members are initialized to zero.

The following declarations are the same:

```
struct test {
             int score;
             int max;
             int min;
             } first = { 89, 100, 0 };

struct test {
             int score;
             int max;
             int min;
             } first = { 89, 100 };
```

Specifying Too Many Arguments is Invalid

If there are too many initializers for the structure, the initializer is invalid.

```
struct test {                  /* Invalid Initialization */
             int score;
             int max;
             int min;
             } first = { 89, 100, 0, 999 };
```

See Also

Refer to “Structure Types” in Section 2 for more information on structures..

Initializing Unions

For a union object, the initializer initializes the member that appears first on the declaration list. The initial value must be a legal value for the type of the member being initialized. The rules for constant expression initializers are the same as for structures.

Example

The following example illustrates the initialization of a union variable:

```
union   {  float real_part;
            int int_part; } un = { 3.14159 } ;
```

See Also

Refer to "Union Types" in Section 2 for more information on unions.

Implicit Declarations

Explicit specification of the storage class and type of an identifier is not always needed in a declaration. The C compiler can determine the storage class from the context in declarations of formal parameters and structure members. If a declaration inside a function specifies a storage class, but not a type, the compiler assumes the type of the identifier is `int`. If a declaration inside a function specifies a type, but not a storage class, the compiler assumes the storage class of the identifier to be `auto.` Functions are the exception to this rule; `auto` functions are meaningless. If the type of the identifier is "function returning *type*", it is implicitly declared to be `extern`.

In an expression, an identifier followed by a left parenthesis and not already declared is contextually declared to be "function returning `int`." For example:

```
error ()
    {
    z(i,j);
    }
```

If z was not declared, the compiler implicitly inserts the following declaration at the beginning of the enclosing block:

```
extern int z ();
```

Allowing a pointer-returning function to be implicitly declared as a function returning `int` has been common practice. However, this affects the portability of a program. The problem with pointer-returning functions is that not all compilers allocate the same size storage to the `int` types as they do to pointer types.

See Also

Refer to "Defining Functions" in Section 7 for more information.

Defining and Declaring External Variables

The declaration of an external variable must be consistent with any other declaration of that external variable in another file. The storage class, type, and initializers for the external variables must be the same in all declarations of the variable.

To avoid confusion, the C language distinguishes between the definition and the declaration of the external variable:

Definition The variable is assigned storage.

Declaration The nature of the variable is stated, but not allocated storage.

External variables are defined at the top level of the program (outside any function). The definition states the type and storage class of the external variable. The definition must not say `extern`. The compiler allocates storage to the external variable at this time. The syntax of a definition is the same as a declaration; however, since it occurs outside a function, the variable is external. To use this variable inside a function, the name of the variable must be known to the function or within the name scope. To make the variable known to the function, the variable is declared inside the function with the `extern` keyword.

In the C language, it is difficult to distinguish between the defining declaration and the external or internal linked declarations. Most C programmers use the following rules to help them separate the definition of a variable from its declaration:

- Define all external variables at a single definition point in a source file. Do not include the `extern` storage class specifier in the definition, but include an explicit initializer.
- Within each function that references the external variable, declare the variable with the `extern` keyword, but do not supply an explicit initializer.
- Declarations with `extern` can be declared repeatedly at file level. The declarations must be identical.

Following these rules makes an A Series C program compatible with many other C compilers.

Examples

The following examples illustrate the definition and declaration of external variables:

```
extern int a;              /* declaration */
int b;                     /* definition */
int c = 5;                 /* definition */
```

Summary of Declarations

Refer to the appropriate section for more information on the storage class, type, declarators, or initializers.

Storage Class	type	Declarator	Initializer
one of:	**one of:**	**either:**	**either:**
`asm`	`char`	simple	simple expression
`auto`	`signed char`	pointer	constant expression
`extern`	`unsigned char`	array	list of initializers
`register`	`short int`	function	literal string
`static`	`signed short`		
`typedef`	`unsigned short`		
`inline`	`int`		
	`signed int`		
	`unsigned int`		
	`long int`		
	`signed long`		
	`unsigned long`		
	`float`		
	`double`		
	`long double`		
	`structure`		
	`union`		
	`enumeration`		

Section 4
Type Conversions

The C language enables a value of one type to be either implicitly or explicitly converted to a value of another type. Implicit conversions are performed primarily to make operands and function arguments conform (if possible) to the type of value expected by the operator and function. A type may be implicitly converted to another type for the following operands and function arguments:

- The operands in arithmetic or logical operations
- A value when it is assigned to a location of a different type
- The actual arguments of a function before a function call
- The return value from a function before the function returns

A value may be explicitly converted from one type to another type with the cast expression.

The representation of an object is the pattern of bits in its storage location.

A conversion need not change the representation of a value. For example:

```
void *f;          /* convert the type of the pointer value */
char *p;          /* but not the                           */
p = (char *)f;    /* representation of the pointer value   */
```

The representation can change when a floating-point value is converted to an integer value:

```
int i;
float f = 1.5;     /* The floating-point value 1.5 is changed to */
i = f;             /* value 1 the integer                        */
```

Notes:

- *In this C language implementation, the enumeration type has the same representation as the* `int` *type. It is poor programming style to assign to an object of enumeration type a value that is not in the list of the enumeration members.*
- *Character types are treated as integer types.*

See Also

Refer to "Cast Operator" in Section 5 for more information.

Integral to Integral

The general rule for converting an integral type to another integral type is to preserve the mathematical value. For example, if an `unsigned int` value of 73 is converted to type `signed int`, the resulting value is also 73.

This C language implementation represents

- Unsigned character types as 8-bit nonnegative values
- Signed character types as 8-bit values in two's complement
- Unsigned integer types as 39-bit nonnegative values
- Signed integer types and enumeration types as 39-bit nonnegative values and a separate sign bit

Conversion between types with the same representation does not change the value. The following lists the value changes when converting between types:

Type Conversions	Values
Unsigned character to signed character	No representation change is made. If the original value is less than 128, the mathematical value is unchanged, otherwise the new mathematical value is the original value minus 256.
Unsigned character to unsigned integer, signed integer	The unused bits are zero filled, the mathematical value is unchanged.
Signed character to unsigned character	No representation change is made. If the original value is nonnegative, the mathematical value is unchanged, otherwise the new value is the original value plus 256.
Signed character to unsigned integer	If the original value is nonnegative, the mathematical value is unchanged, otherwise the new value is the original value plus the maximum value of unsigned integer plus one.
Signed character to signed integer	The mathematical value is unchanged, however, the representation for negative values changes considerably.
Unsigned integer to unsigned character	The new mathematical value is the original value modulo 256. The least significant 8 bits of the representation are preserved, the other bits are lost.
Unsigned integer to signed character	The conversion is equivalent to converting from unsigned integer to unsigned character followed by unsigned character to signed character.

Type Conversions	Values
Unsigned integer to signed integer	The mathematical value is unchanged. The sign is assumed to be zero.
Signed integer to unsigned character	The conversion is equivalent to signed integer to signed character followed by signed character to unsigned character.
Signed integer to signed character	If the value is greater than 127 or less than -128, the mathematical value is the original value modulo 256, then converted as if from unsigned character to signed character. Otherwise the mathematical value is unchanged.
Signed integer to unsigned integer	If the value is nonnegative, the mathematical value is unchanged, otherwise the new mathematical value is the original value plus 2**39.

Note: *Converting a signed integer into an unsigned integer then back to a signed integer produces the same mathematical value for nonnegative values. Negative values put through the same conversions can result in positive values.*

Floating-Point to Integral

The conversion of a floating-point value to an integral value truncates any fractional part of the floating value. When a negative floating-point value is converted to an integer, the result is the smallest integer not less than the floating-point value. For example, -2.8 is converted to -2.

The conversion is undefined if the magnitude of the floating-point value is too large for the integral type.

Pointer to Integral

A pointer can be converted to an integral value; however, since this is not portable, when the compiler detects it a warning is issued.

The conversion of a null pointer to an integral value is a special case. A null pointer is a pointer that never points to any object or function. Create a null pointer through explicit assignment or by initializing it to zero. Converting zero to a pointer results in the null pointer. Converting a null pointer to an integral results in 0.

Example

```
int *p;
p = NULL;
p == 0;     /* will be 1 (true) */
```

Floating-Point to Floating-Point

The general rules for converting floating-point type are as follows:

Floating-Point Type Conversions	Values
`float` or `double` to `long double`	The result value is the same as the original value.
`long double` to `float` or `double`	A long double value is converted to a `float` or `double` value by truncating the 78-bit double mantissa to a 39-bit single mantissa and truncating the 15-bit exponent to a 6-bit exponent. The result value is undefined if the magnitude of the original value is too large for the result type.

Integral to Floating-Point

For this implementation, the floating-point type can exactly represent all integral values. Therefore, the result is the equivalent floating-point value of the integral value.

Structure to Structure

A structure object can be converted only to another structure that has compatible type as the original structure.

The conversion does not change the representation. However it may not preserve the bit patterns in any unused portions or “holes” in the structure.

Union to Union

A union object can be converted only to another union that has compatible type as the original union.

The conversion does not change the representation. However, it may not preserve the bit patterns in any unused portions or “holes” in the union.

Pointer to Pointer

A null pointer constant of any type can be converted to any pointer type. It is still recognized as a null pointer. With this C compiler, the conversion does not change its representation. A null pointer is a pointer that never points to any object or function.

A pointer to an object of one type can be converted to a pointer to an object of another type by use of an explicit cast. The explicit cast is not necessary if one pointer is a pointer to `void`. The resulting pointer might not be valid if it is improperly aligned for the type of object pointed to. It is always true that a pointer to an object of a given alignment can be converted to a pointer to an object of a less strict alignment and back again without change. (An object that has type `char` has the least strict alignment.)

The following cannot be converted:

- "pointer to function" to "pointer to object"
- "pointer to object" to "pointer to function"
- "pointer to function" to "pointer to `void`ö
- "pointer to `void`ö to "pointer to function"

See Also

Refer to "Cast Operator" in Section 5 for more information.

Integral to Pointer

Integral values can be converted to pointer types. However, the result of these conversions is not portable.

An integral constant expression with the value of zero (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 never points to any object or function.

Array to Pointer

The expression "array of *type*" is converted to a value of type "pointer to *type*" by substituting for the expression a pointer to the first element of the array.

The exceptions to this conversion are when the array is

- An argument to the `sizeof` function
- An argument to the address (`&`) function
- A string literal used to initialize a character array

See Also

Refer to "Sizeof Operator" in Section 5 for more information.

Function to Pointer

An expression of type "function returning *type*" is converted to a value of type "pointer to function returning *type*" by substituting for the expression a pointer to the function. The two exceptions to this conversion are:

- As operand of the address (`&`) function
- As operand of the `sizeof` function

Any Type to Array or Function

Conversions to array and function types are not allowed.

Any Type to Void

All values can be converted to type `void` however, the result of a conversion cannot be used for anything. This conversion can be used when an expression value is to be discarded, such as in an expression statement.

See Also

Refer to "Expression Statements" in Section 6 for more information.

Explicit Conversions-Casting Conversions

The implicit conversions previously described in this section can also be explicitly performed with a type cast.

Examples

The following examples illustrate cast expressions. Given the following definitions and declarations:

```
int i, *ip;

char *cp;
```

The following cast expressions are legal:

```
(char) i;          /* converts an integer value to a character value */
cp = (char *)ip;   /* converts a pointer to an integer */
                   /* to a pointer to a character      */
```

See Also

Refer to "Cast Operator" in Section 5 for more information.

Assignment Conversions

In an assignment expression, the type of the expression on the left of the assignment operator should be the same as the type of the expression on the right. If the expressions are not the same type, the compiler attempts to convert the value on the right side of the expression operator to the type of the value on the left side. The following conversions are legal:

Left Side	Right Side
any arithmetic type	any arithmetic type
any pointer type	the integer constant 0
pointer to *type*	array of *type*
pointer to function	function

This C language implementation allows casting conversions in an assignment expression.

See Also

Refer to "Assignment Operators" in Section 5 for more information.

Usual Arithmetic Conversions

This C language implementation performs all conversions before executing the operation. The following rules describe the usual arithmetic conversions:

Step	If . . .	Then . . .
1	Either operand is of type `long double`	The other operand is converted to type `long double`.
2	Either operand is of type `double`	The other operand is converted to type `double`.
3	Either operand is of type `float`	The other operand is converted to type `float`.

Otherwise, the integral promotions are performed on both operands. If this is the case, then the following rules apply:

Step	**If . . .**	**Then . . .**
1	Either operand is of type `unsigned long int`	The other operand is converted to `unsigned long int`.
2	One operand is of type `long int` and the other operand is of type `unsigned int`, and the operand of type `long int` can represent all values of an `unsigned int`	The operand of type `unsigned int` is converted to `long int`.
	One operand is of type `long int` and the other operand is of type `unsigned int`, and the operand of type `long int` **cannot** represent all values of an `unsigned int`	Both operands are converted to `unsigned long int`.
3	Either operand is of type `long int`	The other operand is converted to type `long int`.
4	The compiler reaches this step	Both operands are of type `int`.

Section 5
Expressions and Operators

An expression is a sequence of operators and operands that specifies how to compute a value or, in the case of a `void` expression, how to generate side effects. This section is divided into the following topics:

- Expressions
  - The type and class of expressions
  - The primary and constant expressions
  - The side effects of expressions
  - The compiler evaluation of expressions
- Operators
  - The functional categories of operators
  - The precedence and associativity of operators

Expressions

Every expression has two attributes

- Type
- Class

Type of an Expression

Every expression or subexpression has a type. If the expression is an identifier, its type is determined when it is declared. If an expression is a constant, the rules for typing constants determine its type. If an expression is enclosed in parentheses, its type is that of the expression in parentheses.

Otherwise, the expression must be an operator with subexpressions for operands. Each operator has its own rules for determining its type, based on the types of its operands. Start with the simplest subexpressions and determine their types, then work up through the operators until the type of the whole expression is determined.

Example

For example, the following expression has the constant operands `2.4` with type `double`, `3L` with type `long`, and `2` with type `int`. Subtracting a value of type `int` from a value of type `long` gives a result of type `long`. Adding a value of type `long` to a value of type `double` gives a result of type `double`. So the type of this expression is the type of the addition operator, which is type `double`.

```
2.4 + (3L - 2)
```

See Also

Refer to "Constants" in Section 1 for more information on the different types of constants.

Class of an Expression

Every expression or subexpression is in one of four classes:

Class of Expressions	Description
`void` expression	This expression yields no value.
Lvalue	This expression designates the location of a data object of some type.
Rvalue	This expression yields a value of some data type.
Function locator	This expression designates a function of some type.

Determine the class of an expression by starting with the simplest subexpression and working up through the operators. Enclosing an expression in parentheses does not change its class.

void Expression

A `void` expression is an expression whose type is `void`. As discussed previously, the compiler evaluates `void` expressions for their side effects only. An example of a `void` expression is the C standard library function `srand(0)`. The `srand` function provides an initial value for a random number generator. The `srand` function has type "function returning `void`." If any other class of expression occurs where the compiler expects a `void` expression, the compiler evaluates it and discards the result. The function `printf`, for example, returns a value every time a program calls it, but the expression that calls it often discards that value.

Lvalue

An object is a region of storage that can be examined and altered. An lvalue is the value of an expression or the result of an expression that refers to the location of the object.

The term "lvalue" originally came from the following assignment expression:

```
E1 = E2
```

The expression `E1` is called an lvalue because it can be used on the left side of the assignment expression. Today, however, an lvalue is more difficult to define. An lvalue can also be called a "data object locator." It designates the location of a data object for the purposes of obtaining its address, obtaining its stored value, or altering its stored value.

The following forms of expressions are legal when used where the compiler requires an lvalue:

Expression	Description
Identifier	The name of a variable declared with the arithmetic, pointer, enumeration, structure, or union type is a legal lvalue.
Indirection operator	The result of the indirection operator (*) is an lvalue. For example, the indirection expression `*ptr` produces an lvalue.
Subscript	Since the subscript expression `a[b]` is the same as `*(a+b)`, subscript expressions are legal lvalues.
Parenthesized expression	A parenthesized expression is an lvalue if the expression inside the parentheses produces an lvalue.
Direct member selection	The result of the member selection operator (.) is an lvalue when the left operand is an lvalue. For example, if expression `exp` is an lvalue, then the direct member selection expression `exp.member` is an lvalue. But `f().member` is not an lvalue because `f()` does not produce an lvalue.
Indirect member selection	Since `ptr -> member` is the same as `(*ptr).member`, the result of the indirect member selection operator (`->`) is an lvalue.

The following expressions are not lvalues:

- Names of arrays
- Names of functions
- Names of enumeration constants
- The result of an assignment expression
- The result of the cast operator
- A function call

The following operators require their operands to be lvalues:

- The operand of a unary address operator `&`
- The operand of a postfix or prefix increment `++` or decrement `--`
- The left operand of an assignment operator

A modifiable lvalue is an lvalue that does not have array type and does not have a type declared with the `const` type qualifier.

Gray Code

Gray code is where an lvalue is both accessed and modified in an expression and the order of the access and modify operations is indeterminate. The following expressions all have gray code:

```
a + (a = 3)
(a++) + a
vec [a++] = a
(a = 3) + (a = 4)
```

The following expressions do not contain gray code:

```
a = a + 1
a++, vec[a]
a, vec [a++]
```

The compiler will emit a warning when gray code is detected.

Rvalue

An rvalue is an expression that permits examination, but not alteration of a value. The term "rvalue" originally came from the following assignment expression:

```
E1 = E2
```

The expression `E2` is called the rvalue because it can be used on the right side of an assignment expression.

The following forms of expressions are legal when used where an rvalue is required:

Expression	Description
Identifier	The value stored in the data object designated by the identifier is the rvalue.
Identifier with type "array of *type*"	When the name of an array is used where an rvalue is required, the rvalue is the address of the first element of the array.
Indirection operator	The rvalue of the indirection operator expression `*ptr` is the value stored in the object pointed to by `ptr`.
Parenthesized expression	The rvalue is the value of the enclosed expression.
Direct member selection	The rvalue of the direct member selection operator (`.`) is the value stored in that member. The expression `f().member` produces an rvalue that is the value of member `member` of the structure returned by function `f`.

Expression	Description
Indirect member selection	The rvalue of an expression such as `ptr->member` is the value stored in the member `member` of the structure to which `ptr` points.
Function name	The rvalue is the address of the function and its type is "pointer to function returning *type*," where *type* is the type of the function named.
Enumeration constant	The rvalue is the value of the constant.
Result of an assignment expression	The rvalue is the value that was stored in the assignment expression.

Function Locator

A function locator is an expression that has function type. If an identifier declared as a function locator appears in a context other than as an operand that can or must be a function locator, it is converted to a pointer.

See Also

- Refer to "Array Types," "Integer Types," "Enumeration Types," "Pointer Types," "Structure Types," and "Union Types" in Section 2 for more information.
- Refer to Section 4, "Type Conversions," for more information on pointer conversions.

Primary Expressions

The primary expressions are as follows:

Primary Expression	Description
Identifier	An identifier is a primary expression. If it was declared as an object, it is an lvalue. If it was declared as a function, it is a function locator.
Constant	A constant is a primary expression. Its type depends on its form (refer to "Constants "in Section 1 for more information).
String literal	A string literal is a primary expression. It is an lvalue of type "array of `char`" (refer to "String Constants" in Section 1 for more information on string literals).
Parenthesized expression	A parenthesized expression is an expression enclosed by parentheses. The purpose of a parenthesized expression is to group the operands in an expression. Parenthesized expressions make code more readable and override the default operator precedence.
`__stack_number__`	`__stack_number__` is a value that uniquely identifies the process running. Each client of a library has a different `__stack_number__` (refer to ALGOL's PROCESSID for more information).

Primary Expression	Description
`__user_lock__`	`__user_lock__` locks an interlock associated with the program and returns the number 0 as a value. Attempting to lock the interlock twice causes the program to hang (refer to ALGOL's LOCK statement for interlocks for more information).
`__user_unlock__`	`__user_unlock__` unlocks an interlock associated with the program and returns the number 0 as a value (refer to ALGOL's UNLOCK statement for more information).

The type of the parenthesized expression is the same as the enclosed expression. The compiler does not perform any conversion just because the expression is enclosed in parentheses. The value of the parenthesized expression is the value of the enclosed expression. If the enclosed expression is an lvalue, the parenthesized expression is an lvalue; if the enclosed expression is a function locator, the parenthesized expression is a function locator.

Examples

This expression shows an override of default operator precedence:

```
a = b * ( c + d );
```

This expression shows the use of unnecessary parentheses:

```
( a ) = b;
```

This expression prints `b` before passing it to `f`:

```
f(a, (printf("%d\n",b), b));
```

See Also

Refer to "Constants" and "Identifiers" in Section 1 for more information.

Constant Expressions

A constant expression is an expression that the compiler can reduce to a known value before it reads any more of the source file.

In some contexts, the C language requires expressions that evaluate to a constant. These contexts are as follows:

- When a `case` constant is evaluated in `switch` statements
- When the size of an array is evaluated
- When initializers are evaluated
- When specifying the size of a bit field member of a structure
- When the value of an enumeration constant is evaluated

- When the tested value in the `#if` preprocessor statement is evaluated

The C compiler imposes slightly different restrictions on the form of expression it allows in each context.

If the expression is evaluated by the compiler, the arithmetic precision and range are at least as great as if the expression were being evaluated in the execution environment.

A constant expression can do the following:

- Contain integer expressions
- Use casts to integral types

  (Casts are not allowed in `#if` type expressions.)

- Use the following binary operators:

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

- Use the following unary operators:

```
+   -   ~   !
```

- Use the following conditional operator:

```
?:
```

- Use parentheses for grouping

A function call operator, an increment and decrement operator, and an assignment operator is not allowed in constant expressions.

An integral constant expression must involve only integer, enumeration, or character constants, and casts to integral types. The following operators must not appear in an integral constant expression, unless the expression is the operand of the `sizeof` operator:

- Array subscripting operator (`[ ]`)
- Direct member selection operator (`.`)
- Indirect member selection operator (`->`)
- Address operator (`&`)
- Indirection operator (`*`)

For constant expressions in initializers, floating-point constants and arbitrary casts can be used in addition to integral constant expressions. Lvalues and objects that have static storage duration or function identifiers can also be used to specify addresses, explicitly with the unary `&` operator or implicitly for unsubscripted array identifiers or

function identifiers. A constant difference in the addresses of two members of the same aggregate can be used.

See Also

- Refer to "Constants" in Section 1 for more information.
- Refer to "Enumeration Types," "Bit Fields," "Function Types," and "Integer Types" in Section 2 for more information.
- Refer to "Initializers" in Section 3 for more information.
- Refer to "`#if`, `#else`, `#elif`, and `#endif` Directives" in Section 8 for more information.

Side Effects

A side effect is a change in the value stored in a data object or a change in the state of a file as a byproduct of evaluating an expression. Function calls, the increment and decrement operators, and the assignment operators all cause side effects. A C program evaluates some expressions, known as "`void` expressions," only for their side effects. A `void` expression either has no value because its type is `void` or the program discards its value.

The following expressions all have side effects:

```
printf("hello world\n")

++a

a = 1
```

If an expression has more than one side effect, the order in which the program evaluates any subexpressions can yield unexpected results. In many cases the C language does not restrict order of evaluation enough to make the evaluation of such expressions completely predictable

For example, a program can evaluate the arguments of the following function call in arbitrary order, so there is no way of precisely determining the values of the arguments passed to the function:

```
f(++a, ++a)
```

As a general rule, avoid writing expressions with more than one (related) side effect.

Certain operators provide a predictable order of evaluation of their operands. Use these operators to write expressions with predictable side effects. The logical AND (`&&`), logical OR (`||`), condition (`?:`), and comma (`,`) operators ensure strict left to right evaluation. Therefore, the following expression can be written so that `a[i]` is never outside the "array of `char`ö data object `a`:

```
0 <= i && i < sizeof a && a[i] != 'c'
```

Similarly, in the following expression, the first input character is stored in c1 and the second in c2:

```
c1 = getchar(), c2 = getchar()
```

An expression involving more than one occurrence of the same commutative and associative binary operators (* + & ^ |) can be regrouped arbitrarily, provided the types of the operands or of the results are not changed by this regrouping. To force a particular grouping of operations, either the value of the expression to be grouped can be explicitly assigned to an object or parentheses can be used to group the operations.

The descriptions of the operators later in this section state any restrictions concerning the order of evaluation for a given operator.

Discarded Values

In the following contexts, the compiler evaluates an expression, but discards its values:

- An expression statement
- The first operand of a comma expression
- The initialization and incrementation expressions in a `for` statement

See Also

Refer to "for Statement" in Section 6 for more information.

Compiler Optimization of Source Code

In general, the compiler is free to do the following:

- Generate code that has computational behavior similar to the source code
- Rearrange code

The compiler might not generate any code for an expression if the expression has no side effects and its value is discarded.

See Also

Refer to "Discarded Values" in this section for more information.

Operators

An operator generates a value by performing a defined operation on one or more operands. An operand can consist of declared identifiers, constants, function calls, array elements, and structure or union members. The value of an operand replaces its reference in an expression. After the values are replaced, the operator then performs its operation on these values.

Table 5–1 lists the operators used in expressions. These operators are listed under their functional categories. Each of these operators is described in detail later in this section.

Table 5–1. C Operators

Addressing and Size Operators	
Operator Name	**Operator Symbol**
Array subscripting operator	[ ]
Indirect member selection operator (pointer to structure)	->
Indirection operator (pointer reference)	*
Direct member selection operator	.
Address-of operator	&
sizeof operator	sizeof
Arithmetic Operators	
Operator Name	**Operator Symbol**
Addition operator	+
Subtraction operator	-
Multiplication operator	*
Division operator	/
Modulo operator	%
Increment operator	++
Decrement operator	- -
Unary minus operator	-
Unary plus operator	+
Assignment Operators	
Operator Name	**Operator Symbol**
Simple assignment operator	=
Compound assignment operators	+= -= *= /= %= <<= >>= &= ^= |=
Operator Name	**Operator Symbol**
Bitwise negation operator	~
Bitwise AND operator	&
Bitwise exclusive OR operator	^

Table 5–1. C Operators

Bitwise inclusive OR operator	|
Operator Name	**Operator Symbol**
Bitwise left shift operator	<<
Bitwise right shift operator	>>
Equality Operators	
Operator Name	**Operator Symbol**
Equal to operator	==
Not equal to operator	!=
Logical Operators	
Operator Name	**Operator Symbol**
Logical negation operator	!
Logical AND operator	&&
Logical OR operator	||
Relational Operators	
Operator Name	**Operator Symbol**
Greater than operator	>
Greater than or equal to operator	>=
Less than operator	<
Less than or equal to operator	<=
Miscellaneous Operators	
Operator Name	**Operator Symbol**
Function call operator	( )
Type cast operator	(type)
Conditional expression operator	?:
Comma operator	,

Operators can also be categorized into the following four forms:

Operator Form	Description
Primary	The primary operators qualify an expression. For example, the parentheses operator ( ) is used to modify the precedence within an expression.
Unary	The unary operators require one operand that is either prefixed (occurring to the left of the operator) or postfixed (occurring to the right of the operator). For example, a prefix decrement operator (- -) subtracts the constant 1 from the value of the operand and stores the value back into the operand.
Binary	The binary operators require two operands, one occurring on either side of the operator. For example, the addition operator (+) results in the sum of the operands.
Ternary	The ternary operators requires three operands. For example, the conditional expression operator (?:) returns the value of one of two alternative expressions as its result.

Operator Precedence and Associativity

Each expression operator has a precedence level and associativity. When parentheses do not indicate the exact grouping of operands with operators, the compiler uses the rules of precedence and associativity:

1. The compiler groups the operand with the operator that has the highest precedence.

2. If two operators have the same precedence, the compiler groups the operator with the left or right operand depending on the associativity of the operator. The compiler groups the operand with the right operator if the operator is right-associative and with the left operator if the operator is left-associative.

Table 5–2 shows the precedence level (highest to lowest, in order) and the rules of associativity of operators. The operators grouped within the same table cell have equal precedence.

Table 5–2. Precedence Level and Associativity of Operators

Operator Name	Operator Symbol	Associativity
Function call Array subscripting Indirect/Direct member selection Postfix Increment/Decrement	( ) [ ] -> . ++ --	Left to right

Table 5–2. Precedence Level and Associativity of Operators

Operator Name	Operator Symbol	Associativity
Prefix Increment/Decrement Unary plus/minus Address Indirection Logical negation Bitwise negation sizeof	++ -- + - & * ! ~ sizeof	Right to left
Type cast	(*type*)	Right to left
Multiplication/Division Remainder operator	* / %	Left to right
Binary plus/minus (addition and subtraction)	+ -	Left to right
Bitwise shift (left and right)	<< >>	Left to right
Relational operators (less than, greater than, less than or equal to, and greater than or equal to)	< <= > >=	Left to right
Equal to/Not equal to	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise exclusive OR	^	Left to right
Bitwise inclusive OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional expression	? :	Right to left
Assignment operators	= += -= *= /= %= <<= >>= &= ^= |=	Right to left
Comma	,	Left to right

Examples

The following two expressions are the same because the division operator (/) has higher precedence than the addition operator (+):

```
a / b + c

(a / b) + c
```

The following two expressions are the same because the multiplication operator (*) has higher precedence than both the bitwise inclusive OR (|) and the bitwise AND (&) operators. The bitwise AND operator (&) has higher precedence than the bitwise inclusive OR operator (|):

```
a | b & c * d

a | ( b & ( c * d) )
```

The following two expressions are the same because the operators have the same level of precedence. The compiler uses the rules of associativity to group the operators. The operand `b` is grouped with `a` because the associativity of the remainder operator (`%`) and the multiplicative operator (`*`) is left to right:

```
a % b * c

(a % b) * c
```

The following two expressions are the same because the associativity of the assignment operator (`=`) and the multiply assignment operator (`*=`) is right to left.

```
a = b *= c

a = (b *= c)
```

Addressing and Size Operators

The following addressing and size operators are discussed:

- The array subscripting operator
- The indirect member selection operator
- The indirection operator
- The direct member selection operator
- The address-of operator
- The `sizeof` operator

Array Subscripting Operator

The array subscripting operator ([ ]) uses the values of its operands to produce an lvalue that designates an element of an array *array*[*index*]. Both operands must be rvalues. One operand must be a pointer; the other must have an integral type. Usually, the pointer is written to the left (*array*) and the integer operand is written inside the square brackets (*index*). This notation is similar to that in several other programming languages, where `x[y]` designates the element of the array `x` at offset `y`. If the array is of type "array of *type*" (or the pointer is of type "pointer to *type*"), the result is of type *type*. The expression `x[y]` is entirely equivalent to the expression `*(x + (y))`.

If `E1` is an array (converted to a pointer to the first member of the array) and `E2` is an integer, `E1[E2]` designates the `E2`-th member of `E1` (counting from zero). Because addition is commutative, `E2[E1]` also designates the `E2`-th member of `E1`.

The following example illustrates array subscripts and pointer relationships:

```
char buffer [100], *bptr = buffer;
buffer[0] = '\0';
bptr[1] = bptr[0];
```

Successive subscript operators designate a member of a multidimensional array object. If E is an *n*-dimensional array with dimensions *i* x *j* x...x *k*, then E is converted to a pointer to an (*n*-1)-dimensional array with dimensions *j* x...x *k*. This pointer is then applied to the next subscripting operator and the whole operation is repeated until no more subscript operators are present. Then the unary operator * is applied to the resulting pointer to obtain the desired value. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

An example of an array declaration is the following:

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

In this declaration, `x` is a 3x5 array of integers. Or more precisely, `x` is an array of three objects; each object is an array of five integers. In the expression `x[2]`, which is the same as `*(x + 2)`, `x` is first converted to a pointer to the initial array of five integers. Adding 2 to the pointer gives a result pointing to the third array of five integers. Indirection is applied to yield the third array of five integers. The expression `x[2][3]` yields the fourth `int` in the third array of integers.

However, be careful not to write an expression such as `b[i,7]`. This expression is not equivalent to `b[i][7]`. The comma operator (`,`) causes the compiler to evaluate `i` as a `void` expression, so its value is discarded. The resulting subscript is the right operand of the comma operator, giving an expression equivalent to `b[7]`. The only way to write multiple subscripts is to enclose each in its own square brackets.

See Also

Refer to "Array Types" and "Integer Types" in Section 2 for more information.

Indirect Member Selection Operator

The indirect member selection operator (`->`) produces an lvalue that designates the structure or union member in the structure or union pointed to.

The following two expressions are equivalent:

```
pointer-> member
```

```
(*pointer).member
```

The *pointer* must be a pointer to a structure or a union type. The *member* is an identifier that must be the name of a member of that structure or union type. The result of an indirect selection member expression is the named member of the structure or union. The result is an lvalue.

See Also

Refer to "Structure Types" and "Union Types" in Section 2 for more information.

Indirection Operator

The unary indirection operator (*) indicates indirection.

An expression associated with the unary indirection operator must be a pointer. If the operand is an object pointer that points to a valid object, the result is an lvalue that refers to the object. If the operand is a function pointer that points to a valid function, the result is a function locator.

If the type of the expression is "pointer to *type*," the type of the result is "*type*."

If an invalid value has been assigned to the pointer, the behavior of the unary indirection operator (*) is undefined. Such invalid values include the following:

- An integral constant expression with value 0.
- An address inappropriately aligned for the type of object pointed to.
- The address of a `const` object when used as a modifiable lvalue.
- The address of an object that has automatic storage duration when execution of the block in which the object is declared and all enclosed blocks have terminated.

See Also

Refer to "Pointer Types" in Section 2 for more information on pointers.

Direct Member Selection Operator

The direct member selection operator (`.`) obtains the structure or union member in the structure or union. The first operand of the operator must have a structure or union type. The second operand must name a member of that type. The value of the selection operation is that of the named member. The result is an lvalue only if the first expression is an lvalue (for example, the result of a function call is not an lvalue).

See Also

Refer to "Structure Types" and "Union Types" in Section 2 for more information.

Address Operator

The address-of operator (&) returns a pointer to its operand.

```
&lvalue
```

The operand must be an lvalue. The C language does not allow application of the address-of operator (&) to a `register` variable or a bit field.

If the type of the operand is "*type*," the type of the result is "pointer to *type*."

See Also

- Refer to "Bit Fields" and "Pointer Types" in Section 2 for more information.
- Refer to "Storage Class Specifiers" in Section 3 for more information.

sizeof Operator

The `sizeof` operator returns the size in storage units (bytes) of its operand. A `sizeof` expression has two forms:

- The `sizeof` keyword followed by an expression

  ```
  sizeof expression
  ```

- The `sizeof` keyword followed by a type name in parentheses

  ```
  sizeof ( type-name )
  ```

The expression `sizeof` *`expression`* returns the same result as if it had been applied to the name of the type of the expression. The `sizeof` operator does not itself cause any of the usual arithmetic conversions to be applied to *`expression`*. But if *expression* contains operators that perform arithmetic conversions, those conversions are then considered when determining the type.

The expression `sizeof` (*`type-name`*) returns the size of an object of the type named in parentheses. The type name cannot name an array type with no explicit length, a function type, or the type `void`.

When applied to an operand that has type `char`, `unsigned char`, or `signed char`, the result is 1. When applied to an operand that has an array type, the result is the total

number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in the object, including whatever internal and trailing padding might be needed to align each member in such a structure or union properly.

The size of a string literal includes the terminating null character implicit on all string literals.

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

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

The implementation of the `alloc` function must ensure that its return value is aligned correctly for conversion to a pointer to `double`.

Another use of the `sizeof` operator is to compute the number of members in an array as in the following example:

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

See Also

Refer to "Array Types," "Function Types," "Pointer Types," "Structure Types," "Union Types," and "Void Type" in Section 2 for more information.

Arithmetic Operators

The following arithmetic operators are discussed:

- The addition operator
- The subtraction operator
- The multiplication operator
- The division operator
- The remainder operator
- The increment and decrement operators
- The unary minus and plus operators

Addition Operator

The binary plus sign operator (+) indicates addition. Both operands can have arithmetic types or one operand can have "pointer to data" type and the other can have integral type. The result of the binary plus sign operator (+) is the sum of the operands. The result is not an lvalue.

The usual arithmetic conversions are performed on the operands. If both operands have arithmetic type, the type of the result is that of the converted operands. For integer operands, integer addition is performed. For floating-point operands, floating-point addition is performed.

A pointer to an object in an array and a value of any integral type can be added. The integral value is converted to an address that is offset by multiplying it by the size of the object pointed to. The result is the same type as the original pointer. If the original pointer points to a member of an array object, the result points to another member of the same type array object, appropriately offset from the original member. Thus, if `P` points to a member of an array object, the expression `P + 1` points to the next member of the array object. Unless both the pointer operand and the result point to a member of the same array object, the behavior is undefined if the result is used as the operand of a unary * operator.

The binary plus sign operator (+) is commutative and associative. The compiler may regroup expressions involving several additions at the same level. For example, given the following expression:

```
a + b + c + d
```

The expression might be regrouped in the following way:

```
( d + c ) + ( b + a )
```

In the event of overflow, the binary + operator may produce unpredictable results if the operands, after conversion, are signed integers, floating-point numbers, or pointers. By definition, overflow does not occur for unsigned integers.

Subtraction Operator

The binary minus sign (-) operator indicates subtraction. The operands can be one of the following types:

Subtraction Operand Type	Description
Arithmetic	Both operands can be an arithmetic type.
Pointer	Both operands can be the same pointer type; refer to "Pointer Arithmetic" in Section 2.
Pointer and integer	The left operand can be a pointer and the right operand can have integral type.

The result is not an lvalue.

The usual arithmetic conversions are performed on the operands. If both operands are arithmetic, the type of the result is that of the converted operands. For integer operands, integer subtraction is performed. For floating-point operands, floating-point subtraction is performed.

A pointer to an object in an array and a value of any integral type can be subtracted. The integral value is converted to an address that is offset by multiplying it by the size of the object pointed to. The result is the same type as the original pointer. If the original pointer points to a member of an array object, the result points to another member of the same type array object, appropriately offset from the original member. Thus if `P` points to a member of an array object, the expression `P - 1` points to the previous member of the array object. Unless both the pointer operand and the result point to a member of the same array object, the behavior is undefined if the result is used as the operand of an indirect member selection operator (*).

In the event of underflow, the binary minus sign operator (**-**) may produce unpredictable results if the operands, after conversion, are signed integers, floating-point numbers pointer. By definition, underflow cannot occur for unsigned integers.

See Also

- Refer to "Arithmetic Types," "Array Types," "Enumeration Types," and "Pointer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Multiplication Operator

The binary multiplication operator (*****) indicates multiplication. Each operand must have an arithmetic type. For integer operands, integer multiplication is performed. For floating-point operands, floating-point multiplication is performed. The usual arithmetic conversions are performed on the operands. The result of the binary multiplication operator (*) is the product of the operands. The result is not an lvalue.

The binary multiplication operator (*) is commutative and associative. The compiler may regroup expressions involving several multiplications at the same level. For example, given the following expression:

```
a * b * c * d
```

The expression might be regrouped as follows:

```
( d * c ) * ( b * a )
```

In the event of overflow, the binary multiplication operator (*) may produce unpredictable results if the operands, after conversion, are signed integers or floating-point numbers. By definition, there is no overflow for unsigned integers.

Division Operator

The binary `division (/)` operator indicates division. The result of the binary division operator is the quotient of the operands. The result is not an lvalue. Each operand must have an arithmetic type.

For floating-point operands, floating-point division is performed.

The usual arithmetic conversions are performed on the operands.

In the event of overflow, the binary division operator (/) may produce unpredictable results. Overflow cannot occur if the operands are integers.

In this C language implementation, the result of division by zero is a run-time divide fault.

Remainder Operator

The binary remainder operator (%) computes the remainder from the division of the first operand by the second operand. Each operand must have an integral type. The usual arithmetic conversions are performed on the operands. The type of the result is that of the converted operands. The result is not an lvalue.

When integer operands are positive, the result is positive. If one of the operands is negative, the sign of the result of the remainder operator (%) is the same as the dividend.

In this C language implementation, the result of division by zero is a run-time divide fault.

See Also

- Refer to "Arithmetic Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Increment Operator

The increment operator (++) enables incrementing of an operand by a value of one. The following are the two forms of increment operators:

Increment Operator Form	Description
Prefix	The operator in the prefix form occurs to the left of the operand and the operand is incremented before its value is substituted in the expression.
Postfix	The operator in the postfix form occurs to the right of the operand and the operand is incremented after substituting its original value in the expression.

Prefix Increment Operator

The prefix increment operator (++) performs an "increment before" operation. A prefix increment is an operation that produces side effects. The prefix increment operator does the following:

- Adds the constant 1 to the value of the operand
- Stores the incremented value into the operand, modifying it

The following three expressions are equivalent:

```
x = x + 1

++(x)

(x) += 1
```

The operand of the prefix increment operator must have a scalar type and must be a modifiable lvalue. The result of the prefix increment is the new value of the operand, after it was incremented. The result is not an lvalue.

The usual arithmetic conversions are performed on the operand and the constant 1 before the addition is performed. The usual assignment conversions are performed when storing the sum back into the operand. The type of the result is that of the lvalue operand before the conversion.

The prefix increment operator can produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. If the PORT(UNSIGNED) suboption is set, then the result of incrementing the largest representable value of an unsigned type is 0.

If the operand is a pointer, the prefix increment operator moves the pointer forward beyond the object pointed to, as if to move the pointer to the next element within the array of objects.

See Also

- Refer to Section 2, "Types," for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.
- Refer to Section 10, "Compiler Control Options," for more information on the PORT option.

Postfix Increment Operator

The postfix increment operator (++) performs an "increment after" operation. A postfix increment is an operation that produces side effects. The postfix increment operator does the following:

- Adds the constant 1 to the value of the operand
- Stores the incremented value into the operand, modifying it

The operand of the postfix increment operator must have a scalar type and must be a modifiable lvalue. The result of the postfix increment is the old value of the operand, before it was incremented. The result is not an lvalue.

The usual arithmetic conversions are performed on the operand and the constant 1 before the addition is performed. The usual assignment conversions are performed when storing the sum back into the operand. The type of the result is that of the lvalue operand before the conversion.

The postfix increment operator can produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. If the PORT(UNSIGNED) suboption is set, then the result of incrementing the largest representable value of an unsigned type is 0.

If the operand is a pointer, the postfix increment operator moves the pointer forward beyond the object pointed to, which positions the pointer on the next element within the array of objects. The value of the expression is the pointer before incrementation.

See Also

- Refer to Section 2, "Types," for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.
- Refer to Section 10, "Compiler Control Options," for more information on the PORT option.

Decrement Operator

The decrement operator (**--**) enables decrementing of an operand by a value of one. The following are the two forms of decrement operators:

Decrement Operator Forms	Description
Prefix	The operator in the prefix form occurs to the left of the operand and the operand is decremented before its value is substituted in the expression.
Postfix	The operator in the postfix form occurs to the right of the operand and the operand is decremented after substituting its original value in the expression.

Prefix Decrement Operator

The prefix decrement operator (**--**) performs a "decrement before" operation. A prefix decrement is an operation that produces side effects. The prefix decrement operator does the following:

- Subtracts the constant 1 from the value of the operand
- Stores the decremented value back into the operand, modifying it

The following three expressions are equivalent:

```
x = x - 1

--(x)

(x) -= 1
```

The operand of the prefix decrement operator must have a scalar type and must be a modifiable lvalue. The result of the prefix decrement is the new value of the operand after it was decremented. The result is not an lvalue.

The usual arithmetic conversions are performed on the operand and the constant 1 before the subtraction is performed. The usual assignment conversions are performed when when the difference is stored back into the operand. The type of the result is that of the lvalue operand before the conversion.

The prefix decrement operator can produce unpredictable effects if underflow occurs and the operand is a signed integer or floating-point number. If the PORT(UNSIGNED) suboption is set, then the result of decrementing the value 0 of an unsigned integer type is the largest representable value of that type.

If the operand is a pointer, the prefix decrement operator moves the pointer back over the object pointed to, which positions the pointer on the previous element within the array of objects. The value of the expression is the pointer after modification.

See Also

- Refer to Section 2, “Types,” for more information.
- Refer to “Usual Arithmetic Conversions” in Section 4 for more information.
- Refer to Section 10, “Compiler Control Options,” for more information on the PORT option.

Postfix Decrement Operator

The postfix decrement operator (--) performs a “decrement after” operation. A postfix decrement is an operation that produces side effects. The postfix decrement operator does the following:

- Subtracts the constant 1 from the value of the operand
- Stores the decremented value into the operand, modifying it

The operand of the postfix decrement operator must have a scalar type and must be a modifiable lvalue. The result of the postfix decrement is the old value of the operand, before it was decremented. The result is not an lvalue.

The usual arithmetic conversions are performed on the operand and the constant 1 before the subtraction is performed. The usual assignment conversions are performed when storing the difference back into the operand. The type of the result is that of the lvalue operand before the conversion.

The postfix decrement operator can produce unpredictable effects if underflow occurs and the operand is a signed integer or floating-point number. If the PORT(UNSIGNED) suboption is set, then the result of decrementing the value 0 of an unsigned integer type is the largest representable value of that type.

If the operand is a pointer, the postfix decrement operator moves the pointer back over the object pointed to, which positions the pointer on the previous element within the array of objects. The value of the expression is the pointer before decrementation.

See Also

- Refer to Section 2, "Types," for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.
- Refer to Section 10, "Compiler Control Options," for more information on the PORT option.

Unary Minus Operator

The result of the unary minus operator (-) is the negative (arithmetic complement) of its operand. The form of a unary minus operator is

```
- expression
```

The *expression* can be any arithmetic type. The integral promotions are performed on the operand. The result has the promoted type and is not an lvalue.

The unary minus expression `-E` is the same as the expression `(0 - E)`. The two expressions return the same result.

Unary Plus Operator

The result of the unary plus operator (+) is the value of its operand. The form of the unary plus operator is

```
+ expression
```

The *expression* can be any arithmetic type. The integral promotions are performed on the operand. The result has the promoted type and is not an lvalue.

The unary plus expression `+E` is the same as the expression `(0 + E)`. The two expressions return the same result.

See Also

Refer to "Pointer Arithmetic" in Section 2 for more information.

Assignment Operators

The two classifications of assignment operators are

- Simple assignment operator
- Compound assignment operators

An assignment expression contains a left and right operand separated by an assignment operator. The left operand must be a modifiable lvalue. The type of an assignment expression is the type of the left operand. The value of the assignment expression is the value of the left operand after the assignment. The result of an assignment expression is never an lvalue.

The assignment operator modifies the object designated by the left operand by storing a new value into it. The compound operators differ in how they compute the new value to be stored.

The assignment operators are all at the same level of precedence. However, unlike all other binary operators in the C language, the assignment operators associate right to left.

Examples

These examples illustrate right associativity.

The following two assignment expressions are equivalent:

```
i = j = 0;

i = ( j = 0 );
```

The following assignment expression is illegal:

```
( i = j ) = 0;
```

Simple Assignment Operator

The simple assignment operator (=) indicates simple assignment. In simple assignment, the value of the expression replaces the value of the object referred to by the lvalue.

Both of the operands must have arithmetic type or both must have the same structure, union, or pointer type. In addition, if either operand is an object pointer, the other operand can have type "pointer to `void`." Also, if the left operand is a pointer, the right operand can be a null pointer constant.

If both operands have arithmetic type, the right operand is converted to the type of the left operand before the assignment. If the left operand is a pointer and the right operand is an integral constant expression with the value 0, the constant is converted into a null pointer.

The type of the result is equal to the unconverted type of the left operand. The result is the value stored into the left operand. The result is not an lvalue.

If an object is assigned to another object that overlaps in storage with any part of the object being assigned, the behavior is undefined.

Given the following program fragment:

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

The `int` value returned by the function is converted to a `char` by truncation, then stored in `c`, and finally converted back to `int` prior to the comparison. In this C language implementation, a plain `char` behaves the same as an `unsigned char`; the result of the conversion cannot be negative. The operands of the comparison can never compare as equal. Therefore, for full portability, the variable `c` should be declared as `int`.

See Also

Refer to "Pointer Types," "Structure Types," and "Union Types" in Section 2 for more information.

Compound Assignment Operators

The compound assignment operators indicate compound assignment. The compound assignment operators are

```
+=      -=      *=      /=      %=

<<=     >>=     &=      ^=      |=
```

The following compound assignment expressions are the same. However, `E1` in the second expression is evaluated only once.

```
E1 op = E2

E1 = E1 op (E2)
```

Table 5–3 lists the allowable types for the compound assignment operators.

Table 5–3. Type of Operands for Compound Assignments

Operator	Operand Type
+= -=	Arithmetic †
*= /= %=	Arithmetic
<<= >>= &= ^= |=	Integral

† Also, the left operand can be a pointer and the right operand can be an integral type.

The type of the result is equal to the (unconverted) type of the left operand. The result is the value stored into the left operand. The result is not an lvalue.

See Also

Refer to "Pointer Arithmetic" and "Integer Types" in Section 2 for more information.

Bitwise Operators

The following bitwise operators are discussed:

- The bitwise negation operator
- The bitwise AND operator
- The bitwise exclusive OR operator
- The bitwise inclusive OR operator
- The bitwise left shift operator
- The bitwise right shift operator

Bitwise Negation Operator

The result of the bitwise negation 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. For `signed` types, the sign bit and all 39 bits for the magnitude are negated; for `unsigned` types, only the 39 bits for the magnitude are negated.

An expression associated with the bitwise negation operator must have an integral type. The usual arithmetic conversions are performed on the operand. The result has the promoted type and is not an lvalue.

See Also

- Refer to "Integer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Bitwise AND Operator

The binary bitwise AND operator (&) performs a bitwise AND.

Both operands must have integral type. The usual arithmetic conversions are performed on the operands. The type of the result is the type of the converted operands. The result is not an lvalue.

The result of the binary bitwise AND operator (&) is the bitwise AND of the operands. Each bit in the result is set if and only if each of the corresponding bits in the converted operands is set. That is, if both bits are 1, the bit in the result is a 1. Otherwise, the bit in the result is a 0. For `signed` types, the sign bit and the 39 bits for the magnitude are affected; for `unsigned` types, only the 39 bits for the magnitude are used.

The binary bitwise AND operator (&) is commutative and associative. The compiler may regroup an expression involving several binary bitwise AND operations at the same level.

See Also

- Refer to "Integer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Bitwise Exclusive OR Operator

The binary exclusive OR operator (^) performs a bitwise exclusive OR.

Both operands must have integral type. The usual arithmetic conversions are performed on the operands. The type of the result is the type of the converted operands. The result is not an lvalue.

The result of the binary exclusive OR operator (^) is the bitwise exclusive OR of the operands. Each bit in the result is set if, and only if, exactly one of the corresponding bits in the converted operands is set. That is, if exactly one bit is a 1, the bit in the result is a 1. If both bits are 1 or if both bits are 0, the result bit is a 0. For the `signed` types, the sign bit and the 39 bits for the magnitude are affected; for `unsigned` types, only the 39 bits for the magnitude are used.

The binary bitwise exclusive OR operator (^) is commutative and associative. The compiler may regroup an expression involving several binary exclusive OR operations at the same level.

See Also

- Refer to "Integer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Bitwise Inclusive OR Operator

The binary bitwise inclusive OR operator (|) performs a bitwise inclusive OR.

Both operands must have integral type. The usual arithmetic conversions are performed on the operands. The type of the result is the type of the converted operands. The result is not an lvalue.

The result of the binary bitwise inclusive OR operator (|) is the bitwise inclusive OR of the operands. Each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set. That is, if one of the bits is a 1, the bit in the result is a 1. If both bits are 0, the result bit is a 0. For `signed` types, the sign and the 39 bits for the magnitude are affected; for `unsigned` types, only the 39 bits for the magnitude are used.

The binary bitwise inclusive OR operator (|) is commutative and associative. The compiler may regroup an expression involving several binary bitwise inclusive OR operations at the same level.

See Also

- Refer to "Integer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Bitwise Left and Right Shift Operators

The two bitwise shift operators are

<< left shift
>> right shift

The first operand is the quantity to be shifted. The second operand is the number of bits that the first operand is shifted. The result of `E1 << E2` is that `E1` (interpreted as a bit pattern) is left-shifted `E2` bit positions. Vacated bits are zero-filled. Only the 39 bits for the magnitude are affected; the sign bit is not changed.

The result of the expression `E1 >> E2` is that `E1` (interpreted as a bit pattern) is right-shifted `E2` positions. If `E1` is a signed or unsigned type, the right shift is logical (zero-filled). Only the 39 bits for the magnitude are affected; the sign bit is not changed.

Both operands must have an integral type. The usual binary conversions are not performed on shift operators, but the integral promotions are performed separately on each operand. After the unary conversions are performed, the right operand is converted to `int`. The type of the result is that of the promoted left operand. If the right operand is negative or greater than or equal to the width in bits of the promoted left operand, the result is undefined.

The result is not an lvalue.

The two bitwise shift operators have the same precedence and are left-associative.

See Also

Refer to "Integer Types," "Character Type," and "Enumeration Types" in Section 2 for more information.

Equality Operators

The two equality operators are

- == equal to
- != not equal to

The types of the operands can be as follows:

Equality Operand Type	Description
Arithmetic	Both operands can be an arithmetic type.
Pointers	Both operands can be the same pointer type.

Equality Operand Type	Description
Pointer and pointer to `void`	One operand can be an object pointer and the other can be a pointer to `void`.
Pointer and null pointer constant	One operand can be a pointer and the other operand can be a null pointer constant.

For integral operands, integer comparison is performed. For floating-point operands, floating-point comparison is performed. Pointer operands are considered equal if they point to the same object or if they are both null. A pointer is equal to the integer constant 0 if, and only if, it is a null pointer.

The result of the equal to operator (==) and the not equal to operator (`!`=) is 1 if the specified relation is true. The result is 0 if the specified relation is false. The result has type `int` and is not an lvalue.

The binary equality operators have a lower precedence than the relational operators. But like the relational operators, they are left-associative.

Note: *Do not confuse the = operator with the == operator. The = operator performs simple assignment. The == operator performs equality comparisons.*

See Also

Refer to "Pointer Arithmetic," "Enumeration Types," and "Pointer Types" in Section 2 for more information.

Logical Operators

The following logical operators are discussed:

- The logical negation operator
- The logical AND operator
- The logical OR operator

Logical Negation Operator

The result of the logical negation operator (`!`) is 0 if the value of its operand is nonzero and 1 if the value of its operand is 0.

An expression associated with the logical negation operator must have a scalar type. The usual arithmetic conversions are performed on the operand. The result has type `int` and is not an lvalue. The expression `!E` is the same as the expression `(E==0)`.

See Also

- Refer to Section 2, "Types," for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Logical AND Operator

The logical AND operator (&&) performs a logical AND operation.

Both operands must have scalar type. The type of the result is `int` and the value is zero or one. The result is not an lvalue.

The result of the logical AND operator is one if both of the operands evaluate to nonzero. The left operand is evaluated first. If the left operand is zero, the right operand is not evaluated, and the result is zero. If the left operand does not equal zero, the right operand is evaluated. If the right operand is equal to zero, the result is zero. If the right operand is not equal to zero, the result is one.

Unlike the binary bitwise AND operator (&), the logical AND operator (&&) guarantees left-to-right evaluation and is a sequence point.

See Also

Refer to Section 2, "Types," for more information.

Logical OR Operator

The logical OR operator (||) performs a logical OR operation.

Both operands must have scalar types. The type of the result is `int` and the value is zero or one. The result is not an lvalue.

The result of the logical OR operator is one if either of the operands evaluates to nonzero. The left operand is evaluated first. If the left operand is not equal to zero, the right operand is not evaluated, and the result is one. If the left operand does equal zero, the right operand is evaluated. If the right operand is not equal to zero, the result is one. If the right operand is equal to zero, the result is zero.

Unlike the binary bitwise inclusive OR operation (|), the logical OR operator (||) guarantees left-to-right evaluation and is a sequence point.

See Also

Refer to Section 2, "Types," for more information.

Relational Operators

The four relational operators are

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

Both operands can have an arithmetic type or the same pointer type. If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

For integral operands, integer comparison is performed. For floating-point operands, floating-point comparison is performed. For pointer operands, the result depends on the relative locations in the address space of the objects pointed to. If the objects pointed to are not members of the same aggregate object, the result is undefined. However, if `P` points to the last member of an array object, the pointer expression `P + 1` is greater than `P`, even though `P + 1` does not point to a member of the same array object as `P`.

The result of each of the less than operator (`<`), the greater than operator (`>`), the less than or equal to operator (`<=`), and the greater than or equal to operator (`>=`) is 1 if the specified relation is true. The result is 0 if the specified relation is false. The result has type `int` and is not an lvalue.

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

The binary inequality operators all have the same precedence and are left-associative.

See Also

- Refer to "Pointer Arithmetic" and "Pointer Types" in Section 2 for more information.
- Refer to "Usual Arithmetic Conversions" in Section 4 for more information.

Miscellaneous Operators

The following operators are discussed:

- The function call operator
- The type cast operator
- The conditional operator
- The comma operator

Function Call Operator

The function call operator, `function()`, calls a function, passing it an argument list. The function designator *function*, must be of type "pointer to function returning *type*."

Whenever an identifier or expression of type "function returning *type*" occurs in the context of a function call, it is converted to type "pointer to function returning *type*." (Refer to "Function to Pointer" in Section 4). This conversion allows any number of indirection operators to be placed before the function locator. For example, the following are legal for either a pointer to a function or a function identifier:

```
x(y)
(*x)(y)
(****x)(y)
```

If a declaration is not in scope for an identifier that is first used as the expression in a function call, the identifier is implicitly declared exactly as if the following declaration appeared in the innermost block containing the function call:

```
extern  int x () ;
```

That is, the function is implicitly declared with external linkage and has no information about its arguments and has type returning `int`.

A function definition can serve as a declaration if and only if the definition appears in the source file before any calls to that function.

If the type of the function expression is "pointer to function returning *type*," the result of the function call is of type *type*. The result of a function call is not an lvalue. If *type* is `void`, the function call does not produce a result. Such a function cannot be used in a context that requires the call to yield a result. A function can return a value of any type of object, except array or function.

If a function prototype declarator is in scope, the actual arguments are compared with the formal arguments and are converted accordingly. An argument that has array or function type is converted to a pointer. The ellipsis notation in a function prototype declarator causes argument type checking to stop after the last declared formal argument. Unchecked arguments are converted as if the function prototype declarator were not in scope.

If a function prototype declarator is not in scope, the compiler passes the following:

- Arrays, functions, and pointers as pointers
- `float` as `double`
- All character and short types as `int`
- Everything else as its own type

If the number of actual arguments or their types after conversion do not agree with those of the formal arguments in the function definition, the behavior is undefined.

If a function prototype declarator is not in scope, the compiler may issue a diagnostic message if the number of actual arguments does not match the number of formal arguments of the function being called. The compiler also may issue a diagnostic message or take any special action if the (converted) type of an actual argument does not match the (promoted) type of the corresponding formal argument.

An argument can be any expression other than a `void` expression. In preparing for the call to a function, each actual argument is evaluated and each formal argument is assigned the value of the corresponding actual argument. If the number of actual arguments or their types after conversion do not agree with those of the formal arguments, the behavior is undefined.

A copy is made of each actual argument after it has been evaluated and converted. All arguments are passed by value. Within the called function, the names of the formal arguments are lvalues. A function can change the values of its formal arguments.

Changing a formal argument only changes the value of the formal argument and does not affect the actual argument.

When a pointer is passed as an argument, a copy is made of the pointer itself. A copy is not made of the object the pointer is pointing to. By using pointers, a called function can modify an object supplied by the caller.

Do not mix old style declarations and definitions with prototype declarations and definitions of the same functions.

Declare a function using the old style and define the function with the function prototype format if, and only if, the types of the actual arguments after promotion agree with the types of the formal arguments in the function definition.

Declare a function using the function prototype format and define the function using the old style format if, and only if, the function prototype declares the formal arguments to be the same type as used by the default argument promotions, after the promotions occur.

Except in the cases just described, a function prototype must be in scope of the function call if the function is defined using the function prototype format. Otherwise, the behavior is undefined.

All declarations in the same scope of a particular function declare the identical return type. Each argument type list agrees in the number and types of the arguments and in the use of the ellipsis. The argument type checking includes agreement on the number of dimensions of arrays and on the bounds in each dimension, except the first.

The C language allows recursive function calls, either directly or indirectly, through a chain of other functions.

See Also

- Refer to “Function Types” in Section 2 for information on functions.
- Refer to “Function Declarators” in Section 3 for information on declaring objects of function types.
- Refer to “Function Prototype Format” and “Old Style Format” in Section 7 for information defining functions.

Cast Operator

The result of the cast operator is the conversion of the expression to the specified type name.

Syntax

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

If the *type-name* specifies type `void`, the expression can be any type, including `void`. Otherwise, the *type name* must specify scalar type and the expression must have scalar type.

A `cast-expression` can invoke any permissible conversion. The result of a cast expression is not an lvalue.

Conversions involving pointers (other than a pointer to `void` to or from a pointer to an object) must be specified by means of an explicit cast and have the following aspects:

- A pointer can be converted to an integer. This is not portable and should be avoided whenever possible.
- An arbitrary integer can be converted to a pointer. This is not portable and should be avoided whenever possible.
- A pointer to an object of one type can be converted to a pointer to an object of another type. The resulting pointer might not be valid if it is improperly aligned for the type of object pointed to. It is guaranteed that a pointer to an object of a given alignment can be converted to a pointer to an object of a less strict alignment and back again without change. (An object that has type `char` has the least strict alignment.)
- A pointer to a function of one type (including parameter-type information, if any) can be converted to a pointer to a function of another type.

See Also

- Refer to "Void Types" in Section 2 for information on `void` types.
- Refer to "Type Names" in Section 3 for more information.
- Refer to Section 4, "Type Conversions" for information on conversions.

Conditional Expression Operator

Unlike all other operators in the C language, which are either unary or binary, the conditional expression operator is a ternary operator. That is, it takes three operands. A conditional expression uses the question mark (?) to separate the first and second operands and the colon (:) to separate the second and third operands.

In the following conditional expression, the first operand x determines which of the other two operands is evaluated:

```
x ? y : z
```

The first operand is evaluated first. If the result of the evaluation of the first operand is true (not equal to zero), the second operand, y, is evaluated. The result of the evaluation of the second operand becomes the result of the operation. If the result of the evaluation of the first operand is false (equal to zero), the third operand, z, is evaluated. The result of the evaluation of the third operand becomes the result of the operation. A conditional expression never yields an lvalue.

The first operand must have scalar type. The type of the second and third operands can be one of the following:

- Arithmetic

  Both the second and the third operands can have an arithmetic type. The usual arithmetic conversions are performed on the second and third operands to bring them to a common type. The type of the result is this common type.

- Pointers

  Both the second and third operands can be pointers of the same type. The result is a pointer of the same type.

  One operand can be a pointer (after the usual unary conversions) and the other operand can be a null pointer constant. The result has the type of the pointer.

  If one operand is a pointer to `void`, the other operand is converted to that type. The result is a pointer to `void`.

- Structure, union

  The second and third operands can have the identical structure or union type. The result has the same type as the operands.

- `void`

  Both operands can be `void` expressions. The result is a `void` expression.

  The conditional expression operator associates from right to left. If an expression contains multiple question mark operators (?) , the compiler groups the expressions from right to left. For example, given the following conditional expression:

```
E1 ? E2 : E3 ? E4 : E5
```

This expression is evaluated as follows:

```
E1 ? E2 : ( E3 ? E4 : E5)
```

Because the question mark (**?**) and the colon (**:**) operators bracket the second operand, it can contain any expression and use operators that have lower precedence. However, the first and third operands are not bracketed and therefore cannot contain operators of lower precedence without the use of parentheses.

See Also

Refer to "Pointer Types," "Structure Types," "Union Types," and "Void Type" in Section 2 for more information.

Comma Operator

A comma operator (**,**) is used in comma or sequential expressions to separate a pair of expressions.

The compiler evaluates the pair of expressions separated by a comma left to right. The left operand is evaluated first and all side effects take place. If evaluation of the left operand produces a value, that value is discarded. The type and value of the result of the comma expression are the type and value of the right operand. The result never yields an lvalue.

The following example illustrates the comma operator:

```
if (exchange) temp=a, a=b, b=temp;
```

The comma operand is associative. A single expression can contain any number of comma operators. The compiler evaluates the subexpressions in left to right order. The value of the last subexpression is the value of the entire expression.

In some cases, the comma is used as a punctuator. For example, the comma is a punctuator in lists of actual arguments to functions and in lists of initializers. In general, any place where a comma can be used as a punctuator, the comma is interpreted as a punctuator and not as a comma operator. For example, the following function call is always treated as a call to a function with four arguments:

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

However, to treat the second comma as a comma operator instead of a punctuator, insert the appropriate parentheses as shown:

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

The function now has three arguments. The value of the second argument is 5.

See Also

Refer to "Operators" in Section 1 for more information.

Section 6
Statements

Statements perform actions and determine the flow of control. Unless directed by a particular statement, statements are executed in sequence. Except for a compound statement, statements generally end with a semicolon. To treat any sequence of statements as a single statement, place braces around the sequence.

Note: *In the C language, the semicolon is not a statement separator; it is part of the syntax of some statements.*

The following types of statements are described in this section:

- Labeled statements
- Compound statements
- Expression statements
- Null statements
- Control statements
- Iteration statements
- Jump statements

The following keywords indicate statements:

- `break`
- `continue`
- `do`
- `for`
- `goto`
- `if`
- `return`
- `switch`
- `while`

In the C language, the "control" expressions that appear in conditional or iterative statements are enclosed in parentheses.

See Also

Refer to Section 5, "Expressions and Operators," for more information.

Statements Syntax

The syntax for statements is as follows:

```
statement:
        compound-statement
        control-statement
        expression-statement
        iteration-statement
        jump-statement
        labeled-statement
```

The syntax for the alternative definitions of the statements syntax (compound, control, expression, iteration, jump, labeled, and null) is described under the discussion of each type of statement.

Labeled Statements

A label can precede any statement. The label marks the statement so that a `goto` or `switch` statement can transfer control to that statement.

Syntax

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

A label never appears by itself. It is always attached to a statement. If a label is needed where there is no statement, insert a null statement and attach the label to it.

The C language contains three kinds of labels:

Label	Description
Named label	A named label is an identifier followed by a colon. A name label can appear on any statement. The label is referenced with the `goto` statement.
`case` label	The `case` label can appear only on a statement within a `switch` statement. It consists of the keyword `case` followed by a constant integer expression followed by a colon.
`default` label	The `default` label can appear only on a statement within a `switch` statement. It consists of the keyword `default` followed by a colon.

Examples

This example illustrates the use of a named label:

```
try_again:
    i++;
    if (a[i] !=0)
        goto try_again;
```

This example illustrates a null statement with a label attached to it:

```
nomansland:   ;
```

Compound Statements

A compound statement consists of an optional sequence of declarations followed by a sequence of statements. The declarations and statement sequence are enclosed in braces ({}).

Syntax

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

statement-list:
statement
statement-list statement

declaration-list:
declaration
declaration-list declaration

A compound statement can appear anywhere a statement can appear.

Statements within a compound statement are executed one at a time and in sequence. Execution of the compound statement stops after execution of the last statement in the compound statement or when control is transferred out of the compound statement by execution of a `goto`, `return`, `continue`, or `break` statement.

A `goto` or `switch` statement can jump to a labeled statement within a compound statement. Execution begins at the labeled statement and continues to the end of the compound statement or until a `goto`, `return`, `continue`, or `break` statement transfers control out of the compound statement.

The compound statement can have its own set of declarations. Without declarations, the compound statement is just a group of statements. With declarations, the compound statement creates a new scope. If an identifier is declared at the beginning of a compound statement, its scope extends from its declaration to the end of the compound statement. It is visible throughout that scope, except when hidden by a declaration defining a different entity with the same name in an embedded compound statement.

If . . .	Then . . .
An identifier is declared at the beginning of a compound statement without a storage class specifier and with the type "function returning *type*"	It is assumed to have storage class `extern`. In all other cases, the identifier is assumed to have storage class `auto`.
A variable or function is declared in a compound statement with storage class `extern`	The compiler does not allocate any storage. Also, initialization is not allowed. The declaration refers to an external variable or function defined either in another source file or in a different location in the same source file.
A variable in a compound statement is declared with the storage class `auto` or `register`	Storage is allocated every time the compound statement is entered and deallocated when control exits the compound statement. If an initialization is supplied for these variables, it is performed on each normal entry of the compound statement. The initialization does not occur if the compound statement is entered by execution of a `goto` or `switch` statement to a label within the statement.
A variable in a compound statement is declared with the storage class `static`	The variable is allocated storage only once, before the program begins execution. Any initialization expression is also evaluated only once, before the program begins execution. The variable retains its value from one execution of the compound statement to the next.

See Also

Refer to "Storage Class Specifiers" in Section 3 for more information.

Expression Statements

An expression statement consists of an expression followed by a semicolon.

Syntax

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

An expression statement is evaluated for its side effects such as assignments and function calls. The compiler does not necessarily retain an expression or part of an expression if it does not have a side effect.

Example

```
++y;
printf("Hi there");
x = y + z;
```

See Also

Refer to "Function Call Operator" and "Side Effects" in Section 5 for more information.

Null Statements

A null statement consists of a semicolon only. It has no expression and performs no operations. It only passes control to the next statement. Use a null statement where a statement must be written, but no action is to be performed.

Examples

One use for a null statement is to supply an empty loop body for an iteration statement (`while`, `do`, or `for`):

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

A null statement can also be used to carry a named label just before the closing brace (}) of a compound statement:

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

Control Statements

The control statements perform conditional executions when a particular condition is met. The following control statements are discussed:

- `if` statement
- `switch` statement

The syntax for control statements is as follows:

Syntax

```
control-statement:
       if ( expression ) statement
       if ( expression ) statement else statement
       switch ( expression ) statement
```

if Statement

The `if` statement enables conditional execution of program statements.

For both syntax definitions of the `if` statement, *`expression`* within the parentheses is evaluated first. If the value of *`expression`* is nonzero, the *`statement`* is executed. If the value of *`expression`* is zero and there is an `else` clause, the second `statement` is executed. If the value of *`expression`* is zero, but there is no `else` clause, execution continues with the statement following the conditional statement. The *`expression`* must have a scalar type.

Figure 6-1 illustrates the logic of an `if` statement.

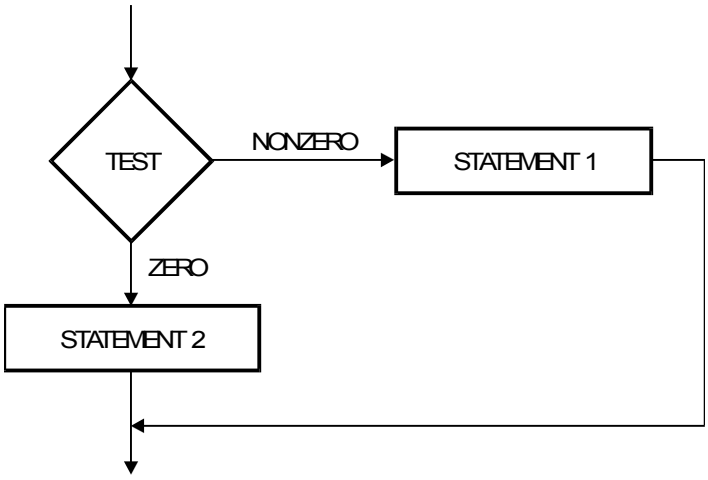


Figure 6–1. if Statement

In *`expression`*, be sure not to confuse the equality operator (`==`) with the assignment operator (`=`).

In some nested `if` statements, it might not be apparent to which `if` statement an `else` belongs. The C compiler always assumes an `else` belongs to the innermost `if` statement; that is, the last `if` statement the compiler saw that does not yet have an `else`.

Example

The following example illustrates the use of the `if` statement:

```
if(a > 0 && a <= 3)
     range = low;
else if (a >= 4 && a <= 7)
     range = middle;
else if (a >= 8 && a <= 10)
     range = high;
else { print_range_error(a);
       return;
     }
```

See Also

- Refer to Section 2, “Types,” for information on scalar types.
- Refer to “Equality Operators” and “Assignment Operators” in Section 5 for more information.

switch Statement

A `switch` statement causes control to jump to none or one of several statements depending on the value of a controlling expression.

A `labeled` statement can be used to mark a statement so that a `switch` statement can transfer control to that statement. The following example shows a typical use of `case` and `default` labels within the body of a `switch` statement:

```
switch ( integer-expression )
      {
      case constant1 :         statement(s)
                                 break;
      case constant2 :         statement(s)
                                 break;
      case constant3 :         statement(s)
                                 break;
      case constantn :         statement(s)
                                 break;
      default :                 statement(s)
      }
```

The `case` and `default` labels must satisfy the following rules:

- The *integer-expression* must have integral type. The integral promotions are performed on *integer-expression*.
- Any *constant* expressions of a `case` label must have an integral type. The compiler evaluates and converts each `case` label constant expression to the same type as the converted *integer-expression*.
- After evaluation and conversion, no two of the `case` constants in the same `switch` statement can have the same value.
- A `default` label can appear anywhere within the `switch` statement. It does not have to appear at the end.
- At most, a `switch` statement can contain only one `default` label.
- The `case` and `default` labels can appear only within a `switch` statement.

The `switch` statement is executed in the following order:

Step	Description
1	The *integer-expression* is evaluated.
2	The *integer-expression* is compared against the *constant* expressions associated with each `case` label.

Step	Description
3	If a match is found, execution begins at the first executable statement following the matching `case` label. Execution continues until a `break` statement or the end of the `switch` statement is found. If a `break` statement does not separate one `case` from the following `case`, execution of the first `case` continues into the next `case` until a `break` statement or the end of the `switch` statement is encountered. If a `break` statement is encountered, execution continues with the first statement after the `switch` statement. All other `case` and `default` labels are ignored.
4	If no match is found, but the `switch` statement contains a `default` label, the statements associated with the `default` label are executed. Execution continues until a `break` statement or the end of the compound statement is found. If no `break` statement is encountered, execution continues into the case following the default case. If a `break` statement is found, execution continues with the first statement following the `switch` statement. All other `case` labels are ignored.
5	If no match is found and the `switch` statement does not contain a `default` label, the body of the `switch` statement is skipped. Execution continues with the first statement following the `switch` statement.

Example

The following example illustrates the use of the `switch` statement:

```
switch(a)
   {  case 1:
      case 2:
      case 3:
           range = low;
           break;
      case 4:
      case 5:
      case 6:
      case 7:
           range = middle;
           break;
      case 8:
      case 9:
      case 10:
           range = high;
           break;
      default:
           print_range_error(a);
           return;
   }
```

See Also

Refer to “Integer Types” in Section 2 for more information.

Iteration Statements

Iteration statements cause a statement called the loop body to be executed repeatedly depending on a controlling expression. The C language contains three iteration statements:

Iteration Statement	Description
`while` statement	The `while` statement tests for the exit condition *before* each execution of a statement.
`do` statement	The `do` statement tests for the exit condition *after* each execution of a statement.
`for` statement	The `for` statement initializes and updates one or more controlling variables and tests for a continuation condition.

The syntax for iteration statements is a follows:

Syntax

iteration-statement:

`while (` *expression* `)` *statement*

`do` *statement* `while (` *expression* `) ;`

`for (` $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *statement*

while Statement

The `while` statement provides a way for executing the loop body after a test of the exit condition.

Figure 6-2 illustrates the logic of a `while` statement.

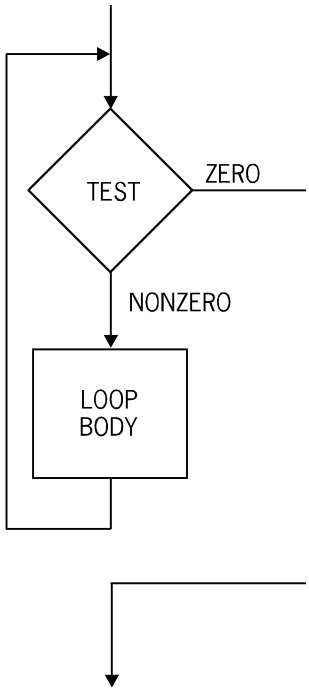


Figure 6–2. while Statement

The `while` statement is executed in the following order:

Step	Description
1	The *`expression`* is evaluated; it must have scalar type.
2	If *`expression`* results in a true condition (it evaluates to a nonzero value), *`statement`* (loop body) is executed. Steps 1 and 2 are repeated.
3	If *`expression`* evaluates to zero, execution of the `while` statement is complete. Execution of the `while` statement is also terminated when control is transferred out of the loop body by a `goto`, `return`, or `break` statement.

Note: The `continue` statement terminates the current iteration of the innermost *`while`*, *`do`*, or *`for`* statement that contains the *`continue`* statement.

The `while` statement tests the exit condition before executing *`statement`*. Therefore, *`statement`* might never be executed.

Example

The following example illustrates the use of the `while` statement:

```
char *p, s[80];
int i=0;
while(*p != '\0') s[i++] = *p++;
     /* copies a character string */
```

do Statement

The `do` statement provides a way to execute a loop body and then test for the exit condition.

Figure 6-3 illustrates the logic of a `do` statement.

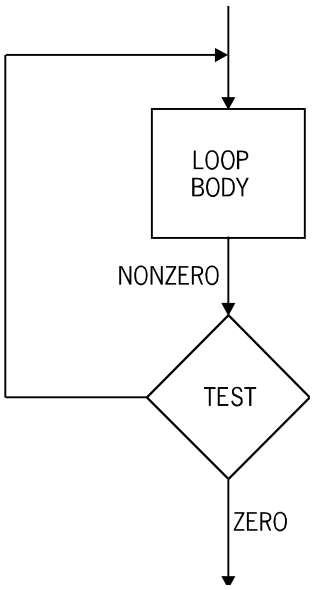


Figure 6–3. do Statement

The `do` statement is executed in the following order:

Step	Description
1	The *`statement`* (loop body) is executed.
2	The *`expression`* is evaluated; it must have scalar type.
3	If the value of *`expression`* is nonzero, Steps 1 and 2 are repeated.
4	If the value of *`expression`* is zero, the execution of the `do` statement is complete. Control can also transfer out of the loop body by a `goto`, `return`, or `break` statement.

Note: *The `continue` statement terminates the current iteration of the innermost `do`, `while`, or `for` statement that contains the `continue` statement.*

The `do` statement differs from the `while` statement in that the `do` statement always executes the loop body at least once. The `while` statement might never execute its loop body.

A loop body must be present, although it can be a `null` statement.

Within the *`expression`*, be sure not to confuse the equal-to operator (==) with the simple assignment operator (=).

Examples

The following examples illustrate the execution of the `do` statement:

```
char *p, c[132];
p = c;
do;  /* null statement */
while((*p++ = getchar()) != '\n');
      /* reads input line into array */
      /* one character at a time */
      /* all work done in condition statement */

do
   putchar ('_');
while (count ++ < graphcount);
```

See Also

Refer to “Equality Operators” and “Assignment Operators” in Section 5 for more information.

for Statement

The `for` statement provides a way for executing a loop body after testing for a continuation condition. The `for` statement also allows for the initialization of variables, modification of variables, and the continuation condition.

Figure 6-4 illustrates the logic of the `for` statement. The three optional expression are referred to as *`init-exp`*, *`test-exp`*, and *`modify-exp`* respectively. The following *`statement`* is called the loop body.

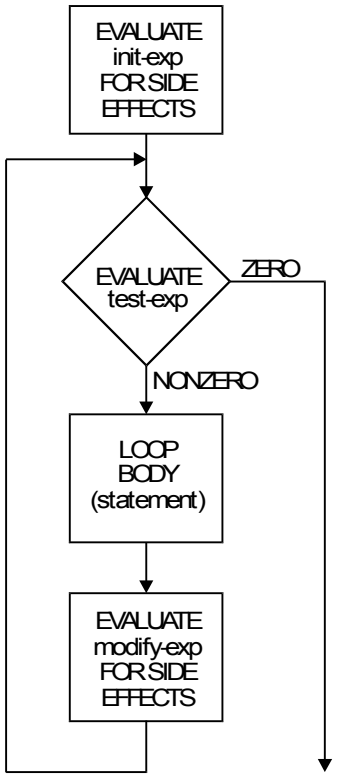


Figure 6–4. for Statement

The `for` statement is executed in the following order:

Step	Description
1	If present, *`init-exp`* is evaluated; it may have any type.
2	If present, *`test-exp`* is evaluated; it must have scalar type. ***Note:*** *If `test-exp` is not present, it is treated as being true. An infinite loop occurs unless a `return`, `break`, or `goto` statement is used within the loop.*
3	If *`text-exp`* is false (evaluates to zero), the loop body is terminated and the execution of the `for` statement is complete. If *`test-exp`* is true (evaluates to nonzero), the loop body is executed and *`modify-exp`*, if present, is evaluated. *`Modify-exp`* may have any type.
4	Control returns to Step 2.
5	Execution of the `for` statement ends when *`test-exp`* evaluates to zero or when control is transferred out of the `for` statement by a `goto`, `return`, or `break` statement. ***Note:*** *The `continue` statement terminates the current iteration of the innermost `while`, `do`, or `for` statement that contains the `continue` statement. `Modify-exp` is executed and control returns to Step 2.*

Each of the three expressions within the parentheses is optional. However, the semicolons and parentheses are required even if the expressions are absent.

Because the `for` statement tests before executing the loop body, the loop body might never be executed.

The *init-exp*, *test-exp*, and *modify-exp* can each consist of multiple expressions separated by the comma operator.

Example

The following example illustrates the use of the `for` statement:

```
short *sp, s[10];
double *dp, d[10];
for(sp = s, dp = d; *sp != 0; sp++, dp++)
    *dp = *sp;
         /* copies values from array of short to array of double */
         /* until a zero is found in array of short */
```

The `for` statement can be illustrated in terms of the `while` statement. The following two code sequences are equivalent:

`while` statement

```
x=0;
i=0;
while (i<10)
      { x = x + i;
        i++;
      }
```

`for` statement

```
x=0;
for (i=0; i<10; i++)
     x = x + i;
```

See Also

Refer to "Comma Operator" in Section 5 for more information.

Jump Statements

A jump statement causes an unconditional jump to another statement. The following keywords indicate jump statements:

- `break`
- `continue`
- `goto`
- `return`

The syntax for jump statements is as follows:

Syntax

jump-statement:
break ;
continue ;
goto *identifier* ;
return $expression_{opt}$;

break Statement

A break statement immediately terminates execution of the innermost do, for, switch, or while statement that contains the break statement. A break statement can appear only in the body of a switch statement or a loop body.

When a break statement is used in a for, do, or while statement, the entire loop is terminated. When a continue statement is used in a for, do, or while statement, only the current iteration of the loop body is terminated.

A break statement is the same as a goto statement that branches to an appropriately labeled null statement immediately following the terminated do, for, switch, or while statement.

Example

The following example illustrates the use of the break statement:

```
struct t *pst;
while (pst -> next)
     {
          if (pst -> tag == error_successor)
              break;
          pst = pst -> next;
     }

/* pst points to last valid structure */
```

continue Statement

A continue statement immediately terminates the current iteration of the innermost do, for, or while statement that contains the continue statement. Execution resumes at the test *expression* in the while and do statements and at the *modify-exp* in the for statement. A continue can appear only in a loop body.

When a break statement is used in a for, do, or while statement, the entire statement is terminated. When a continue statement is used in a for, do, or while statement, only the current iteration of the loop body is terminated.

Example

The following example illustrates the use of the continue statement:

```
for (i = first; i == last; i++);
     {
          if (data[i].status == dont_touch)
              continue;
          .
          .
          .
     }
```

goto Statement

A `goto` statement transfers control from any statement in a function to any other statement in that function.

Execution of the `goto` statement forces an immediate transfer of control to the point in the function indicated by the *`identifier`*. The statement labeled with *`identifier`* is executed next.

Since labels have function scope, a `goto` can transfer control only to a statement within the same function.

The use of a `goto` statement should be avoided. In general, the use of `goto` is considered a poor programming practice that makes programs difficult to understand and maintain.

The branching that a `goto` statement performs can hinder compiler optimizations. To write a C program that executes as fast as possible, limit the number of labels. Any label, whether explicit, named, or implicit, can inhibit compiler optimizations and slow down a C program.

The following guidelines are intended to provide a clear use of the `goto` statement and reduce any confusion that `goto` branching can cause:

- Do not branch into the `if` or `else` clause of an `if` or `if-else` statement from outside the clause.
- Do not branch from the `if` clause to the `else` clause or from the `else` clause to the `if` clause.
- Do not branch into the body of a `switch` or iteration statement from outside the statement.
- Do not branch into a compound statement from outside the statement.
- Use `break`, `continue`, and `return` statements in preference to a `goto` statement.

Example

The following example illustrates the use of the `goto` statement:

```
for (i = 1; i < count; i++)
     if(a[i] == lookup) goto done;
printf("item not found");
done:;
```

See Also

Refer to "Scope" in Section 1 for more information.

return Statement

The `return` statement terminates execution of the current function and returns control to its caller.

Execution of a `return` statement forces execution of the current function to terminate. Control is transferred to the caller of the function and the next statement executed is the statement following the function call.

The *`expression`* is optional. If the `return` statement does not contain an *`expression`*, the function does not return a value. If the context of the function requires that a value be returned, the value of the function is undefined. The returned value must be compatible with the type returned by the function.

A `return` statement without an associated value is automatically issued for functions that "drop off the end" of their definition (that is, they reach the closing brace that terminates the function).

Example

The following example illustrates the use of the `return` statement:

```
double sine(double x)
      {
           if(fabs(x) < epsilon)
                  return x - (x * x * x) / 6.0;
           else
              {
                x = sine (x/3.0);
                return x * (3.0 - 4.0 * x * x);
              }
      }
```

See Also

- Refer to "Function Types" in Section 2 for more information on functions.
- Refer to "Function Call Operator" in Section 5 for more information on functions.
- Refer to "Defining Functions" and "Return Values" in Section 7 for more information on functions.

Section 7
Functions

A function is an invoked routine called by another function. A function can be called as many times as required. Each time a function is called the statements within a function are executed.

This section discusses the following concepts of functions:

- Defining functions
- Passing arguments
- Returning values

Defining Functions

When defining a function, supply the following information:

- The name of the function
- The type and number of formal arguments
- A storage class specifier
- The statements to be executed when the function is called
- The type of the value returned by the function, if any

Function definitions have the following two formats:

- Old style format
- Function prototype format

Function Syntax

The following is the syntax for defining old style and function prototype formats:

```
function-definition:
        declaration-specifiers(opt) declarator function-body

function-body:
        declaration-list(opt) compound-statement
```

```
declaration-list:
        declaration
        declaration-list declaration
```

The function body consists of optional local declarations and definitions, statements to be executed when the function is called, and an optional return statement.

See Also

- Refer to "Function Types" in Section 2 for more information on functions.
- Refer to "Function Declarators" and "Storage Class Specifiers" in Section 3 for more information on declaration-specifiers.
- Refer to "Compound Statements" in Section 6 for information on compound statements.
- Refer to "External Definitions" in Appendix G for more information on the syntax of function definitions.

Old Style Format

In this format, the formal arguments are declared in two parts:

1. The names of the arguments are listed within the parentheses after the function name.
2. The arguments are declared (given their attributes) in the argument section.

In the old style format, an empty argument list means that the function has no arguments. Otherwise, write the argument identifiers alone within the argument list. The following example illustrates a typical old style format for defining functions:

```
storage-class type name ( arg-name1 arg-name2, ... arg-namen)
storage-class type arg-name1;
storage-class type arg-name2;
 .
 .
 .
storage-class type arg-namen;
```

Declare the arguments in the argument section immediately following the declarator. Each argument must be declared at most once. Declare arguments only in the argument section. If an argument is not declared, the compiler implicitly declares it to have type `int`. The function body, enclosed in braces, terminates the argument level declarations.

Example

The following example illustrates the old style format of a function:

```
int fxd(x,y)
    int x;
```

```
    double y;
{ return x*y; }
```

For the old style definition and declaration, the compiler makes some adjustments in the type of function arguments according to the default argument promotions. These adjustments occur on the actual arguments when the function is called.

The adjustments to the type of the actual arguments for functions using the old style format are:

- Actual arguments of type `char` are implicitly promoted to be type `int`.
- Actual arguments of type `short` are implicitly promoted to be type `int`.
- Actual arguments of type `float` are implicitly promoted to be type `double`.

A formal argument declared to be of type "array of *type*" is treated as if it were declared to be type "pointer to *type*." Formal arguments of type "function returning *type*" are implicitly converted to type "pointer to function returning *type*."

Once the arguments are passed, they are then converted to the type of the formal arguments as specified in the parameter list.

When a function is defined, it can be declared to be either `extern` or `static`. The `extern` storage class specifier is the default. A function defined with the `static` storage class specifier can be called only from within the same file where the function appears. The names of functions defined with the `extern` storage class specifier, and functions that do not have a specified storage class specifier, are exported to the Binder and can be called by functions in other files.

Both the type specifier and the declarator determine the type of a function. If a type specifier is not specified, `int` is assumed.

The type of the return value can be any type, except "array of `type`" or "function returning `type`." In other words, functions cannot return arrays or other functions. But they can return pointers to arrays or functions. If a function is not intended to return a value, specify the type of the return value to be `void`.

See Also

Refer to "Function Declarators" and "Storage Class Specifiers" in Section 3 for more information.

Function Prototype Format

In the function prototype format, a function and the type of its arguments are declared in the argument list. Use the function prototype format when the compiler is to check the number and types of arguments, or to avoid the widening done by the default argument promotions. The following example illustrates a typical function prototype format for defining functions:

storage-class type name (type arg_1*, type* arg_2*, ... type* arg_n*)*

The benefit of the function prototype format is that the compiler converts the actual arguments to the type specified by the formal argument in the prototype. The compiler also checks for type compatibility and number. However, the compiler

cannot check for agreement of actual and formal arguments and perform the conversion unless the function prototype declaration is in scope at the point of the call.

The formal arguments are listed and declared in the function declarator. The scope of these formal argument identifiers ends with the end of the function body:

```
max(int *a, float b);
```

Formal argument declarations are similar to definitions of ordinary variables, except the only storage-class specifier allowed is `register`. The `register` specifier indicates that an argument is heavily used. A formal argument can be any type, except `void`.

To declare a function with a variable length argument list, write a function prototype whose argument list ends with a comma followed by ellipsis marks. For example:

```
int printf(char *fmt, ...)
```

At least one argument must be specified in the function prototype. On a function call, the compiler checks the types and number of all arguments that are specified in the function prototype. A function call with more arguments than specified in the function prototype is allowed, but not a function with fewer arguments than specified. All function calls must be in scope of such a function prototype. For undeclared arguments, the compiler adjusts the type of function arguments according to the default argument promotions. These adjustments occur on the actual arguments when the function is called, and are the same as the adjustments for functions that use the old style format. Refer to Volume 2, "Headers and Functions," for information about the header `<stdarg.h>` and referencing undeclared arguments.

A function prototype for a function `f` with no arguments is written as follows:

```
f(void)
```

Because array expression and function identifiers as arguments are always converted to pointers before the function call, a declaration of a formal argument as "array of *type*" is always adjusted to read "pointer to *type*." A declaration of a formal argument as "function returning *type*" is always adjusted to read "pointer to function returning *type*."

Once the arguments are passed, they are then converted to the type of the formal arguments as specified in the parameter list.

When a function is defined, it can be declared to be either `extern` or `static`; `extern` is the default. A function defined with the `static` storage class specifier can be called only from within the same file where the function appears. Functions defined with the `extern` storage class specifier or functions that do not have a specified storage class specifier can be called by functions in other files because the function name is exported to the Binder.

Both the type specifier and the declarator determine the type of a function. If no type specifier is specified, `int` is assumed.

The type of the return value can be any type, except "array of *type*" or "function returning *type*." That is, functions cannot return arrays or other functions. But they can return pointers to arrays or functions. If it is not intended that a function return a value, specify the type of the return value to be `void`.

See Also

- Refer to “Function Prototype” and “Storage Class Specifiers” in Section 3 for more information on functions.
- Refer to “Function Call Operator” in Section 5 for more information.

Mixing Formats

It is not recommended to mix old style declarations and prototype declarations of the same functions.

A function using the old style can be declared and defined with the function prototype format if, and only if, the type of the actual arguments after promotion agrees with the type of the formal arguments in the function definition.

A function using the function prototype format can be declared and defined using the old style format if, and only if, the function prototype declares the formal arguments to be the same type as used by the default argument promotions, after the promotions occur.

Except in the cases described previously, a function prototype must be in scope of the function call if the function is defined using the function prototype format. Otherwise, the behavior is undefined.

Argument Passing—Call-by-Value

Function arguments are passed “by value.” By-value means that the function is given the value of its arguments in a storage area local to the function rather than their addresses. A function cannot alter an actual argument in the calling function; it can alter only its local copy of the argument.

When necessary, the called function can alter the calling function’s data. To do this:

- The calling function must explicitly pass the address of the arguments to be altered.
- The called function must declare the arguments to be pointers and reference the actual argument indirectly through them.

Example

The following program sequence illustrates argument passing:

```
int func (int a, char *arr)
      {   *arr = 'b';
          a = 10;
      }

main()
      {   char actual_array[3] = "abc";
          int actual_var = 5;
          func(actual_var, actual_array);
          /* After call to 'func', 'actual_var' is still 5,
```

```
                but the contents of 'actual_array' is "bbc" */
        }
```

See Also

Refer to "Pointer Types" in Section 2 for more information on pointers.

Return Values

A function can be defined to return a value of any type, except an array or function. It can, however, return pointers to arrays or functions. The actual value returned by the function, if any, is specified by an expression in the `return` statement. The `return` statement causes the function to terminate. If a function "falls off the end," it is the same as if the following statement were executed:

```
return;
```

The value returned by a function is not an lvalue. Therefore, a function call cannot be the left side of an assignment operator.

If a function is declared with a return type `void`, any `return` statement within the function must not have an expression. If a function is declared as returning a non-`void` type, the `return` statement may or may not contain an expression. If the expression is present, its type must be convertible by assignment to the return type. If the `return` statement does not contain an expression, the value returned is undefined. It is recommended that a `return` function with no expression be used only with a function that is declared to have a `void` return type.

Examples

The following examples illustrate `return` statements:

```
int **func(int a)
    {    static int var = 1;
         static int *var1 = &var;
         static int **var2 = &var1;
         return var2;
    }

void func2()
    {    static int a = 0;
         a++;
         return;
    }
```

See Also

- Refer to "Void Type" in Section 2 for more information.
- Refer to "Assignment Operators" in Section 5 for more information on simple and compound assignments.
- Refer to "Return Statement" in Section 6 for more information.

Linkages to Non-C Functions

Non-C functions can become available to a C program through binding or by using the A Series library facility.

By default, C passes function arguments by value. This can be too restrictive for calling a function that is not written in C. The *linkage specification* provides a mechanism that enables the programmer to bypass the default calling conventions when calling non-C functions.

The linkage specification describes the language in which the function is written and is used by the C compiler to determine the parameter modes for that function. For example, arrays are typically passed to C functions by passing the array's address by value; entry points declared with ALGOL, Pascal, or COBOL linkage can be passed arrays by reference.

Syntax

The syntax for a linkage specification is as follows:

```
linkage-specification:
      extern string-literal { declaration-list }
      extern string-literal declaration
```

The string literal contains the name of the language and must be one of the following (case is not significant):

```
"ALGOL"
"Pascal"
"FORTRAN"
"COBOL"
"C"
```

ALGOL linkage can be used for ALGOL and the ALGOL-type languages (such as NEWP). FORTRAN linkage can be used for FORTRAN and FORTRAN77. COBOL linkage can be used for COBOL74 and COBOL85. C linkage is provided for completeness (in the absence of a linkage specification, all functions have C linkage).

Examples

The first form of the linkage specification enables the declaration of several functions with the same linkage, for example:

```
extern "ALGOL" {
     int     p1 (int);
     void    p2 (int, int);
     static  double p3 (double);
     int     abc [100];
}
```

Note that a linkage specification does not apply to functions declared `static` (such as function `p3` in the previous example). Also, linkage specification does not apply to data objects (such as array `abc` in the previous example).

A pointer to a function with ALGOL linkage can be declared as follows:

```
extern "ALGOL" (*pf) (int, int);
```

Note that for data objects, the `extern` keyword in a linkage specification **does not** specify the object's storage class. To declare `pf` with storage class `extern`, another `extern` must be added as shown in the following example:

```
extern "ALGOL" extern (*pf) (int, int);
```

Unlike objects, functions declared with a linkage specification, but without a storage class, are implicitly `extern`.

A linkage specification may occur only at file scope; for example, the declaration of `pf` may not occur within a function.

See Also

- Refer to the *Binder Programming Reference Manual* for more information on binding other languages.
- Refer to Appendix A, "Interface to the Library Facility," in this manual for more information about library calls.

Parameter Passing

The & operator is used to declare parameters passed by reference. The & operator is used in much the same way as the * operator is used when declaring a pointer. For example:

```
extern "ALGOL" int F (int, float&, int (&)[ ], char (&)[ ]);
```

The preceding example declares an ALGOL integer procedure `F` taking the following four parameters:

1. Integer by value
2. Real by reference
3. Integer array by reference
4. EBCDIC array by reference

The corresponding ALGOL procedure is declared as follows:

```
INTEGER PROCEDURE F (I, D, IA, CA);
     VALUE   I;
     INTEGER I;
     REAL    D;
     INTEGER ARRAY IA [*];
     EBCDIC  ARRAY CA [*];
```

The following declaration declares a Pascal procedure G, which takes two parameters: a by-reference array of integers and a by-reference integer function that takes two by-value integers:

```
extern "Pascal" void G (int (&)[ ], int (&) (int, int));
```

The corresponding Pascal procedure could be declared as follows:

```
type intarray = array [1..100] of integer;
procedure G (var i : intarray;
             function f (x, y: integer): integer);
```

Note the following considerations for declaring entry points with linkage specifications:

- A by-reference function argument with a return type of `void` is treated as an integer procedure parameter, even though a `void` function with non-C linkage is treated as an untyped procedure
- A single `char` cannot be passed by reference
- By-reference arrays, structures, and unions are passed with lower bounds

COBOL does not make use of an array parameter's lower bound; instead, it assumes a lower bound of zero. Since most C arrays, structures, and unions passed by reference have a non-zero lower bound, passing these types of C parameters to COBOL is not recommended. Unexpected results, including corruption of the C heap, can occur if the COBOL program stores data into the array.

- By-reference parameters are not valid for functions with C linkage
- FORTRAN arguments are treated differently from those for ALGOL or Pascal. Simple variables and expressions are passed using the FORTRAN method of call-by-value-result. Character arrays are passed as FORTRAN strings, with the length being that of the actual array and not the length of the C string contained in the array. All other arrays are passed by reference. Note that it is not required to specify the parameter passing mode with the `&` operator. For example, the following example passes an integer by-value-result and a real array by reference to subroutine `SUB1`:

  ```
  extern "FORTRAN" void SUB1 (int, double [ ]);
  ```

A function declaration with a linkage specification must use the function prototype format. If no arguments are passed, the argument list must be explicitly specified as `void`, as in the following example:

```
extern "ALGOL" F (void);
```

Parameter Type Matching

Tables 7-1 through 7-4 illustrate the type matching between C and other languages. The C type `double` is the same as either `float` or `long double` depending on the value of the DBLTOSNGL compiler control option.

Table 7–1. C to ALGOL Types

C Argument Type	ALGOL Parameter Type
`char`	INTEGER by VALUE
`int, short, long`	INTEGER by VALUE
`pointer`	INTEGER by VALUE
`float`	REAL by VALUE
`long double`	DOUBLE by VALUE
`int&, short&, long&`	INTEGER by REFERENCE
`pointer&`	INTEGER by REFERENCE
`float&`	REAL by REFERENCE
`long double&`	DOUBLE by REFERENCE
`char (&) [ ]`	EBCDIC ARRAY [*]
`int (&) [ ]`	INTEGER ARRAY [*]
`float (&) [ ]`	REAL ARRAY [*]
`long double (&) [ ]`	REAL ARRAY [*]
`struct&, union&`	REAL ARRAY [*]
`int (&) ( )`	INTEGER PROCEDURE
`float (&) ( )`	REAL PROCEDURE
`long double (&) ( )`	DOUBLE PROCEDURE
`void (&) ( )`	INTEGER PROCEDURE
`struct (&) ( ),` `union (&) ( )`	PROCEDURE (PTR); VALUE PTR; POINTER PTR;

Table 7–2. C to Pascal Types

C Argument Type	Pascal Parameter Type
`char`	integer

Table 7–2. C to Pascal Types

C Argument Type	Pascal Parameter Type
`int, short, long`	integer
`pointer`	integer
`float`	real
`int&, short&, long&`	var integer
`pointer&`	var integer
`float&`	var real
`char (&) [ ]`	var packed array of char
`int (&) [ ]`	var array of integer
`float (&) [ ]`	var array of real
`long double (&) [ ]`	var array of real
`struct&, union&`	var record
`int (&) ( )`	integer function
`float (&) ( )`	real function
`void (&) ( )`	integer function

Table 7–3. C to FORTRAN Types

C Argument Type	FORTRAN Parameter Type
`char`	INTEGER
`int, short, long`	INTEGER
`pointer`	INTEGER
`float`	REAL
`long double`	CHARACTER* (*) (FORTRAN77 only)
`char [ ]`	INTEGER (*)

Table 7–3. C to FORTRAN Types

C Argument Type	FORTRAN Parameter Type
`int [ ]`	REAL (*)
`float [ ]`	REAL (*)
`long double [ ]`	REAL (*)
`struct, union`	REAL (*)

Table 7–4. C to COBOL Types

C Argument Type	COBOL85 Parameter Type
`char`	77 BINARY †
`int, short, long`	77 BINARY †
`pointer`	77 BINARY †
`float`	77 REAL †
`long double`	77 DOUBLE †
`int&, short&, long&`	77 BINARY
`pointer&`	77 BINARY
`float&`	77 FLOAT
`long double&`	77 DOUBLE

Legend

† The argument is actually passed by value. In a bound COBOL procedure, the COBOL declaration should read “BY CONTENT”.

Result Type Matching

Function results declared as `char`, `int`, or pointer types map to INTEGER in other languages, `float` maps to REAL, and `long double` maps to DOUBLE.

A `void` function with non-C linkage maps to an untyped procedure in other languages, but a `void` function with C linkage maps to an INTEGER procedure in other languages. A function with COBOL linkage must have a result type of `void`.

A struct or union function maps to an untyped procedure with an additional first argument, which is an ALGOL POINTER passed by value. The POINTER points at a

place for the function to put its result. The following example shows a C function declaration followed by the ALGOL definition of the function. The function returns a struct containing three integers given as arguments to the function.

C Declaration

```
typedef struct S {int a, b, c;} S;
extern "ALGOL" S F (int a, int b, int c);
```

ALGOL Procedure

```
PROCEDURE F (RESULT_PTR, A_VALUE, B_VALUE, C_VALUE);
   VALUE RESULT_PTR, A_VALUE, B_VALUE, C_VALUE;
   POINTER RESULT PTR;
   INTEGER A_VALUE, B_VALUE, C_VALUE;
BEGIN
REPLACE RESULT_PTR BY A_VALUE, B_VALUE, C_VALUE;
END;
```

Section 8
The C Preprocessor

The C language contains a preprocessor that translates the source code of a C program before it is compiled. The C preprocessor is capable of macro substitution, conditional compilation, and inclusion of named files. These capabilities are called preprocessing because conceptually they occur before translation of the resulting source code.

Programmers use the preprocessor to develop programs that are easier to read, develop, modify, and transport to another system. A programmer can also use the preprocessor to customize the C language for a particular application.

The C preprocessor acts on directives. These directives are on special preprocessor control lines in the source file. The control lines begin with the number sign character (#). In C, the number sign character (#) is used only on preprocessor control lines. Lines that do not contain preprocessor commands are called source lines.

All preprocessor control lines are removed from the source file and the actions that the directives indicate (such as expansion of macro calls that occur within the source text) are performed on the source code. After preprocessing, the source text must not contain a number sign character (#), except in string and character constants. The result of preprocessing is source text that the compiler begins to translate.

Also discussed in this section is CPREP, a C compiler utility that enables you to create output source files by manipulating input source files through preprocessor directives.

Lexical Conventions of the C Preprocessor

The lexical conventions of the preprocessor are similar to the lexical conventions of the C compiler. The preprocessor breaks the source text up into tokens so it can locate macros. The following are the lexical conventions of the preprocessor:

- The preprocessor recognizes the following:
  - Identifiers
  - Comments
  - Integer, floating-point, character, and string constants

  Note: *Numeric constants used by the preprocessor, called pp-numbers, are different from numeric constants used by the C language, called numbers. The pp-numbers start with an optional period, a digit (0..9), followed by zero or more letters, underscores, digits, periods, and* `e+`, `e-`, `E+`, *or* `E-` *character sequences. The pp-numbers lexically include all floating and integer constant tokens; they also include such tokens as the following:*

  ```
  0xABCDE+123
  ```

  Appropriate use of white space prevents pp-numbers from using characters beyond the end of the number:

  ```
  0xABCDE + 123
  ```

- The preprocessor does not look for directives within the following:
  - Character and string constants
  - Comments
- A preprocessor control line is a line whose first non-preprocessor white-space character is a number sign (#), followed by more preprocessor white space (optional), followed by the preprocessor directive.
- Terminating a preprocessing control line with a backslash continues the line to the next line. Within a preprocessing control line, the backslash (\) is discarded and the following line is treated as part of the line that ended with the backslash. This means that the following line is not treated as a separate preprocessor control line, even if it begins with the number sign character (#).

See Also

Refer to “Comments,” “Constants,” and “White Space” in Section 1 for more information.

The C Preprocessor Directives

The C preprocessor contains four types of preprocessor directives.

Preprocessor Directives	Description
Conditional inclusion directive	This directive enables compilation of only selected lines of the source code.
Macro directive	This directive creates or discards a macro definition.
File inclusion directive	This directive includes text from other files.
Information directive	This directive obtains information from the compiler and sends information to the compiler.

Table 8–1 summarizes the preprocessor directives.

Table 8–1. Preprocessor Directives

Conditional Inclusion Directives	
Directive Name	**Description**
`#if`	Conditionally includes some text, based on the value of a constant expression.
`#else`	Alternately includes some text, if the previous #if, #elif, #ifdef, or #ifndef test failed.
`#elif`	Alternately includes some text based on the value of another constant expression.
`#endif`	Terminates conditional text.
`#ifdef`	Conditionally includes some text, based on whether a macro name is defined.
`#ifndef`	Conditionally includes some text, with the sense of the test opposite that of #ifdef.
File Inclusion Directive	
Directive Name	**Description**
`#include`	Inserts text from another file.
Macro Directives	
Directive Name	**Description**
`#define`	Defines a preprocessor macro.
`#undef`	Removes a macro definition.
Information Directives	
Directive Name	**Description**
`#error`	Produces user error messages.
`#line`	Sets the value of `_LINE_` and `_FILE_` macros.
`defined`	Determines if a name is defined as a preprocessor macro.
`#pragma`	Implementation-defined behavior.

Conditional Inclusion

The preprocessor conditional compilation directives enable source lines to be included or excluded from a compilation. The conditional inclusion directives are

- `#if`
- `#else`
- `#elif`
- `#endif`
- `#ifdef`
- `#ifndef`

#if, #else, #elif, and #endif Directives

The `#if`, `#else`, `#elif`, and `#endif` directives enable source statements to be conditionally included in a source program.

The following program fragment illustrates a typical use of the `#if`, `#else`, `#elif`, and `#endif` directives:

```
#if constant-exp1
   source-lines /* included if constant-exp1 is nonzero */
#elif constant-exp2
   source-lines /* included if constant-exp2 is nonzero and */
                /* constant-exp1 is zero */
      .
      .
      .
#elif constant-expn
   source-lines /* included if constant-expn is nonzero and */
                /* constant-exp1 through constant-expn-1 are zero */
#else
   source-lines /* included if all constant-exps are zero */
#endif
```

The `#else` and `#elif` directives and their corresponding statement groups are optional.

Each *constant-exp* is evaluated until one produces a nonzero (true) value. The *source-lines* associated with the control line that contained the nonzero *constant-exp* are included. All other *constant-exps* are ignored and *source-lines* associated with their control lines are discarded. If no *constant-exp* produces a nonzero (true) value, the *source-lines* associated with the `#else` control line are included. If the conditional construct does not contain a `#else` control line, none of the *source-lines* is included.

Examples

The following example illustrates the conditional inclusion directives:

```
#define init 1

#if init
       int table [50];
       void initialize_table();
#else
       int table[50];
#endif
```

Conditional directives can be nested. In the following example, the `#else`, `#elif`, and `#endif` bind to the closest `#if` that has not already been closed by `#endif`:

```
#if defined NEED_DECLARE
int foo(int i, char c)
        #if defined NEED_BODY
          { return i + c; }
        #else
          ;
        #endif
#endif
```

As previously stated, all statements not to be included are discarded. However, before being discarded, any preprocessor lines within the group are checked. Therefore, all preprocessor lines within a discarded group must be legal preprocessor directives. This is because only the `#if`, `#elif`, `#else`, and `#endif` directives are recognized as delimiters. These directives are used to keep track of nested conditionals. The following example produces an error saying `garbage` is an invalid directive, even though compilation is turned off:

```
#if 0
#garbage
#endif
```

The constant expression that controls conditional inclusion must be an integral constant expression that does not contain a `sizeof` operator, a cast, or an enumeration constant. It can contain unary expressions of the following format:

```
defined identifier
defined (identifier)
```

These expressions produce the value 1 if the identifier is currently defined as a macro name (that is, it is the subject of a `#define` directive without an intervening `#undef` directive) or the value 0 if the identifier is not currently defined as a macro name (refer to "defined Operator" in this section).

On the `#if` and `#elif` control lines, macro names not associated with the `defined` operator are macro replaced. Therefore, macro calls can be included in the *constant-exp*. Any other identifier not defined as a macro is replaced with the value 0.

See Also

Refer to "Constant Expressions" in Section 5 for more information.

#ifdef and #ifndef Directives

The `#ifdef` and `#ifndef` directives test whether an identifier is currently defined as a preprocessor macro.

Syntax

if-group:

`#ifdef` *identifier newline* $group_{opt}$

`#ifndef` *identifier newline* $group_{opt}$

If *identifier* is not currently defined as a macro name, the *source-lines* are included.

To be defined, the *identifier* must appear on a `#define` control line without being the subject of an intervening `#undef` directive (refer to "#undef Directive" in this section).

The `#ifdef` and `#ifndef` directives are variations of the `#if` directive and the `defined` operator.

The following control lines are equivalent:

```
#ifdef identifier

#if defined(identifier)
```

Also, the following control lines are equivalent:

```
#ifndef identifier

#if !defined (identifier)
```

Examples

The following example illustrates the use of the `#ifdef` directive:

```
#define debug(mark) printf( " executing at spot %i\n", mark);
#ifdef debug
      debug(1)
   #undef debug
#endif
#ifdef debug
      debug(2)
#endif
```

File Inclusion Directive

The file inclusion directive contains the `#include` directive, which enables the preprocessor to include source code into your program when it is compiled.

#include Directive

The `#include` directive causes the `#include` control line to be replaced with the contents of the specified file.

Syntax

control-line:
`#include` *pp-tokens library-linkage*$_{opt}$ *newline*

The *pp-tokens* are macros. The expanded macros must result in one of the following two forms:

`#include "`*source identifier*`"` *library-linkage*$_{opt}$
`#include <`*source-identifier*`>` *library-linkage*$_{opt}$

The semantics of the two forms differ in the manner in which the files are searched. The "*source-identifier*" form searches for a file with the title *source-identifier*. If a file is not found, *source-identifier* is compared to the list of headers. If found, the header file is included, otherwise the compiler either waits on a NO FILE condition (if the compiler control option ANSI is **not** set) or flags the missing file as an error (if the ANSI option is set).

The <*source-identifier*> form compares *source-identifier* to the list of headers. If found, the header file is included, otherwise the source file is processed as if the header were as follows:

`#include "`*source-identifier*`"`

Before a file is searched for, the *source-identifier* is edited to help it conform to the required title format. Any characters inside double quotes (`" "`) are left "as is" except any period (`.`) or backslash (\), which is translated to a slash (/), except when that would produce an illegal file title, in which case the period or backslash is dropped. The SEARCH compiler control option may be used to further edit the file title and to search in multiple places. The TITLESYNTAX compiler control option specifies which file name attribute, title or pathname, and parsing rules to apply when a file name is found in the source. Refer to Section 10, "TITLESYNTAX Option," for more information on compiler control options.

A comment, continued line, or conditional inclusion cannot cross a boundary of an included file. For example, a comment cannot start in an included file and end outside of the included file.

This C language implementation enables 32 nesting levels for `include` files before it issues an error message and terminates the compilation.

The optional *library-linkage* is an A Series extension that enables the included file to specify entry points in an A Series library. Refer to Appendix A, "Interface to the Library Facility," for information about these libraries.

Syntax

```
library-attributes:
       bytitle = string-literal , intname = string-literal
       byfunction = string-literal , intname = string-literal
       byinitiator , intname = string-literal
```

The first syntax specifies that the library is located by title and the code file that should be linked to. The *string-literal* should be a valid A Series file title. The SEARCH compiler control option is not used when searching for a library.

The second syntax specifies that the library is located by a Support Library function name. The system command SL maps a function name onto a file by title. The *string-literal* should be a letter followed by 1 to 16 letters or digits.

The third syntax is used when the program is itself a library and wants to link back to the program that caused it to be initiated.

All forms specify the internal library name that is used by the Binder to merge entry points from different files. The *string-literal* should be a letter followed by 1 to 16 letters or digits.

Example

The following example illustrates inclusion of files. Assume that files FILE/C and STDLIB/H exist and that no file exists with the names STDIO/H or NONE/H.

```
#include "file.c"        /* file FILE/C included */
#include "stdlib.h"      /* file STDLIB/H included */
#include "stdio.h"       /* header stdio.h included */
#include "none.h"        /* compiler waits on a no-file */
                         /* or gives an error if ANSI option is set */

#include <file.c>        /* file FILE/C included */
#include <stdlib.h>      /* header stdlib.h included */
#include <stdio.h>       /* header stdio.h included */
#include <none.h>        /* compiler waits on a no-file */
                         /* or gives an error if ANSI option is set */
```

Macro Directives

The macro directives enable the preprocessor to define and remove a macro.

#define Directive

The #define directive creates preprocessor macros by associating a token sequence, called the macro body, with an identifier (macro name). When the macro name is recognized, it is replaced with the macro body.

Syntax

control-line:
#define *identifier replacement-list newline*
#define *identifier lparen identifier-list*$_{opt}$*)replacement-list newline*

The #define directive can be defined in two ways:

- A simple macro definition
- A macro definition with parameters

The coding style of completely capitalizing macro identifiers is recommended. This style makes it easier to distinguish macro identifiers from other identifiers.

Simple Macro Definitions

The simple macro definition begins with a #define directive, followed by an identifier that is the macro name, followed by a *replacement-list* that is the macro body. White space preceding or following *replacement-list* is not considered part of the macro body.

Whenever a defined macro name is encountered, the macro name is replaced with *replacement-list*. After replacement, the macro body is rescanned for more defined macro names. Any macro names inside a string or character constant are not replaced.

To continue a macro definition to another line, end the line to be continued with a backslash.

One use of this form of the #define directive is to define the value of a program constant. The value of the constant can be defined in one place and referred to by name in the program. To change the value of the constant, change only the #define directive.

```
#define TABSIZE 100
int table[TABSIZE];
```

Macro Definitions with Parameters

The macro definition with parameters defines a function-like macro with arguments similar to a function call. The arguments are specified by the optional list of identifiers whose scope extends until the termination of the #define directive. The argument list must immediately follow the macro name; no space is allowed. If a space or spaces separate *identifier* from the left parenthesis, the definition is considered to be a simple definition and the argument list to be part of the macro body.

When the preprocessor encounters the macro name immediately followed by a left parenthesis, it replaces the macro name and argument list with the processed macro body. The left parenthesis that begins the argument list must immediately follow the *identifier* (macro name).

To continue a macro definition to another line, end the line to be continued with a backslash.

The actual arguments in the call are nonempty token strings separated by commas. The number of actual arguments must match the number of formal arguments. Actual arguments can contain parentheses, if they are properly nested and balanced, and commas, if contained within nested parentheses.

Macro expansion is the replacement of a macro call with the processed copy of its body. The processed copy of its body is called the expansion of the macro call. In a macro expansion:

1. The actual arguments are associated with their corresponding formal arguments

2. A copy of the macro body is created

   In this copy, each formal argument is replaced with the actual argument associated with it. Unless preceded by a number sign character (#) or double number sign character (##) token or followed by a double number sign character (##) token, the actual arguments are subject to macro replacement before they are substituted. That is, the tokens that make up the actual arguments are checked for macros and expanded before being substituted in the macro body.

   Any macro names or argument names inside a string or character constant are not replaced.

3. The modified copy of the macro body replaces the macro call

The following macro definition defines a macro whose body provides source code to calculate the maximum value of its arguments:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

This macro definition has the advantages of working for any compatible types of arguments and of generating in-line code without the overhead of a function call. It has the disadvantages that during execution of the code one of the original arguments is evaluated a second time (side effects occur again).

Preprocessor Operator

Each # preprocessor token in the macro body of a function-like macro must be followed by a formal argument as the next preprocessing token in the macro body.

When a formal argument is immediately preceded by a # preprocessing token in a macro body, both the formal argument and the # token are replaced by a single string literal that contains the spelling of the actual argument (that is, the actual argument is not expanded before replacing the formal argument and the # token in the macro body). White space before the first token and after the last token comprising the actual argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the string literal. This requires special handling for producing the spelling of string literals and character constants: a backslash character is inserted before each double quotation mark and backslash character of a character constant or string literal (including the delimiting double quotation mark characters).

Example

The following example illustrates the # preprocessor operator:

```
#define stringize(a) #a
char c[] = stringize(Every "string" ends with '\0');
```

The resulting source code is as if the following had been written:

```
char c[] = "Every \"string\" ends with '\\0'";
```

The `strlen` of the string is 29.

Preprocessor Operator

For both the simple macro and parameterized macro invocations, each instance of a ## token in a macro body is deleted and the tokens preceding and following it are concatenated. If the tokens to be concatenated are formal arguments, they are replaced by the actual arguments before concatenation occurs. If the result of the concatenation is not a valid preprocessing token, an error is issued. The resulting token is available for further macro replacement. The concatenation takes place before the macro body is rescanned for more macro names.

A ## preprocessing token cannot occur at the beginning or end of a replacement list for either form of macro definition.

Rescanning for Macro Calls

After all arguments in the macro body have been substituted, the resulting token sequence is rescanned, within the context of the source file, for more macro names to be replaced. Any macro names inside a string or character constant are not replaced.

If the name of the macro being replaced is found during the scan of the replacement list, it is not replaced. Further, if any nested replacements encounter the name of a macro being replaced, it is not replaced. (This prevents direct and indirect recursive macro expansion.)

Example

The following example illustrates macro replacement:

```
#define mac a+mac2
#define mac2 b+mac

                    /* results in a+b+mac */
mac                 /* Note that mac did not get expanded when */
                    /* it was seen in the replacement tokens. */
```

Macro body scanning and replacement does not occur until the macro name is called. The macro body is not scanned during definition (on the control line). Macro names are recognized in the body only when the macro has been expanded by a macro call. Macro names within the actual arguments are expanded before the parameters are replaced, except where the # or ## operators are used.

Example

The following example illustrates the rescanning and replacement of macro names:

```
#define A  3
#define F(x) F(A * (x))
F(y + 1) + F(F(z))
```

After expansion, the result is the following:

```
F(3 * (y + 1)) + F(3 * (F(3 * (z))))
```

#undef Directive

The `#undef` directive removes the current definition of a preprocessor identifier.

Syntax

control-line:
`#undef` *identifier newline*

Given this directive, the current definition of the macro named *identifier* is forgotten. *identifier* can now be redefined with the `#define` directive.

Undefining an identifier that is not currently defined as a macro name has no effect.

A macro definition lasts, independent of block structure, until a corresponding `#undef` directive is encountered. If none is encountered, the definition lasts until the end of the source file, including all included files.

A macro can be redefined without an intervening `#undef` if the token sequence of the macro definitions are identical. This includes using the same formal argument names and the same white space separations where all white space separations are considered identical. That is, any amount of white space is considered identical to any other amount of white space, except no white space. For example, the following macro definitions are **not** identical:

```
#define x (a)

#define x (  a  )
```

However, the following macro definitions **are** identical:

```
#define x (   a   )

#define x (  a        )
```

Example

The following example defines and undefines an identifier:

```
#define TEST
#undef TEST
```

Predefined Macro Names

Table 8–2 lists the predefined macro names.

Table 8–2. Predefined Macro Names

Predefined Macro Name	Description
__ASERIES__	Defined when compiling on the A Series system.This macro name can be used with `#ifdef` to conditionally compile code specifically for A Series applications. This macro name gives the same string value as __COMPILER_VERSION__.
__COMPILER_VERSION__	A string literal representing the current compiler version. The string is of the form “rr.ccc.ppp” where r=release, c=cycle, p=patch. This is an A Series C extension.
__DATE__	The date of translation of the source file (a 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 blank if the value is less than 10).
__FILE__	The presumed name of the source file (a string literal).
__LINE__	The line number of the current source line (a decimal constant).
__LINENUMBER__	The sequence number associated with the current source line (a decimal constant). This is an A Series C extension.
__STDC__	The decimal constant 1 (for an implementation that conforms to the ANSI C standard).
__TIME__	The time of translation of the source file (a string literal of the form “hh:mm:ss”).
__VERSION__	A string literal of the form "rr.ccc.pppp" where r=release, c=cycle, and p=patch number taken from the VERSION compiler control option of the source program. This is an A Series C extension.

None of these macro names, nor the identifier `defined`, can be the subject of a `#define` or a `#undef` preprocessing directive. Predefined macros always begin with `double` underscores. Therefore, always name macros with names that do not begin with `double` underscores to avoid conflict with predefined macros.

Macro Definition and Replacement Examples

The following examples illustrate the replacement results for macro definitions.

Example

The following example illustrates the rules for redefinition and reexamination:

```
#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);
```

Replacement Result

```
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);
```

Example

The following example illustrates the rules for creating string literals and concatenating tokens:

```
#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      vers1

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') /* continues on next line */
          ==0) str(: @\n), s);
#include xstr(INCFILE.h)
#include str(INCFILE.h)
```

Replacement Result

```
printf("x" "1" "= %d, x" "2" "=%s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers1.h"         /* after macro replacement, before file access */
#include "INCFILE.h"
```

The preceding result is identical to the following statements because the adjacent string literals are concatenated:

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers1.h"         /* after macro replacement, before file access */
#include "INCFILE.h"
```

Space around the number sign (#) and double number sign (##) tokens in the macro definition is optional.

Example

To demonstrate the redefinition rules, the following sequence is valid:

```
#define OBJ_LIKE        (1-1) /* no white space in (1-1)  */
#define OBJ_LIKE        /* white space */ (1-1) /* other */
#define FTN_LIKE(a)    (  a   +   b)
#define FTN_LIKE(a)    ( a +         b)
```

In the last two examples, if one white-space character exists between tokens, any amount of white space can exist between the same tokens and it is still considered identical.

Example

The following sequence is an invalid sequence of redefinitions of the macros just defined:

```
#define OBJ_LIKE          (0)        /* different token sequence */
#define OBJ_LIKE          (1  -  1) /* here white space exists */
#define FTN_LIKE(a)     ( #a )     /* different formal parameter usage */
#define FTN_LIKE(c)     ( a + c)  /* different formal parameter spelling */
```

Information Directives

The information directives produce error messages, determine if a name is defined as a preprocessor macro, and sets the value for _LINE_ and _FILE_ macros.

#error Directive

The #error preprocessing directive causes a diagnostic message that includes the token sequence that follows the #error directive.

#line Directive

The #line directive sets the value of the predefined macro __LINE__ and optionally __FILE__.

Syntax

```
control-line:
      #line  pp-tokens newline
```

The *pp-tokens* are macros. The expanded macros must result in one of the following two forms.

```
#line  digit-sequence
#line  digit-sequence  string
```

Both forms set the macro __LINE__ to be *digit-sequence*, starting at the next line. The macro __LINE__ continues to be incremented by one for every line read. The second form also sets the macro __FILE__ to be *string*. The *string* can be any C string or concatenation of C strings. If the compiler control option ANSI is set, the value of *digit sequence* must be between 1 and 32767 inclusive. If the ANSI option is reset, the value must be between 1 and 99999999 inclusive. The macro __LINE__ ceases to increment once the maximum value is reached.

#pragma Directive

The #pragma directive verifies that object files being bound together are compiled with the same set of compile-time options. The compiler supports the following #pragma directive and ignores all others:

Syntax

```
#pragma binder_match = (<string1>, <string2>)
```

The BINDER_MATCH compiler control option is added to the name of *string1* and to the value of *string2*. *String1* names that begin with an asterisk (*) are reserved for use by Unisys only.

Example

The following example shows the use of the #pragma directive:

```
#pragma binder_match = ( "XPG", "false" )
#pragma binder_match=("Compiled Date",__DATE__)
```

See Also

Refer to Section 10, “Compiler Control Options,” for more information on the BINDER_MATCH compiler control option.

defined Operator

The defined operator can appear only in #if and #elif preprocessor directives. The defined operator tests for the definition of a preprocessor identifier definition.

Syntax

```
defined identifier
defined ( identifier )
```

The `defined` operator produces the value 1 (true) if *identifier* is defined in the preprocessor or 0 (false) if *identifier* is not defined.

Example

The following example illustrates the use of the `defined` operator:

```
#define mac(var,string) char var[] = string;
#define macZ(a) int a;
#if defined(mac) && defined macZ
   mac(string_var, "This is a string")
   macZ(integer)
#else            /* would produce: */
int b;           /* char string_var[ ] = "This is a string"; */
#endif           /* int integer; */
```

Syntax

The syntax of the preprocessor directives is independent of the syntax of the rest of the C language, although in some ways it is similar to the syntax of the C language. The syntax of the preprocessor is as follows:

preprocessing-file:
   group$_{opt}$

group:
  group-part
  group group-part

group-part:
  pp-tokens$_{opt}$ *newline*
  if-section
  control-line

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
  #if *constant-expression newline group*$_{opt}$
  #ifdef *identifier newline group*$_{opt}$
  #ifndef *identifier newline group*$_{opt}$

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  #elif *constant-expression newline group*$_{opt}$

else-group:
  #else *newline group*$_{opt}$

endif-line:
  #endif *newline*

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

lparen:
  the left parenthesis character without preceding white space

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

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

preprocessing-token:
  character-constant
  header-name /* only on a #include line */
  identifier
  keyword
  operator
  pp-number
  punctuator
  string-literal
  each non-white-space character that cannot be one of the previous

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

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

h-char:
   any character in the source character set, except the
      newline character and >

newline:
   the newline character

```
pp-number:
   digit
   . digit
   pp-number digit
   pp-number non-digit
   pp-number e sign
   pp-number E sign
   pp-number .

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

q-char:
  any character in the source character set,
     except the newline character and "

library-linkage:
  (library-attributes)

library-attributes:
  bytitle = string-literal , intname = string-literal
  byfunction = string-literal , intname = string-literal
  byinitiator , intname = string-literal
```

The C Preprocessor and Compiler Control Options

For this C language implementation, compiler control options provide many of the capabilities provided by preprocessing directives. All compiler control options are processed before the preprocessing directives.

Example

```
#if 0
$ SET FLAG1
#endif
$ IF NOT FLAG1
#error Hi there
$ END
```

The compiler control option `FLAG1` is always set and the `#error` is never processed.

The CPREP Compiler Utility

CPREP is a C compiler utility that allows you to create an output source file by manipulating an input source file through C preprocessor directives. The input file may be any source file.

The input source to the CPREP utility can be any statements including C compiler control records (CCRs) and C preprocessor statements. CPREP will accept any compiler control option; however, only those options that control input source and output source have any meaning to CPREP. The C preprocessor statements work in CPREP the same way they work in the C compiler. They can be used to control source inclusion/exclusion and they can be used for macro substitution. For all other statements other than comments, CPREP simply breaks the source into tokens. Tokens that match macro names are replaced by macro definitions. Macros are defined through use of C preprocessor statements.

The output source file generated by CPREP is the same as the output file XSOURCE, which is generated by the C compiler when the compiler control option XSOURCE is enabled. The output source file contains all statements except CCRs, C preprocessor, and comments.

Functionally, the use of CPREP is the same as the use of the C compiler when the option XSOURCE is enabled. However, CPREP differs in the following ways:

- CPREP does no syntax checking on non-CCI and non-preprocessor statements.
- CPREP allows in addition to C-style comments, single-line comments using a double slash (//). Any text to the right of a double slash is considered a comment.
- CPREP does not generate object code files.

CPREP Input and Output Files

The input files to CPREP are the same as the input files to the C compiler. The output files from CPREP are the same as the output files from the C compiler except that

CPREP does not create object code files. Refer to "Understanding Compiler Files" in Section 9 for more information on input and output files.

Running CPREP

CPREP is a compiler and as such can be invoked in the same manner in which the C compiler is invoked. In the COMPILE command, you simply substitue `cprep` for `cc`.

Refer to "Compiling and Executing A Series C Programs" in Section 9 for more information on invoking the C compiler.

Compiling from CANDE

To compile with CPREP from CANDE, do the following:

```
c with cprep
c source/prog with cprep
```

Compiling from WFL

To compile with CPREP from WFL, do the following:

```
COMPILE <anyname> WITH CPREP LIBRARY;
COMPILER FILE CARD(KIND=DISK, TITLE=SOURCE/PROG);
```

Naming XSOURCE

The primary output from CPREP is the file XSOURCE. To give XSOURCE another name, add the following:

```
COMPILER FILE XSOURCE = output_source_file
```

where `output_source_file` is a user-supplied file name.

For example, if you are compiling from CANDE, do the following:

```
c source/prog with cprep; compiler file xsource = xsource/prog
```

If you are compiling from WFL, do the following:

```
COMPILE <anyname> WITH CPREP LIBRARY;
COMPILER FILE CARD(KIND=DISK, TITLE=SOURCE/PROG);
COMPILER FILE XSOURCE(TITLE=XSOURCE/PROG);
```

Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

Section 9
Compiler Operations

This section describes the compiler operations for this C language implementation. It contains the following two parts:

- Understanding Compiler Files

  This part of the section describes the input and output files of the compiler.

- Compiling and Executing A Series C Programs

  This part of the section describes the syntax and semantics for:

  – Creating an object code file using the compiler

  – Executing the object code file generated by the compiler

Refer to Section 10, "Compiler Control Options," for detailed information on the compiler control options that are available in this C language implementation.

Understanding Compiler Files

Table 9–1 summarizes the input and output files used by the C compiler.

Table 9–1. Summary of Compiler Files

Input Files		
Internal File Name	**Usage**	**Kind**
CARD	This is the primary input file.	READER for batch compilation. DISK for interactive compilation.
SOURCE	This is an optional secondary input file when MERGE is enabled.	DISK
INITIALCCI	This is an optional file containing initial compiler option settings.	DISK

Table 9–1. Summary of Compiler Files

Output Files		
Internal File Name	**Usage**	**Kind**
CODE	This file contains the object code.	DISK
ERRORS	This file contains all error and warning messages issued by the compiler.	DISK for batch compilation. REMOTE for interactive compilation.
LINE	This is an optional file that contains a listing of program text, object code, error messages, and diagnostics.	PRINTER
NEWSOURCE	This is an optional source file containing merged CARD and SOURCE input files.	DISK
XSOURCE	This is an optional file containing an expanded source.	DISK

Input Files

The compiler can accept at least three input files. These input files are:

- CARD file
- SOURCE file
- INITIALCCI file

The compiler accepts input from one or more input files. The primary input file is assigned to the internal file CARD. If a secondary input file is specified (through the presence of a MERGE compiler control record), that file is assigned to SOURCE (or TAPE, if there is no file equation for SOURCE).

The merging of primary and secondary input files produces what is known as a "virtual" input file. Merging proceeds based on the sequence number fields of the file records: The compiler serially processes the input files, selecting at each step the record with the lowest sequence number. If both files have records with the same sequence number, the record from the primary file is read and the record from the secondary file is skipped. Since the merging depends on the presence of sequence numbers, files with FILEKIND DATA cannot be used for merging.

The CARD File

The CARD file is the primary input file of the compiler. It must be present for each compilation. The default KIND attribute of the CARD file depends on how the compiler is initiated. If the compiler is initiated through a batch compilation, the CARD file is assumed to be a card reader file. If the compiler is initiated through an interactive compilation, the CARD file is assumed to be a disk file.

The SOURCE File

The SOURCE file is an optional secondary input file when the MERGE compiler control option is enabled. If no file equation statements are used, the SOURCE file is assigned to TAPE.

The INITIALCCI File

The INITIALCCI file provides a means whereby compiler options can be assigned initial settings for each invocation of the compiler. Moreover, the initial option settings can be established on a language-by-language basis.

The INITIALCCI file consists of slightly modified compiler control records. Each individual compiler user can have an individual INITIALCCI file or a site can install a system-wide (nonusercoded) *INITIALCCI file. The compiler searches for the INITIALCCI file using the usual usercode and family name conventions; if it is not found, the file is done without (INITIALCCI is the INTNAME of the file; hence, the file can be file equated). The FILEKIND of the INITIALCCI file need not match that of the program being compiled.

In the INITIALCCI file, the compiler control records need not begin with a dollar sign ($), as they normally must. An INITIALCCI record may contain an identifier followed by a colon (:). If the identifier matches the compiler language, then the compiler options following the colon are processed; otherwise, the remainder of the record is ignored. If the identifier is BATCH, the options are processed only for compilations initiated through WFL; if the identifier is INTERACTIVE, the options are processed only for interactive compilations.

Before the INITIALCCI file is read, all compiler options are set to their initial values. INCLUDE cards are not allowed in INITIALCCI files. The INITIALCCI file is read before the compiler input files are read.

Example

The following is an example of an INITIALCCI file:

```
C:            RESET LINEINFO
C:            OPTIMIZE XREFFILES ANSI
FORTRAN77:    SET XREF
INTERACTIVE:  ERRORLIMIT = 5
BATCH:        ERRORLIMIT = 10
              RESET LIST SET NEW
```

This INITIALCCI file has the following effects:

- If the CARD file is a C source file, the LINEINFO option is disabled and the OPTIMIZE, XREFFILES, and ANSI options are enabled.
- If the CARD file is a FORTRAN77 source file, the XREF option is enabled.
- If the compilation is through CANDE, Editor, or the MAKE utility, the ERRORLIMIT is set to 5; if the compilation is through WFL, the ERRORLIMIT is set to 10.
- For all compilations, the LIST option is disabled and the NEW option is enabled.

Output Files

The compiler can generate at least five output files. These output files are:

- CODE file
- LINE file
- ERRORS file
- NEWSOURCE file
- XSOURCE file

In addition, the compiler generates cross-reference files if the XREFFILES compiler option is enabled. Refer to "XREFFILES Option" in Section 10 for more information.

The CODE File

The CODE file is created by the compiler and contains the compiled object code. If syntax errors occur during compilation, this file is not created.

The ERRORS File

The ERRORS file receives all error and warning messages issued by the compiler. Creation of this file is controlled by the ERRORLIST compiler option. By default, the ERRORS file is a disk file if the compilation is initiated through WFL and a remote file if the compilation is initiated through CANDE.

The LINE File

The LINE file is the output listing file and is by default a PRINTER file. The output listing may contain source listings, code listings, error messages, and diagnostic information. The CODE, ERRORLIST, LIST, LISTP, LISTINCL, and LISTINITIALCCI compiler options control the generation of this information and are fully described in Section 10, "Compiler Control Options."

The NEWSOURCE File

The NEWSOURCE file is created if the NEW option is enabled. This file contains most of the records that appear in the virtual input file. Temporary dollar-cards (as defined in Section 10, "Compiler Control Options") are not written to the NEWSOURCE file. INCLUDE directives are handled specially: If the INCLNEW option is enabled, the included records are written to NEWSOURCE, but the INCLUDE card is not; otherwise, the INCLUDE card is written to NEWSOURCE, but the included records are not. The NEWSOURCE file is not created if syntax errors are encountered.

The XSOURCE File

The XSOURCE file is generated if the XSOURCE option is enabled. It is equivalent to the NEWSOURCE file with all macro definitions expanded and comments deleted.

The XSOURCE file is also generated by the CPREP compiler utility. Refer to "The CPREP Compiler Utility" in Section 8 for more information.

Controlling Compiler Input

The source language input processed by the compiler can be controlled in the following ways:

- Setting the correct FILEKIND attribute to customize C commands
- Making compiles consistent through the INITIALCCI file

Customizing Commands through the FILEKIND Attribute

Both C source files and the header files should have their FILEKIND attribute set to CCSYMBOL. While not required, having the correct FILEKIND enables the Editor and CANDE to customize commands for C.

The FILEKIND can be specified by the CANDE *MAKE* command using the following syntax:

```
make file-title cc
```

The FILEKIND can also be specified by any program that creates a source file. Refer to Appendix C, "Porting C Applications to A Series Systems," for examples.

Making Compilations Consistent through the INITIALCCI File

The INITIALCCI file provides a means by which compiler control options can be assigned initial settings for each invocation of the C compiler. One common use for it is to make compilations under CANDE more consistent with compilations under WFL. If the INITIALCCI file contains the following WFL statement, WFL compilations start with the same initial option settings as Editor or CANDE compilations:

```
BATCH: RESET LIST SET LINEINFO ERRORLIST ERRORLIMIT = 10
```

Refer to "The INITIALCCI File" in this section for more information on the format and use of the INITIALCCI file.

Compiling and Executing A Series C Programs

The C compiler can be invoked from the following interactive environments:

- The Command and Edit (CANDE) environment
- The Editor environment
- The Work Flow Language (WFL) batch environment

The syntax and semantics are slightly different for each environment and are described briefly in this section.

Compiling from CANDE

CANDE customizes its environment based on the FILEKIND attribute; files compiled from CANDE should have their FILEKIND set to CCSYMBOL. CANDE translates lowercase letters to uppercase in commands, except inside double-quoted (") strings. Lowercase is used in the examples, but either case can be used.

The CANDE *COMPILE* Command

The short form syntax of the CANDE *COMPILE* command, which requires that the FILEKIND attribute of the source file be CCSYMBOL, is the following:

```
c source-file-title
```

For compiling the work file, the short form syntax of the COMPILE command is:

```
c
```

The long form syntax of the CANDE *COMPILE* command, which works with files that have any FILEKIND value, is any of the following:

```
c source-file-title with cc
c source-file-title : cc
c source-file-title cc
```

If compiling the work file, the long form syntax of the CANDE *COMPILE* command is any one of the following:

```
c with cc
c : cc
c cc
```

Note that the last command prevents the source file "cc" from being compiled using the short form; the file title can be quoted to avoid ambiguity with the long form as shown in the following syntax:

```
c "CC"
```

The Default Code File Title

The default code file title is OBJECT/*source-file-title*. Errors are listed on the terminal and the source listing is not generated, unless specified otherwise through a compiler option.

For compiling a patch file against a source file, the title can be provided either through file equation or through the MERGE compiler control option. In the short form, where the work file is the patch file, use the following syntax:

```
c; cc file source = source-file-title
```

If the patch file already exists, use the following syntax:

```
c patch-file-title; cc file source = source-file-title
```

Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

Compiling from the Editor

The Editor customizes its environment based on the FILEKIND attribute. To compile from the Editor, FILEKIND should be set to CCSYMBOL.

Note: *Editor translates lowercase letters to their uppercase in commands, except inside strings that are delimited by double quotation marks(" "). Lowercase is used in these examples, but either case can be used.*

The current work file can be compiled in the Editor by the following command:

```
]comp
```

If any errors are detected by the compiler, the Editor positions the cursor on the first line that is in error. The command `]+err` can be used to move to the next line that is in error.

When compiling a patch file against a source file, the Editor automatically inserts the MERGE compiler control option and applies the correct file equation.

When the Editor finishes, if the code file represents the latest version, it is passed back to CANDE. The CANDE *SAVE* command saves both the source file and the code file.

Compiling from WFL

WFL does not customize its environment based on the FILEKIND attribute of the file; however, the FILEKIND should be set in case the file is used in other environments.

Note: *WFL requires that all commands be given in uppercase; there is no translation from lowercase to uppercase.*

When a file is compiled through WFL, you must provide a CARD file. If a SOURCE file is required (because the MERGE compiler control option is enabled), the title can be provided either through file equation or through the MERGE compiler control option. The CARD file defaults to KIND=READER; all other files default to KIND=DISK. The defaults can be overridden by file equation.

To compile an existing source file in WFL, use the following COMPILE statements:

```
COMPILE code-file-title WITH CC LIBRARY;
COMPILER FILE CARD(KIND=DISK, TITLE= source-file-title);
```

The file *source-file-title* is compiled and the object code is placed in file *code-file-title*. Although not required, the usual convention is to give *code-file-title* the title OBJECT/*source-file-title*.

To compile a patch file against a source file in WFL, use the following COMPILE statements:

```
COMPILE code-file-title WITH CC LIBRARY;
COMPILER FILE CARD(KIND=DISK, TITLE=patch-file-title);
COMPILER FILE SOURCE(TITLE=source-file-title);
```

To include the patch as part of the WFL job, use the following COMPILE statements:

```
COMPILE code-file-title WITH CC LIBRARY;
COMPILER FILE SOURCE(TITLE=source-file-title);
COMPILER DATA
patch card images

.
.
.

i
```

The *i* is an illegal WFL character (usually a question mark (?)).

When compiling under WFL, many of the compiler control options have different initial values than when compiling under CANDE. If the compiler control option LIST is not explicitly reset, a listing of the source, including any errors, is printed on the LINE file. If the ERRORLIST option is explicitly set, errors are printed on the ERRORS file. If the LIST option is reset and ERRORLIST is not set, errors are not printed. Refer to Section 10, “Compiler Control Options,” for more information on compiler control options and compiler files.

Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

Compiling POSIX Code

When you compile POSIX code, you must direct the compiler to locate the header files. Not all the headers are available internal to the compiler—headers new to POSIX are available only as external source files. Use the SEARCH compiler control option to direct the compiler to look for the files under specific directories.

For example, if the header files are located in the directory (CC)SYMBOL/CC/LIBRARY/= ON SYMBOLS, place the following line in the INITIALCCI file (or in the source file before the first `#include`) so the compiler can locate the external header files:

```
$SEARCH="(CC)SYMBOL/CC/LIBRARY/= on SYMBOLS"
```

The resulting object file, however, is not suitable for direct execution. The following actions must also be done:

- Compile the source file SYMBOL/CC/LIBRARY (usually installed in the same location as the header files). Also do the following:
  - Specify the same compiler control options when you compile the SYMBOL/CC/LIBRARY that you specified in the program.
  - Define the macros `_POSIX_SOURCE` and `_ASERIES_SOURCE` in SYMBOL/CC/LIBRARY the same way you defined them in the source file.
  - Compile both the source file and SYMBOL/CC/LIBRARY the same way (using the same compiler, etc.).
- Bind the object file SYMBOL/CC/LIBRARY with the one or more object files of the program.

Binding C Programs

A C program that consists of multiple C source files must be bound before it can be executed. A program is bound using the A Series Binder.

The Binder takes as input the following files:

- Two or more object files, which contain the separately compiled variable declarations and functions that are to be bound together. One of these object files must contain the procedure `main`.
- A file containing Binder statements, which instruct the Binder which object files are to be bound.

The Binder produces as output a code file consisting of the bound object files. An external variable or function is considered resolved if it has been successfully bound; otherwise it is considered unresolved. An unresolved reference is not necessarily fatal to the execution of the bound program; the program terminates with a run-time error only if the code containing the unresolved reference is executed.

Note that a bound code file cannot be used again as a Binder object file.

Object Files

The description of a variable or function in the object file in which it is referenced must match the description of the variable or function in the file in which it is defined. During binding, if a mismatch occurs between function or variable types or between the number or types of a function's arguments, the Binder issues an error message and leaves the reference to the variable or function in the bound code file unresolved.

The following rules are used by the Binder when performing type-checking on variables, functions, and function arguments:

- A single-word operand (`char`, `short`, `int`, `long`, `pointer`, `float`, `double`) matches any other single-word operand.
- A double-word operand (`long double`) matches any other double-word operand.
- An aggregate type (`array`, `struct`, `union`) matches any other aggregate type. The actual length is the greater of the two matched types.

Binder Statement File

The Binder statement file is the primary input file to the Binder. This file contains statements that tell the Binder which object files are to be bound, the settings of various Binder options, and other directives.

The following Binder statement directs the Binder to treat all functions and variables defined in *object-file-title* as candidates for binding:

```
BIND ? FROM object-file-title;
```

Given a C program that consists of separate source files F1/C, F2/C, F3/C, and MAIN/C, the following Binder input binds the corresponding object files into an executable code file:

```
BIND ? FROM OBJECT/F1/C;
BIND ? FROM OBJECT/F2/C;
BIND ? FROM OBJECT/F3/C;
BIND ? FROM OBJECT/MAIN/C;
```

Notes:

- For C programs, the BIND = syntax, as described in the *Binder Programming Reference Manual*, cannot be used.

- Be sure that the compiler control options BYTEADDRESS, DBLTOSNGL, and FARHEAP have the same setting for all the object files being bound; otherwise, the Binder issues an error.

- Also be sure that the MEMORY_MODEL compiler control option has the same setting for all of the object files being bound unless the FARHEAP compiler control option is enabled. When the FARHEAP option is enabled, it is possible to bind subprograms compiled with any memory model to a C host compiled with the TINY or SMALL memory model. It is also possible to bind subprograms compiled with the LARGE or HUGE memory model to a C host compiled with the LARGE or HUGE memory model.

Example 9–1 shows a sample C program, composed of two separate source files, that sorts a list of random numbers, followed by the Binder input to bind the objects into an executable code file.

The file SYMBOL/CC/LIBRARY can be compiled to create bindable versions of the standard library functions. If a standard library is included with the `#include` directive, it does not need to be bound; otherwise, a Binder statement of the form as shown in the following example is required:

```
BIND ? FROM SYSTEM/CC/LIBRARY;
```

SORTNUMS/C:

```
#include <stdio.h>
extern initialize (int *pa, int len);
extern sort (int *pa, int len);
main () {
   #define MAXNUM 100
   int a [MAXNUM], i;
   initialize (a, MAXNUM);
   sort (a, MAXNUM);
   printf ("sorted array: ");
   for (i = 0; i < MAXNUM; i++) printf (" %i", a [i]);
   putchar ('\n')
```

SORT/C:

```
#include <stdlib.h>
initialize (int *pa, int len) {
   int i;
   /* fill *pa with random numbers */
   for (i = 0; i < len; i++) {
      *pa = rand ();
      pa++;
   }
}

int compare (void *p1, void *p2) {
   if      (*(int *)p1 < *(int *)p2) return -1;
   else if (*(int *)p1 > *(int *)p2) return 1;
   else                              return 0;
}
sort (int *pa, int len) {
   /* sort *pa using qsort intrinsic from <stdlib.h> */
   qsort (pa, len, sizeof (int), &compare);
}

SORTNUMS/BIND:

BIND ? FROM OBJECT/SORT/C;
BIND ? FROM OBJECT/SORTNUMS/C;
```

Example 9–1. Sample C Program and Binder Input

In the previous example, the SYSTEM/CC/LIBRARY is the object file for SYMBOL/CC/LIBRARY. For more information on binding standard library functions, refer to Volume 2, "Headers and Functions."

See Also

Refer to the *Binder Programming Reference Manual* for more information on both binding C to C and binding from other languages to C.

Binding from CANDE

The CANDE *BIND* command is very similar to the COMPILE command. If a source file has a FILEKIND of BINDERSYMBOL, then you can invoke the Binder using the following syntax:

```
b source-file-title
```

If the work file has a FILEKIND of BINDERSYMBOL, the short form of the BIND command is

```
b
```

The default code title is OBJECT/*source-file-title*. This file contains the bound program. Using Example 9–1, the following command produces the executable code file OBJECT/SORTNUMS/BIND:

```
b sortnums/bind
```

Binding from WFL

Binding from WFL is very similar to compiling from WFL. The CARD file is the input file containing Binder statements; it is assumed to be KIND=READER unless specified otherwise.

To bind in WFL, use the following BIND statements:

```
BIND code-file-title BINDER LIBRARY;
BINDER FILE CARD (KIND=DISK, TITLE=source-file-title);
```

The file *source-file-title* contains the Binder input.

Alternatively, the Binder input can be supplied as part of the WFL job, shown as follows:

```
BIND code-file-title BINDER LIBRARY;
BINDER DATA
binder statements
?
```

Running C Programs

A program written in C can be run from many different environments, including the following:

- The batch environment of WFL
- User programs
- The interactive environments of CANDE, MARC, APLB, and the Editor

This section briefly describes how to run programs from some of these environments. Refer to the reference manuals for each environment for detailed information.

The object file from the C compiler can be run directly if the source file defines the procedure `main`. Calling a procedure that is not defined causes a run-time error. Referencing data that is not defined causes unpredictable results. A standard library function can be called without a bind step, provided its header file was included.

If the C source exists in separate source files or does not include the library headers, then the code file must be bound before it can be executed.

Program Parameters

When a C program is invoked, optional arguments to the program can be supplied on the command that invoked the program. These are called command-line-arguments and they are allowed only when the function `main` is declared with the `argc, argv` parameters. In this case, the C compiler generates a program with a REAL ARRAY parameter with an asterisk (*) lower bound. This is the parameter expected by the various environments that can supply command-line-arguments.

If `main` is declared with no parameters, the program that is generated does not have any parameters. The environments signal an error if such a program is invoked with command-line-arguments.

The C program parses the command-line-arguments into separate strings based on the rules described in Table 9–2. The documentation for each environment describes any editing done by the environment on the command-line-arguments before they are passed into the C program.

Note: *The redirection of the standard input and output files using input-file-title and output-file-title is currently not supported; the angle bracket and the file title are passed as parameters into the program. This might change in the future and users are urged not to rely on parameters with leading angle brackets being passed to the program.*

Command Line Parsing

The environment that invoked the C program first edits the command-line-arguments as described in "Running a C Source File from CANDE," "Running a C Source File from the Editor," "Running a C Source File from WFL," and "Running a C Source File from Other Environments," later in this section. Once the command-line-arguments are given to the C program, the program can specify one of four different parsing rules through the COMMANDLINE compiler control option. The default parsing rules are the WILD parsing rules.

Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

Table 9–2 describes the command line parsing rules.

Table 9–2. Command Line Parsing Rules

Parsing Rules	Used by	Description
CANDE Parsing Rules		No additional parsing is done. The entire command line argument (possibly empty) is provided as one string pointed to by `argv[1]`.
ORIGINAL Parsing Rules	These parsing rules are used by the 1.0 and 3.9 C compiler.	1. Parse the longest "reasonable" file title, where reasonable means ignore the length limit of 17 characters in a node and limit of 12 nodes in a title and allow lowercase letters anywhere. 2. Parse the longest "word", where a word is one of the following: – A single quoted (`' '`) string: The quotes are removed. – A double-quoted (`" "`) string: The quotes are preserved. – All characters before a blank or one of the following: ; & ( ) | < > ^ - \ 3. Take the longest string pointed to by `argv`.

Table 9–2. Command Line Parsing Rules

Parsing Rules	Used by	Description
MINIMAL Parsing Rules	These parsing rules modify the ORIGINAL parsing rules to conform to the proposed POSIX 1003.2 standard. While the rules are described quite differently, most strings would be parsed the same way.	1. Parse the longest word. A word consists of a single quoted string (' '), double-quoted string (" "), double ampersand (&&), double vertical bar (||), double greater than (>>), or string of characters that does not contain any of the following characters: `&|<>'";` or a space Remove the quotation marks around a single quoted string. 2. Concatenate words until one of the following characters is detected: `&|<>;` or a space If compiler control option $TITLESYNTAX is set to TITLE, adjacent arguments can be combined. If a space is detected and the next word is on ignoring case and the following word is a valid family name, concatenate spaces, on, and family name. If a space is detected and the next word is at ignoring case and the following word is a valid host name, concatenate spaces, at, and host name. 3. The next `argv` element points to the concatenated string.
WILD Parsing Rules	These parsing rules extend the MINIMAL parsing rules with wild character expansion for file titles. Wild character expansion occurs when a question mark (?), tilde (~), or equal sign (=) appears outside of single quoted strings (' '). The file title is used as a pattern to find matching titles of existing files.	1. A question mark (?) matches any single character except the slash (/). 2. A tilde (~) matches any string of characters except the slash. The empty string is considered a match. 3. An equal sign (=) matches any string of characters, including the slash. An empty string is considered a match.

Examples of Parsing Rules

The following pages show examples of the ORIGINAL, WILD, and MINIMAL parsing rules.

ORIGINAL Parsing Rules Examples

File titles can contain optional blanks between nodes and blanks are required around the `ON` and `AT` keywords. Furthermore, a double-quoted (`"`) string can be a reasonable file title; if it is, the double quotation marks are preserved.

The following examples show which `argv` values are produce when input is parsed according to the ORIGINAL parsing rules:

Example 1

This input:

```
u foo (")(")"()" on at at on
```

produces these `argv` values:

```
argv[1] = (")(")"()" ON AT AT ON
                /* Where usercode is ")("  */
                /*       filename is "()"  */
                /*       family name is AT */
                /*       host name is ON   */
```

Example 2

This input:

```
r foo("on on on on on at at at at at")
```

produces these `argv` values:

```
argv[1] = on on on
argv[2] = on on at at at
argv[3] = at
argv[4] = at
```

Example 3

This input:

```
u foo abc,def<hij>klm
```

produces these `argv` values:

```
argv[1] = on on on
argv[1] = ABC,DEF
argv[2] = <HIJ
argv[3] = >KLM
```

MINIMAL Parsing Rules Examples

The following examples show which `argv` values are produce when input is parsed according to the MINIMAL parsing rules:

Example 1

When $TITLESYNTAX is TITLE, this input:

```
r foo("a/b on c at d a / b on")
```

produces these `argv` values:

```
argv[1] = a/b on c at d
argv[2] = a
argv[3] = /
argv[4] = b
argv[5] = on
```

Example 2

When $TITLESYNTAX is either PATHNAME or NXPATHNAME, this input:

```
r foo("a/b on c at d a / b on")
```

produces these `argv` values:

```
argv[1] = a/b
argv[2] = on
argv[3] = c
argv[4] = at
argv[5] = d
argv[6] = a
argv[7] = /
argv[8] = b
argv[9] = on
```

Example 3

This input:

```
u foo abc,def<hij>klm
```

produces these `argv` values:

```
argv[1] = ABC,DEF
argv[2] = <
argv[3] = HIJ
argv[4] = >
argv[5] = KLM
```

WILD Parsing Rules Examples

If matching occurs, the original pattern is replaced by one or more strings, each representing a file title. If no matching occurs, the original pattern is passed as the string.

Wild character expansion does not occur in the family name or host name. If a host name is specified, wild character expansion will not occur anywhere.

Assume the following files exist:

```
A           AB          AC          ABC
A/B         A/B/C       A/C
B/C
```

The following inputs to CANDE produce the following `argv` values:

Inputs	**Resulting `argv` Values**
`u foo ?`	`argv[1] = A` `argv[2] = B`
`u foo A?`	`argv[1] = AB` `argv[2] = AC`
`u foo ~B`	`argv[1] = AB`

`u foo =B=`	`argv[1] = A/B`
	`argv[2] = A/B/C`
	`argv[3] = B/C`
	`argv[4] = AB`
	`argv[5] = ABC`

Running a C Source File from CANDE

The following table shows the commands for running C source files that have been compiled into different code files from CANDE:

If a C source file exists as . . .	**And is compiled into the code file . . .**	**Then run the file from CANDE using . . .**
`NOARG/C`	`OBJECT/NOARG/C` Assume that `main` is defined with no arguments.	`r noarg/c`
`ARGS/C`	`OBJECT/ARGS/C` `main` is defined with the `argc`, `argv` parameters.	`r args/c ("command-line -arguments")`
`ARGS/C` (alternative)	`OBJECT/ARGS/C` `main` is defined with the `argc`, `argv` parameters.	`u args/c command-line-a rguments`

Lowercase letters and uppercase letters are preserved. The double quote character (") cannot appear within the command-line-arguments.

A string within double quote characters (" ") has blanks and lowercase letters preserved. Outside of double-quoted strings, lowercase letters become uppercased. A string within single quote characters (' ') is not interpreted as possible commands by CANDE. If the command-line-arguments start with a single quote, it must end with the single quote; embedded single quotes are not permitted, but double quotes are permitted.

Note: *Running POSIX source files requires special handling. See "Running POSIX Source Files" in this section for more information.*

The following examples show which argv values are produced when the C source file is run from CANDE:

Example 1

This input:

```
u args/c 'help "help" "on" pack'
```

produces these argv values:

```
argv[1] = HELP     /* uppercased */
argv[2] = "help"   /* file title */
argv[3] = "on"     /* file title */
argv[4] = PACK     /* uppercased */
```

Example 2

This input:

```
u args/c eat at joe's
```

produces these argv values:

```
argv[1] = EAT AT JOE /* file title with host name */
argv[2] = 'S         /* mismatched quoted string  */
```

Running a C Source File from the Editor

The following table shows the commands for running C source files that have been compiled into different code files from the Editor.

If the current work file is . . .	And is compiled into the code file . . .	Then run the file from the Editor using . . .
NOARG/C	OBJECT/NOARG/C Assume that main is defined with no arguments.	]run
ARGS/C	OBJECT/ARGS/C main is defined with the argc, argv parameters.	]run]run "command-line-arguments"

The following table shows the commands for running C source files that have not been compiled into different code files from the Editor.

If the current work file is not . . .	Then run the file from the Editor using . . .
NOARG/C	]run noarg/c
ARGS/C	]run args/c]run args/c "command-line-arguments"

Lowercase letters and uppercase letters are preserved. The double quote character (") cannot appear within the command-line-argument*s*.

Running a C Source File from WFL

The following tables shows the commands for running C source files that have been compiled into different code files from WFL.

If a C source file exists as . . .	And is compiled into the code file . . .	Then run the file from CANDE using . . .
NOARG/C	OBJECT/NOARG/C Assume that main is defined with no arguments.	RUN OBJECT/NOARG/C
ARGS/C	OBJECT/ARGS/C main is defined with the argc, argv parameters.	RUN OBJECT/ARGS/C("command-line-arguments")

Lowercase letters and uppercase letters are preserved. The double quote character (") cannot appear within the command-line-arguments.

A C program that references an A Series run-time library can be run from WFL using the following command:

```
PROCESS RUN OBJECT/PROGRAM/C
```

An error is generated if a RUN is requested instead of a PROCESS RUN.

Note: *Running POSIX source files requires special handling. See "Running POSIX Source Files" in this section for more information.*

Running POSIX Source Files

When compiling for POSIX, programs that are not libraries behave as if `main` is always declared with the `argc` and `argv` arguments. A program is a library when one or more functions are defined with the `asm` storage class. The same rules apply to running POSIX libraries as non-POSIX libraries.

Running from CANDE

To run a POSIX source file in CANDE, use the following syntax even though the parameters are ignored:

```
r foo/c("parameters")  or  u foo/c parameters
```

Running from WFL

To run a POSIX source file from WFL, use the following syntax even though the parameters are ignored:

```
RUN OBJECT/FOO/C("parameters")
```

Running a C Source File from Other Environments

The C program is invoked just like any other program with either no parameters or a single REAL ARRAY with star lower bounds parameter. For information, refer to the documentation for each environment.

File Equation

Using standard file equation, the standard input, output, and error files can be redirected from a remote file to some other system file. The file equation can be done at run-time and only affects that run or it can be done at compile-time and affects all runs. Refer to the reference manuals for each environment for information on how to apply file equation at compile-time.

In the CANDE and WFL environments, apply file equation at run-time by following the command with a semicolon (`;`) and the file equation list. For example, in CANDE, enter the following:

```
run noargs/c; file stdout(kind=printer)
run args/c("Hello world"); file stdin(kind=disk, title=script)
u args/c 'arg1;arg2'; file stderr(kind=printer)
```

The INTNAMEs for the three standard files are `stdin`, `stdout`, and `stderr` and can be file equated as previously shown. Files that are explicitly opened are randomly assigned an INTNAME and cannot directly be file equated. The `fopen` function enables attributes to be specified and one of the attributes can be INTNAME. If INTNAME is set, the file

attributes are reset to their default values and then any file equation specified for that INTNAME is applied. (The specifications of attributes is an extension of A Series C.) For example, if the following `fopen` function is specified:

```
fopen("PRINT", "w,INTNAME=LPR");
```

Then, you can run the program with file equation applied to LPR as shown in the following example:

```
run;file lpr(kind=printer);
```

To change any of the standard files from a remote file to a disk file, both the KIND and TITLE attributes need to be specified. For explicitly opened files, only the TITLE attribute is required; the KIND attribute defaults to disk.

Section 10
Compiler Control Options

Compiler control options affect the compilation process in the following ways:

- Instructs the compiler to read optional input files
- Produces optional output files
- Includes or excludes certain source records
- Specifies a character set
- Designates a target machine

Using Compiler Control Records

Compiler control options are specified on compiler control records (CCRs). CCRs begin with a dollar sign ($) at the start of the input record (and are hence referred to as "dollar cards" or "$-cards"). Records with a dollar sign only in the first column of the text field (that is, column 1 for C source files) are considered "temporary", while records with a blank in the first column and a dollar sign in the second column or a double dollar sign ($$) in the first two columns are considered "permanent." Permanent records are written to the new source file (NEWSOURCE) and the program listing, while temporary records are not (the LISTDOLLAR option causes temporary records to be written to the program listing).

Following the dollar signs is a succession of option clauses. A single CCR may contain as many clauses as will fit; one CCR with several clauses is logically equivalent to several CCRs each containing a single clause.

A CCR consisting of a single dollar sign character and no option clauses is called a "null option record" and has no effect; such a record can be used to delete records from the secondary input file when merging. Refer to "MERGE Option" in this section for more details.

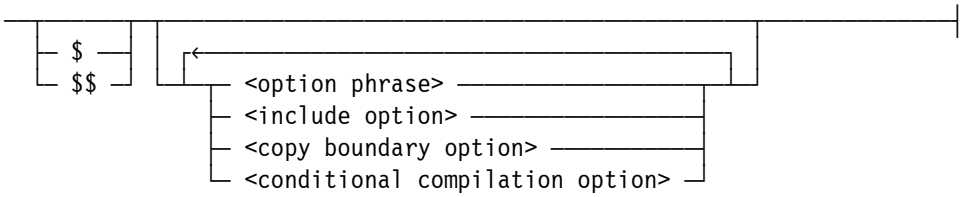
Options and keywords in CCRs may appear in any combination of uppercase and lowercase characters.

A percent sign (%) is used to introduce comments on a CCR; the text following the % sign is ignored. A % sign occurring inside a string is treated as a character in the string, not a comment delimiter.

Syntax

The syntax for a CCR is defined as follows:

<compiler control record>



Explanation

<option phrase>	The <option phrase> provides a means for setting the values of compiler options.
<include option>	The <include option> temporarily redirects the input stream to an alternate source.
<copy boundary option>	The <copy boundary option> defines regions of a source file that can be used by an <include option>.
<conditional compilation option>	The <conditional compilation option> provides a convenient method for optionally omitting source images.

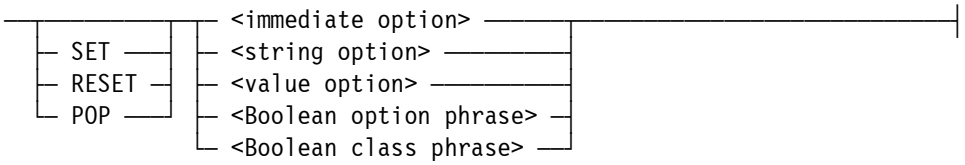
The syntax for each of these options is outlined in the alphabetical listing of compiler options.

Using the Option Phrase

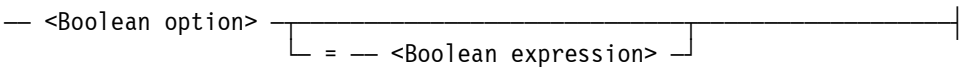
The option phrase provides additional features that apply to compiler control options.

Syntax

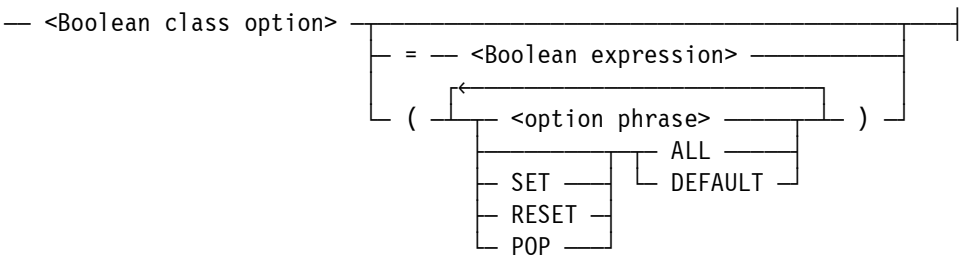
<option phrase>



<Boolean option phrase>



<Boolean class phrase>



Explanation

The options of the option phrase are discussed in the following pages.

The Immediate Option

An immediate option causes the compiler to perform an action independent of subsequent processing, for example, causing a page eject in a program listing.

The String Option

A string is a sequence of characters delimited at either end by a single or double quote. The same delimiter must be used at each end of the string and the string may not contain the delimiter as an embedded character. The contents of the string are converted to uppercase characters; characters that are not printable EBCDIC characters are translated to a question mark (**?**). If string is to be used as a file title, all backslashes (****) and periods (**.**) are changed into slashes (/). Refer to "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

A string option associates an internal compiler variable with a string value. String options may either be set to a value, using the = syntax, or may have a string concatenated onto the end of its present value, using the += syntax.

Examples

The first two string options are equivalent to the third in the following examples:

```
$ XXX  = "ABC"
$ XXX  += "DEF"
$ XXX = "ABCDEF"
```

Some Boolean options may also have an associated <string> value. In the following example, the MERGE option is enabled and the title of the merge file is set to BLIT:

```
$SET MERGE = "BLIT"
```

Value Option

A <value option> sets the value of an internal compiler variable, for example, setting the number of errors that can be encountered before terminating a compilation.

Boolean Option

A Boolean option phrase can enable a Boolean option (set it to TRUE), disable a Boolean option (set it to FALSE), or restore the setting of a Boolean option to a prior value. The presence of a SET, RESET, or POP keyword establishes a mode that controls the setting of the Boolean options that follow. The compiler control options (also called dollar-card options) are initially processed in the SET mode. For example, the following dollar-card options are equivalent; they each set option XYZ and reset option ABC:

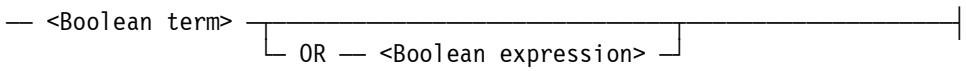
```
$ SET XYZ RESET ABC

$ XYZ RESET ABC
```

Each Boolean option is managed as a stack of up to 31 values plus the current value. When a Boolean option is SET or RESET, the option is enabled or disabled, respectively, and the previous setting is pushed onto the stack. When a Boolean option is the object of a POP action, the current setting is discarded and the previous setting is popped off the stack. If more than 31 values are pushed, the oldest are lost.

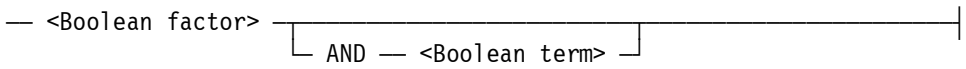
Syntax

The syntax for a <Boolean expression> is as follows:

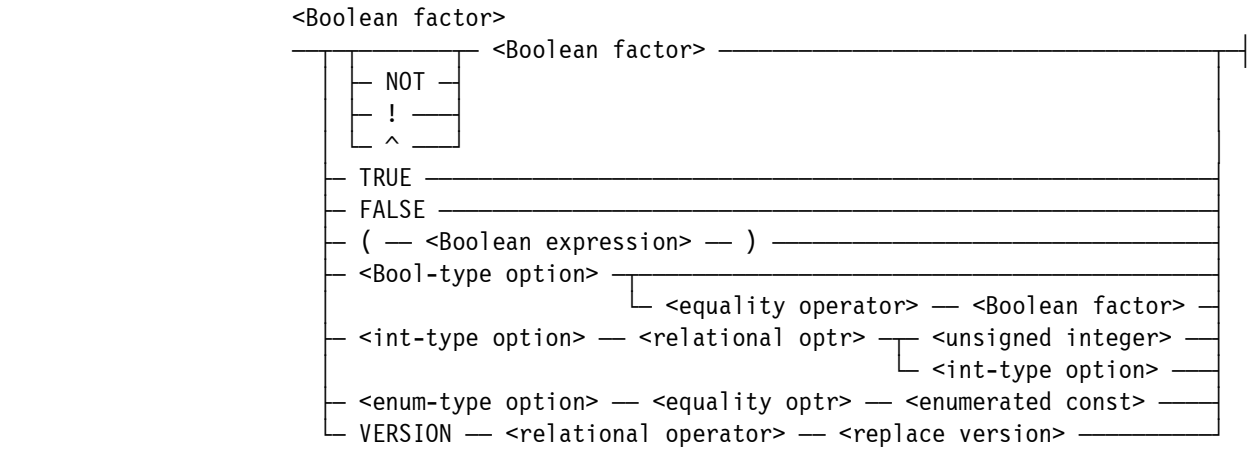
<Boolean expression>



<Boolean term>



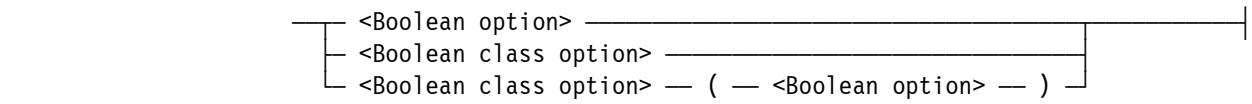
<Boolean factor>



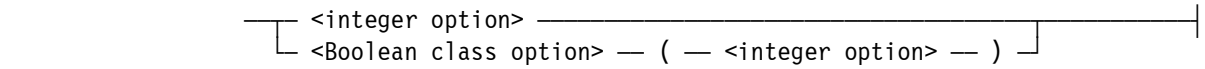
<replace version>

Refer to "VERSION Option" later in this section for a description of <replace version>.

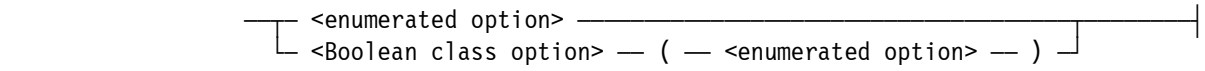
<Bool-type option>



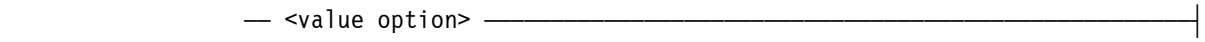
<int-type option>



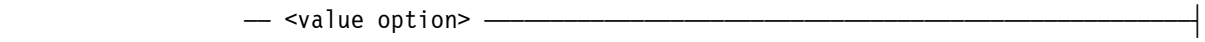
<enum-type option>



<integer option>



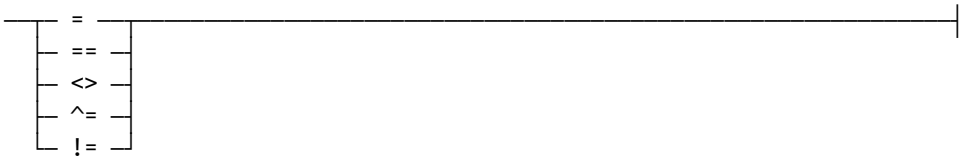
<enumerated option>



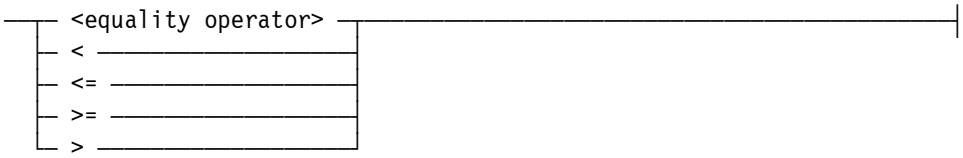
<enumerated constant>

An enumerated constant is a symbolic value that is assignable to the previous enumerated option.

<equality operator>



<relational operator>



The SET and RESET values are treated as TRUE and FALSE respectively for the purposes of evaluating the Boolean expression. Note that when a Boolean option is followed by the = <Boolean expression> clause and the mode is SET, the option is set to the result of the Boolean expression rather than the constant value TRUE.

Boolean Class Option

A Boolean class phrase is used to set the value of a Boolean option and its associated subordinate options. The Boolean option may be followed by a parenthesized list of option phrases, each phrase referencing suboptions of the class option. If a parenthesized list does not appear, the suboptions are not affected. For example, the following option phrase enables the OPTIMIZE option, but disables the optimization suboption UNRAVEL:

```
$SET OPTIMIZE (RESET UNRAVEL)
```

The following option phrase enables the OPTIMIZE option, but leaves the suboptions at their current settings:

```
$SET OPTIMIZE
```

If ALL appears in the option phrase list, then all of the Boolean suboptions are set to the current mode. For example, the following option phrase enables the OPTIMIZE option and all of its Boolean suboptions:

```
$SET OPTIMIZE (ALL)
```

The following option phrase enables the OPTIMIZE option, but disables all of its Boolean suboptions:

```
$SET OPTIMIZE (RESET ALL)
```

If DEFAULT appears in the option phrase list, then all suboptions are restored to their initial default values. For example, the following option phrase enables the OPTIMIZE option and sets all of its suboptions to their default values:

```
$SET OPTIMIZE (DEFAULT)
```

Reserved Compiler Control Options

The following options, though not currently used by the C compiler, are reserved:

AUTOBIND
BINDER
CLEAR
CODEGEN
CORRECTOK
CORRECTSUPR
DEBUG
LANGUAGE
LONG
MAKEHOST
OPTIMIZER
PATH
SEPARATE
SEPCOMP
SUPPORT

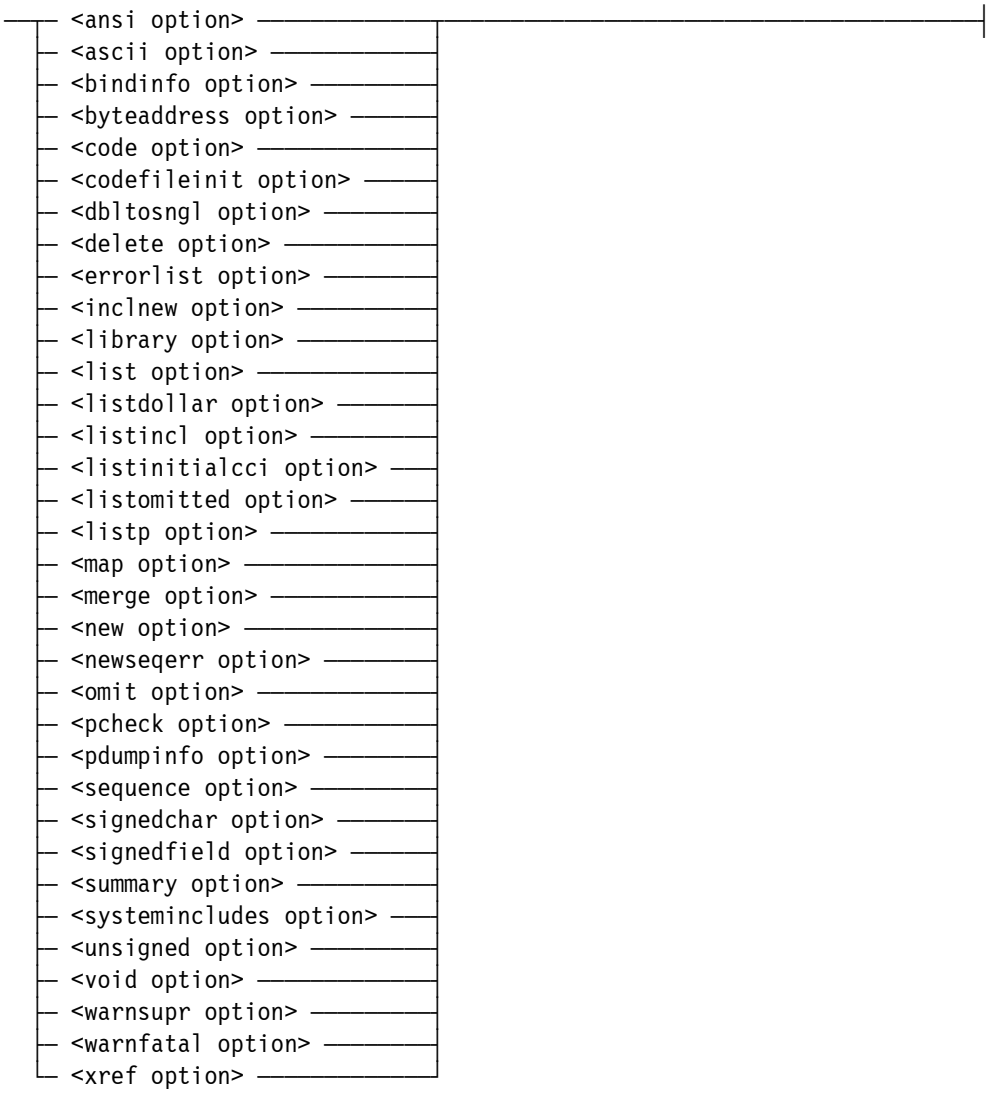
Types of Compiler Control Options (CCRs)

There are six types of compiler control options:

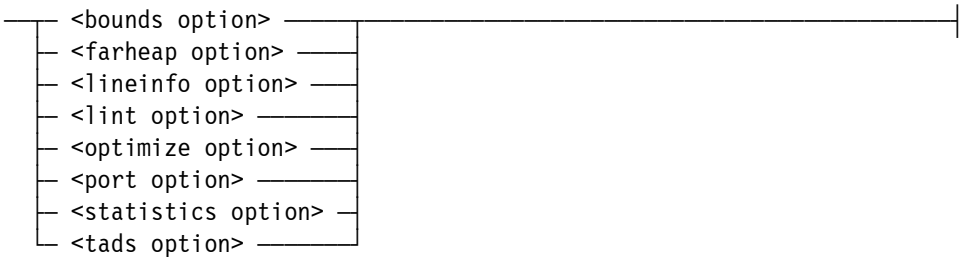
Compiler Control Option Types	Description
Boolean	A Boolean option is either enabled (set to TRUE) or disabled (set to FALSE). When the Boolean option is set to TRUE, the compiler applies the option to all subsequence processing until the option is set to FALSE.
Boolean class	A Boolean class option is a Boolean option with subordinate options. It is used to enable a general feature. When the Boolean class option is set to TRUE, the subordinate options control specific details of the general feature. When the Boolean class option is set to FALSE, the general feature is disabled and all the subordinate options are ignored.
Boolean title	A Boolean title option sets the value of a Boolean option and optionally associates a file name with the option.
Immediate	An immediate option is applied by the compiler when the option is encountered in source code. The function performed by an immediate option is independent of any subsequence processing by the compiler. Immediate options can have associated parameters.
String	A string option is an option to which a string is associated. The string can be delimited by either single or double quotation marks, so long as they are used consistently. A string delimited at the beginning by a single quotation mark must be delimited at the end by a single quotation mark.
Value	A value option stores a specific value that the compiler uses when applying the option.

The following is a grouping of the compiler options by type. An alphabetical listing of the individual options and their syntax comprise the rest of this section. Each entry includes the syntax, type, default, and a description of the syntax of the option.

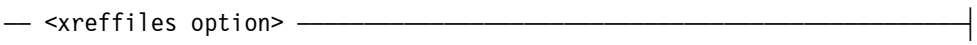
<Boolean option>



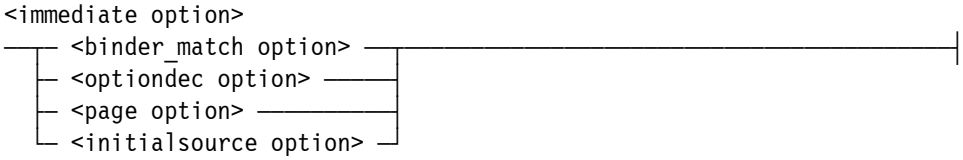
<Boolean class option>



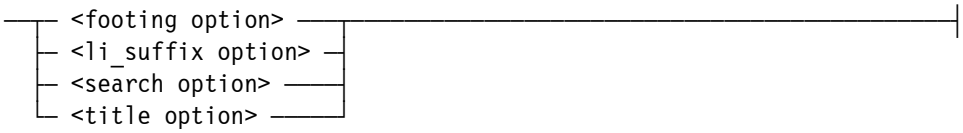
<Boolean title option>



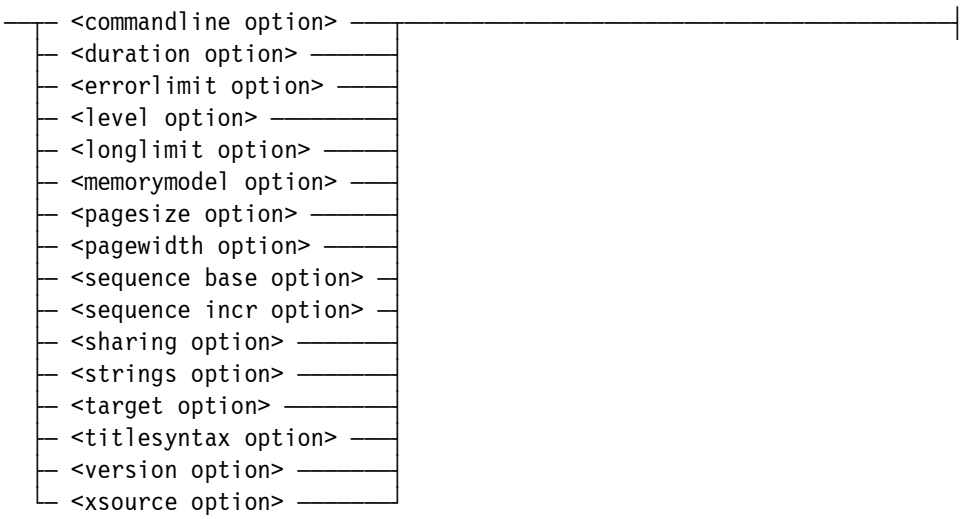
<immediate option>



<string option>



<value option>



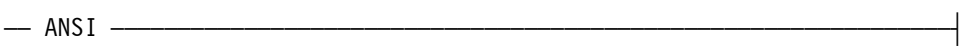
Syntax for Compiler Control Options

The following is an alphabetical listing of the individual compiler control options and their syntax.

ANSI Option

The ANSI option, when enabled, causes a syntax error to be issued for source code that does not conform to the ANSI C standard.

<ansi option>



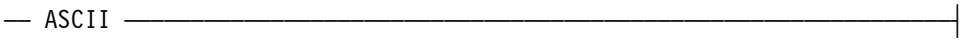
(Type: Boolean; Default: FALSE)

In order for the execution of a C program to conform to the ANSI C Standard, the PORT(CHAR2) and PORT(UNSIGNED) compiler control options must be set to TRUE. If the STRINGS compiler control option is set to EBCDIC (the default), then the PORT(SIGNEDCHAR) compiler control option must be left with its default value set to FALSE.

ASCII Option

The ASCII option, when enabled, specifies that the ASCII character set is to be used to represent the strings in the object program. The ASCII option should appear before any program text is encountered.

<ascii option>



(Type: Boolean; Default: FALSE)

Example

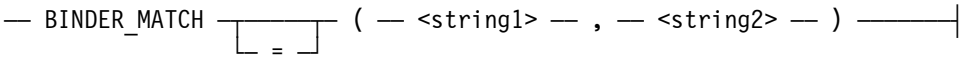
```
$SET ASCII is equivalent to $STRINGS = ASCII.
```

Note: *Care should be taken when invoking this option in conjunction with the* `setlocale` *function for locales other than the C language locale. For example, selecting ASCII strings for a program that calls* `setlocale` *to select a locale whose base character set is not ASCII may produce undesirable results.*

BINDER_MATCH Option

The BINDER_MATCH option is used to verify that different object files were compiled with the same set of compile-time options.

<binder_match option>



(Type: Immediate)

Setting the BINDER_MATCH option adds two strings to the object file. When two BINDER_MATCH options have identical first strings (string1), the Binder verifies that the second strings (string2) are also identical. If not, the Binder prints an error message and the bind is aborted. Strings must match exactly, including uppercase and lowercase letters.

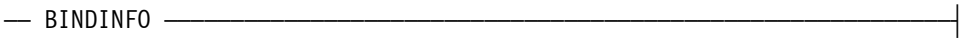
The first string should not begin with an asterisk (*). First strings that begin with an asterisk are for use by Unisys only.

You can specify multiple BINDER_MATCH options. Each option is stored and tested separately. A compile-time error occurs when there are two BINDER_MATCH options in the same file with the same first string and different second strings.

BINDINFO Option

Setting the BINDINFO option causes the compiled code file to be suitable for use by the Binder and the Printbindinfo Utility. This option should appear before any program text. Normally, this option need only be set if binding is required for a standalone program (that is, a program with no external references).

<bindinfo option>

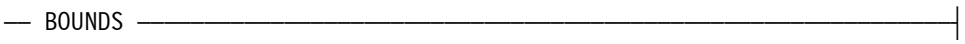


(Type: Boolean; Default: TRUE if `main` is not defined or if there are external references; otherwise FALSE)

BOUNDS Option

The BOUNDS option, when disabled, prevents the compiler from generating the range-checking code that it would otherwise normally generate.

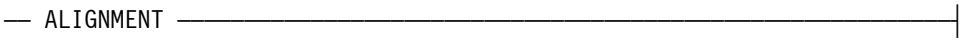
<bounds option>



(Type: Boolean class; Default: TRUE)

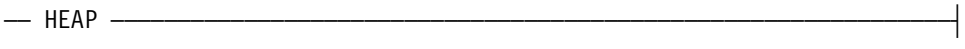
The BOUNDS option has the following subordinate options:

- ALIGNMENT
- HEAP
- STACK



(Type: Boolean; Default: FALSE)

The ALIGNMENT subordinate option controls the generation of code that traps alignment errors during pointer type conversions. For example, when a pointer is cast into a pointer with a more strict alignment, the resulting pointer might not be aligned with the original pointer. The ALIGNMENT subordinate option traps this error.



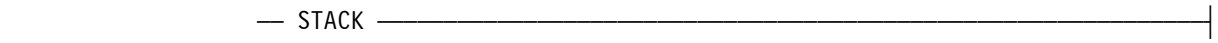
(Type: Boolean; Default: TRUE)

The HEAP subordinate option traps memory accesses when C heap control structures have become invalid.

When enabled, memory allocation or deallocation will cause a C program to fault with "DYNAMIC ALLOCATION AREA CORRUPTED", if the heap control structures are invalid due to a corrupted heap. The FARHEAP option must be set for the HEAP subordinate option to have any affect.

There will be a slight performance improvement in C heap allocation and deallocation if this subordinate option is turned off.

Note: *When this subordinate option is disabled it is possible for the C heap manager to return invalid data or enter an endless loop due to corrupted heap control structures. It is highly recommended that this option remain enabled until a program is fully debugged.*



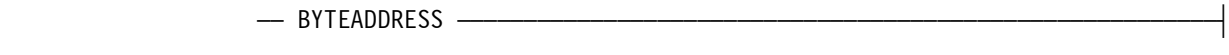
(Type: Boolean; Default: TRUE)

The STACK subordinate option controls the generation of code that traps overflow of the software stack. Refer to "MEMORY_MODEL Option" in this section for details on the software stack.

BYTEADDRESS Option

The BYTEADDRESS option, when enabled, causes all pointers to types `int`, `float`, `long double`, and `struct` to be the number of bytes from the start of addressable memory. When disabled, pointers to `int` and `float` are the number of words, and pointers to `long double` and `struct` are the number of double words, from the start of addressable memory.

<byteaddress option>



(Type: Boolean; Default: TRUE)

With BYTEADDRESS enabled, there is a performance gain for most programs on MCP systems. The amount of gain is dependent on the structure of the programs.

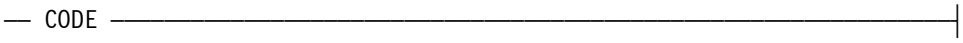
Modules with the BYTEADDRESS option enabled may not be bound with modules with the BYTEADDRESS option disabled.

When a C module is compiled with the BYTEADDRESS option enabled, there is possible impact on the use of pointers to `struct`, `long double`, `float`, and `int` data items when passed outside of C, such as to a non-C library. To use these pointers as word addresses or offsets, the pointers must be divided by 6.

CODE Option

If both the LIST and CODE options are enabled, the object code generated by the compiler is included in the program listing. The CODE option is ignored if the LIST option is disabled.

<code option>

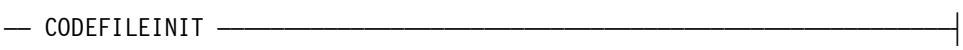


(Type: Boolean; Default: FALSE)

CODEFILEINIT Option

The CODEFILEINIT option is used to specify how variables with constant initializers are assigned. If the option is disabled, each variable, array element, or structure field is assigned individually. If the option is enabled, the initializers of an array or structure are read in as a block from the code file.

<codefileinit option>



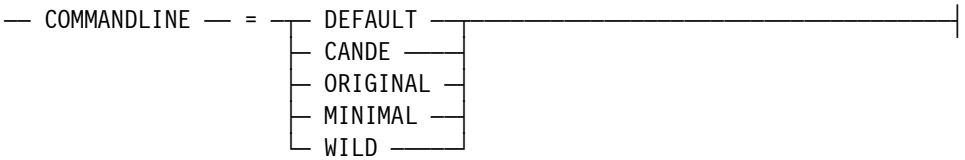
(Type: Boolean; Default: TRUE)

For optimal performance, programs with few array or structure initializers should disable this option; programs with a large number of array or structure initializers should enable this option.

COMMANDLINE Option

The COMMANDLINE option is used to specify how the command line arguments are parsed.

<commandline option>



(Type: Value; Default: DEFAULT)

CANDE	CANDE specifies no parsing: the entire command line argument is provided as one string.
ORIGINAL	ORIGINAL specifies the parsing rules used by the 1.0 and 3.9 versions of the C compiler.
MINIMAL	MINIMAL specifies the POSIX conforming parsing rules.
WILD	WILD specifies the POSIX conforming parsing rules with wild character expansion for file titles.

For more information on the various parsing rules, refer to “Command Line Parsing” in Section 9.

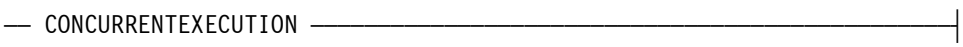
Refer to “TITLESYNTAX Option,” for more information on the file title changes that occur during compilation.

CONCURRENTEXECUTION Option

The CONCURRENTEXECUTION option should be set if more than one task executes within a program or library at the same time. A C Program that is compiled with $SET CONCURRENTEXECUTION and #define _ASERIES_SOURCE 461 (or larger) can start and manage threads.

Note: *Threads are functions initiated by a program that not only run concurrently with the program but also share the same data as the program.*

<concurrentexecution option>



(Type: Boolean; Default: TRUE if SHARING = SHAREDBYALL, otherwise FALSE)

Setting this option causes system functions, such as memory allocation, to be protected from concurrent tasks. Failing to set the option when there are concurrent tasks can result in data corruption or program failure.

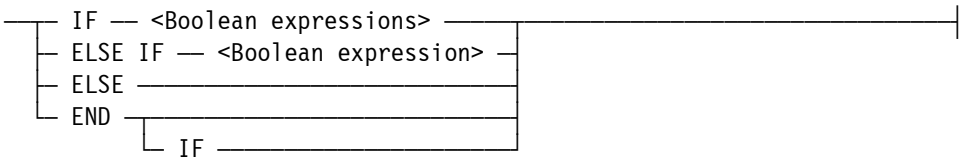
This option should be set before any program text, and in separately compiled modules that are bound into programs supporting concurrent execution.

If the CONCURRENTEXECUTION option is set or $SHARING = SHAREDBYALL, the MEMORY_MODEL compiler option must be SMALL or HUGE.

CONDITIONAL COMPILATION Options

The CONDITIONAL COMPILATION options are used to conditionally include or omit certain source records in the compilation. These options serve the same function as the OMIT option, but are more intuitive.

<conditional compilation options>



(Type: Immediate)

If the Boolean expression in the IF clause is TRUE, the records between the IF and a subsequent ELSE IF, ELSE, or END are compiled and all records between any subsequent ELSE IF, ELSE, and END cards are ignored. Similarly, if the IF clause is FALSE, but a subsequent ELSE IF clause is TRUE, then the records between the ELSE IF and subsequent ELSE or END are compiled, with the other cards omitted. If all IF and

ELSE IF clauses are FALSE, the records between the ELSE and END are compiled. (Note that the ELSE IF and ELSE clauses are optional.)

Compiler control records encountered in the source language input between any IF, ELSE IF, ELSE, or END compiler control option are always processed in the normal fashion, regardless of the value of the Boolean-expression in the IF option.

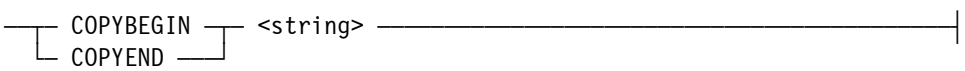
In the following example, if OP1 or OP2 is enabled, records 1 through 10 are compiled. Otherwise, if OP3 is enabled, records 11 through 16 are compiled; otherwise, records 17 through 20 are compiled.

```
$ IF (OP1 OR OP2)
     <source records 1-10>
$ ELSE IF OP3
     <source records 11-16>
$ ELSE
     <source records 17-20>
$ END
```

COPY BOUNDARY Option

The COPYBEGIN and COPYEND options are used to symbolically demarcate regions of a file.

<copy boundary option>



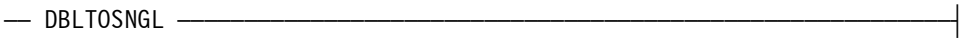
(Type: Immediate)

If the string matches that which is used in an INCLUDE, the region of the file between the COPYBEGIN and COPYEND records is included. Otherwise, COPYBEGIN and COPYEND cards are treated as comments. Refer to “INCLUDE Option” for more information.

DBLTOSNGL Option

When the DBLTOSNGL option is enabled, variables of type `double` are given the same type and range as those of type `float.` When disabled, variables of type `double` are given the same type and range as those of type `long double`.

<dbltosngl option>



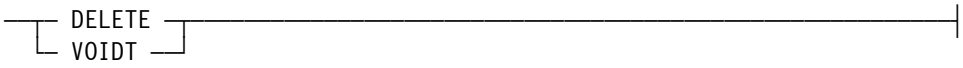
(Type: Boolean; Default: TRUE)

If the range of type `float` is suitable for variables declared with type `double`, then this option should be left enabled for improved run-time performance. This option should appear before any program text.

DELETE Option

When the DELETE option is enabled, all source records from the secondary input file (SOURCE) are ignored until the option is disabled. DELETE has no effect unless the MERGE option is enabled. VOIDT is a synonym for DELETE.

<delete option>



(Type: Boolean; Default: FALSE)

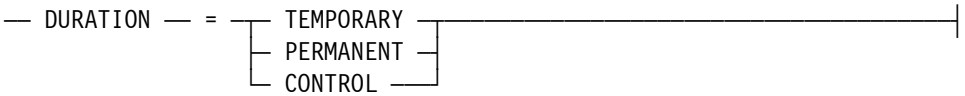
The source records that are discarded as a result of this option are not written to the new source file (NEWSOURCE) if NEW is enabled, nor do they appear in the program listing.

Note: *This option can appear only on a compiler control record in the primary input file (CARD).*

DURATION Option

The DURATION option specifies the lifetime of a C library. This option should appear before any program text.

<duration option>



(Type: Value; Default: TEMPORARY)

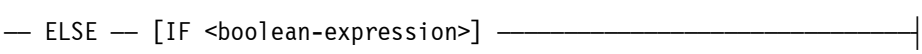
TEMPORARY	The library "thaws" when it has no users linked to it.
PERMANENT	Once the library is initiated, it runs even though there are no users linked to it. The THAW system command will make the library temporary.
CONTROL	The entry point `main` is called after the library is frozen. If the library exits `main`, it is treated as a temporary library.

Refer to the *System Commands Operations Reference Manual* for more information on the THAW system command.

ELSE and ELSE IF Options

The ELSE and ELSE IF options are conditional compilation options used to conditionally include or omit certain records in the compilation. Compiler control options encountered in the source language input between any IF, ELSE IF, ELSE, END, or END IF compiler control options are always processed in the normal fashion, regardless of the value of the Boolean-expression of the IF option.

<else options>



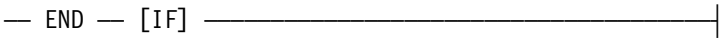
(Type: Immediate)

Refer to "IF Option" for a more detailed description of the conditional compilation options.

END and END IF Options

The ELSE and ELSE IF options are conditional compilation options used to conditionally include or omit certain records in the compilation. Compiler control options encountered in the source language input between any IF, ELSE IF, ELSE, END, or END IF compiler control options are always processed in the normal fashion, regardless of the value of the Boolean-expression of the IF option.

<end options>



(Type: Immediate)

Refer to “IF Option” for a more detailed description of the conditional compilation options.

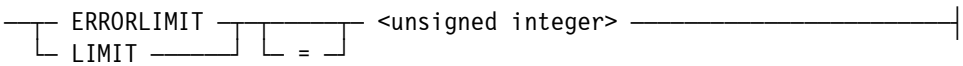
ERRLIST Option

ERRLIST is a synonym for ERRORLIST. Refer to “ERRORLIST Option” for details.

ERRORLIMIT Option

The ERRORLIMIT option specifies the number of errors that can occur before a compilation is terminated.

<errorlimit option>



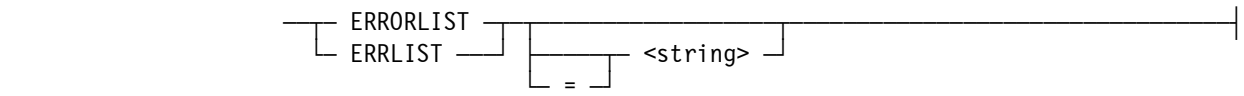
(Type: Value; Default: 10 if compilation is initiated interactively; otherwise 100)

If the error limit is exceeded when the NEW option is enabled, the new source file (NEWSOURCE) is not created. The error limit is ignored by setting ERRORLIMIT to 0.

ERRORLIST Option

When the ERRORLIST option is enabled, error and warning messages are written to the error file (ERRORS). If a <string> is associated with the option, the error file is given the file title specified by the <string>.

<errorlist option>



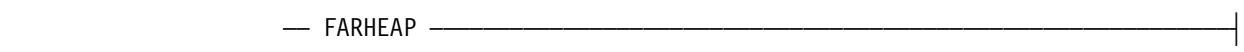
(Type: Boolean; Default: TRUE if the compilation is initiated interactively; otherwise FALSE)

If this option is enabled and LIST is disabled, error and warning messages appear only in the error file. If both this option and LIST are enabled, error and warning messages appear in both the error file and the program listing (LINE).

FARHEAP Option

The FARHEAP option enables a program or library to select the far heap management mechanism instead of the default heap management mechanism. This option must be set before any program text.

<farheap option>



(Type: Boolean class; Default: TRUE)

You should set the FARHEAP option in the following cases:

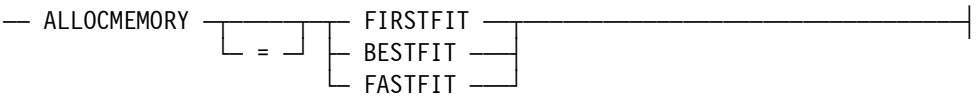
- A C program or library addresses memory in non-C code or another C program or library.
- A C program or library requires more memory than is available with the default heap management mechanism or benefits from separate array row allocation of data.
- Near data, far data, near pointer, or far pointer declarations appear in a C program or library, or the `<alloc.h>` header is included in a C program or library.
- A separately compiled module is bound into a program that uses the far heap management mechanism.
- A C program or library requires the shared memory feature.

- The C program is a library shared by two or more processes at once. (The CONCURRENTEXECUTION option should also be set.)
- The C program creates threads. (The CONCURRENTEXECUTION option should also be set.)

The FARHEAP option has the following subordinate options:

- ALLOCMEMORY
- INSTALLMEMORY
- ONE
- RESIZEMEMORY
- STACKSIZE

These subordinate options can be set only before any program text, and they are ignored if the FARHEAP option is not set. The subordinate options are discussed in the following paragraphs.



(Type: Value; Default: FIRSTFIT)

The ALLOCMEMORY option enables you to select the allocation algorithm that works best with your C application.

If FIRSTFIT is selected, available blocks of memory are kept in an unsorted list with the most recently freed block at the front of the list. Requests to allocate memory result in searching the available list for the first block that is large enough to satisfy the request. If a large enough block is found, it is removed from the list and divided into two blocks if it exceeds the requested size. The remaining portion, if any, is inserted back on the front of the available list. If none of the blocks on the available list is large enough, memory is allocated from the top of the heap.

If BESTFIT is selected, available blocks of memory are kept in an unsorted list with the most recently freed block at the front of the list. Requests to allocate memory result in searching the available list for the block that best fits the request. If a block is found, it is removed from the list and divided into two blocks if it exceeds the requested size. The remaining portion, if any, is inserted back on the front of the available list. If none of the blocks on the available list are large enough, memory is allocated from the top of the heap.

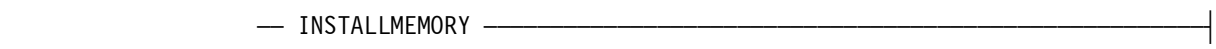
If FASTFIT is selected, available blocks of memory are kept in an unsorted list with the most recently freed block at the front of the list. Requests to allocate memory result in searching the first two blocks on the available list for the first block that is large enough to satisfy the request. If a large enough block is found, it is removed from the list and divided into two blocks if it exceeds the requested size. The remaining portion,

if any, is inserted back on the front of the available list. If neither of the two blocks at the front of the available list is large enough, the smaller of the two is moved to the end of the available list. Memory is then allocated from the top of the heap.

Of the three choices, BESTFIT makes the best use of memory. It makes use of previously freed memory whenever possible and keeps heap fragmentation to a minimum. However, BESTFIT does perform slower because of the overhead that results from searching through the entire available list.

The best performance choice is FASTFIT because it does not use a sorted list and only searches a small portion of the available list each time. However, FASTFIT prefers to use new memory instead of memory that has been previously freed. The total amount of memory used by a program may be higher if this option is used.

In terms of memory utilization and performance, the FIRSTFIT option falls somewhere between the BESTFIT and FASTFIT options.



(Type: Boolean; Default: FALSE)

The INSTALLMEMORY option causes the INSTALLMEMORY function to be exported automatically from libraries and programs that call libraries. The INSTALLMEMORY function is defined as follows:

```
INTEGER PROCEDURE INSTALLMEMORY (SEG, CHAR_ARRAY);
       VALUE SEG;  INTEGER SEG;
       EBCDIC ARRAY CHAR_ARRAY[*];
```

The INSTALLMEMORY function installs CHAR_ARRAY as a segment in the heap of the C program and returns a far pointer to CHAR_ARRAY[0]. If SEG is zero (0), the next available heap segment number is used. Otherwise, SEG is used as the segment number. If all segments are in use or if the segment specified by SEG is in use, then the array is not installed in the heap and the function returns zero.

If the C program is a library, then the INSTALLMEMORY function is exported. It can be called by a program in another language in order to share data. It can also be called by another C program in order to install a heap segment from the heap of the other program. The other C program can use this declaration:

```
extern "ALGOL" void _far *INSTALLMEMORY (int, char(&)[]);
```

Bound programs in other languages can also use the INSTALLMEMORY option. The name of the function in the Binder is __INSTALLMEMORY. The following USE statement can be added to a bind deck in order to bind the function as INSTALLMEMORY:

```
USE __INSTALLMEMORY FOR INSTALLMEMORY;
```

The INSTALLMEMORY function can also be passed to an external function with a non-C linkage class as the hidden parameter __install_memory_t. Refer to Appendix A, "Interface to the Library Facility," for more information on the use of hidden parameters.

Restrictions

Abuse of the INSTALLMEMORY option may violate system integrity and may cause system dumps. Therefore, if the INSTALLMEMORY option is set, the object code file is marked by the compiler as unsafe and non executable. The INSTALLMEMORY option is unsafe because the program activation owning an array installed in the heap might exit while the array is part of the heap, thus causing the heap to become invalid. The object file can be made executable by using the *MP* ODT system command, or it can be enabled as a library with the *SL* ODT system command. You should not use INSTALLMEMORY if you cannot meet the following restrictions:

- Ensure that no array installed in the heap is ever deallocated before being freed from the heap.
- An array installed in the C heap can never be resized while in the heap. If resizing is necessary, first ensure that the array is not concurrently being accessed through the heap. Free the array from the heap using the `_sfree` function in the `<alloc.h>` header, resize it, and then use the INSTALLMEMORY option again to put it back into the heap.

ONE

(Type: Boolean; Default: FALSE)

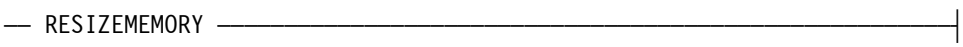
Setting the ONE option in all of the C modules of a program enables the heap to grow up to the maximum array size allowed by the MCP and the machine. This option should be set when the C program requires `malloc()` larger than the limit imposed by the FARHEAP option with the ONE option reset.

Refer to "MEMORY_MODEL Option" for more information on the various limits on the heap size.

Restrictions

Setting the ONE option imposes the following restrictions:

- All bound C object files must have the ONE option set.
- The MEMORY_MODEL option must be set to TINY or SMALL. The heap is restricted to one array row.
- The INSTALLMEMORY option must be reset. Calls on INSTALLMEMORY, `_smalloc()`, `_sfree()`, `_srealloc()`, and `_scalloc()` signal an error at compile time.
- The RESIZEMEMORY option must be reset.
- The first argument for calls on the `FP_MK()` function in `<alloc.h>`, also known as the segment number, must be 0. Otherwise, a run-time assertion fault occurs.

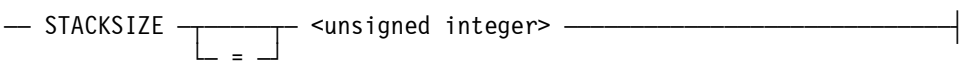


(Type: Boolean; Default: FALSE)

Setting the RESIZEMEMORY option in all of the C modules of a program enables the number of heap segments to be increased as needed.

Restriction

Before using the RESIZEMEMORY option, you must ensure that no code other than the C heap management mechanism ever performs a resize or deallocation of any of the heap segments or arrays. A resize or deallocation can violate system integrity and can cause system dumps. Thus, if the RESIZEMEMORY option is set, the object code file is marked by the compiler as unsafe and non executable. The object code file can be made executable by using the *MP* ODT command, or it can be enabled as a library with the *SL* ODT command.



(Type: Value; Default: 0)

The STACKSIZE value is used to reserve a fixed number of words of memory for the addressable stack of the program in either the near or far heap as determined by the MEMORY_MODEL option. When a STACKSIZE value is not specified, the compiler generates code that is executed when entering a function that checks to see if enough stack remains for the function. If there is not enough stack for the function, the compiler allocates a new piece by calling a heap management routine.

Companion code is also generated by the compiler and executed just before exiting the function that deallocates any stack pieces that may have been allocated upon

entering the function. All of the checking, allocation, and deallocation can be avoided for the addressable stack contained in the default heap by providing the compiler with a fixed STACKSIZE value. This will decrease the size of the generated object code and may improve execution-time performance. A further reduction in the amount of checking code generated can be achieved by resetting the STACK subordinate option of the BOUNDS compiler control option.

If a STACKSIZE value is specified and the STACK option of the BOUNDS compiler control record is set, the following fault message occurs if the addressable stack of the program exceeds the number of words specified for the STACKSIZE value:

```
Soft stack overflow
```

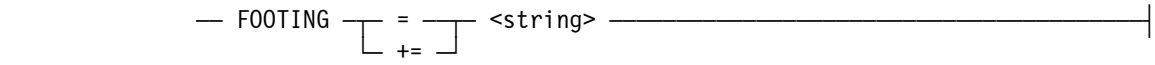
If the STACK option of the BOUNDS compiler control record is reset and the STACKSIZE value is exceeded, then the result is unpredictable. However, the most likely result is a fault occurring at some point later in the program because of data that is corrupted.

The range of the STACKSIZE value depends on the MEMORY_MODEL option that determines the size and structure of the heap. The heap contains the addressable stack and the dynamic memory area. The larger the specified STACKSIZE value, the smaller the space for dynamic memory allocation. If the STACKSIZE value exceeds the smaller of 16,777,215 words and the total initial heap size, the maximum value allowed for the chosen memory model is used in its place. If the STACKSIZE value is specified for multiple object files being bound together that have all been compiled with the same memory model, the largest value is used by the bound program. If the object files are compiled with different memory models, the largest near and far STACKSIZE values are used by the bound program. Refer to “MEMORY_MODEL Option” in this section for more information.

FOOTING Option

The FOOTING option specifies a string that is to be printed at the bottom of each page of the program listing, preceded by a blank line. The compiler uses the last value declared as the footing on the compilation listing.

<footing option>



(Type: String)

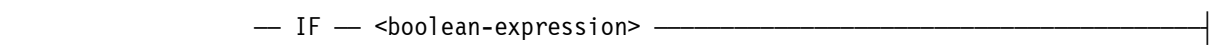
If the LIST option is disabled or if the PAGESIZE is less than 6, this option has no effect.

IF Option

The IF option is one of the conditional compilation options used to conditionally include or omit certain records in the compilation. Compiler control options encountered in the source language input between any IF, ELSE IF, ELSE, or END

compiler control options are always processed in the normal fashion, regardless of the value of the Boolean-expression of the IF option.

<if option>



(Type: Immediate)

If the IF option Boolean-expression is TRUE, the records between the IF option and a subsequent ELSE, ELSE IF, or END compiler option are compiled and all records between any subsequent ELSE or ELSE IF and END compiler options are ignored.

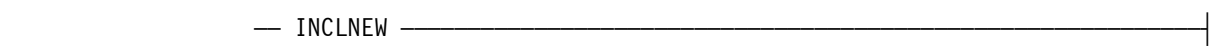
If the IF option Boolean-expression is FALSE but a subsequent ELSE IF option is TRUE, then the records between the ELSE IF and subsequent ELSE or END are compiled but the records between the IF and the ELSE IF are ignored.

If the both the IF option and subsequent ELSE IF option are FALSE, the records between the ELSE (if specified) and END options are compiled.

INCLNEW Option

When the INCLNEW option is enabled, source language records that are read from an INCLUDE file are written to the new source file (NEWSOURCE). If the NEW option is disabled, INCLNEW has no effect.

<inclnew option>



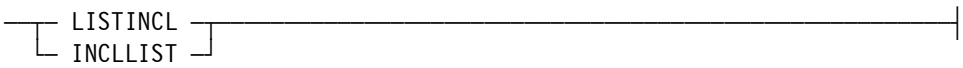
(Type: Boolean; Default: FALSE)

If this option is enabled and the SEQUENCE option is also enabled, the source records from the included file are assigned new sequence numbers; otherwise, their sequence numbers remain unchanged.

INCLLIST Option

The INCLLIST option is a synonym for the LISTINCL option. Refer to "LISTINCL Option" in this section for details.

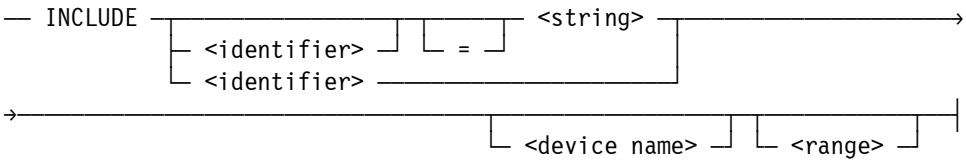
<listincl option>



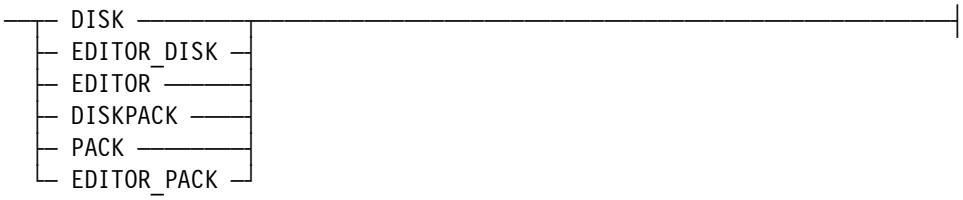
INCLUDE Option

The INCLUDE option is used to temporarily redirect the input stream to an alternate source. When that source is exhausted, input resumes with the source record immediately following the INCLUDE.

<include option>



<device name>



The device names (DISK, EDITOR_DISK, EDITOR, PACK, DISKPACK, EDITOR_PACK) are included only to facilitate the migration of code from V Series platforms to A Series platforms. Although a warning will be generated, including a device name has no effect on the compiler.

<range>

(Type: Immediate)

If an identifier is supplied on the INCLUDE card, that identifier is used as the INTNAME of the file to be included. The INTNAME of the file can then be file equated at compile-time. For example, given the following INCLUDE card:

```
$ INCLUDE THATFILE
```

The identifier THATFILE can be file equated to the actual file to be included in the following compile statement:

```
COMPILE MYFILE; C FILE THATFILE (TITLE=MYINCLUDEFILE)
```

If a string is supplied as the argument to INCLUDE, the string, along with the partial file titles supplied in the SEARCH path, is used to form the title of the file to be included (refer to "SEARCH Option" in this section for details on search paths). For example, the following include option contains the file named MYINCLUDEFILE ON MYPACK.

```
$ INCLUDE "MYINCLUDEFILE ON MYPACK"
```

The INCLUDE card may contain a range that specifies the region of the file to be included. The range is indicated by a string, a sequence number, or a pair of sequence numbers. If no range is supplied, the entire file is included.

If the range consists of a string, the region of the file demarcated symbolically through the COPYBEGIN and COPYEND options is included. For example, assume the file XYZ contains the following region:

```
$ COPYBEGIN "string procedures"
     <some lines of code>
$ COPYEND   "string procedures"
```

Also assume, the INCLUDE card is of the following form:

```
$ INCLUDE "XYZ" ("string procedures")
```

In this example, the region of file XYZ between the COPYBEGIN and COPYEND records is included. The strings must match exactly, including the casing of any letters, and only the first 30 characters are used to form the unique string.

If a single sequence number is supplied in the range, the record with the matching sequence number in the file is included. If a pair of sequence numbers is supplied, the records whose sequence numbers fall within that range are included. If only the second sequence number is supplied, the first sequence number is assumed to be that which terminated the most recent range from the same included file. The following example includes the record with sequence number 3000 from XYZ:

```
$ INCLUDE "XYZ" 3000
```

The following example includes the records with sequence numbers 4000 through 5500:

```
$ INCLUDE "XYZ" 4000 TO 5500
```

The following example includes the records with sequence numbers 5501 through 8000:

```
$ INCLUDE "XYZ" TO 8000
```

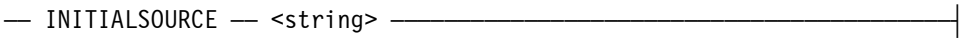
Note: *An INCLUDE card may not appear in an INITIALCCI file. Refer to "The INITIALCCI File" in Section 9 for a description of INITIALCCI files.*

Refer to "TITLESYNTAX Option" for more information on the file title changes that occur during compilation.

INITIALSOURCE Option

The INITIALSOURCE option specifies a string that represents a C source line to be parsed and compiled before the first source token of the program.

<initialsource option>



The string represents either a preprocessor directive or a C line. Compiler control options cannot be specified. The INITIALSOURCE option can be specified multiple times. The list of source lines are parsed in the same order as specified. The parsing is done after all the leading comments, white spaces, and compiler control options are read, and occurs before the first source token of the program.

An equal sign (=) cannot occur between INITIALSOURCE and <string>. If <string> contains an error, the error is reported against the line that contains the first source token.

Examples

The INITIALSOURCE option can be specified in the INITIALCCI file:

```
C:  INITIALSOURCE '#define _POSIX_SOURCE'
```

The option can also be specified in the compiler's TASKSTRING attribute:

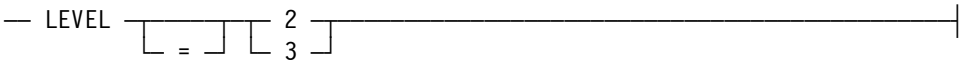
```
compile abc;c taskstring="INITIALSOURCE '#define abc 3'"
```

The option can also be specified in the source file before the first source token:

```
/* comment */
$INITIALSOURCE '#include <float.h>'
$INITIALSOURCE 'double MAX = DBL_MAX;'
#include <stdio.h>
main() {
        printf("Double Maximum = %E\n", MAX);
}
```

LEVEL Option

The LEVEL option controls the level at which program globals occur in the program stack. Globals normally occur at level 2. However, by setting the level to 3, you can create a nested C program. The LEVEL option should appear before any program text.



(Type: Value; Default: 2)

To use a nested C program, you must bind the program as a single function into a level 2 C program. The arguments of the function and the return type must match those of the nested program's `main` function. Calling the function causes the nested program to initialize and run. All external variables and functions in the nested program that are not resolved within the nested program are mapped by the Binder to variables and functions in the level 2 program.

Variables and functions that are resolved, including all static variables and functions, are local to the nested program. Therefore, the variables and functions are automatically allocated for each call of the nested program.

The nested program uses the same heap as the level 2 program, so data pointers can be shared between them. However, a function pointer is usable only in the program where it is created. A nested program can make a pointer point to an external level 2 function, but using the pointer outside of the nested program will produce undetermined results.

When a nested program exits by calling the `exit` function or by returning from its `main` function, the status value passed to the `exit` function or returned from the `main` function is returned as the function result of the nested program. All functions registered by the `atexit` function are called in reverse order of their registration. Files are not automatically closed because files actually occur at level 2.

Example

The following is an example of a level 2 host program, a nested program, and the Binder input to combine the two files:

File 1: EXAMPLE/HOST

```
extern nestedProgram ();
int global;
main ()
    {nestedProgram ();
     nestedProgram ():}
```

File 2: EXAMPLE/LEVEL3/PROGRAM

```
$$set level 3
extern int global;
int local;
main ()
    {static int staticLocal;
     global++;        /* adds 1 to global on each call      */
     local++;         /* sets local to 1, because it gets
                         initialized on each call           */
     staticLocal++;   /* sets staticLocal to 1, like local  */
     return;}
```

File 3: EXAMPLE/BIND

```
BIND ? FROM OBJECT/EXAMPLE/HOST;
BIND nestedProgram FROM OBJECT/EXAMPLE/LEVEL3/PROGRAM;
```

Example 10–1. LEVEL Option Example Files

To bind a nested program, give the name of the corresponding level 2 function in the BIND statement. The Binder can bind multiple nested programs into a level 2 program, but it cannot bind multiple level 3 subprograms to create a nested program. If you want to create a nested program from multiple source modules, then you must combine the source modules (by using the `#include` directive or by editing) so that they are compiled together.

LIBRARY Option

Setting the LIBRARY option causes the compiler to generate code for each function separately, with the resulting code stored in a single code file. Also, this option causes binding information to be generated. For more information, refer to "BINDINFO Option" in this section.

<library option>

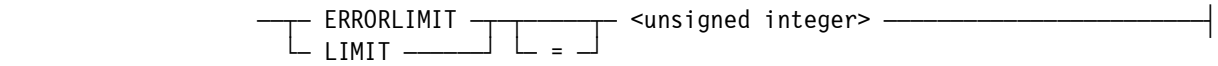
(Type: Boolean; Default: TRUE if `main` is not declared; otherwise FALSE)

This option should appear before any program text. Normally, this option does not need to be set explicitly.

LIMIT Option

The LIMIT option is a synonym for the ERRORLIMIT option. Refer to "ERRORLIMIT Option" in this section for details.

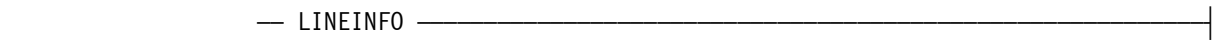
<errorlimit option>



LINEINFO Option

When the LINEINFO option is enabled, the compiler saves source record sequence numbers in the CODE file.

<lineinfo option>

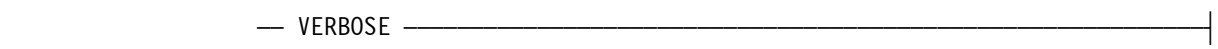


(Type: Boolean; Default: TRUE if the compilation is initiated interactively; otherwise FALSE).

If the program terminates abnormally, the source record sequence number associated with the point of program termination is displayed. If an LI_SUFFIX string has been specified, the string appears after the sequence number.

The LINEINFO option can be changed only before any program text.

There is one subordinate option:



(Type: Boolean; Default: TRUE)

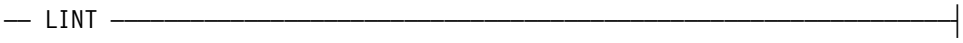
If the VERBOSE option is set, the displayed stack history will include the names of the functions in the call chain up to the point of termination. Note that this will cause a larger code file to be generated than if the option were disabled.

A function name longer than 40 character is truncated to 40 in the LINEINFO.

LINT Option

The LINT option controls the generation of some compile-time warnings. Setting the LINT subclass options can make porting an application to A Series easier. This option should appear before any program text.

<lint option>

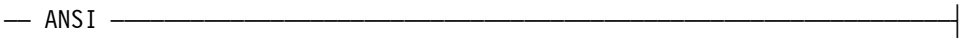


(Type: Boolean class; Default: TRUE)

When the LINT option is reset, suboption values are ignored by the compiler. The LINT option has the following subordinate options:

- ANSI
- _ASERIES_SOURCE
- BITWISE
- CAST
- FUNCTION
- KNR_EXTERN
- KNR_FUNCTION
- KNR_KW
- KNR_POINTER
- KNR_STDFUNC
- _POSIX_SOURCE
- PRAGMA
- SETJMP
- UNUSEDOBJECT
- UNUSEDSTATICFUNCTION

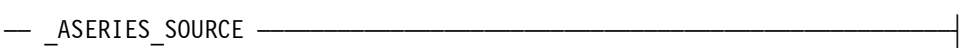
ANSI



(Type: Boolean; Default: TRUE)

When the ANSI option is enabled, a warning message is generated for ANSI syntax warnings.

_ASERIES_SOURCE



(Type: Boolean; Default: TRUE)

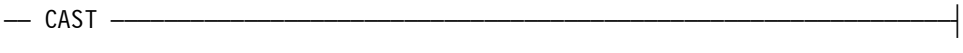
The _ASERIES_SOURCE option is no longer used. Setting and resetting it is ignored.

BITWISE

(Type: Boolean; Default: FALSE)

When enabled, a warning is generated when a signed type is used in an expression with the bitwise operators >>, <<, or ~.

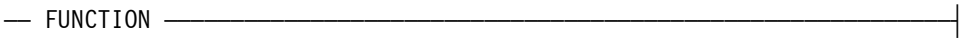
CAST



(Type: Boolean; Default: FALSE)

When enabled, explicit nonportable type casts generate a warning. Implicit nonportable type casts always generate a warning, regardless of the setting of this option.

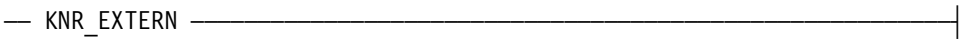
FUNCTION



(Type: Boolean; Default: TRUE)

When enabled, a warning is generated when a function is referenced that has not yet been declared.

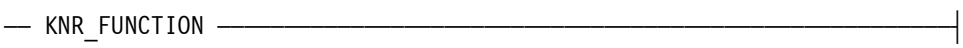
KNR_EXTERN



(Type: Boolean; Default: TRUE)

When the KNR_EXTERN option is enabled, a warning message generated for PORT(KNR_EXTERN) displays; otherwise, the warning message is suppressed. Refer to "PORT Option" in this section for more information.

KNR_FUNCTION



(Type: Boolean; Default: TRUE)

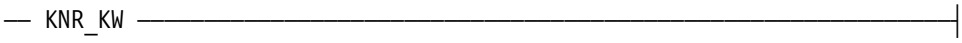
When the KNR_FUNCTION option is enabled, a warning message generated for PORT(KNR_FUNCTION) displays. Refer to "PORT Option" in this section for more information.

Examples

```
$PORT (KNR_FUNCTION)
$LINT (RESET KNR_FUNCTION)
```

When the $RESET ANSI option is set, the compatibility checking between a function prototype and an old-style function definition is relaxed. When two functions have equivalent parameter type lists, the functions are treated as compatible functions.

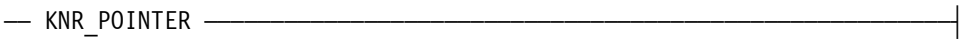
KNR_KW



(Type: Boolean; Default: TRUE)

When the KNR_KW option is set, the warning message for $PORT(KNR_KW) is suppressed.

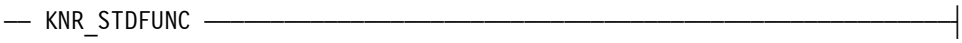
KNR_POINTER



(Type: Boolean; Default: TRUE)

When the KNR_POINTER option is enabled, a warning message displays if strict pointer rules are relaxed when the PORT(KNR_POINTER) option is set.

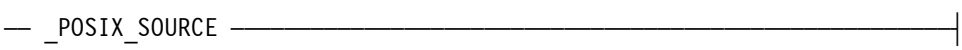
KNR_STDFUNC



(Type: Boolean; Default: TRUE)

When the KNR_STDFUNC option is enabled, a warning message displays when an internal header is implicitly included for a standard function that is referenced but not yet declared.

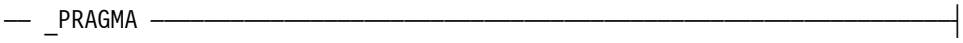
_POSIX_SOURCE



(Type: Boolean; Default: TRUE)

The _POSIX_SOURCE option is no longer used. Setting and resetting it is ignored.

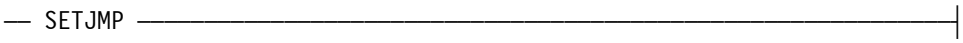
PRAGMA



(Type: Boolean; Default: TRUE)

When the PRAGMA option is enabled, a warning message is generated for unknown pragmas.

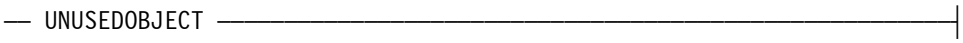
SETJMP



(Type: Boolean; Default: TRUE)

When the SETJMP option is enabled, a warning is generated when a call to the `setjmp` macro does not conform to the ANSI C Standard.

UNUSEDOBJECT



(Type: Boolean; Default: TRUE)

When enabled, a compiler warning message is generated when declared objects are not used. Declared objects that are initialized are considered to have been referenced and do not generate a warning message.

UNUSEDSTATICFUNCTION

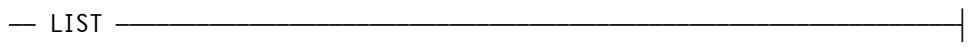
When enabled, a warning message is generated when declared static functions are not used.

(Type: Boolean; Default: TRUE)

LIST Option

When the LIST option is enabled, the compiler writes a listing of the program to the output listing file (LINE). The listing includes the source records, error and warning messages, and a compilation summary.

<list option>

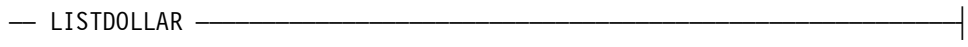


(Type: Boolean; Default: FALSE if the compilation is initiated interactively; otherwise TRUE)

LISTDOLLAR Option

When the LISTDOLLAR option is enabled, temporary compiler control records are written to the program listing (LINE) and the settings of the compiler options that appear in the program are included in the compilation summary. If the LIST option is disabled, this option has no effect.

<listdollar option>

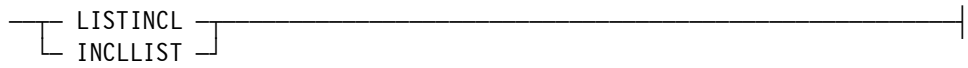


(Type: Boolean; Default: FALSE)

LISTINCL Option

The LISTINCL option, when enabled, causes the compiler to write all source records that were read as the result of an INCLUDE statement to the output listing (LINE). This option is independent of the LISTOPTION.

<listincl option>

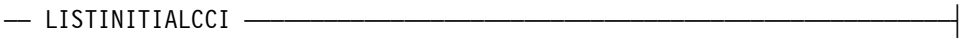


(Type: Boolean; Default: FALSE)

LISTINITIALCCI Option

The LISTINITIALCCI option, when enabled, causes the compiler to write the contents of the INITIALCCI file to the first page of the program listing.

<listinitialcci option>



(Type: Boolean; Default: FALSE)

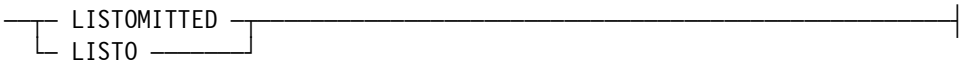
The LISTINITIALCCI compiler control record must appear in the INITIALCCI file for this option to take effect. If the LIST option is disabled, this option has no effect.

Refer to "The INITIALCCI File" in Section 9 for more details.

LISTOMITTED Option

The LISTOMITTED option, when enabled, causes the compiler to write all source records omitted through the OMIT option to the program listing (LINE) and to mark those records as omitted. If the LIST option is disabled, this option has no effect.

<listomitted option>

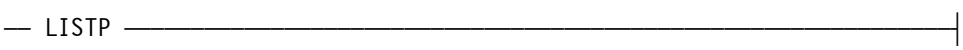


(Type: Boolean; Default: FALSE)

LISTP Option

The LISTP option, when enabled, causes source records from the primary input file (CARD) to be written to the program listing (LINE).

<listp option>



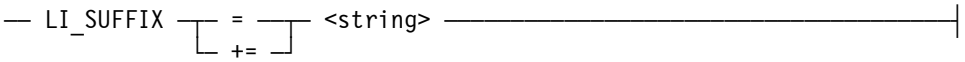
(Type: Boolean; Default: FALSE)

Source records from the secondary input file (SOURCE) are not written. The LIST option, when enabled, overrides the setting of LISTP, that is, the LIST option causes both primary and secondary source records to be written.

LI_SUFFIX Option

The LI_SUFFIX option enables the specification of a string of characters to be written to the code file along with LINEINFO.

<li_suffix option>



(Type: String)

If the LINEINFO option is disabled, this option has no effect. Each sequence number written to the code file is appended with the LI_SUFFIX string. Refer to "LINEINFO Option" in this section for additional information.

Parentheses, commas, and periods should not be used in LI_SUFFIX strings. These characters are used to delimit LINEINFO and code addresses in stack histories, so their presence in LINEINFO can cause problems for existing software such as CANDE which parses stack histories.

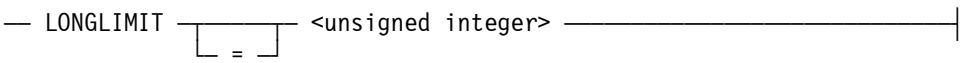
If an LI_SUFFIX string is longer than 40 characters, only the first 40 characters are put in the LINEINFO.

All statements will use the most recently-set LI_SUFFIX value.

LONGLIMIT Option

The LONGLIMIT option specifies the maximum length (in words) of a nonpaged array.

<longlimit option>



(Type: Value; Default: 10922)

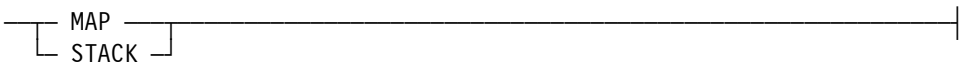
Nonpaged arrays (sometimes called LONG arrays) provide greater efficiency in accessing elements at the expense of increased actual memory usage at any given instant. Arrays that are less than or equal to the LONGLIMIT are not paged, while all other arrays are paged.

The value of <unsigned integer> must be in the range 1024-65536. In addition, an individual installation may have a system-wide LONG array limit that is less than the setting of LONGLIMIT. Setting the LONGLIMIT greater than the site limit causes the object program to abnormally terminate at run-time.

MAP Option

Enabling the MAP option causes the compiler to write information regarding the allocation of variables and procedures to the output listing (LINE). This option is independent of the LIST option.

<map option>

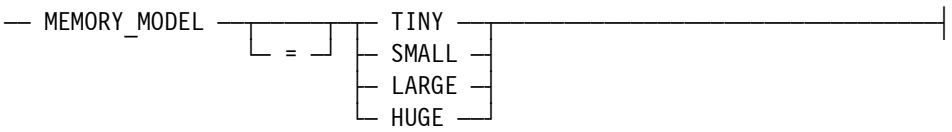


(Type: Boolean; Default: FALSE)

MEMORY_MODEL Option

The MEMORY_MODEL option controls the size and structure of the heap provided for the program.

<memorymodel option>



(Type: Value; Default: TINY)

Each C program is provided an area for user data known as the "heap". The heap consists of two parts: the software stack and the dynamic memory allocation area. When the FARHEAP option is reset, these two areas grow toward each other from opposite "ends" of the heap. Otherwise, the software stack is allocated in chunks from the dynamic memory allocation area. When the stack bounds-checking code (the BOUNDS subordinate option STACK) is enabled, the program terminates and a stack overflow error message is issued if the software stack collides with the boundary of the dynamic memory allocation area.

Data objects that are not appropriate for the hardware stack are put in the software stack, such as objects that are addressed, arguments in variable length argument lists, and so forth. The software stack increases and decreases in size as procedures or functions with such data objects are entered and exited.

The dynamic memory allocation area is used by language functions that provide such capability (such as `malloc` and `free`). As memory is allocated, this area is enlarged. If such an allocation causes a collision with the software stack, the allocation fails. As the memory is deallocated, the dynamic memory allocation area shrinks, leaving more space for the software stack.

The MEMORY_MODEL option provides four alternatives that determine the size and structure of the heap. They differ in two respects: the number of array rows used for the heap and the size of each array row. The choice of memory model can have a dramatic effect on program performance and utilization of system resources. This option should appear before any program text.

The behavior of the MEMORY_MODEL option depends on the following:

- The value of the TARGET option at compile time and the machine level at run time. Refer to "TARGET Option" in this section.

- Whether or not the FARHEAP option is set.

If TARGET specifies only LEVEL4 machines and the FARHEAP option is reset, models LARGE and HUGE are treated as model SMALL. If a program with model SMALL runs on a LEVEL4 machine, the heap size is 16,777,215 words. Otherwise, the behavior is as described in the following paragraphs.

Models TINY and LARGE use nonpaged array rows. These arrays are always present in memory when the program is active, making access to them relatively fast. Models SMALL and HUGE, on the other hand, use paged array rows. These rows have a larger capacity, but have a slower access time.

Models TINY and SMALL have less capacity, by virtue of being one-dimensional arrays, than LARGE and HUGE. However, the additional capacity of LARGE and HUGE comes with a performance cost due to the additional overhead of indexing two-dimensional arrays. Therefore, the best rule in selecting a memory model is to choose the smallest option that still allows a given program to compile and execute correctly.

If the FARHEAP option is set, all four memory models use a two-dimensional array for the heap. If the ONE option is set, the number of rows is fixed at one. If the RESIZEMEMORY option is set, the number of rows will grow as required up to 32,768 on all memory models. Otherwise, the TINY and SMALL models have 32 rows and the LARGE and HUGE models have 1,024 rows. Refer to "FARHEAP Option" in this section for more information.

During run time, if the FARHEAP option is set and the MEMORY_MODEL is TINY or LARGE, the rows of the heap will be segmented as needed if a program uses more than the LONGLIMIT words of heap space. For example, given the following compiler control options in a program:

```
$SET FARHEAP
$MEMORY_MODEL = TINY
```

If a program uses more than LONGLIMIT words of heap space during run time, the farheap manager will resize the heap to the same size as the SMALL memory model. That is, the program will run under the TINY memory model until it needs more than LONGLIMIT words of heap, then it automatically switches to the SMALL memory model. If this situation occurs when the FARHEAP option is **not** set, the program is programmatically discontinued when it exceeds LONGLIMIT words of heap. Likewise, if the LARGE model is specified and insufficient space remains for an allocation request, then the heap will be resized to the HUGE memory model one row at a time as needed. Refer to "LONGLIMIT Option" for more information.

Table 10–1 summarizes the four memory models.

Table 10–1. Memory Models with FARHEAP reset

Model	Rows	Row Size (words)	Performance Characteristics
TINY	1	LONGLIMIT	Nonpaged one-dimensional array; fastest access; minimum space
SMALL	1	16,777,215	Paged one-dimensional array
LARGE	1	16,777,215	Paged one-dimensional array
HUGE	1	16,777,215	Paged one-dimensional array

Tables 10–2 and 10-3 summarize the four memory models for the FARHEAP option with the ONE option reset and set, respectively.

Table 10–2. Memory Models for the FARHEAP Option (ONE reset)

Model	Rows	Row Size (words)	Performance Characteristics
TINY	32–32,768	LONGLIMIT–2,796,202	Initially nonpaged, two-dimensional array; near pointers and data by default.
SMALL	32–32,768	2,796,202	Paged two-dimensional array; near pointers and data by default.
LARGE	1,024–32,768	LONGLIMIT–2,796,202	Initially nonpaged, two-dimensional array; far pointers and data by default.
HUGE	1,024–32,768	2,796,202	Paged two-dimensional array; far pointers and data by default.

***Note**: If the FARHEAP option is **not** set, it is possible for programs to compile or execute under the SMALL model that do not compile or execute under the LARGE model. This is due to the fact that objects are never split across row boundaries. Therefore, a large object that fits in the single bigger array row of the SMALL model may not fit in the smaller array rows of the LARGE model.*

Table 10–3. Row Sizes for Machines with FARHEAP option set (ONE set) †

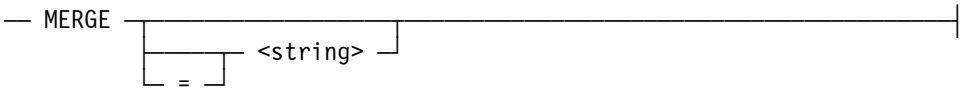
Machines	Memory Model	Rows	Row Size (words)
LX5000	TINY, SMALL	1	134,217,727
A2800, NX4800, NX56xx, and NX58xx	TINY, SMALL	1	268,435,455

† MEMORY_MODEL cannot be LARGE or HUGE.

MERGE Option

The MERGE option, when enabled, causes the compiler to merge source records from the primary input file (CARD) with source records from the secondary input file (SOURCE).

<merge option>



(Type: Boolean; Default: FALSE)

If a string is specified, then the string, along with the partial file titles supplied in the search path (if any), is used as the file title for SOURCE (refer to "SEARCH Option" in this section for more details on SEARCH paths). Refer to "Input Files" in Section 9 for more details on how input files are merged.

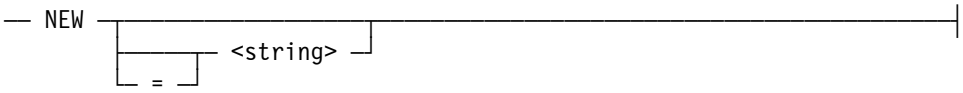
Refer to "TITLESYNTAX Option" for more information on the file title changes that occur during compilation.

Note: *This option remains in effect throughout the compilation and cannot be disabled. It may appear only in the CARD file.*

NEW Option

Enabling the NEW option causes the compiler to create a new source file (NEWSOURCE) containing all source records that were accepted for compilation.

<new option>



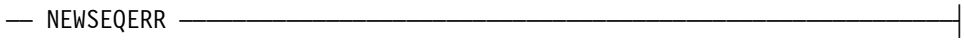
(Type: Boolean; Default: FALSE)

The new file includes all records that were omitted by the OMIT option and all permanent compiler control records, but does not include any records that were discarded by enabling the DELETE or VOID options. If a string is specified, then the title of the new source file is set to <string>. This option should appear before any program text.

NEWSEQERR Option

Enabling the NEWSEQERR option causes an error to be issued if the sequence number on a record of the new source file (NEWSOURCE) is not strictly greater than the sequence number of the preceding record.

<newseqerr option>

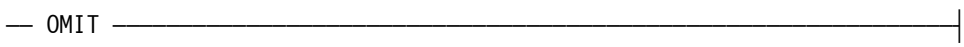


(Type: Boolean; Default: FALSE)

OMIT Option

Enabling the OMIT option causes all source records to be ignored until the option is disabled. This option may be enabled from either the primary (CARD) or the secondary (SOURCE) input files.

<omit option>



(Type: Boolean; Default: FALSE)

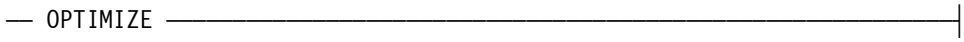
Omitted records are written to the new source file (NEWSOURCE) if the NEW option is enabled. Omitted records are not listed in the program listing unless the LISTOMITTED option is enabled.

Compiler control options encountered in the source language input while this option is TRUE are processed in the normal manner.

OPTIMIZE Option

Enabling the OPTIMIZE option causes optimizations to be performed prior to code generation.

<optimize option>



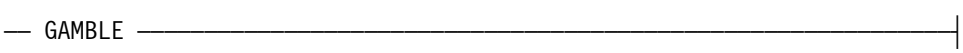
(Type: Boolean class; Default: FALSE)

The option is examined at the beginning of each function and, if enabled, optimizations are performed for that function. Although the setting of the option can be changed at any time, only its setting at the beginning of a function is significant. The optimizations performed include the following:

- Moving invariant expressions out of loops
- Reordering expressions to take advantage of the commutative and associative properties
- Constant evaluation (for example, multiplying by -1)
- Dead code elimination
- Loop optimization (loop unraveling, strength reduction, elimination of duplicate induction variables)
- Constant folding
- Common sub-expression folding
- Variable substitution
- Replacing arrays by simple variables
- Elimination of dead variables

There are three subordinate options:

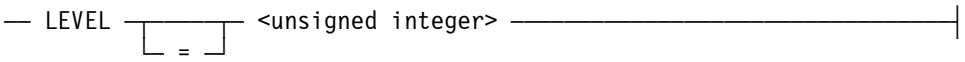
- GAMBLE
- LEVEL
- UNRAVEL



(Type: Boolean; Default: TRUE)

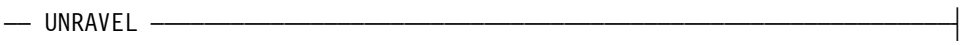
Enabling the GAMBLE option allows the compiler to make certain assumptions in order to perform some of the optimizations previously described, including:

- Applying the associative law to floating-point types, that is, changing a division to a multiplication by a reciprocal
- Applying the associative law to integral types; that is, changing a subtraction to an addition of an inverse
- Moving conditionally executed invariant expressions outside of loops
- Assuming array indices are within-bounds
- Assuming that variable strides are positive



(Type: Value; Default: 5)

The LEVEL option controls the amount of effort expended by the compiler in optimizing a function. In general, the higher the LEVEL, the greater the optimization effort. Higher LEVELs yield a potentially reduced run-time at the expense of an increased compile-time. The LEVEL must range from 0 to 10.



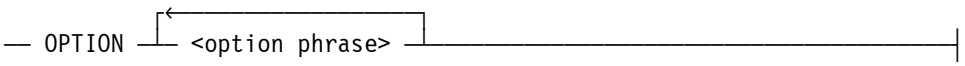
(Type: Boolean; Default: TRUE)

The UNRAVEL option, when enabled, allows loops to be unraveled and certain functions to be generated in-line.

OPTION Option

The OPTION option is provided as a means for declaring user options.

<option option>



(Type: Immediate)

User options are Boolean options and may be set, reset, or popped just like any other Boolean option. The user options must appear in the <option phrase>. The initial mode setting may also appear in the <option phrase>. If a mode setting is not supplied, the mode prior to the left parenthesis is used.

For example, the following statement declares user option IOLIST, which is disabled, and ABCLIST, which is enabled:

```
$ SET LIST OPTION (RESET IOLIST SET ABCLIST)
```

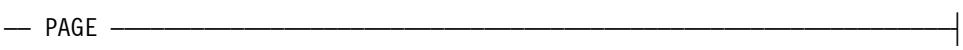
The following statement declares user option OP1, which is enabled, and OP2, which is disabled:

```
$ SET NEW OPTION (OP1 RESET OP2)
```

PAGE Option

The PAGE option causes the output listing to skip to the top of a new page.

<page option>



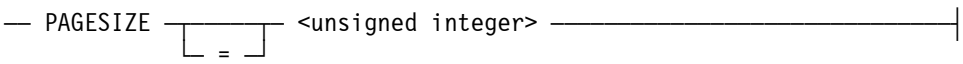
(Type: Immediate)

If the LIST option is disabled, this option has no effect. If the OMIT option is enabled, this option is ignored.

PAGESIZE Option

The PAGESIZE option sets the number of lines per page that are to be printed on the output listing.

<pagesize option>



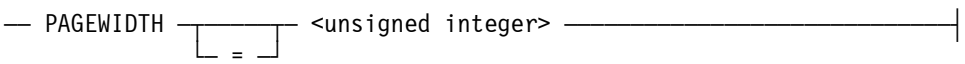
(Type: Value; Default: 58)

If PAGESIZE is set to a value less than 6, the listing is a contiguous unpaged stream and the FOOTING and TITLE option settings are ignored. The PAGESIZE includes two lines for the TITLE and two lines for the FOOTING, but only if a FOOTING is specified. If the LIST option is disabled, this option has no effect.

PAGEWIDTH Option

The PAGEWIDTH option sets the number of columns per line that are to be printed on the output listing. If the LIST option is disabled, this option has no effect.

<pagewidth option>

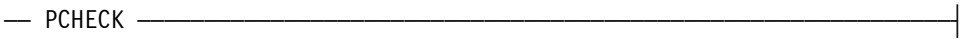


(Type: Value; Default: 132)

PCHECK Option

The PCHECK option, when enabled, causes strict run-time comparison of the actual and the formal parameters of functions invoked by function pointers.

<pcheck option>



(Type: Boolean; Default: TRUE)

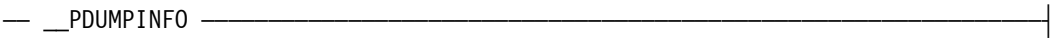
When the PCHECK option is enabled, actual and formal parameters must match in number and type. When PCHECK is disabled, the run-time comparison of the actual and formal parameters must match in number. Any single-word parameter type (`char, short, int, long, pointer, float, double`) matches any other single-word parameter type, and any double-word parameter type (`long double, long long`) matches any other double-word parameter type.

__PDUMPINFO Option

The __PDUMPINFO option is used for debugging programs with the PRINTHEAP (Print Heap) command in DUMPANALYZER.

Note: *The __PDUMPINFO option name requires two leading underscores.*

<pdumpinfo option>



(Type: Boolean class; Default: FALSE)

The __PDUMPINFO option causes the compiler to generate additional and more specific bindinfo for use by the DUMPANALYZER utility. For information about DUMPANALYZER, refer to the *System Software Utilities Operations Reference Manual*.

The __PDUMPINFO option has the following limitations:

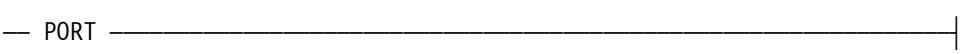
1. Local directories are generated for static functions only in nested programs (see the $LEVEL=3 option description in this section). Local directories provide information about local variables.

2. Bindinfo generated for function arguments is limited to word and double word operands or pointer addresses of items stored in the C heap.

PORT Option

The PORT option is used to port a C program developed for some other computing environment onto an A Series system. This option should appear before any program text.

<port option>



(Type: Boolean class; Default: TRUE)

The PORT option has the following subordinate options:

- CHAR2
- KNR_EXTERN
- KNR_FUNCTION
- KNR_KW

- KNR_POINTER
- KNR_STDFUNC
- SIGNEDCHAR
- SIGNEDFIELD
- TEXTASBINARY
- UNSIGNED

These subordinate options are discussed in the following paragraphs.

CHAR2

(Type: Boolean; Default: FALSE)

The CHAR2 option, when enabled, causes `unsigned chars` to be stored as two's complement values.

KNR_EXTERN

(Type: Boolean; Default: FALSE)

When the KNR_EXTERN option is enabled, an external variable found in a function body is promoted to the file level. A warning, which can be suppressed, is also issued. The LINT suboption with the same name is also used to suppress the warning message.

Refer to "LINT Option" in this section for more information.

KNR_FUNCTION

When the KNR_FUNCTION option is enabled, no type checking of parameters occurs when a function is called. If an actual argument is found to be incompatible with the formal parameter, a warning message displays.

— KNR_KW ——┤

(Type: Boolean; Default: FALSE)

When the KNR_KW option is enabled, the following ANSI keywords are used as normal identifiers:

- `const`
- `enum`
- `signed`
- `void`
- `volatile`

A warning message displays to indicate that ANSI keywords are used as identifiers.

— KNR_POINTER ————————————————————————————————————┤

(Type: Boolean; Default: FALSE)

When the KNR_POINTER option is enabled, pointers of one type are allowed to assign to pointers of another type without using explicit cast. A warning message is issued to warn you of possible improper alignment. The warning message can be suppressed by the LINT suboption with the same name. Refer to "LINT Option" in this section for more information.

— KNR_STDFUNC ————————————————————————————————————┤

(Type: Boolean; Default: FALSE)

When the KNR_STDFUNC option is enabled and a standard function is called but not yet declared, the internal header for that function is implicitly included. Currently, only the following headers are supported:

- `<ctype.h>`
- `<local.h>`
- `<math.h>`

- <signal.h>
- <stdio.h>
- <stdlib.h>
- <string.h>
- <time.h>

All other headers are not supported because they have type or macro definitions only. When a header is implicitly included, a warning message such as the following is issued:

```
"stdio.h" is implicitly included.
```

SIGNEDCHAR

(Type: Boolean; Default: FALSE)

The SIGNEDCHAR option controls the default sign attribute for the `char` type. When disabled, the plain `char` type is unsigned; when enabled, the plain `char` type is signed.

SIGNEDFIELD

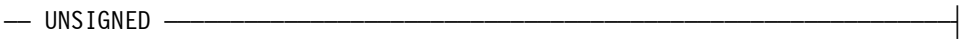
(Type: Boolean; Default: FALSE)

The SIGNEDFIELD option controls the default sign attribute for structure or union bit fields of type `int`. When disabled, a plain `int` bit field in a structure or union is unsigned; when enabled, a plain `int` bit field in a structure or union is signed.

TEXTASBINARY

(Type: Boolean; Default: FALSE)

The TEXTASBINARY option controls the semantics of I/O streams. If this option is enabled, text streams are treated as though they are binary streams, meaning that data bytes are transferred as is to or from the host environment with no interpretation. This option applies to disk files only.



(Type: Boolean; Default: FALSE)

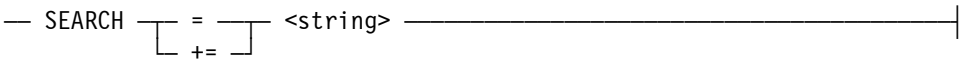
The UNSIGNED option controls the semantics of the sign attribute for integer types. If this option is disabled, unsigned integers are treated as signed integers; that is, normal signed-magnitude arithmetic is performed. If the option is enabled, unsigned integers are emulated as two's-complement quantities. Enabling this option could slow down performance and should be used only when absolutely necessary.

SEARCH Option

The SEARCH option is used to specify a search path used when processing an INCLUDE or MERGE option.

Note: *The following discussion makes references to INCLUDE, but such references apply to MERGE as well.*

<search option>



(Type: String)

The SEARCH string consists of a list of partial file titles separated by semicolons (;). A partial file title, when combined with the title string of an INCLUDE, must form a legitimate file title.

When processing an INCLUDE, the INCLUDE title is combined with each partial file title specified by the SEARCH string until a file title is formed that matches the name of an existing file; that file is then included. If the SEARCH string is exhausted without finding a match, if the SEARCH string is null, or if the INCLUDE title begins with a slash, the INCLUDE title itself is used to find the INCLUDE file.

An INCLUDE title may begin with one or more greater-than-signs (>). Each > causes the search to skip a partial file title in the SEARCH string.

If the partial file title consists of a single dollar sign ($), the partial title is treated as though it is the name of the primary input file (CARD), followed by the /= symbol. For each dash (-) that follows the $, a node is dropped from the end of the primary input file name and the /= is implicitly appended.

To form the final INCLUDE file title, the partial file title is scanned for an equal sign (=); the first occurrence of = is replaced with the INCLUDE title.

Refer to “TITLESYNTAX Option” for more information on the file title changes that occur during compilation.

The following example illustrates the use of the SEARCH option. Assume the title of the CARD file is “GRAPHICS/EXE” and compiler option TITLESYNTAX equals TITLE or PATHNAME.

```
$ SEARCH = "$-;SYSTEM/UTILITIES/= ON UTILPACK"
$ SEARCH += ";(Charlie)utilities/="
$ INCLUDE "GRAPH.C"
```

The previous sequence causes in the following file titles to be formed when processing the INCLUDE file title (the search stops as soon as a title matches an existing file):

```
GRAPHICS/GRAPH/C
SYSTEM/UTILITIES/GRAPH/C ON UTILPACK
(CHARLIE) UTILITIES/GRAPH/C
GRAPH/C
```

If POSIX APIs are used (the macro _POSIX_SOURCE or _ASERIES_SOURCE <release>, where <release> is 46.1 or greater), then partial file titles used in the SEARCH option must conform to absolute PATHNAME rules.

See *ClearPath HMP NX and A Series C Programming Reference Manual, Volume 2: Headers and Functions,* for more information.

The following example illustrates the use of the SEARCH option with POSIX APIs and a complier option TITLESYNTAX that equals NXPATHNAME.

```
$INITIALSOURCE '#define _POSIX_SOURCE'
$SET TITLESYNTAX=NXPATHNAME
$SET SEARCH="/-/USERS/USERCODE/JONES/=;"
$SET SEARCH+="/-/DISK/SYMBOL/CC/LIBRARY/=;"

#include <jni.h>
#include "Dummy.h"
```

The previous sequence forms the following absolute PATHNAMEs when processing `#include <jni.h>:`

```
/-/USERS/USERCODE/JONES/"JNI.H"
/-/DISK/SYMBOL/CC/LIBRARY/"JNI.H"
```

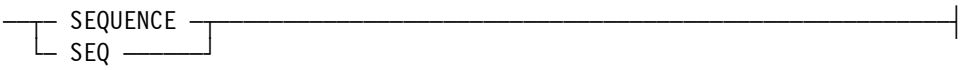
The previous sequence forms the following absolute PATHNAMEs when processing `#include "Dummy.h":`

```
/-/USERS/USERCODE/JONES/"DUMMY.H"
/-/DISK/SYMBOL/CC/LIBRARY/"DUMMY.H"
```

SEQUENCE Option

Enabling the SEQUENCE option causes the compiler to assign new sequence numbers to the source records accepted for compilation.

<sequence option>



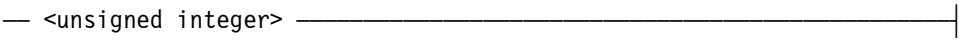
(Type: Boolean; Default: FALSE)

This option affects only those source records read by the compiler following a MERGE option and includes those records omitted by the OMIT option. The SEQUENCE option assigns the current sequence base to the current source record and increments the base by the current sequence increment (refer to "SEQUENCE INCREMENT Option" in this section).

SEQUENCE BASE Option

The SEQUENCE BASE option sets the value of the current sequence base, used by the SEQUENCE option, to <unsigned integer>. The maximum value of <unsigned integer> is 99999999.

<sequence base option>

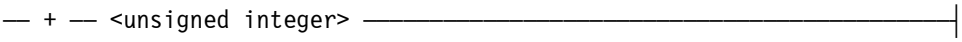


(Type: Value; Default: 100)

SEQUENCE INCREMENT Option

The SEQUENCE INCREMENT option sets the value of the current sequence increment, used by the SEQUENCE option, to <unsigned integer>. The maximum value of <unsigned integer> is 99999999.

<sequence incr option>

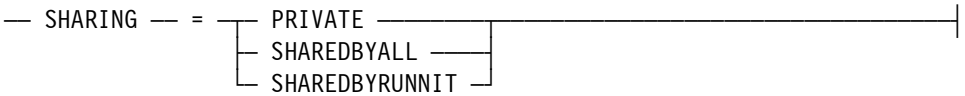


(Type: Value; Default: 100)

SHARING Option

The SHARING option is used in a C library to specify how other programs are to share the library. This option should appear before any program text.

<sharing option>



(Type: Value; Default: PRIVATE)

PRIVATE	A separate instance of the library is started for each invocation of the library. Any changes made to static variables and variables with file scope in the library by the program invoking the library apply only to that particular calling program.
SHAREDBYALL	A single copy of the library is shared by all users. Any changes made to static variables and variables with file scope in the library by the program invoking the library apply to all calling programs.
SHAREDBYRUNUNIT	A run unit consists of a program and all libraries that are called, either directly or indirectly, by that program. A "program" in this context excludes both a library that is not frozen and any tasks that are initiated by the program.

SIGNEDCHAR Option

This option is obsolescent; the PORT option should be used instead.

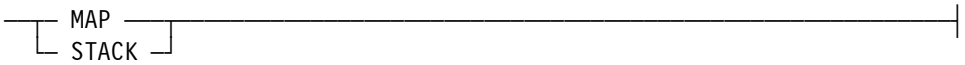
SIGNEDFIELD Option

This option is obsolescent; the PORT option should be used instead.

STACK Option

Stack is a synonym for MAP. Refer to "MAP Option" for details.

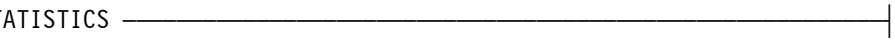
<stack option>



STATISTICS Option

The STATISTICS option, when enabled, causes timing statistics to be gathered for a specified function.

<statistics option>



(Type: Boolean class; Default: FALSE)

The option is examined at the beginning of each function and, if enabled, statistics are gathered for that function. Although the setting of the option can be changed at any time, only its setting at the beginning of a function is significant. At program termination, the statistics information is printed out to the TASKFILE.

Additional characters may precede some function names in the statistics data:

If the function name is preceded by . . .	Then it. . .
M:	is an MCP function that was added by the compiler.
S:	is a SLICESUPPORT function that was added by the compiler.
__	may be an internal C function that was created by the compiler.

The statistics include the number of times the function is called and the amount of time spent in the function, both inclusive and exclusive of the amount of time spent in the functions called by that function.

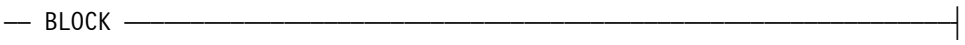
The STATISTICS option can be used with individual subprograms of a bound program. When the bound program terminates, statistics are printed independently for each subprogram compiled with the option.

The STATISTICS option and the TADS option cannot be set at the same time. If both options are set at the same time, the STATISTICS option resets, the production of statistics information is halted, and the following warning is issued.

```
$STATISTICS and $TADS may not be used together.
$STATISTICS has been reset.
```

There are three subordinate options to STATISTICS:

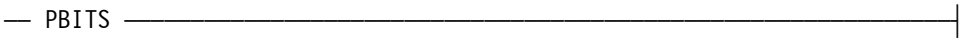
- BLOCK
- PBITS
- TERSE



(Type: Boolean; Default: FALSE)

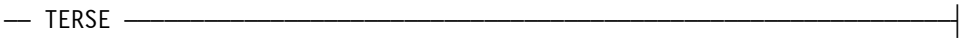
If the BLOCK option is set, statistics are gathered for each execution path in the code. These statistics include the number of times each line of code is executed and the amount of time spent executing that line.

The statistics information includes the frequency of execution of each function or block and the amount of time spent in the function or block.



(Type: Boolean; Default: FALSE)

If the PBITS option is set, statistics are gathered about the initial pbits and other pbits occurring in the program.



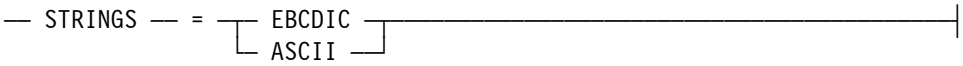
(Type: Boolean; Default: FALSE)

If the TERSE option is set, functions which were not called will not be listed in the statistics output.

STRINGS Option

The STRINGS option specifies the character set to be used to represent strings in the object program. The STRINGS option should appear before any program text.

<strings option>



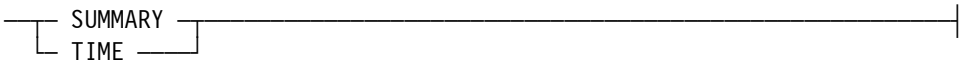
(Type: Value; Default: EBCDIC)

Note: Care should be taken when invoking this option in conjunction with the `setlocale` function for locales other than the C language locale. For example, selecting ASCII strings for a program that calls setlocale to select a locale whose base character set is not ASCII may produce undesirable results.

SUMMARY Option

The SUMMARY option, when enabled, causes a compilation summary to be written to the output listing (LINE), regardless of the setting of the LIST option.

<summary option>

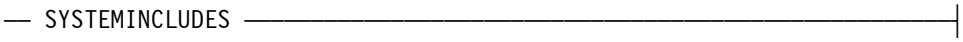


(Type: Boolean; Default: FALSE)

SYSTEMINCLUDES Option

When the SYSTEMINCLUDES option is enabled, a standard library header is **not** read when a standard library is through the `#include` directive; rather, the definitions in the header file are made known to the compiler internally. When the option is disabled, the compiler searches for a library header file to read (subject to the current SEARCH path setting; refer to "SEARCH Option" for details).

<systemincludes option>

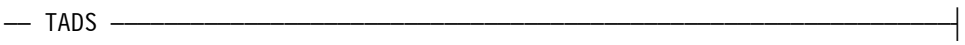


(Type: Boolean; Default: TRUE)

TADS Option

The TADS option is used to compile a program for use with the C Test and Debug System (TADS).

<tads option>

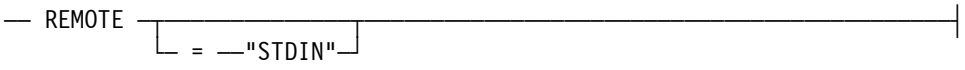


(Type: Boolean class; Default: FALSE)

The TADS option and the STATISTICS option cannot be set at the same time. If both options are set at the same time, the STATISTICS option is reset, the production of statistics information is halted, and the following warning is issued.

```
$STATISTICS and $TADS may not be used together.
$STATISTICS has been reset.
```

For more information, refer to the *C Test and Debug System (TADS) Programming Reference Manual*. There is one subordinate option:



The REMOTE option enables TADS to share a remote input file with the program being tested. Sharing may be necessary if the program opens a remote input file because only one remote input file can be open on a station at any given time. The optional string “STDIN” indicates the internal file name (INTNAME) of the file to be shared with TADS (currently, no file other than the `stdin` stream can be shared).

TARGET Option

The TARGET option designates a specific computer system or group of systems as the target for which the generated object code is optimized. This option can be used to specify all machines that must run the code file.

<target option>

```
── TARGET ── = ── <target-1> [(<target-2>[,<target-3>. . .])]──┤
```

(Type: Enumerated; Default: Installation-defined)

Specification of a secondary target is optional. If specified, a secondary target must be enclosed in parentheses. If more than one secondary target is specified, then the additional targets must be separated from each other by a comma, and the entire list must be enclosed in parentheses.

If a secondary target is specified, the compiler does not generate any operators that are valid for the system or systems identified by the primary target, and that are invalid for the system or systems identified by the secondary target.

For a complete list of allowed target values, refer to the COMPILERTARGET system command description in the *System Commands Operations Reference Manual*.

Examples

```
TARGET=THIS
```

The compiler optimizes the object code file for the system on which it is compiled.

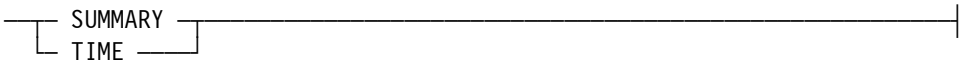
```
TARGET=THIS (ALL)
```

The compiler optimizes the object code file for the system on which it is compiled, but does not generate any operators that are invalid for other machines.

TIME Option

TIME is a synonym for SUMMARY. Refer to "SUMMARY Option" in this section for details.

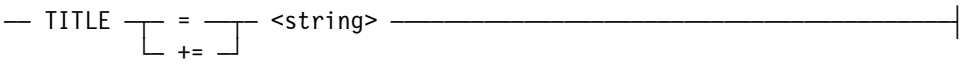
<summary option>



TITLE Option

The TITLE option specifies a string that is to be printed at the top left corner of each page of the program listing. The compiler uses the last value declared as the title on the compilation listing.

<title option>



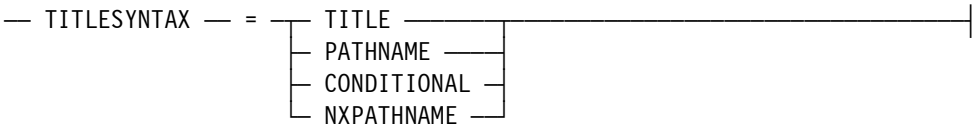
(Type: String)

By default, <string> is the name of the compiler. If the LIST option is disabled or if the PAGESIZE is less than 6, this option has no effect.

TITLESYNTAX Option

The TITLESYNTAX option specifies which file name attribute, title or pathname, and parsing rules to apply when a file name is found in the source.

<titlesyntax option>



(Type: Value; Default: CONDITIONAL)

TITLE	Always use file attributes LTITLE and SEARCHRULE=NATIVE.
PATHNAME	Always use file attributes PATHNAME and SEARCHRULE=POSIX.
CONDITIONAL	Use file attributes PATHNAME and SEARCHRULE=POSIX if one of the following conditions is met: • _ASERIES_SOURCE is greater than or equal to 423 • _POSIX_SOURCE is defined and _ASERIES_SOURCE is not defined Otherwise, use file attributes LTITLE and SEARCHRULE=NATIVE.
NXPATHNAME	Always use file attribute PATHNAME.

C specifies a file name in the following places:

- `#include "<file name>"`
- `fopen ("<file name>" ...)`
- `freopen ("<file name>", ...)`
- `remove ("<file name>")`
- `rename ("<file name>", "<file name2>")`
- `tmpnam (<file name buffer>)`

File Titles

When TITLESYNTAX has the following values, it affects file titles as follows.

TITLE

Symbols	Case	Nodes	File Attribute	Examples

Converts periods (.) and backslashes (\) to slashes (/).	Converts lowercase letters to uppercase letters.	Does not automatically quote nodes if required.	Uses LTITLE.	`a\b.c -> A/B/C`
				`a/b.c -> A/B/C`
				`a/"b.c" -> A/"b.c"`

PATHNAME

Symbols	Case	Nodes	File Attribute	Examples

Converts periods (.) and backslashes (\) to slashes (/).	Converts lowercase letters to uppercase letters.	Does not automatically quote nodes if required.	Uses TITLE.	`a\b.c -> A/B/C`
				`A/b.c -> A/B/C`
				`A/"b.c" -> A/"b.c"`

NXPATHNAME

Symbols	Case	Nodes	File Attribute	Examples

Converts backslashes (\) to slashes (/).	Converts lowercase letters to uppercase letters.	Automatically quotes nodes if required.	Uses PATHNAME.	`A\b.c -> A/"B.C"`
				`A/b.c -> A/"B.C"`
				`A/"b.c" -> A/"b.c"`

File Attributes

The following file attributes affect the file titles as follows.

LTITLE

The LTITLE attribute allows larger limits on the length of a node and on the number of nodes in the file title if the system option SYSOPS LONGFILENAMES is set.

If LONGFILENAMES is reset and a node is longer than 17 characters, the LTITLE attribute generates an attribute error instead of truncating the node.

The code generated by the compiler uses either LTITLE or PATHNAME, depending on whether the code is compiled for ANSI or for POSIX, respectively. Like LTITLE, the PATHNAME attribute allows a larger limit. The arguments in a C program also have their wildcard expansion done using larger limits.

PATHNAME

Since LTITLE cannot be used for POSIX, the file attribute PATHNAME allows a new syntax for file names, wherein files can be specified relative to a current working directory. In this new syntax, file names cannot be altered and family substitution and usercode/* substitution is not allowed.

SEARCHRULE

File attribute SEARCHRULE is used to specify which look-up rules are applied. If SEARCHRULE equals NATIVE, use normal family substitution and usercode/* substitution. If SEARCHRULE equals POSIX, use current working directories, but do not use family substitution or usercode/* substitution.

Refer to the *ClearPath HMP NX and A Series File Attributes Programming Reference Manual* for explanations of file attributes LTITLE, PATHNAME, and SEARCHRULE.

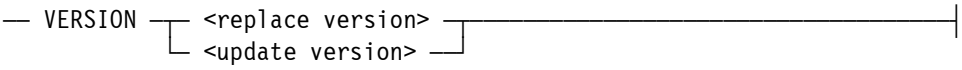
UNSIGNED Option

This option is obsolescent; the PORT option should be used instead.

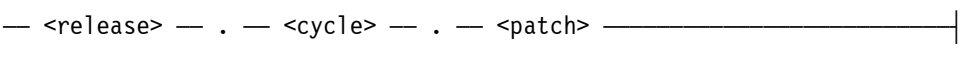
VERSION Option

The VERSION option is used to record information in the mark field of source records that are written to the new source file (NEWSOURCE) or to create new VERSION cards reflecting the current version number of a program symbolic in the NEWSOURCE file.

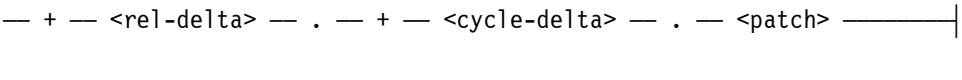
<version option>



<replace version>



<update version>



(Type: Value; Default: 0.000.0000)

The values <release>, <cycle>, <patch>, <rel-delta>, and <cycle-delta> must all be unsigned integers. The <release> option can be no greater than 99. The <cycle> option can be no greater than 999. The <patch> option can be no greater than 9999.

The version information contained on the first version card found by the compiler is saved as the most current version information and is used for the rest of the compilation. The <replace version> sets the new version number to <release>.<cycle>.<patch>. The <update version> adds the <rel-data> to the current release value and the <cycle-delta> to the current cycle value to build the new version number.

If the MERGE and NEW options are enabled, any subsequent permanent version cards are updated with the new version when that card is written to the NEWSOURCE file. If a source record read from the CARD file has a patch mark in the mark field, that record's mark field is updated with a version number containing the current release and cycle numbers plus the patch mark before the record is written to the NEWSOURCE file. (A patch mark is a single unsigned integer in the mark field. Certain system utilities, including the Editor and PATCH, assist in creating patch marks. Refer to the *Editor Operations Guide* and the *System Software Utilities Operations Reference Manual* for more information on these utilities.) The format of the version number written to the mark field is RR.CCC.PPP or RRCCCPPPP (the latter if the patch number requires four significant digits).

For example, given the following SOURCE file and CARD file :

SOURCE File	**CARD File**
`$$ VERSION 2.123.456`	`$VERSION +1.+3.44`

The resulting NEWSOURCE file contains the following version card with the same sequence number as the VERSION card in the SOURCE file:

```
$$ VERSION 3.126.0044
```

Each source record from the CARD file that has a number (in this case, the patch number, 44) in the mark field has the version number 03.126.044 in its mark field in the NEWSOURCE file.

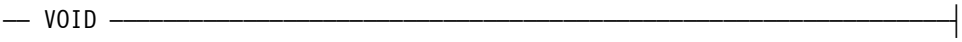
VERSION can be tested in the <Boolean expression> used in compiler control options. VERSION can be compared to a constant with the same syntax as <replace version>. For example:

```
$$SET SSR423 = VERSION >= 42.300.0 AND VERSION <= 42.399.9999
$$IF VERSION >= 43.5.34
    . . .
$$END IF
```

VOID Option

The VOID option, when enabled, causes all input records other than compiler control records to be ignored by the compiler until the option is disabled.

<void option>



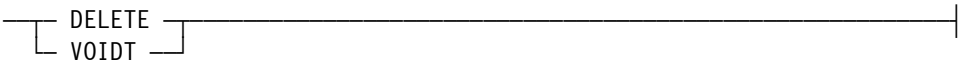
(Type: Boolean; Default: FALSE)

The ignored source is neither listed in the program listing (LINE) nor included in the new source file (NEWSOURCE), regardless of the settings of LIST and NEW. Once the VOID option is enabled, it can be disabled only by a compiler control record in the primary input file (CARD).

VOIDT Option

VOIDT is a synonym for DELETE. Refer to "DELETE Option" in this section for details.

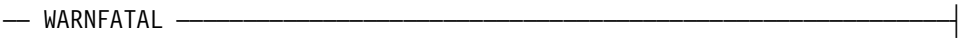
<delete option>



WARNFATAL Option

The WARNFATAL option, when enabled, causes warnings to be treated like errors.

<warnfatal option>

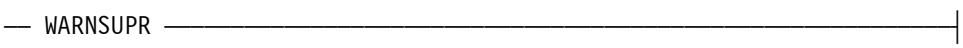


(Type: Boolean; Default: FALSE)

WARNSUPR Option

Enabling the WARNSUPR option prevents the compiler from issuing warning messages.

<warnsupr option>

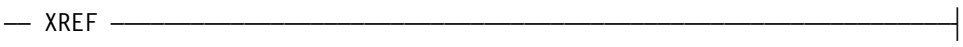


(Type: Boolean; Default: FALSE)

XREF Option

Enabling the XREF option causes the compiler to generate a printed listing containing cross-reference information. This option must appear before any program text.

<xref option>



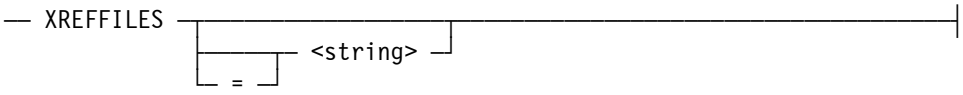
(Type: Boolean; Default: FALSE)

The information is written to the output listing (LINE). No listing is produced if a syntax error occurs during compilation.

XREFFILES Option

The XREFFILES option, when enabled, causes the compiler to generate files that contain cross-reference information.

<xreffiles option>



(Type: Boolean title; Default: FALSE)

These files can be used by the Editor or INTERACTIVEXREF. This option should appear before any program text.

The titles of the generated files are XREFFILES/<code file name>/DECS and XREFFILES/<code file name>/REFS. If a string is specified, the files are XREFFILES/<string>/DECS and XREFFILES/<string>/REFS.

XREF file titles are affected by the value of the compiler option TITLESYNTAX as follows.

Value	File Title
TITLE	XREFFILES/<file>/DECS XREFFILES/<file>REFS
PATHNAME	XREFFILES/<file>/DECS XREFFILES/<file>/REFS
NXPATHNAME	<file>.DECS <file>.REFS

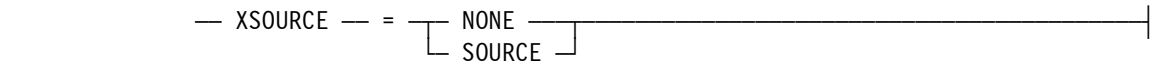
See "TITLESYNTAX Option" for a description of the effects of TITLE, PATHNAME, and NXPATHNAME on node <file>.

No cross-reference files are generated if a syntax error occurs during compilation.

XSOURCE Option

The XSOURCE option controls the generation of macro-expanded source.

<xsource option>



(Type: Value; Default: NONE)

When XSOURCE is set to SOURCE, the compiler generates a file (XSOURCE) that contains the original source with all macros expanded. If XSOURCE is set to NONE, no such file is generated.

Appendix A
Interface to the Library Facility

The A Series library facility is a feature that can be used to structure processes. A library is a program containing one or more functions that can be called by other programs, which are referred to as "calling programs." These functions are "entry points" into the library. Unlike a regular program, which is always entered at the beginning, a library can be entered at any entry point.

Libraries provide all the benefits of functions plus the added advantages that they can be reused and they can be shared by a number of programs. Consolidating logically related functions into a library can make programming easier and program structure more visible. Standard packages of functions (such as plotting and statistics) need not be copied into any calling programs.

Functional Description of Libraries

A subset of the full library capabilities is presented here, along with a description of their use in C programs.

Library Programs

A library program is a program that specifies functions as entry points for use by calling programs. A function in a library is specified to be an entry point by being "exported." A library program becomes a library after it "freezes."

Calling Programs

A calling program is a program that calls entry points provided by a library. In the calling program, an entry point is an external function declared to belong to the library in which the actual function resides.

A library can itself act as a calling program and call other libraries.

Library Directories and Templates

The information used by the Master Control Program (MCP) to match entry points in a library with entry points declared in a calling program is contained in a pair of data structures called the "directory" and the "template", which are built by the compilers.

When a program exports entry points, the object file for that program contains a library directory. A library directory contains a description of all the entry points in the library. This description includes the following information:

- The name of the entry point

- The type of the entry point
- A description of the arguments of the entry points
- Information on how the entry point is provided (refer to "Linkage Provisions" in this appendix)

When a program declares a library and entry points in that library, the object file for the program contains a library template that describes the library and the declared entry points. One template exists for each library declared in the calling program. A template contains the following information:

- A description of the attributes of the library
- A description of all the entry points of the library that are declared by the program. Each description includes the following information:
  - The name of the entry point
  - The type of the entry point
  - A description of the arguments of the entry point

Library Identification

A calling program identifies the library through one of the following:

By title	The file title of the object file is given.
By function	The name of the Support Library function name is given. The system command SL maps a function name onto the file title of the object file.
By initiator	The calling program was activated because a second program tried to do a library link to it. The calling program should link back to the second program.

Library Initiation

On the first call to an entry point of a library or at an explicit linkage request, the calling program is suspended. The description of the entry point in the library template of the calling program is compared to the description of the entry point with the same name in the library directory associated with the referenced library.

If the entry point does not exist in the library or if the two entry point descriptions are not compatible, a run-time error is given and the calling program is terminated. If the entry point exists and the two entry point descriptions are compatible, the MCP automatically initiates the library program (if it has not already been initiated). The library program runs normally until it "freezes," which makes the entry points available. All of the entry points of the library that are declared in the calling program are linked to the calling program and the calling program resumes execution.

If a calling program declares an entry point that does not exist in the library, no error is generated when the library is initiated; however, a subsequent call on that entry point

causes a "MISSING ENTRY POINT" run-time error and the calling program is terminated.

A library can be specified to be permanent or temporary. A permanent library remains available until it is terminated either by the system commands DS (Discontinue) or THAW or by execution of a CANCEL statement. A temporary library remains available as long as users of the library remain. A temporary library that is no longer in use "unfreezes" and resumes running as a regular program.

Linkage Provisions

In the library facility, entry points declared in a calling program are linked to corresponding entry points provided by a library in one of three ways (C only provides direct linkage):

- Directly
- Indirectly
- Dynamically

Direct linkage occurs when the library program contains the function that is exported.

Indirect linkage occurs when the library program exports a function that is declared as an entry point of another library. The MCP then attempts to link the calling program to this second library, which can provide the entry point directly, indirectly, or dynamically.

Dynamic linkage allows a program to determine at link time which library task the calling program will be linked to. The library program must provide a selection function that accepts information provided by the linker and informs the MCP which library to link to.

Sharing Specifications

Users of a library can be restricted through the normal file access features provided by the system. The allowed simultaneous usage of a library can be specified by the creator of the library at compile-time. The library sharing can be PRIVATE, SHAREDBYALL, or SHAREDBYRUNUNIT.

PRIVATE	A separate instance of the library is started for each invocation of the library. Any changes made to static variables and variables with file scope in the library by the program invoking the library apply only to that particular calling program.
SHAREDBYALL	A single copy of the library is shared by all users. Any changes made to static variables and variables with file scope in the library by the program invoking the library apply to all calling programs.
SHAREDBYRUNUNIT	A run unit consists of a program and all libraries that are called, either directly or indirectly, by that program. A "program" in this context excludes both a library that is not frozen and any tasks that are initiated by the program.

Discontinuing Linkage

In some languages, a program can explicitly delink from a library program by canceling the library or delinking from the library. The C compiler does not support explicit delinking; instead, a C program implicitly delinks from a library program by terminating.

Duration Specifications

You can specify the duration of a library at compile-time by using the DURATION compiler control option. A library's duration can be set to one of the following options:

TEMPORARY	The library "thaws" when it has no users linked to it.
PERMANENT	Once the library is initiated, it runs even though there are no users linked to it. The THAW system command will make the library temporary.
CONTROL	The entry point `main` is called after the library is frozen. If the library exits `main`, it is treated as a temporary library.

Refer to the *System Commands Operations Reference Manual* for more infomation on the THAW system command.

Error Handling

Any fault caused (and ignored) by a function in a library that is invoked by a calling program is treated as a fault in the calling program. If ignored by the calling program, this fault causes the calling program to be terminated, but has no effect on the status of the library.

If a library program faults (or is otherwise terminated) before freezing, then all calling programs that are waiting to link to that library program are also terminated.

If a library is terminated while calling programs are linked to it, those programs are also terminated.

The first call on an entry point in a library causes library linkage to be made. In this phase, an attempt is made to locate and establish links to all entry points declared by the calling program. If an entry point declared in the calling program does not exist in the library, the linkage cannot be established and any subsequent calls to that entry point result in a "MISSING ENTRY POINT" error.

Creating C Libraries

A library program is created by declaring, in the source file that declares `main`, at least one procedure with the storage class `asm`. The duration of the library program is by default "temporary" and can be changed with the DURATION compiler control option. The sharing specification is by default "private" and can be changed with the compiler control option SHARING. In the case of temporary or permanent libraries, the freeze is done after `main` executes, but before any functions enrolled by the `atexit` function are called. In the case of control libraries, `main` is the control function.

When the last user of the library delinks, the library thaws, if temporary, and resumes running as a normal program. Any functions enrolled by the `atexit` function are called.

The program that links to a C library may itself be a C library. Or the program may process internal procedures as separate processes. In either case, it is possible to have two entry points executing simultaneously. If this occurs, undefined behavior can result. The CONCURRENTEXECUTION compiler control option should be set if more than one task executes within the program or library at the same time. When this option is set, the system functions such as memory allocation are protected from concurrent tasks. Refer to "CONCURRENTEXECUTION Option" in Section 10 for more information.

Referencing Libraries from C

Referencing non-POSIX Libraries

To declare a library and its entry points, a C program uses an extension of the `#include` preprocessor directive. The extended syntax specifies how the library is located, the name of the library, and the INTNAME of the library. The INTNAME is used to determine the library template in which the entry points are stored; entry points from libraries with different INTNAMEs are placed in different templates.

The INTERFACENAME of the library may be optionally specified after the INTNAME. The INTERFACENAME identifies a particular connection library in a connection library program.

Some examples of `#include` directives used to declare libraries are:

```
#include "mylib.h"      (bytitle="OBJECT/MYLIB", intname="MYLIB")
#include "mathlib.h"    (byfunction="MATHSUPPORT", intname="MATH")
#include "stuff.h"      (byinitiator, intname="STUFF")
#include "mycl.h"       (bytitle="OBJECT/MYCL", intname="MYCL",    \
                         interfacename="CL100")
```

The included file contains the declarations of the library entry points. Any function declared with the storage class `extern` is considered an entry point to the library. (Other non-`extern` declarations may also be present in the file.)

Entry points with linkage specifications are identified in the library template with their names uppercased if the specified language is case-insensitive.

Referencing POSIX Libraries

When you are compiling code for POSIX, libraries no longer access their own files when called in an entry point. Instead, the library uses the same file descriptors used by the program.

> **Caution**
>
> A POSIX library must not use functions in `<stdio.h>` to do I/O operations while in an entry point. Use the POSIX I/O functions `open`, `close`, `read`, and `write` (which are defined in `<fcntl.h>` and `<unistd.h>`) instead.

For more information regarding referencing POSIX libraries, see the *C Programming Reference Manual, Volume 2*.

Hidden Parameters

C programs rely heavily on pointers for data manipulation and parameter passing. While this may be considered a strength of the language, it is also a limitation when interacting with programs written in other languages. A C pointer is implemented as an index into C's run-time data area called the *heap*; unlike an ALGOL pointer, a C pointer cannot be dereferenced outside the scope of the C program that "owns" the pointer.

To bypass these limitations and provide maximum flexibility when calling library entry points, "hidden" parameters have been supplied. These parameters consist of both variables and procedures that can be passed to a library entry point to allow the entry point direct access to the C heap. The hidden parameters are declared in the function prototype of the library entry point, but (because they are hidden) do not appear in the actual entry point call. Of course, within the library entry point itself, corresponding formal parameters must be declared.

The following describes the name and purpose of each hidden parameter, along with a description of the corresponding ALGOL formal parameter (although the hidden parameters may apply to other languages as well; see Tables 7-1 through 7-4 under "Parameter Type Matching" in Section 7). The examples at the end of this appendix illustrate how some of the hidden parameters may be used.

__copy_to_ptr_t

This parameter is a procedure that copies data from a character array to a C pointer. The formal parameter is:

```
PROCEDURE COPYTOPTR (LEN, BUFF, OFF, PTR);
   VALUE   LEN, OFF, PTR;
   INTEGER LEN, OFF, PTR;
   EBCDIC  ARRAY BUFF [*];
   FORMAL;
```

The procedure copies `LEN` bytes beginning at `BUFF[OFF]` to the C pointer `PTR`, which should be a `void` pointer. This provides a convenient method for copying data into the C program's heap.

__copy_from_ptr_t

This parameter is a procedure that copies data from a C pointer into a character array. The formal parameter is:

```
PROCEDURE COPYFROMPTR (LEN, PTR, BUFF, OFF);
   VALUE   LEN, OFF, PTR;
   INTEGER LEN, OFF, PTR;
   EBCDIC  ARRAY BUFF [*];
   FORMAL;
```

The procedure copies `LEN` bytes from the C pointer `PTR` into `BUFF[OFF]`. The C pointer should be a `void` pointer. This provides a convenient method for copying data out of the C program's heap.

__errno_t

The formal parameter should be a by-reference integer. This hidden parameter passes the C variable `errno` with external linkage to the entry point.

__file_t

The formal parameter should be a by-reference file. This parameter allows the user to pass a file from a C program to an ALGOL entry point (files cannot be passed to Pascal or FORTRAN). Unlike the other hidden parameters, an actual parameter is required, which must be the `_file_no` field of a `FILE` struct. For example:

```
extern "ALGOL" algolproc (__file_t);
...
f = fopen ("blit", "r");
algolproc (f->_file_no);
```

This example passes the file `f` to the procedure `ALGOLPROC`. This can be used for obtaining file attribute information that is not directly available from C.

Notes:

- *The FILE structure contains file state information not accessible outside the C program; it is therefore potentially dangerous to perform I/O to a particular file from both a C program and a library entry point.*

- *You cannot use the hidden parameter __file_t when compiling code for POSIX.*

__free_t

This parameter is the C `free` function. The formal parameter is:

```
INTEGER PROCEDURE FREE (CPTR);
   VALUE   CPTR;
   INTEGER CPTR;
   FORMAL;
```

Calling this function from the library entry point deallocates the space in the C program's heap pointed to by `CPTR`. The value 0 is returned.

__heap_t

The formal parameter should be a by-reference EBCDIC array. This hidden parameter is the C run-time heap. The heap can only be passed from C programs compiled with the TINY or SMALL memory model.

Note: *Using this parameter is potentially dangerous since storing data directly into the heap can lead to data corruption. Use it carefully.*

__heap_to_ptr_t

This parameter is a procedure that converts a C pointer into an ALGOL pointer. The formal parameter is:

```
INTEGER PROCEDURE HEAPTOPTR (CPTR, APTR);
   VALUE   CPTR;
   INTEGER CPTR;
   POINTER APTR;
   FORMAL;
```

`CPTR` is the C pointer and `APTR` is the ALGOL pointer. The C pointer should be a `void` pointer. The procedure returns 0.

__install_memory_t

This parameter is a procedure that installs an array row into the heap. The INSTALLMEMORY compiler control option must be enabled to use this parameter. The formal parameter is:

```
INTEGER PROCEDURE INSTALLMEMORY (SEG, CHAR_ARRAY);
   VALUE SEG;
   INTEGER SEG;
   EBCDIC ARRAY CHAR_ARRAY[*];
   FORMAL;
```

The INSTALLMEMORY function installs `CHAR_ARRAY` as a segment in the heap of a C program and returns a far pointer to `CHAR_ARRAY[0]`. If `SEG` is zero (0), the next available heap segment number is used. Otherwise, `SEG` is used as the segment number. If all segments are in use or if the segment specified by `SEG` is in use, then the array is not installed in the heap and the function returns zero. Refer to "FARHEAP Option" in Section 10 for information on the INSTALLMEMORY compiler control option.

__malloc_t

This parameter is the C `malloc` function. The formal parameter is:

```
INTEGER PROCEDURE MALLOC (BYTES);
   VALUE   BYTES;
   INTEGER BYTES;
   FORMAL;
```

Calling this function from the library entry point allocates space in the C program's heap and returns a C pointer to the allocated space.

These hidden parameters are also available through the normal library interface. Whenever a C program imports or exports library entry points, it also exports entry points for `MALLOC`, `FREE`, `HEAPTOPTR`, `COPYTOPTR`, `COPYFROMPTR`, and `INSTALLMEMORY`. The library can import these entry points by declaring them as being from a library with linkage "byinitiator". A circular library linkage is established between the C program and the library. Importing the C procedures can reduce the parameter-passing overhead of hidden parameters, as well as eliminate the hidden parameters from the entry point prototypes.

Examples

C Program Calling ALGOL Library

Library Header File

```
/*File CTOA/H */
extern "ALGOL" {
   void  fill (int (&) [ ], char (&) [ ]);
   int   print (int*, int, char*, int (&) (int*, int, char*));
   char* stationname (__file_t);
}
```

C Program

```
#include "ctoa.h" (bytitle="OBJECT/CTOA/A", intname="LIB")
#include <stdio.h>

int print_em (int *pi, int pi_size, char *pc) {
/* print contents of pi and pc*/
   int i, k;
   k = 0;
   for (i = 0; i < pi_size; i++) {
      k += printf ("%i", *pi);
      pi++;
   }
   k += printf ("\n%s\n", pc);
   return (k);
}

main ( ) {
   #define IA_SIZE 10
   #define IC_SIZE 27
   int  ia [IA_SIZE], n;
```

```
    char ca [IC_SIZE], *ps;


    /* call FILL to fill arrays with data */
    fill (ia, ca);

    /* print them via PRINT */
    n = print (&ia [0], IA_SIZE, &ca [0], print_em);
    printf ("chars printed = %i\n", n);


    /* STATIONNAME gets stdout's stationname */
    ps = stationname (stdout ->_file_no);
    printf ("station name=%s\n", ps);
 }
```

ALGOL Library

```
BEGIN
   LIBRARY CLIB (LIBACCESS = BYINITIATOR);

   INTEGER PROCEDURE MALLOC (BYTES);
      VALUE    BYTES;
      INTEGER  BYTES;
      LIBRARY  CLIB;
   PROCEDURE COPYTOPTR (LEN, BUF, OFF, PTR);
      VALUE    LEN, OFF, PTR;
      INTEGER  LEN, OFF, PTR;
      EBCDIC   ARRAY BUF [*];
      LIBRARY  CLIB;
   PROCEDURE FILL (IA, CA);
   % fill arrays IA and CA with data.
      INTEGER ARRAY IA [*];
      EBCDIC  ARRAY CA [*];
   BEGIN
     REAL I;
      FOR I := 0 STEP 1 UNTIL 10 DO IA [I] := I;
      REPLACE CA [0] BY "abcdefghijklmnopqrstuvwxyz", 48"00";
   END FILL;

    INTEGER PROCEDURE PRINT (PI, PI_SZ, PC, PRINT_FCN);
    % call PRINT_FCN to print contents of PI and PC
       VALUE   PI, PI_SZ, PC;
       INTEGER PI, PI_SZ, PC;
       INTEGER PROCEDURE PRINT_FCN (PI, PI_SZ, PC);
          VALUE   PI, PI_SZ, PC;
          INTEGER PI, PI_SZ, PC;
          FORMAL;
    BEGIN

       PRINT := PRINT_FCN (PI, PI_SZ, PC);
    END PRINT;
```

```
   INTEGER PROCEDURE STATIONNAME (F);
   % return C pointer to file F's station name.
      FILE F;
   BEGIN
      EBCDIC  ARRAY TMP [0:300];
      POINTER P;
      INTEGER PTR, LEN;
      REPLACE P:TMP BY F(1).STATIONNAME;
      REPLACE P-1 BY 48"00";
      LEN := OFFSET (P);
      PTR := MALLOC (LEN);
      COPYTOPTR (LEN, TMP, 0, PTR);
      STATIONNAME := PTR;
   END STATIONNAME;

   EXPORT FILL, PRINT, STATIONNAME;
   FREEZE (TEMPORARY);
END.
```

C Program Calling Pascal Library

Library Header File

```
/* File CTOP/H */
extern "Pascal" {
   char *upcase (char (&) [ ], int, __malloc_t, __copy_to_ptr_t);
   void inc_x (struct s &);
}
```

C Program

```
#include "ctop.h" (bytitle="OBJECT/CTOP/P", intname="LIB")
#include <stdio.h>

main ( )  {
   char ca[ ] = "abcdefghijklmnopqrstuvwxyz";
   char* up_ca;
   struct s { double y; int x; } ss = { 3.14159, 25 };

   /* make a copy of ca, upcasing each character */
   up_ca = upcase (ca, sizeof (ca) -1);
   printf ("up_ca=%s\n", up_ca);

   /* pass struct to INC_X, which increments field 'x' */
   inc_x (ss);
   printf ("ss.y=%f ss.x=%i\n", ss.y, ss.x);
}
```

Pascal Library

```
library p (output);
   interface
      type char_array = packed array [1..65535] of char;
```

```
      type struct_s   = record
                           y : real;
                           x : integer;
                        end;
      function upcase (var ca : char_array;
                       sz     : integer;
                       function malloc (i : integer) : integer;

                       procedure copytoptr (len      : integer;
                                            var buff : char_array;
                                            off, ptr : integer)
                       ) : integer;
      procedure inc_x (var ss : struct_s);
   end;
   function upcase;
   { converts contents of ca to upper case, returning C pointer to new array }
   var
      i, p : integer;
      b    : char_array;
   begin
      p := malloc (sz);
      for i := 1 to sz do begin
         b [i] := chr (ord (ca [i]) + 64);
      end;
      copytoptr (sz, b, 0, p);
      upcase := p;
   end;

   procedure inc_x;
   { increments field 'x' of record ss }
   begin
      ss.x := ss.x +1;
   end;

begin
   freeze;
end.
```

C Program Calling FORTRAN77 Library

Library Header File

```
/* File CTOF/H */
extern "FORTRAN" {
   void  scale (double [ ], int);
   int   strlen (char [ ]);
}
```

C Program

```
#include "ctof.h" (bytitle="OBJECT/CTOF/F", intname="LIB")
#include <stdio.h>
```

```
main ( ) {
   double da [ ] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
   char   s  [ ] = "this is a string of 34 characters";
   int    n;

   /* pass da to SCALE, which multiplies each element by 3 */
   scale (da, (sizeof(da) / sizeof (double)));
   for (n = 0; n < 5; n++) printf ("da[%i]=%.2f\n", n, da [n]);

   /*use STRLEN to calculate the length of string s */
   n = strlen (s);
   printf (">%s< is %i chars long\n", s, n);
}
```

FORTRAN77 Library

```
  BLOCK GLOBALS
       EXPORT (SCALE, STRLEN)
    END

    PROGRAM MAIN
       CALL FREEZE ('TEMPORARY')
    END

    SUBROUTINE SCALE (DA, DALEN)
       REAL DA (*)
       INTEGER DALEN
       DO 10 I=1, DALEN
          DA (I) = DA (I) * 3
10     CONTINUE
    END

    INTEGER FUNCTION STRLEN (S)
       CHARACTER*(*) S
       I = 1
10     IF (S(I:I).EQ.CHAR(0)) GOTO 20
       I = I + 1
       GOTO 10
20     STRLEN = I
    END
```

C Program Calling COBOL Libraries

Library Header Files

```
/* File CTOCB/C74/H */
extern "COBOL" void proceduredivision (int &, double &);

/* File CTOCB/C85/H */
extern "COBOL" void cobolproc (int &, double &);
```

C Program

```
#include "ctocb.c74.h" (bytitle="OBJECT/CTOCB/C74", intname="LIB")
#include "ctocb.c85.h" (bytitle="OBJECT/CTOCB/C85", intname="LIB2")
#include <stdio.h>

main ( ) {
   double d = 3.141;
   int    i = 72;

   /* add i to d */
   proceduredivision (i, d);
   printf ("i=%i d=%f\n", i, d);

   /* add i to d */
   cobolproc (i, d);
   printf ("i=%i d=%f\n", i, d);
}
```

COBOL74 Library

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
   77 II BINARY  PIC 9(11) RECEIVED BY REFERENCE.
   77 XX REAL              RECEIVED BY REFERENCE.
PROCEDURE DIVISION USING II, XX.
P1.
   ADD II TO XX.
   EXIT PROGRAM.
```

COBOL85 Library

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MSGLIB IS LIBRARY PROGRAM.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROGRAM-LIBRARY SECTION.
LB MSGLIB EXPORT
    ATTRIBUTE SHARING IS PRIVATE.
    ENTRY PROCEDURE COBOLPROC.
PROCEDURE DIVISION.
P1.
    CALL SYSTEM FREEZE TEMPORARY.
    STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLPROC.
ENVIRONMENT DIVISION.
DATA DIVISION.

WORKING-STORAGE SECTION.
    77 II BINARY  PIC 9(11) RECEIVED BY REFERENCE.
```

```
    77 XX REAL              RECEIVED BY REFERENCE.
PROCEDURE DIVISION USING II, XX.
P1.
    ADD II TO XX.
    EXIT PROGRAM.
END PROGRAM COBOLPROC.
```

ALGOL Program Calling C Library

ALGOL Program

```
BEGIN
   LIBRARY CLIB (LIBACCESS=BYTITLE, TITLE="OBJECT/STREAM/C.");

   % entry points exported by C "for free"
   INTEGER PROCEDURE MALLOC (BYTES);
      VALUE   BYTES;
      INTEGER BYTES;
      LIBRARY CLIB;
   INTEGER PROCEDURE FREE (CPTR);
      VALUE   CPTR;
      INTEGER CPTR;
      LIBRARY CLIB;
   INTEGER PROCEDURE HEAPTOPTR (CPTR, APTR);
      VALUE   CPTR;
      INTEGER CPTR;
      POINTER APTR;
      LIBRARY CLIB;

   % C library entry point
   INTEGER PROCEDURE WRITELINE (CPTR);
      VALUE   CPTR;
      INTEGER CPTR;
      LIBRARY CLIB;

   PROCEDURE XFER (S);
      VALUE  S;
      STRING S;

   BEGIN
      POINTER APTR;
      INTEGER CPTR;
      % allocate room in C heap for string
      CPTR := MALLOC (LENGTH (S) + 1);
      HEAPTOPTR (CPTR, APTR);
      % move string into heap
      REPLACE APTR BY S, 48"00";
      % write the string
      WRITELINE (CPTR);
      % deallocate the string
      FREE (CPTR);
   END XFER;
```

```
   XFER ("HELLO WORLD");
   XFER ("THIS IS AN EXAMPLE");
   XFER ("GOODBYE WORLD");
END.
```

C Library

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

asm WRITELINE(char * pc) {
   puts (pc);
}

void cleanup (void) {
   /* called after main exits */
   puts ("all done");
}

main ( ) {
   atexit (cleanup);
}
```

Appendix B
Internal Compiler Limits

The ANSI C Standard specifies certain internal translation limits that a conforming C compiler must meet. In every case, the A Series implementation exceeds the stated limits. In some cases, the actual limit varies depending on the amount of stack space allocated to the compiler when it is invoked. Also, the use of one C construct may have an effect on the limits for other C constructs, since they may share certain data structures within the compiler. Table B-1 provides the translation limits given by the ANSI C Standard beside the corresponding limits for this implementation.

Table B–1. A Series C Compiler Limits

Limit Description	ANSI Standard	A Series C
Nesting levels of compound statements, iteration control structures and selection control structures	15	> 60
Nesting levels of conditional inclusion	8	> 250
Pointer, array and function declarators (in any combination) modifying an arithmetic, a structure, a union or an incomplete type in a declaration	12	> 240
Declarators nested by parentheses within a full declarator	31	> 500
Expressions nested by parentheses within a full expression	32	> 160
Significant internal characters in an internal identifier or a macro name	31	250
Significant initial characters in an external identifier	6	250
External identifiers in one translation unit	511	> 13,500
Identifiers with block scope declared in one block	127	> 13,500
Macro identifiers simultaneously defined in one translation unit	1,024	> 10,000
Arguments in one function definition	31	255
Arguments in function call	31	255
Arguments in one macro definition	31	> 100
Arguments in one macro invocation	31	> 100
Characters in a logical source line	509	2047

Table B–1. A Series C Compiler Limits

Limit Description	ANSI Standard	A Series C
Characters in a character string literal or wide string literal (after concatenation)	509	> 3,000
Bytes in an object	32,767	
In-heap object: TINY memory model		65,358
SMALL memory model		100,663,295
LARGE memory model		65,532
HUGE memory model		1,048,572
Non-heap object		1,048,572
Nesting levels for #included files	8	32
Case labels for a switch statement (excluding those for any nested switch statements)	257	1,021
Members in a single structure or union	127	> 10,000
Enumeration constants in a single enumeration	127	> 10,000
Levels of nested structure or union definitions in a single struct-declaration-list	15	98

Appendix C
Porting C Applications from Other Systems

This appendix contains information you can use to port a C application (written for another computing environment) onto an A Series system. It describes how to get the source onto the system and compile it. It also describes any differences that can be expected when running the application on the A Series system.

Getting Source Onto A Series Systems

One of the first problems with porting software is getting the source onto the A Series system. The method of transferring source files from another system is to upload the files through a data communications line.

Source Files

Format

Source files on the enterprise server are stored as zero or more fixed-length records. The default character set is EBCDIC, but ASCII is also supported. The internal and external character coding of files is controlled by using the file attributes INTMODE and EXTMODE respectively. (See the *File Attributes Programming Reference Manual* for information on INTMODE and EXTMODE file attributes.)

Uploaded files are generally translated to EBCDIC for use and then translated back to ASCII when downloaded back into the original environment. The C Compiler, CANDE, EDITOR, and SYSTEM/PATCH utilities use the EBCDIC character set. However, these softwares can access files that have their physical format set to ASCII (EXTMODE= ASCII) and their logical format set to EBCDIC (INTMODE=EBCDIC) by allowing the system to translate the file between the internal and external character encoding.

Each record contains program text in columns 1 through 72, padded with trailing blanks as necessary. The newline character (commonly found in other environments) is not stored because it is implied by the end of the fixed length record. Source lines from other environments that extend beyond 72 characters must be imported so that a trailing backslash (\) is placed in column 72 to indicated that the line is continued at column 1 of the next source record.

A sequence number is stored in columns 73 through 80 and is used to identify the line in cross-reference information, patching control, and error reports. Unlike other systems, this sequence number is associated with a particular source record and is not a relative line number within the file. Adding new lines within the source by using standard enterprise server tools does not automatically change the sequence numbers of subsequent records.

The last 10 characters, columns 81 through 90, are used to store the version information that is shown on compiler listings. This practice allows the history of changes to the source to be shown; it may be left blank.

Source lines for C in other systems may be longer than 72 characters. To handle these lines, the following convention is used by the C compiler. The same convention is also used for text stream files; however, the length of the data field in the record depends on the text stream FILEKIND attribute.

- Lines longer than 72 characters are folded at 71 characters and a trailing backslash (\) is appended. The remainder of the line starts at column 1 on the next record. This is repeated as many times as necessary.

- If a line is supposed to end in a backslash (not a valid C source line, but can be done in text streams), two backslashes are used instead. The next line should be all blanks. When read, the last backslash causes the next line to be read, but since it is all blanks, it is ignored. The next-to-last backslash remains.

C source lines that end in a backslash are not valid in the sense that the C preprocessor removes such lines before the C compiler reads them. However, from a C source file and C programmer perspective, C macros can and do have lines in them that end in a backslash that indicates the macro continues on the next line.

In both C source and in text streams, trailing blanks on a line are ignored. Also, the last line in a file is always assumed to have a newline character at the end, regardless of whether one was written.

Titles and Directories

An A Series file title consists of the following:

- A usercode that identifies the owner of the file
- One to twenty nodes separated by slashes
- A familyname that identifies the pack or packs on which the file resides

File titles used inside a C program can also have a hostname that identifies which system the file resides on. The hostname syntax in a file title is

```
(usercode)filename on packname at hostname
```

Use of the hostname syntax is supported in the following functions: fopen(), frename(), rename() and remove(). Also, when the command line parser is invoked, the 'at hostname' is included as part of the file title.

For example, assume a C program with the declaration `main(int argc, char *argv[])` is run as follows:

```
u filename name1 (usercode)filename on packname at hostname name2
```

The command line parser would return the following result:

```
argv[0] = "filename"
argv[1] = "name1"
argv[2] = "(usercode)filename on packname at hostname"
argv[3] = "name2"
```

Most enterprise server softwares, including the C compiler and C programs, capitalize letters in titles found outside of quoted strings. A directory (node) does not have to be created before a file is stored under it. A directory is automatically removed when the last file under it is removed. A directory and a file can have the same name. There is no "current directory"; the complete file title has to be specified every time. Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

FILEKIND

Many other systems use a convention that stores the kind of the file as a period suffix onto the last node of the file title. For example, “example.c” could be a C source file, where “example.h” would be the corresponding header file and “example.o” would be the corresponding object file. While it is possible to use this naming convention on the enterprise server, it is also necessary to specify the FILEKIND attribute, indicating the type of contents of the file. In the preceding example, “example.c” and “example.h” would be FILEKIND=CCSYMBOL and “example.o” would be FILEKIND=CCCODE.

By convention on the enterprise server, object files are prefixed by OBJECT/. As a result, the files in the previous paragraph could be EXAMPLE/C, EXAMPLE/H and OBJECT/EXAMPLE/C. The addition of the object file prefix is performed by default by many enterprise server products. Note that the object file is the complete source name prefixed by OBJECT/.

In general, the C compiler changes a period (.) or backslash (\) to a slash (/) when it finds them in a file, except when that produces an illegal enterprise server file title, in which case it is dropped. The C preprocessor makes the same changes to the titles of files being included by using the #include directive or to file titles listed as arguments to the fopen(), frename(), rename(), and remove() functions. Refer to Section 10, “TITLESYNTAX Option,” for more information on the file title changes that occur during compilation.

Data Communications

There are several ways to transfer files to the enterprise server:

- Use a terminal emulator such as INFOConnect version 2.0 or later or INFOVIEW II 3.5 or later to transfer files over Local Area Network (LAN) connections.
- Use the enterprise server COPY command if the system uploading files has File Transfer Protocol (FTP) software running.
- Use a custom file transfer program.

File Transfer Program

Transferring a file through data communications requires a properly configured data communications line. The line should have a TTY or APL/ASCII algorithm. It should be in half duplex. Pseudotabs should be reset. If the APL/ASCII algorithm is being used, invisible mode should be set (it may also be set dynamically by the PC or B20 or by the A Series program). The baud, parity, and number of stop bits should match the sending line. Most lines are configured for ASCII 7-bit characters, but other character codes can be supported.

Any program that drives a data communications line to must conform to A Series timing rules. Generally characters within a line can be sent at full line speed. However, it may be necessary to impose a small delay between the first character and the second character.

End-of-line is indicated by sending up the carriage return character (ASCII value '\x0D'). After sending the carriage return character, a delay of at least 1/2 second should be imposed before the first character of the next line is sent.

The following C program sends a prompt of "]]]START" down the line, then reads from the data communications line until "]]]END" is received. The file UPLOAD is created and each line is stored in it with a sequence number and a blank mark field.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main() {
        FILE * upload;
        char buf[4000];
        upload = fopen("UPLOAD", "w, FILEKIND=CCSYMBOL");
        if (upload == NULL) {
              perror ("Cannot create UPLOAD");
              exit (EXIT_FAILURE);
        }
        puts("]]]START");
        while (fgets(buf, 4000, stdin) !=NULL) {
               if (strcmp(buf, "]]]END\n") == 0) break;
               fputs(buf, upload);
        }
        fclose(upload);
}
```

Common Portation Problems

You might encounter the following problems when porting C code to an enterprise server:

1. Pointer alignment—Many programs assume that all pointers can be treated in the same manner. You can detect these problems at run-time by setting the $BOUNDS(ALIGNMENT) compiler control option.

2. Signed magnitude representation—Encryption (and other) algorithms might assume that integers are stored in a particular format.

3. Unsigned types—Since the enterprise server stores the sign bit separately from the data, different behavior can occur when casting between signed and unsigned types.

When you are porting an application for the first time, it is strongly recommended that you set the BOUNDS(ALIGNMENT) compiler control option. When the application is fully tested, the option can be reset to remove the performance penalty associated with its use.

For more information on portation issues, see "Language Differences" later in this section.

Compiling/Linking/Running Source Files

An example of developing the “hello world” program is presented first. It is assumed the station is logged-on under CANDE. Lines that start with a number sign (#) are responses from the system. The leading numbers are the sequence numbers; they precede lines of text that are entered in CANDE.

```
make test/c cc
#WORKFILE TEST/C: CC
100#include <stdio.h>
200main() {
300     printf("Hello world\n");
400}
save
#UPDATING
#WORKSOURCE TEST/C SAVED
compile test/c
#COMPILING 1234
#ET=1.7 PT=0.0 IO=0.5
run test/c
#RUNNING 1235
Hello world
#ET=0.2 PT=0.0 IO=0.1
```

Compiling

Traditionally, an A Series program is developed as a single file. Compilation of 100,000 to over 700,000 lines is routine. As a result, the tools available for program development are oriented towards handling a single large file.

Systems created in C tend to be written as many small files. The MAKE Utility, CC tool, and LD tool (see the discussion of the MAKE Utility, CC tool, and LD tool later in this section) can be used for C file compile- and run-time dependency control, compilation, and binding, respectively. However, tools for managing many small files do not currently exist.

INITIALCCI

One of the tools of the C compiler is the INITIALCCI (Initial Compiler Control Images) file. The file contains compile-time options to be used for all compiles by the user. The following is an example INITIALCCI file; refer to Section 9 for the meaning of the various options.

```
search   = '$-/=;'
search   += 'UNIX/=;'
BATCH:    reset list set lineinfo
longlimit = 18000
```

Running, Including File Equation

If the C program exists as one source file and if any library function used has its header file included using the `#include <...>` syntax, the object file can be run directly. Otherwise an explicit Binder step is required.

The definition of the function `main` determines how the C program is invoked. If `main` has no arguments, run the program `OBJECT/FOO/C` from CANDE by entering the following:

```
r foo/c
```

Then, run the program `OBJECT/FOO/C` from WFL by entering the following:

```
RUN OBJECT/FOO/C;
```

If `main` is declared with the `argc, argv` arguments, run the program from CANDE by entering the following:

```
r foo/c("parameters, lower case is preserved")
```

or

```
u foo/c parameters, lower case is capitalized
```

Then, run the program from WFL by entering the following:

```
RUN OBJECT/FOO/C("Parameters, lowercase is preserved");
```

The parameter string is parsed into separate `argv` strings, each separated by one or more blanks. However, if two or more `argv` strings look like an A Series file title they are combined. For example, the following parameter string is parsed as two `argv` strings, "FRANK ON PACK AT HOST" and "HARRY AT FRANKS":

```
u foo/c frank on pack at host harry at franks
```

Binding

Binding (synonymous with linking in UNIX, B20, and PC environments) is required if the program consists of multiple separately compiled source files or if standard library functions are used without including the corresponding header files by using the #include <. . .> syntax. See the *C Programming Reference Manual, Volume 2: Headers and Functions* for more information about binding requirements.

Refer to Section 9 of this manual for a description of how to bind C object files.

Language Differences

The A Series C compiler is based on the ANSI C Standard; it therefore differs in many ways from early compilers based on *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or the Portable C Compiler (PCC) program. This appendix identifies the major characteristics of the A Series C compiler that are different from other C compilers.

ANSI C Standard Conformance

To flag code that does not conform to the ANSI C standard, use the ANSI compiler control option. If your program uses some of the Kernighan and Ritchie features that are not part of ANSI C, you can use the KNR subordinate options of the PORT compiler control option to loosen some of the ANSI restrictions.

Identifiers

Identifiers can be up to 250 characters long. External linkage is significant up to the full identifier length instead of the first 6 or 8 characters. Internal linkage is significant up to the full identifier length instead of only the first 31 characters as required by the ANSI C Standard.

Lowercase and uppercase letters are preserved and are significant. Dollar signs are not allowed in identifiers.

Types

The following types are discussed:

- Char types
- Integer types
- Floating types
- Pointer types
- Common type portation problems

Char Types

Characters are 8 bits wide. The plain `char` type is unsigned by default. This default can be changed to be signed by the $PORT (SIGNEDCHAR) option. This affects all variables of type `char`, even in arrays and structures.

Signed characters are stored in two's-complement format. The values 0 through 127 have the same bit pattern for signed and unsigned types. Unsigned characters are stored in two's complement format if the $PORT (CHAR2) option is enabled.

Variables of type `char` that are declared as `extern` in a module are used in that module based on the PORT suboption of that module. Therefore, the use of char objects that are declared `extern` could produce undesirable results if the PORT suboptions SIGNEDCHAR, UNSIGNED, and CHAR2 differ between the modules in a bound program where the char object is declared and where it is used.

The default character set used at run time is EBCDIC. This can be changed by the $ SET STRINGS=ASCII option. When ASCII is set, all characters are stored using the ASCII character set and all I/O is translated to or from ASCII if necessary.

Six characters are stored for each word instead of the usual four or two. It is not valid to compare multiple characters at once by casting a character pointer into an integer

pointer and doing integer comparisons. This comparison results in a run-time error if the BOUNDS(ALIGNMENT) compiler control option is set; otherwise undefined behavior is likely to occur.

Integer Types

Integer type representation differs between A Series C and C language on most other machines. A Series C uses a signed-magnitude representation for integers instead of two's-complement representation. Furthermore, A Series C integers use only 40 of the 48 bits in the word: a separate sign bit and the low order 39 bits for the absolute value.

Unsigned types in A Series C use the same representation as signed types, except that the sign bit is always zero. Negative values, when casted to an unsigned type, are added to `(INT_MAX+1)`, producing a value within the signed integer range. This value does not change when cast back to a signed type.

The types `short`, `int`, and `long` are all the same size.

Bit operations (bitwise AND, OR, exclusive OR, and NOT) on signed values affect only the 40 bits used by integers. Bit operations on unsigned values conform to the mathematical definitions given in the ANSI C standard. Because the sign bit is not adjacent to the other bits, it is not possible to shift into or out of the sign bit.

Operations on unsigned integer types are more expensive than on signed types. The $RESET PORT (UNSIGNED) option makes unsigned equivalent to signed types and should be used on programs that do not depend upon the wraparound or bit operation properties of unsigned types.

By default, bit fields in structures or unions that are of type plain `int` are unsigned. The default can be changed to signed by the $PORT (SIGNEDFIELD) option.

Floating Types

By default, `double` type is the same size and range as `float` type. Note that the A Series `float` type has about 11 digits of precision. The default can be changed to be the same size and range of `long double` type by the $RESET DBLTOSNGL option (Double to Single).

Pointer Types

Pointers are internally stored as integer values. A pointer to a `char` is the number of bytes from the start of addressable memory (the C heap), not the machine memory. A pointer to an `int` or a `float` is the number of words, and a pointer to a `long double` is the number of double words, from the start of addressable memory. Implicit and explicit casts between pointers of different types adjust the value. Casts that are invisible to the compiler must be avoided, such as an invisible cast declaring a prototype to an external procedure as taking a `char *` parameter, but defining the procedure in another compilation unit as taking an `int *` parameter. See "Pointer Alignment" in this section for the implication of implicit and explicit casts between pointers of differing types.

Any pointer that is itself pointed at or any pointer stored in an array, structure, or union is always stored as if a `void` cast were done. These pointers may be cast safely, either visibly or invisibly.

Pointer arguments to old-style functions are always passed as if a `void *` cast were done.

The allocation of objects is not necessarily consecutive. Bumping a pointer beyond the end of an object does not cause the pointer to point to the next object declared. This is especially true for function arguments.

Problems with implicit and explicit casts between pointers of different types can possibly be avoided through use of the $BYTEADDRESS compiler control option.

Common Type Portation Problems

The following common type portation problems are discussed:

- Signed and unsigned type comparisons
- Scalar type size assumptions
- Structure alignment
- Pointer alignment

Signed and Unsigned Type Comparisons

The comparison of negative numbers with unsigned types may behave differently on A Series systems than on other platforms. Platforms that store arithmetic values in one's or two's complement form store the sign when assigning to an unsigned type. On A Series systems, the sign is not stored when assigning to an unsigned type. On A Series systems, comparing a negative value with the value in an unsigned type will never be true unless the CHAR2 or UNSIGNED subordinate options of the PORT compiler control option are used to emulate two's complement arithmetic on unsigned types. Another workaround is to change negative literals to their one's complement values (for example, -1 becomes 0xFF).

Scalar Type Size Assumptions

The representation of scalar types differs between A Series systems and other platforms. On A Series systems, integer types are 6-bytes in length and start on word boundaries. Any calculations that assume the size of an integer type to be anything other than 6 bytes will result in problems when portation occurs.

Examples

The following examples illustrate these type of size assumptions:

Using the `sizeof` operator in combination with numeric literals to calculate the number of bytes in an item of scalar type can cause problems. In the following program fragment, the intent is to calculate the number of long elements occupied by `SomeObject`:

```
long Ary[ (sizeof(SomeObject)+3) / sizeof(long)];
```

The addition of the numeric literal 3 to the size of the type of `SomeObject` assumes that `sizeof(long)` returns 4. In A Series C, `sizeof(long)` returns 6, causing the calculation to return unexpected results. The following code fragment illustrates how you can avoid this type of portation problem:

```
long Ary[ (sizeof(SomeObject)+sizeof(long)-1) / sizeof(long)];
```

Problems can also occur when pointers are incremented to access arrays or members of structures if assumptions are made about the alignment and packing of the individual elements. The following program fragment uses a pointer to access a member of a structure:

```
struct{ unsigned short Type;
        unsigned char SeqNum;
      } Info;
char *InfoPtr = &Info;
seq = InfoPtr[2];
```

Using `InfoPtr[2]` to access `SeqNum` assumes that bytes 0 and 1 contain the structure member `Type` and that byte 2 contains the structure member `SeqNum`. On A Series systems, `Type` occupies bytes 0 to 5, and `SeqNum` occupies byte 6, which means that `InfoPtr[2]` accesses the wrong data.

Pointer Alignment

When a pointer is implicitly or explicitly cast from a pointer to an object of a given alignment to a pointer to an object of a more strict alignment, the resulting pointer may not be aligned with the original pointer. For example, casting a pointer to `char` to a pointer to `int` causes the pointer index to be divided by 6 to change from character to word units. The divide by 6 operation causes the pointer to `int` to point to the beginning of the word. If the `char` pointer points to the middle of the word, the two pointers do **not** point to the same data. To catch pointer alignment errors of this type at run time, use the $BOUNDS(ALIGNMENT) compiler control option.

Two's Complement Arithmetic

Two's complement arithmetic is used on many platforms. On A Series systems, arithmetic is performed on data in signed-magnitude form. This can cause discrepancies in algorithms that depend on the two's complement representation. For example, C applications that use encryption algorithms to match data, such as passwords, between client and server must perform the encryption the same way on the client-end and the server-end. The differences between the two's complement and signed-magnitude representation may result in different values when fragments of data are extracted, encrypted, and reinserted into the data.

To obtain matching results, you can define macros that return arithmetic results in two's complement form. The following example illustrates macros for two's complement addition and subtraction:

```
#define tc_add(arg1, arg2)  (((arg1) + (arg2)) & 0xFF)
#define tc_sub(arg1, arg2)  (((arg1) + (0x100 - (arg2))) & 0xFF)
```

Date/Time Representation

The internal representation of the date and time differs on different platforms. On UNIX systems, the internal representation of date/time is the number of seconds elapsed since the epoch date of Thursday, January 1, 1970 at 00:00:00. The MS-DOS representation is a 16-bit value for both date and time with an epoch date of Tuesday, January 1, 1980 at 00:00:00. On A Series systems, the actual date and time is returned in a structure. When performing timestamp matching between client and server, algorithms are required to convert date/time to a common format.

Type Matching

The A Series C compiler and Binder require that the number of arguments be correct. The prototype syntax "`,...`" must be used for varying numbers of arguments. Within a source file, the types of the arguments must also agree. Across source files, the types can disagree, except that `long double` types must match.

Object Sizes

By default, the TINY memory model is used. The default can be changed by the $ SET MEMORY_MODEL = SMALL, = LARGE, or = HUGE options. The smallest model within which the data fits should be chosen. Refer to Section 10, “Compiler Control Options,” for more details on memory models.

Library Procedures

The A Series C compiler provides only the library functions specified in the ANSI C Standard. Other libraries, like “curses” and “shared memory”, are not currently provided.

Headers

Whenever a standard library function is used, its header should be included. The compiler produces faster code and does not require a bind step for the function.

When including a nonstandard header file, the $ SET SEARCH= option can be used to control where to search for the file. The search path can be made relative to the directory of the source file. A search for included files inside an include file is always based on the original source file.

Multiple Clients in a C Library

The macro `errno` is defined as an array indexed by the unique stack number of the execution process when $SET CONCURRENTEXECUTION or $SHARING=SHAREDBYALL. As a result, each client of a C library has a unique `errno` location.

Although each `errno` location starts at zero, the location is not reset to zero when a client delinks from a library or when another client links with the same stack number. In order to avoid this issue, the C library should explicitly set `errno` to zero before calling a function that sets `errno`.

Note: *The macro* `errno` *cannot be used as a declarator when $CONCURRENTEXECUTION or $SHARING=SHAREDBYALL and any of the standard heads have been included.*

File System Differences

The following file system differences are discussed:

- Text streams
- Binary streams
- LTITLE attribute

Text Streams

If the $ SET ANSI option is used, the horizontal tab feature is preserved; otherwise, it is replaced by spaces up to the next column that is a multiple of 8. All other control characters are stored without translation.

By default, files created with text streams can hold about 1 million lines. The default can be changed by setting the AREASIZE attribute on the `fopen` function. The AREASIZE value (default is 1,008) multiplied by 1,000 is the number of lines allowed. The maximum value for AREASIZE is 1,048,575, allowing over 1,000 million lines.

Files created using text streams are compatible with most other A Series tools such as CANDE and the Editor. For example, a C program can be written to disk using a text stream and then turned into a CCSYMBOL file simply by using the CANDE TYPE command. (Setting the FILEKIND attribute to CCSYMBOL on the `fopen` function has the same effect.)

Binary Streams

By default, binary streams are stored as 90 byte records, with automatic wraparound as necessary. The default file size is 90 million bytes. Note that this easily fits within 2**39-1, the maximum integer, allowing `ftell` and `fseek` to be used.

The maximum size of a file is over 400,000,000 million bytes and is achieved by setting AREASIZE to its maximum of 1,048,575 and MAXRECSIZE and BLOCKSIZE to their maximum of 65,535. This is larger than the maximum integer and the `fgetpos` and `fsetpos` functions must be used instead of `ftell` and `fseek`.

LTITLE Attribute

The C compiler uses the LTITLE attribute instead of the TITLE attribute. The LTITLE attribute allows larger limits on the length of a node and on the number of nodes in the file title if the system option SYSOPS LONGFILENAMES is set.

Refer to Section 10, “TITLESYNTAX Option,” for more information about the LTITLE attribute.

Data Communications Differences

The following data communications differences are discussed:

- Input
- Output

Input

The A Series system does not support unbuffered data communications where each keystroke is sent immediately to the system. Full duplex and half duplex are supported, but the characters are collected in the data communications processor (DCP) and are sent to the system only when the Return key is pressed. Input is limited to MAXINPUT characters, a value defined for each station. A line of input can be continued by terminating the input with a backslash (\). Another line is read and appended to the previous line.

The DCP translates a horizontal tab into enough blanks to position the cursor at the next tab stop. Linefeed characters are ignored and the backspace character causes the previous character to be deleted. The delete character causes the input to be discarded. Other characters are not changed.

Output

Output is automatically sent when the output limit is reached or a new line is displayed. Unbuffered output is not supported, but a partial line can be displayed by flushing the output file: The next character written appears immediately after the last character displayed, with no new line. (On the TD/MT/ET style of terminals, unexpected behavior may result.)

Any line longer than the declared terminal width, usually 80, is folded at the width. The backslash (\) character does not appear. A horizontal tab is translated by the data communications hardware into enough blanks to position the cursor at the next tab stop. Other characters are sent without editing.

Extensions

The A Series C compiler supports many extensions that can make porting easier.

Preprocessor

The source program with all macros expanded and nonsystem `include` files inserted, can be saved by using the $ SET XSOURCE=SOURCE option. The resulting file is an equivalent C program that can be compiled instead of the original source. It is useful in determining exactly how macros are expanded.

Cross Reference Files

Cross-references for all identifiers in the source file can be printed by using the $ SET XREF option. The $ SET XREFFILES option can be used to store cross-reference information in a file for later use by the Editor. (The Editor can show all references to a

variable, function, define, and inside defines. It can also jump to the next or previous reference, jump to the beginning or end of the function, and so forth. Refer to the documentation on the Editor.)

Memory Layout

The layout of structures, arrays, and variables can be displayed by using the $ SET MAP option.

Statistics

The number of times each function is called, as well as the time spent in the function, can be printed after each run by compiling with the $ SET STATISTICS option. Time is calculated from each function entry to function exit, inclusive and exclusive of time spent in any other function called.

The frequency count of each line executed can be shown by the $ SET STATISTICS(BLOCK) option. The reported time values are in seconds of processor time.

Run Time Libraries

It is possible at run-time to dynamically link to an A Series Library written in ALGOL. Library linkage can be used instead of compile-time binding. A library can also provide a shared facility, such as a message passing system. Refer to Appendix A, "Interface to the Library Facility," for additional information.

Enhancing Performance

The standard library functions should be used whenever possible. They have been hand-coded to take full advantage of the A Series instruction set. The header files should be included using the `#include <...>` syntax whenever possible; it allows the code to be inserted inline, possibly avoiding the Binder step.

The most efficient compiler control values are given in the following list. They should be used whenever possible. Each default is given in parentheses afterwards.

```
$$ RESET ANSI               (RESET)
$$ PORT (RESET UNSIGNED)    (RESET)
$$ PORT (RESET SIGNEDFIELD) (RESET)
$$ PORT (RESET SIGNEDCHAR)  (RESET)
$$ SET STRINGS=EBCDIC       (EBCDIC)
$$ SET DBLTOSNGL            (SET)
$$ SET MEMORY_MODEL=TINY    (TINY)
```

Variables outside of functions should be declared `static` whenever possible.

Array indexing should be done instead of pointer dereference whenever possible (as is true for any optimizing compiler). Very small functions that are called frequently from the same source file should be defined with the `inline` storage class specifier.

MAKE Utility

In most computing environments that support the C language, it is typical for a C application to be developed as numerous small interdependent modules. As the size and complexity of the project increases, so does the difficulty in keeping the various files up-to-date. The MAKE utility improves the maintenance of such a project by providing a method for creating and updating the necessary files with a minimum of effort.

The A Series MAKE utility provides a subset of the MAKE features familiar to most C programmers. The MAKE utility does the following:

- Determines if a source file has changed since its last compilation and, if so, recompiles it.
- Determines if a program must be bound to produce a current version and, if so, invokes the Binder.
- Enables the user to invoke Work Flow Language (WFL) commands conditionally.
- Provides macro substitution.

MAKE Command Line

The MAKE command line takes zero or more flags, followed by zero or more macro definitions, followed by zero or more targets. The flags may occur in any order, in either uppercase or lowercase letters. The syntax for MAKE is as follows:

```
make [flags] [macro definitions] [targets]
```

The permissible MAKE command line flags are

Flag	Argument	Description
-d	none	Specifies the debug mode; detailed information regarding the MAKE process is printed.
-f	name	Specifies the name of the makefile. A file name of '-' denotes stdin. The default name is MAKEFILE.
-i	none	Specifies that errors that are fatal to the MAKE should be ignored, for example, a missing source file.
-n	none	Specifies no execute mode; commands are printed, but not executed.
-s	none	Specifies silent mode; commands are not printed before they are executed.

The macro definitions follow the flags. A macro definition takes the same form as that in the makefile:

```
name=value
```

The definition of a macro on the command line overrides the definition of the macro in the makefile. Note that on the command line, a macro definition that has embedded blanks must be enclosed within quotation marks.

The targets to be made follow the macro definitions and are processed in left-to-right order. If no targets are supplied, then the first target that appears in the makefile is made.

The following is an example of a makefile and how to invoke MAKE:

```
OBJECTS=prog.o init.o
CFLAGS= -s 'reset list' -d abc=3

appl:   $(OBJECTS)
        ld -o appl.o $(OBJECTS)
prog.o: prog.c
        cc $(CFLAGS) prog.c
save:
        WFL COPY OBJECT/APPL/C TO MYPACK (PACK);
```

From CANDE, issue the following:

```
r make ("-f appl/make -s 'CFLAGS=-d abc=0' appl save")
```

or

```
u make -f appl/make -s 'CFLAGS=-d abc=0' appl save
```

This invokes the MAKE command, which processes the makefile `APPL/MAKE` with the silent mode enabled. The definition for `CFLAGS` in the command line overrides the definition of `CFLAGS` in the makefile. A MAKE command of `appl` is processed first. Assuming that all target files are out-of-date, the file `init.c` is compiled first by virtue of the default rule. The file `prog.c` is compiled next, using the CC command that follows the `prog.o` target line. The LD tool is then invoked to bind the objects. The MAKE command of `save` is processed next, which causes the WFL COPY command to be invoked.

From WFL, issue the following:

```
PROCESS RUN OBJECT/MAKE
   ("-f appl/make -s 'CFLAGS=-d abc=0' appl save")
```

Makefile

The makefile is a text file containing target lines, command sequences, and macro definitions.

Makefile Rules

The following rules apply:

- Comments are delimited by a number sign character (#).
- Any text following the number sign is ignored.
- Blank lines are ignored.
- A logical line may consist of several physical lines by terminating all but the last physical line with a backslash (\).

File Name Conventions

To ease the porting of C applications to the A Series system, file names that appear in a makefile may be either in the standard A Series file name format or in an alternative format familiar to other C environments. The alternative format takes the form:

name.suffix

Typically, a suffix of c indicates a C source file and a suffix of o indicates an object file. Such file names are converted by MAKE to an equivalent A Series file name, using the following rules:

- Periods (.) and backslashes (\) are translated to slashes (/)
- Lowercase letters are translated to uppercase letters
- A file name of the form *x*.o becomes OBJECT/*X*/C.

The following examples illustrate the transformation rules:

MAKE File Name	A Series File Name
prog.c	PROG/C
prog.o	OBJECT/PROG/C
mcp\37\src.x	MCP/37/SRC/X

For clarity and compatibility with other C environments, the remaining discussion of the MAKE utility in this section contains file names in the alternative format.

File names that are prefixed with a dollar sign ($) are not translated since a single dollar sign in MAKE signifies a macro.

Note: *When using the MAKE utility, prefix your file name with two dollar signs ($$) to indicate an actual dollar sign.*

Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

Makefile Directories

Unlike other C environments, A Series systems do not support the concept of a "current" directory. Such a concept becomes extremely useful when dealing with projects that are comprised of numerous files. To work around this, a makefile may contain a DIR directive.

Using the DIR directive

The DIR directive takes the following form:

.DIR: *directory-name*

.	The period should appear in column 1.
directory-name	The *directory-name* must terminate with an equal sign (=). After MAKE processes this directive, all subsequent MAKE file names are prefixed with *directory-name*.

For example, when given the following directive:

```
.DIR: MCP/37/=
```

The following transformations occur:

MAKE File Name	A Series File Name
prog.c	MCP/37/PROG/C
prog.o	OBJECT/MCP/37/PROG/C

Not using the DIR directive

If a DIR directive does not appear, MAKE uses the directory of the makefile as the *directory-name*. For example, if the name of the makefile is MY/TEST/MAKE, then:

MAKE File Name	A Series File Name
prog.c	MY/TEXT/PROG/C
prog.o	OBJECT/MY/TEST/PROG/C

When the CC tool is invoked, a SEARCH compiler control record is automatically generated with the *directory-name*. The *directory-name* will be searched when the #include directive is processed. Refer to Section 10, "Compiler Control Options," for details on the SEARCH option.

Target Line

File interdependencies are indicated on target lines. A target line contains a list of target files optionally followed by a list of source files on which the targets are dependent. The initial target name must begin in column 1. The syntax for a target line is as follows:

```
target [target...] : [source...]
```

The brackets indicate optional items. If a MAKE has been specified for a target that is older than any of the target's sources, then a "match" for that target line occurs.

An example target line is:

```
prog.o: prog.c stuff.h
```

This line indicates that prog.o is dependent on both prog.c and stuff.h. A change to either prog.c or stuff.h causes prog.o to become out-of-date; a MAKE of prog.o therefore results in a match for this target line.

The process whereby MAKE matches targets in the makefile is recursive: Each source file can also be a target that is dependent on other sources. When a MAKE is requested for target *x*, MAKE, in effect, performs a depth-first search of the dependency tree rooted at *x*, matching all targets that are older than their sources.

For example, given the following target lines and a request to MAKE the target `appl`:

```
appl:     prog.o init.o
prog.o:   prog.c stuff.h
init.o:   init.c stuff.h
```

If `appl` is older than `prog.o`, then line 1 matches. If in turn `prog.o` is older than either `prog.c` or `stuff.h`, then line 2 matches. Similarly, if `appl` is older than `init.o`, then line 1 matches and if `init.o` is older than either `init.c` or `stuff.h`, then line 3 matches.

Once a match occurs, the command sequence following the target line is executed. Commands are executed in the reverse order in which the matches occur. Using the prior example, the command sequence for `init.o` is executed first, followed by the command sequence for `prog.o`, and finally the command sequence for `appl`.

Command Sequence

A command sequence consists of one or more command lines following a target line. A command line must begin with one or more blanks to distinguish it from a target line and contain a command to be invoked when the corresponding target line matches.

Acceptable commands

The MAKE utility accepts the following commands:

- A WFL command, which is passed to the WFL compiler for execution. A WFL command is preceded by the keyword `WFL`, for example:

  ```
  WFL COPY OBJECT/APPL/C TO MYTAPE
  ```

  A file name in a WFL command must appear in the A Series format, with its complete directory.

- A CC command, which is processed by the CC tool. The CC tool invokes the C compiler in a manner similar to that used by other C environments.

- An LD command, which is processed by the LD tool. The LD tool invokes the Binder to bind object files into an executable code file in a manner similar to that used by other C environments.

The CC and LD tools are described in detail later in this appendix.

Command line options

A command may be prefixed by as many as three command line options. Possible options are

- The hyphen (-). This option signifies that any error found while executing the command is ignored.

- The "at" sign (@). This option signifies that the command is not written before it is executed.

- The plus sign (+). This option signifies that the command is executed even if the -n option is specified.

Example

The following is a sample makefile with appropriate command sequences:

```
appl:    prog.o init.o
         ld -l+ -o appl prog.o init.o  # binds prog.o & init.o into appl
prog.o:  prog.c stuff.h
         cc -v prog.c                  # compiles prog.c into prog.o
init.o:  init.c stuff.h
         cc -v init.c                  # compiles init.c into init.o
```

Macro Substitution

The MAKE utility supports a simple macro substitution mechanism for both target lines and command lines. A macro name is any string of characters. A macro can be defined either in the makefile or on the command line when MAKE is invoked. In the makefile, a macro definition takes the form:

name=value

The macro name is in column 1. A macro that is not defined explicitly has the null string as its value.

The following are legal macro definitions:

```
CFLAGS= -v
3=abc.c
X=prog.o init.o
```

Macros are invoked in the makefile by preceding the macro name with a dollar sign ($). To specify a leading dollar sign, use two dollar signs ($$). The name must be parenthesized if it is longer than one character.

The following are legal macro invocations:

```
$(CFLAGS)
$3
$(X)
$$3
```

Example

The following is a sample makefile with some macro definitions:

```
CFLAGS= -v
OBJECTS=prog.o init.o
P=appl

$P:      $(OBJECTS)
         ld -o $P $(OBJECTS)    # binds OBJECTS to produce P
prog.o:  prog.c stuff.h
         cc $(CFLAGS) prog.c    # compiles prog.c into prog.o
init.o:  init.c stuff.h
         cc $(CFLAGS) init.c    # compiles init.c into init.o
```

Predefined Macros

The MAKE utility includes four predefined macros. They are:

Macro Name	Default	Description
CC	On	Compilation tool. Note that this macro is invoked as part of the MAKE utility default rule.
CFLAGS	Null	Compilation flags. Note that this macro is invoked as part of the MAKE utility default rule.
MAKE	Current object	MAKE execution object. This macro may be used to invoke MAKE from a MAKE file. For example: `$(MAKE) $(MAKEFLAGS) -F MAKEFILE MAKE2` invokes MAKE with the current flags to process target MAKE2 in MAKE file MAKEFILE.
MAKEFLAGS	Current command line	MAKE command line flags and macro definitions. This macro may also be used to invoke MAKE from a MAKE file using the example shown for the MAKE macro.

CC Tool

The CC tool can be used in a makefile to invoke the A Series C compiler. CC enables you to specify compiler control options, C preprocessor directives, and the names of source files to be compiled. The CC tool provides a convenient alternative to the WFL *COMPILE* command.

The CC command line takes zero or more flags, followed by the names of one or more source files to be compiled. The flags may occur in any order in either uppercase or lowercase characters. The syntax for CC is as follows:

```
cc [flags] file [file...]
```

The permissible flags are

Flag	Argument	Description
-c	*name*	Specifies name as the name of the compiler to invoke. The default compiler is SYSTEM/CC.
-d	*id[=text]*	Allows specification of a #define *id* for the C preprocessor, giving it the value *text* if specified and 1 otherwise. There may not be blanks around the '='.
-o	*name*	Specifies *name* as the title of the object file for the first source file on the command line.
-p	*name*	Specifies *name* as a directory. File names appearing on the cc command line will be prefixed with *name*.
-s	*option*	Allows specification of a compiler option. If the option contains embedded blanks, it must be quoted with single quotes (').
-u	*id*	Allows specification of an #undef *id* for the C preprocessor.
-v	none	Causes names of the files that are passed to the C compiler to be displayed.
-x	none	Indicates a syntax-only compilation.

The following example illustrates how to use the CC tool:

```
cc -s optimize -d abc=5 -u xyz -o mylib.o lib.c main.c
```

This command results in the following:

- The compiler option OPTIMIZE is enabled
- A preprocessing directive of the form #define abc 5 is passed to the C preprocessor
- A preprocessing directive of the form: #undef xyz is passed to the C preprocessor
- The file LIB/C is compiled, producing the object file OBJECT/MYLIB/C
- The file MAIN/C is compiled, producing the object file OBJECT/MAIN/C

Note that the CC tool does not generate an object file named a.out if only one source file is specified. This is contrary to the behavior of CC in other C programming environments.

Refer to Section 10, "TITLESYNTAX Option," for information on the file title changes that occur during compilation.

LD Tool

The LD tool can be used in a makefile to invoke the A Series Binder. LD enables you to specify Binder options and the names of the object files to be bound all on the same command line, providing a convenient alternative to the WFL *BIND* command.

The LD command line takes zero or more flags, followed by the names of one or more object files to be bound. The flags may occur in any order in either uppercase or lowercase characters. The syntax for LD is as follows:

```
ld [flags] file [file...]
```

The permissible flags are

Flag	Argument	Description
`-c`	name	Specifies *name* as the name of the Binder to invoke. The default Binder is `SYSTEM/BINDER`.
`-l`	any letter; must be adjacent to the `l`	Indicates that the C standard library functions are to be bound from the file `SYSTEM/CC/LIBRARY`. The argument has no effect, but conforms to the convention used in other C environments.
`-o`	name	Specifies *name* as the title of the bound code file.
`-p`	name	Specifies *name* as a directory. File names appearing on the `ld` command line will be prefixed with *name*.
`-s`	option	Allows specification of a Binder option. If the option contains embedded blanks, it must be quoted with single quotes (`'`).
`-v`	none	Causes the names of the files that are passed to the Binder and the Binder input file to be displayed.
`-x`	none	Indicates a syntax-only bind.

If the `-o` flag does not appear, the bound code file has the title `OBJECT/A/OUT`.

The following example illustrates how to use the LD tool:

```
ld -S MAP -o appl.o -lx prog.o init.o
```

This command results in the following:

- The Binder option MAP is enabled
- The standard library functions are bound from SYSTEM/CC/LIBRARY
- The object files OBJECT/PROG/C and OBJECT/INIT/C is bound, producing the code file OBJECT/APPL/C

Refer to Section 10, "TITLESYNTAX Option," for information on the file title changes that occur during compilation.

Default Rule

The MAKE utility invokes a default rule when a command sequence is not explicitly provided for a file. The default rule is that a target of the form *x*.o has a source of the form *x*.c and is created with the following command:

```
$(CC) $(CFLAGS) x.c
```

Thus, a makefile can consist only of the following without any target lines for prog.o or init.o.:

```
appl: prog.o init.o
      ld -o appl.o prog.o init.o
```

If the target line matches during a MAKE, then MAKE examines prog.o and init.o to determine if either is older than its respective source file and recompile them if necessary.

Mnemonic Targets

A mnemonic target is a target that is not followed by any sources. It is not the name of an actual file, but is used instead as a convenient way of invoking a specific command sequence. If a target line matches, but does not contain any sources, then the command sequence following the target line is executed. For example, given the following makefile:

```
appl: prog.o init.o
      ld -o appl.o prog.o init.o
save:
      WFL COPY OBJECT/APPL/C TO MYTAPE;
```

Then, issuing a MAKE command for the target save copies the file OBJECT/APPL/C to a tape.

Appendix D
Implementation-Defined Items

This appendix contains items that the American National Standard Institute (ANSI) C allows each implementation of C to define. Each ANSI C item is followed by the A Seriesk C implementation description.

Translation

ANSI C Standard

How a diagnostic is identified

A Series C Implementation

Diagnostics will begin with either "ERROR:," "FATAL ERROR:," or "WARNING:." When possible, a numeral will precede the diagnostic, indicating line position in the source file as in the following example diagnostic:

```
  00005400          int_ptr = &int_array[0];
                                      1

 1) ERROR: Name not declared = "int_array".
```

Environment

ANSI C Standard

The semantics of the arguments to `main`

A Series C Implementation

Refer to "Program Parameters" in Section 9 of this manual for the semantics of the arguments to `main`.

ANSI C Standard

What constitutes an interactive device

A Series C Implementation

A Series C supports fully buffered and line buffered streams. Unbuffered streams are not supported by A Series C. The semantics of output to unbuffered streams can be simulated with the `fflush` function.

Identifiers

ANSI C Standard

The number of significant initial characters (beyond 31) in an identifier without external linkage

A Series C Implementation

The first 250 characters of an internal identifier are significant.

ANSI C Standard

The number of significant initial characters (beyond six) in an identifier with external linkage

A Series C Implementation

The first 250 characters of an external identifier are significant to the Binder.

ANSI C Standard

Whether case distinctions are significant in an identifier with external linkage

A Series C Implementation

Case distinctions are significant in external identifiers, but since most other languages either translate lowercase to uppercase or else do not allow lowercase characters, external identifier should not use lowercase.

Characters

ANSI C Standard

The members of the source and execution sets, except as explicitly specified in the standard

A Series C Implementation

For a detailed description of the source set, refer to "Character Sets" in Section 1 of this manual. The execution set is whatever can be represented in eight bits.

ANSI C Standard

The shift states used for the encoding of multibyte characters

A Series C Implementation

Multibyte characters are handled through the CENTRALSUPPORT library. For a complete description of the use of internationalization features, refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide*.

ANSI C Standard

The number of bits in a character in the execution character set

A Series C Implementation

A character in the execution character set contains eight bits

ANSI C Standard

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set

A Series C Implementation

EBCDIC is used as both the source and execution character set, where applicable. Regardless of the character set used, the mapping is a one-to-one translation of the source code to the execution set.

ANSI C Standard

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant

A Series C Implementation

The value is equivalent to the encoding in the source character set.

ANSI C Standard

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character

A Series C Implementation

The A Series C compiler allows multicharacter constants like `ABC`. The purpose of this is to allow programmers to create an integer value (not a string) from the characters. Character constants are type `int`. The value of the character constant is the numerical value of that character in the EBCDIC character set. The compiler allows character constants up to four characters long. Because characters have different sizes on different computers, the use of this feature limits the portability of a program.

ANSI C Standard

The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant

A Series C Implementation

For more information on Internationalization, refer to Appendix E, "Internationalization," in this manual, and to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide.*

ANSI C Standard

Whether a "plain" `char` has the same range of values as signed `char` or unsigned `char`

A Series C Implementation

An object declared as a character (`char`) is large enough to store any member of the source character set. These values range from 0 to 255. The compiler control option PORT(SIGNEDCHAR) may be used to change the type of `char` from unsigned values to signed values.

Integers

ANSI C Standard

The representations and sets of values of the various types of integers

A Series C Implementation

For a complete description of the various types of integers, refer to "Integer Types" in Section 2. Refer to Tables 2-3 and 2-4 for information on the size and range of each of the integer types.

ANSI C Standard

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, if the value cannot be represented

A Series C Implementation

For more information on type conversions, refer to "Integral to Integral" in Section 4 of this manual.

ANSI C Standard

The results of bitwise operations on signed integers

A Series C Implementation

For bitwise shifts, only the 39 bits for the magnitude are affected; the sign bit is unchanged. For bitwise ANDs and ORs, the sign bit and the 39 bits for the magnitude are affected.

ANSI C Standard

The sign of the remainder on integer division

A Series C Implementation

When the integer operands to the binary operator % are both positive, the result is positive. If one of the operands is negative, the sign of the result of the % operator is the same as the dividend.

ANSI C Standard

The result of a right shift of a negative-valued signed integral type

A Series C Implementation

Since a bitwise shift does not affect the sign bit on A Series C, the negative sign will be retained.

Floating Point

ANSI C Standard

The representations and sets of values of the various types of floating-point numbers

A Series C Implementation

For more information on floating-point numbers, see "Floating-Point Types" in Section 2 of this manual.

ANSI C Standard

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

A Series C Implementation

Since the A Series C floating-point type can exactly represent all integral values, the result is the equivalent floating-point value of the integral value. No truncation is performed.

ANSI C Standard

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

A Series C Implementation

A `long double` value is converted to a `float` or `double` value by truncating the 78-bit double mantissa to a 39-bit single mantissa and truncating the 15-bit exponent to a 6-bit exponent. The result value is undefined if the magnitude of the original value is too large for the result type.

Arrays and Pointers

ANSI C Standard

The type of integer required to hold the maximum size of an array--that is, the type of the `sizeof` operator, `size_t`

A Series C Implementation

The type of `size_t` is unsigned `int`.

ANSI C Standard

The result of casting a pointer to an integer or vice versa

A Series C Implementation

Both of these castings are allowed, but the results of such castings are not portable.

ANSI C Standard

The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t`

A Series C Implementation

The type of `ptrdiff_t` is `int`.

Registers

ANSI C Standard

The extent to which objects can actually be placed in registers by use of the `register` storage class specifier

A Series C Implementation

Use of the `register` storage class specifier will never result in slower code. It does not result in any faster code, except that the address is never taken. Any variable whose address is not taken is optimized in the same manner.

Structures, Unions, Enumerations, and Bit-Fields

ANSI C Standard

A member of a union object is accessed using a member of a different type

A Series C Implementation

The bit pattern currently contained within the union is interpreted as a value of the type of the member used in the access.

ANSI C Standard

The padding and alignment of members of structures

A Series C Implementation

The A Series C compiler assigns members in increasing memory addresses in a strict left to right order. The first member starts at the beginning address of the structure. Holes can appear between two consecutive members of a structure to allow for the proper alignment of the members in storage.

ANSI C Standard

Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field

A Series C Implementation

A "plain" `int` bit-field is unsigned. Use of the compiler control option PORT(SIGNEDFIELD) may be used to make "plain" integers act as signed integers.

ANSI C Standard

The order of allocation of bit-fields within a unit

A Series C Implementation

Bit-fields are packed in adjacent bits within a machine word from left to right.

ANSI C Standard

Whether a bit-field can straddle a storage-unit boundary

A Series C Implementation

A field that does not fit into the space remaining in a word is put into the next word. No field can be wider than the normal width of the declared type, or 39 bits, whichever is smaller.

ANSI C Standard

The integer type chosen to represent the values of an enumeration type

A Series C Implementation

The identifiers in an enumerator list are declared as constants that have type `int`. They can appear wherever type `int` is permitted.

Qualifiers

ANSI C Standard

What constitutes an access to an object that has volatile-qualified type

A Series C Implementation

Any read or write operation

Declarators

ANSI C Standard

The maximum number of declarators that may modify an arithmetic, structure, or union type

A Series C Implementation

The maximum number of declarators that can modify a basic type depends on the size of the compiler's internal data structures, typically over 100.

Statements

ANSI C Standard

The maximum number of case values in a `switch` statement

A Series C Implementation

The maximum number of case values allowed in a `switch` statement exceeds 2000.

Preprocessing Directives

ANSI C Standard

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value

A Series C Implementation

Character constants in preprocessor statements have the same value as identical constants in other statements. Such a constant may not have a negative value.

ANSI C Standard

The method for locating includable source files

A Series C Implementation

For a complete description of the file inclusion search method, refer to "#include Directive" in Section 8 of this manual.

ANSI C Standard

The support of quoted names for includable source files

A Series C Implementation

Quoted names are supported for includable source files.

ANSI C Standard

The mapping of source file character sequences

A Series C Implementation

Before a file is searched for, the file name is edited to help it conform to A Series title format. Any characters inside double quotes (") are left as is. Otherwise, any period (.) or backlash (\) is translated to a slash (/), except when that would produce an illegal file title, in which case the period or backslash is dropped. The SEARCH compiler control option may be used to further edit the file title and to search in multiple places. Refer to Section 10, "TITLESYNTAX Option," for more information on the file title changes that occur during compilation.

ANSI C Standard

The behavior on each recognized `#pragma` directive

A Series C Implementation

The `#pragma` directive verifies that object files being bound together are compiled with the same set of compile-time options. The compiler supports the following `#pragma` directive and ignores all others:

```
#pragma binder_match = (<string1>, <string2>)
```

ANSI C Standard

The definitions for__DATE__and__TIME__ when respectively, the date and time of the translation are not available

A Series C Implementation

The date and time of translation are always available.

Library Functions

ANSI C Standard

The null pointer constant to which the macro NULL expands

A Series C Implementation

A null pointer constant is an integral constant expression with the value of 0.

ANSI C Standard

The diagnostic printed by and the termination behavior of the `assert` function

A Series C Implementation

When the expression passed to the `assert` function is false, the following message is printed and the abort function is called:

"Assert failure in File x at Line y with Condition expression"

The x and y are the values of the preprocessing macros `__FILE__` and `__LINE__`.

ANSI C Standard

The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions

A Series C Implementation

The sets of characters tested for by the `isalnum`, `isalpha`, `islower`, and `isupper` functions are defined in Volume 2, "Headers and Functions." The following is the set of characters tested for by the `iscntrl` function:

NUL, SOH, STX, ETX, HT, DEL, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, NL (EBCDIC only), BS, CAN, EM, FS, GS, RS, US, LF, ETB, ESC, ENQ, ACK, BEL, SYN, EOT, DC4, NAK, and SUB

The following is the set of characters tested for by the `isprint` function:

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

ANSI C Standard

The values returned by the mathematics functions on domain error

A Series C Implementation

The mathematics functions return a value of zero when a domain error occurs.

ANSI C Standard

Whether mathematics functions set the integer expression `errno` to the value of the macro ERANGE on underflow range errors

A Series C Implementation

`errno` is set to ERANGE on underflow range errors.

ANSI C Standard

Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero

A Series C Implementation

If the second argument to `fmod` is zero, a domain error occurs and a value of zero is returned.

ANSI C Standard

The set of signals for the `signal` function

A Series C Implementation

The set of signals is defined in Volume 2, "Headers and Functions."

ANSI C Standard

The semantics for each signal recognized by the `signal` function

A Series C Implementation

The semantics for each `signal` is described in Volume 2, "Headers and Functions."

ANSI C Standard

The default handling and the handling at program startup for each `signal` recognized by the `signal` function

A Series C Implementation

The default handling and handling at program startup is described in Volume 2, "Headers and Functions."

ANSI C Standard

If the equivalent of `signal (sig, SIG_DFL)` is not executed prior to the call of a signal handler, the blocking of the signal that is performed

A Series C Implementation

The equivalent of `signal (sig, SIG_DFL)` is executed prior to the call of a signal handler.

ANSI C Standard

Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function

A Series C Implementation

The default handling is reset in this case.

ANSI C Standard

Whether the last line of a text stream requires a terminating new-line character

A Series C Implementation

The final line of a text stream does not require a terminating new-line character.

ANSI C Standard

Whether space characters that are written out to a text stream immediately before a new-line character appear when read in

A Series C Implementation

Spaces wrriten before a newline character are **not** retained.

ANSI C Standard

The number of null characters that may be appended to data written to a binary stream

A Series C Implementation

Null characters are appended to binary streams to fill out the current record.

ANSI C Standard

Whether the file position indicator of an append mode stream is initially positioned at the beginning or end-of-the file

A Series C Implementation

The file position indicator is initially at end of file.

ANSI C Standard

Whether a write on a text stream causes the associated file to be truncated beyond that point

A Series C Implementation

The associated file will not be truncated beyond that point.

ANSI C Standard

The characteristics of file buffering

A Series C Implementation

For a complete description of buffering in A Series C, refer to Volume 2, "Headers and Functions," for more information on the use of buffers in the header `<stdio.h>`.

ANSI C Standard

Whether a zero-length file actually exists

A Series C Implementation

A zero-length file exists. The file exists as soon as it is opened.

ANSI C Standard

The rules for composing valid file names

A Series C Implementation

A file name should be in the form of a standard A Series file name (optional usercode, name, and optional family name) and may contain both upper and lowercase characters. In addition, a file's host name can be specified by an optional AT <hostname> suffix in the file name.

ANSI C Standard

Whether the same file can be open multiple times

A Series C Implementation

A file may be opened multiple times.

ANSI C Standard

The effect of the `remove` function on an open file

A Series C Implementation

An open file may be removed.

ANSI C Standard

The effect if a file with the new name exists prior to a call to the `rename` function

A Series C Implementation

The `rename` succeeds. The existing file is replaced by the file specified by the `rename`.

ANSI C Standard

The output for %p conversion in the `fprintf` function

A Series C Implementation

The argument is taken to be a (`void *`) pointer to an object. The value of the pointer is printed as a signed decimal (d format).

ANSI C Standard

The input for %p conversion in the `fscanf` function

A Series C Implementation

A pointer is expected. The subsequent argument must be a pointer to a pointer to void. The format of the input field is a decimal integer. It is the same as that produced by the %p conversion of `fprintf`.

ANSI C Standard

The interpretation of a minus sign (-) character that is neither the first nor the last character in the scanlist for %[conversion in the `fscanf` function.

A Series C Implementation

A minus sign (-) character is treated like any other character, that is, it is part of the scan set.

ANSI C Standard

The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure

A Series C Implementation

On failure, `errno` can be set to the values of any of the following `#defines`:

`#define`	Meaning
`EBADF`	File is not open
`ENOFILEPOS`	File does not support positioning requests
`EIOERROR`	I/O error
`EDATAERR`	I/O data error
`EPARITYERROR`	I/O parity error

ANSI C Standard

The messages generated by the `perror` function

A Series C Implementation

For the possible values of `errno` and their related strings, refer to Volume 2, "Headers and Functions," for more information on the header `<errno.h>`.

ANSI C Standard

The behavior of the `calloc`, `malloc`, or `realloc` functions if the size requested is zero

A Series C Implementation

This is not allowed by the compiler. A null pointer is returned.

ANSI C Standard

The behavior of the `abort` function with regard to open and temporary files

A Series C Implementation

Open output streams are not flushed; temporary files are not removed.

ANSI C Standard

The status returned by the `exit` function if the value of the argument is other zero, EXIT_SUCCESS, or EXIT_FAILURE

A Series C Implementation

If the value of the argument is anything other than EXIT_SUCCESS, the program goes to abnormal termination (P-DSed) and the task attribute is set to the value of the argument.

ANSI C Standard

The set of environment names and the method for altering the environment list used by the `getenv` function

A Series C Implementation

A Series C does not support an environment list. Therefore, `getenv` always returns a null pointer.

ANSI C Standard

The contents and mode of execution of the string passed by the `system` function

A Series C Implementation

For a complete description of the system function, refer to "system Function" in Volume 2.

ANSI C Standard

The contents of the error message strings returned by the `strerror` function

A Series C Implementation

For the possible values of `errno` and their related strings, refer to Volume 2, "Headers and Functions," for more information on the header `<errno.h>`.

ANSI C Standard

The local time zone and Daylight Savings Time

A Series C Implementation

Time Zone is not determinable in A Series C. Daylight Savings Time can be checked through the following member of the `tm` structure `<time.h>`:

```
int tm_isdst; /* Daylight Savings Time flag */
```

The value of `tm_isdst` is positive if Daylight Savings Time is in effect, zero if it is not in effect, and negative if it cannot be determined.

ANSI C Standard

The era for the `clock` function

A Series C Implementation

The `clock` function returns the best approximation to the processor time used by the program since the invocation of the program. The value returned must be divided by the value of the macro CLOCKS_PER_SEC to obtain time in seconds. The value (`clock_t`)-1 is returned of the processor time used is not available.

Appendix E
Internationalization

Internationalization refers to the software, firmware, and hardware features that enable the development and execution of application systems that can be customized to meet the needs of a specific language or business environment. The internationalization features provide support for several character sets, different international business and cultural conventions, extensions to data communications protocols, and the ability to use one or more natural languages concurrently.

This section describes one of two sets of internationalization features used to customize an application for the language and conventions of a particular locale. Using these features to write or modify an application is termed *localization*. The MultiLingual System (MLS) environment enables the information processing necessary to localize applications. Some of the localization methods provided in the MLS environment include the following:

- Translating messages to another language
- Choosing a particular character set to be used for data processing
- Defining date, time, number, and currency formats for a particular business application

To supplement the information described in this section, refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for additional information pertaining to the MLS environment. The MLS Guide provides definitions for and detailed information about the ccsversions, character sets, conventions, and languages provided in the MLS environment. It also provides procedures for setting system values for the internationalization features.

Accessing the Internationalization Features

Two distinct approaches are available for localizing a program in this C language implementation. The first approach is to utilize the `<locale.h>` header file and related functions. This provides a set of localization functions defined by the ANSI C standard. The `setlocale` function, declared in `<locale.h>` allows the program to select, dynamically, one of a set of system defined "locales" to become the current active locale. Certain other system library functions are "locale-sensitive", which means that their behavior is determined by the current locale setting. For example, the `strcoll` function in `<string.h.>` compares two strings in a way appropriate to the collation sequence of the currently selected locale. Since the `<locale.h>` header file and the locale-sensitive functions are defined by the ANSI C standard, use of these functions should be portable to other ANSI compliant C compilers. If portability is an issue, this approach is probably the most desirable.

For more information on this approach, refer to Volume 2, "Headers and Functions," for information on the header `<locale.h>`. For a detailed description of the implementation of the C locale-sensitive functions in terms of the MLS environment, particularly the CENTRALSUPPORT library, refer to the MLS Guide which includes a description of the process of defining the set of specific locales available on a particular system.

The second approach for localizing an A Series C program is to make use of the CENTRALSUPPORT system library directly. This approach is the focus of the rest of this section. The CENTRALSUPPORT system library provides a set of procedures that may be called from a user program to perform localization functions. For example a program can access a procedure called `cnv_formattime` that formats the time according to the language and convention specified in the procedure call. While this approach does not provide the same level of portability as the first one discussed, it does have the advantage that the CENTRALSUPPORT interfaces are almost identical to those used by other languages on the enterprise server. Also, there may be some desired functionality provided by the CENTRALSUPPORT approach that is not available through the `<locale.h>` facilities.

A file is provided on the system, SYMBOL/INTL/C/PROPERTIES, that provides the necessary C language declarations to access the available functions in the CENTRALSUPPORT library. This file may be included into the program's C source code by means of an `#include` statement. Refer to Appendix A, "Interface to the Library Facility," in this manual for general information on calling ALGOL library procedures within a C program.

Note: A program is not affected by the internationalization features unless the features are specifically invoked by the program. Any programs that already exist and do not make use of internationalization features are not affected by them.

Understanding Default Settings and Hierarchy

Default settings have been preset on the system for some of the internationalization options. These can be changed either by the user or system administrator, depending upon the hierarchy or priority of the levels.

The default setting for localization can be established at the following levels:

Default Level	Explanation
Task	Established as task initiation.
Session	Handled by MARC or CANDE commands or programs that support sessioning.
Usercode	Established in the USERDATAFILE.
System	Established with a system or MARC command.

Each of these levels has a priority, or a hierarchical rank, associated with it. A setting at the task level overrides a setting at the session level. A setting at the session level overrides a usercode level. A setting at the usercode level overrides a setting at the system level and so on. Default language and convention can be established at any of the levels, but the default ccsversion is limited to the system level.

When coding a program, in calls to the CENTRALSUPPORT library either the default values or particular settings may be used. The task level and system level are probably the most useful levels for most programs.

The language and convention features have defined task attributes. These attributes can be accessed or set by programs to enable changes to the language, the convention, or both. While C programs do not have direct access to task attributes, these may be manipulated; for example, by writing a simple ALGOL library procedure and calling it from C. Refer to Appendix A, "Interface to the Library Facility," in this manual for more information. As an alternative to using the LANGUAGE and CONVENTION task attributes, the program may pass explicitly the desired language or convention name when calling CENTRALSUPPORT procedures.

The LANGUAGE task attribute can be used to establish the language used by a program at run-time. The CONVENTION task attribute can be used to establish the convention used by a program at run-time. For example, an international bank may have a program to print bank statements for customers in different countries. This program could have a general routine to format dates, times, currency, and numbers according to the selected conventions. To print a bank statement with the French language and conventions, this program could set both the LANGUAGE task attribute and the CONVENTION task attribute to the appropriate French settings and process the general routine.

The current default setting for the system can be determined by using one of the following methods:

- The program calls the `centralstatus` procedure in the CENTRALSUPPORT library.
- A system administrator, a privileged user, or a user who is allowed to use the system console can use MARC menus and screens or the SYSTEMOPTIONS system command. Refer to the MLS Guide or the *Menu-Assisted Resource Control (MARC) Operations Guide* for the instructions to display the default ccsversion, language, or convention with MARC.

Using Default Settings and Hierarchy

When calling a CENTRALSUPPORT library procedure, the default settings for ccsversion, language, and convention can be used without knowledge of their settings.

The default settings for ccsversion, language, and convention are shown as follows:

Option	Default Setting
Ccsversion	ASeriesNative
Language	English
Convention	ASeriesNative

The hierarchy of the levels must be considered before changing the default settings for localization. As an example, the default ccsversion is only changeable at the system operations level. For any procedure that accepts a ccsversion number as an

input parameter, the constant value CS_VSN_NOT_SPECIFIEDV can be specified as input to indicate that the system default value should be used. For any procedure that accepts a ccsversion name as an input parameter, all blanks or all nulls can be specified as inputs to indicate that the task attribute should be used. If the ccsversion name specified by the task attribute is not available, the CENTRALSUPPORT library searches down the hierarchy until a usable value is found.

Refer to "Understanding Default Settings and Hierarchy" in this appendix for more information on levels.

Understanding Components of the MLS Environment

Support for different languages and cultures comes from four components of the MLS environment:

- Coded character sets
- Ccsversions
- Languages
- Conventions

Coded Character Sets and Ccsversions

A *coded character set* or CCS is a set of rules that establishes a character set and the one-to-one relationship between the characters of the set and their code values. The same character set can exist with different encodings. For example, the LATIN1-based character set can be encoded in an International Organization for Standardization (ISO) format or an EBCDIC format. Coded character sets are defined in the MLS Guide. A coded character set name or number is given to each unique coded character set definition.

A coded character set name or number can also be used to set the INTMODE or EXTMODE file attribute values for a file. For more information on how to use the INTMODE and EXTMODE file attributes, refer to the *File Attributes Programming Reference Manual*.

This implementation supports three categories of CCS—8-bit, 16-bit, and mixed multibyte.

- An 8-bit CCS uses a one-byte (8-bit) code to represent each character in the set. Up to 256 characters can be represented. These character sets are fully described in Sections 9 through 12 of the *MLS Administration, Operations, and Programming Guide*.
- A 16-bit CCS uses a two-byte (16-bit) code to represent each character in the set. Approximately 64 thousand characters can be represented. These character sets are described in Section 13 of the *MLS Administration, Operations, and Programming Guide.*

- A mixed multibyte CCS consists of at least one 8-bit CCS and at least one 16-bit CCS, each of which is defined as a subCCS. SubCCS records specify the individual character sets and define how each is invoked. Mixed multibyte character sets are described in Section 13 of the *MLS Administration, Operations, and Programming Guide*.

A ccsversion is a collection of information necessary to apply a coded character set in a given country, language, or line of business. This information includes the processing requirements such as data classes, lowercase to uppercase mapping, ordering of characters, and the presentation set and escapement rules necessary for output. A ccsversion name and number are given to each unique group of information. A ccsversion name or number can also be used to set the CCSVERSION file attribute for a file. For more information, refer to the *File Attributes Programming Reference Manual*.

Each A Series system includes a data file, SYSTEM/CCSFILE, containing all coded character sets and ccsversions that are supported on the system. A system default coded character set cannot be chosen directly, but by choosing a ccsversion, the default coded character set is implicitly designated for the system.

Data can be entered and manipulated in only one coded character set and ccsversion at a time. Although there are many ccsversions that can be accessed, there is only one ccsversion active for the entire system at one time, which is called the system default ccsversion. For more information, refer to "Using the Ccsversion, Language, and Convention Default Settings" and "Understanding the Hierarchy for Default Settings" in this appendix.

There are several ways to determine which coded character sets and ccsversions are available on the system:

- Refer to the MLS Guide.
- Use the MARC menus and screens or the system command SYSTEMOPTIONS. For more information, refer to the MLS Guide or the *System Commands Operations Reference Manual*.
- Call the `ccsvsn_names_nums` CENTRALSUPPORT library procedure.

All coded character set and ccsversion information on the system can be accessed by calling CENTRALSUPPORT library procedures. To call these CENTRALSUPPORT library procedures, a C program must first include system-supplied file that declares all the library procedures. This file is called *SYMBOL/INTL/C/PROPERTIES. The syntax for including this file in a C program is as follows:

```
#include "*SYMBOL/INTL/C/PROPERTIES"       \
                 ( byfunction="CENTRALSUPPORT",  \
                   intname   ="CENTRALSUPPORT"   )
```

Many of the procedures require the specification of a coded character set or ccsversion as an input parameter. A program can choose a specific coded character set or ccsversion by calling the procedures using the name or number of the coded character set or the ccsversion as an input parameter. For example, by calling the `vsnordering_info` procedure with the ccsversion name ASeriesNative, then calling the `vsnordering_info` procedure again with the ccsversion name SWITZERLAND, a

program can access data in the ASeriesNative ccsversion and then access data in the Switzerland ccsversion. A program can also use the system default setting by using predefined values as input parameters. Refer to "Input Parameters" in this appendix for specific information about those parameters.

Mapping Tables

Many CENTRALSUPPORT library procedures store coded character sets and ccsversion information in translation tables that are used to process and map data. A translation table is used to translate one group of characters to another group of characters or another representation of the original characters, such as translating lowercase characters to uppercase characters.

The internationalization procedures provide access to the following translation tables that apply to data specified in coded character sets or specified ccsversions:

- Mapping data from coded character set to coded character set
- Mapping data from lowercase to uppercase characters
- Mapping data from uppercase to lowercase characters
- Mapping ccsversion-defined alternate numeric digits to U.S. EBCDIC numeric digits
- Mapping U.S. EBCDIC numeric digits to ccsversion-defined alternate numeric digits
- Mapping characters to their escapement values

Procedures from the CENTRALSUPPORT library may be used to access these translation tables or to process data using these tables. Use the `ccstoccs_trans_text` or the `ccstoccs_trans_text_complex` procedure to translate data from one coded character set to another coded character set and the `vsntrans_text` procedure to map lowercase data to uppercase data. Refer to the MLS Guide for definitions of mapping tables for each coded character set and ccsversion.

Data Classes

A data class is a group of characters sharing common attributes, such as alphabetic, upon which membership tests can be made. A truthset is a method of storing the declared set of characters upon which membership tests can be made. Many CENTRALSUPPORT library procedures store ccsversion information in ALGOL truthset tables as a way to define ccsversion data classes.

The internationalization features provide access to the following truthsets that apply to a specified ccsversion:

- Ccsversion alphabetic
- Ccsversion numeric
- Ccsversion graphics
- Ccsversion spaces

- Ccsversion lowercase
- Ccsversion uppercase

The alphabetic truthset contains those characters that are considered alphabetic for a specified ccsversion. The numeric truthset contains those characters that are considered numbers for a specified ccsversion, and so on.

Procedures from the CENTRALSUPPORT library may be used to access these truthsets or process data using these truthsets. For example, if a program has the text "1A23" it may use the `vsninspect_text` CENTRALSUPPORT library procedure to compare the text to the numeric truthset to determine if all the characters are numeric.

Refer to the explanation of the `vsninspect_text` procedure in this section for more information about using truthsets. Refer also to the MLS Guide for definitions of ccsversions and data classes.

Text Comparisons

A text comparison can be required for sorting text or for comparing relationships between two pieces of text.

The traditional method for handling text comparisons is based on a strict binary comparison of the character values. The binary method of comparison is not meaningful when used for sorting text if the binary ordering of the coded characters does not match the ordering sequence of the language's alphabet. This is the situation for most coded character sets.

Since the binary method is not sufficient for all usage requirements, two other levels of text comparison are supported:

1. Ordering

   Each character gets an *Ordering Sequence Value (OSV)*. An OSV is an integer from 0 through 255, assigned to each code position in a character set. The OSV signifies a relative ordering value of a character. An OSV of 1 indicates that the character comes before a character with an OSV of 2. More than one character may be assigned the same OSV.

   Only the character NUL may have an OSV of 0.

2. Collating

   Each character gets an OSV and a *Priority Sequence Value (PSV)*. A PSV is an integer from one to five that is assigned to each code position in a character set. The PSV is a relative priority value within each OSV. Each character with a unique OSV has a PSV of 1; however, if two characters have the same OSV, they have different PSVs for differentiation.

When comparing two strings of data, a comparison that uses only the first level, the Ordering level, is called an *equivalent comparison*. A comparison that utilizes both levels, Ordering and Collating, is called a *logical comparison*.

Specify the following three types of ordering for text comparisons when calling the procedures of the CENTRALSUPPORT library:

Order Type	Explanation
Binary	A comparison based on the binary values of the characters.
Equivalent	A comparison based on the ordering sequence values of the characters.
Logical	A comparison based on the ordering sequence values plus the priority sequence values of the characters.

In addition to the three types of ordering, the following types of character substitution are supported:

Substitution	Explanation
Many to One	A predetermined string of up to three characters can be ordered as if it were one character, assigning it a single OSV and PSV pair. Even if a character is part of a predetermined string of characters that are ordered as a single value, the character still has an OSV and a PSV pair assigned to it to allow for cases in which the character appears in other strings or individually. For example, in Spanish, the letter pair *ch* is ordered as if it were a single letter, different from either *c* or *h*, and ordering between *c* and *d*.
One to Many	A single character can generate a string of two or three OSV and PSV pairs. For example, the German sharp S character is ordered as though it were ss.

For a description of how the characters in each ccsversion are ordered and how text comparisons work, refer to the MLS Guide.

Providing Support for Natural Languages

The natural language feature enables users of an application program to communicate with the computer system in their natural language. A natural language is a human language in contrast to a computer programming language.

A program must be written in the subset of the standard EBCDIC character set defined by the C language. Only the contents of string literals, data items with variable character data, or comments can be in a character set other than EBCDIC.

If a program interacts with a user, has a user interface with screens or forms, displays messages, or accepts user input, then those aspects of the program should be in the natural language of the user. For example, French would be the natural language of a person born and raised in France. Refer to the MLS Guide for a list of user interface tools that can be localized. The following text explains how to develop a C application program, in conjunction with an ALGOL library, that supports interaction in the natural language of the user.

Creating Messages for an Application Program

In the MLS environment, the messages handled by the application program are grouped into the following categories:

Message Category	Explanation
Output message	A message that an application program displays to the user. Some examples of output messages are error messages and prompts for input. An output message is localized so that it can be displayed in the language of the user.
Input message	A message received by an interactive program either from a user or from another program in response to a prompt for input. The input message may be in a language that the program cannot recognize. In this instance, the message must be translated so that it can be understood by the program.

The localization process is easier if input and output messages are stored within an ALGOL output message array. When messages are in an output message array, the translator can use the MSGTRANS utility to localize the messages into one or more natural languages.

Create an output message array by creating an ALGOL library that contains OUTPUTMESSAGE ARRAY declarations. These output message arrays may contain output messages or translated input messages. The library can then be called from a C application program. The MLS Guide describes the procedures for creating and utilizing output message arrays.

In addition, a program is provided in the MLS Guide that demonstrates how to create an ALGOL library containing output message arrays and how to display the messages contained in the output message arrays from a C application. This program, called EXAMPLE/MLS/ALGOL/LIBRARY, is an example of how to use message arrays to make a C program multilingual.

Guidelines for Creating Multilingual Messages

The following are guidelines for creating multilingual messages:

- Put all output messages in output message arrays.
- Accept or display any messages through a library interface similar to that described previously.
- Use complete sentences for messages because phrases are hard to translate accurately.
- Do not use abbreviations because they also are hard to translate.
- Allow extra space for translated messages. The English language is more compact than many other natural languages and a message in English generally becomes 33 percent longer after it is translated into another language. For example, if a program can display an 80-character message, an English message should be only

60 characters long so that the translated message can expand by one-third and not exceed the maximum display size.

Business and Cultural Conventions

The business and cultural features enable users of an application program to display and receive data according to local conventions. A convention consists of formatting instructions for date, time, numbers, monetary amounts, and page size. Each system includes standard convention definitions that enable a user to choose among many formatting styles. For example, a user may choose formatting styles for Denmark, Italy, United Kingdom1, or Turkey.

Each system includes a data file called SYSTEM/CONVENTIONS, containing all convention definitions that are supported on the system. All conventions can be accessed by calling the CENTRALSUPPORT library procedures, requesting either explicit conventions by name or the default convention. Refer to the MLS Guide for complete information on all available conventions. To access the names of the conventions available on the host computer, use the `cnv_names` CENTRALSUPPORT library procedure. Use the `centralstatus` library procedure to find the name of the default convention on the system.

A new convention can be defined if none of the available conventions are satisfactory. This is done by creating a custom template. A template is a group of predefined control characters that describe the components for date, time, numeric, or monetary values; it is used in the CENTRALSUPPORT library procedures. For example, with the data item, 02251990, and the template, !0o!/!dd!/!yyyy!, the formatted date, 02/25/1990 is produced.

Refer to the MLS Guide for information on control characters, creating templates, and defining conventions.

Formatting Date and Time

The following types of CENTRALSUPPORT library procedures are available to format the date and time:

- The program supplies the convention name and the value for the date or time. The procedure returns the date or time value in the format used by the convention.
- The program supplies the format for the date or time in a template parameter, as well as the value for the date or time. Predefined control characters must be used to create the template. These control characters are described in the MLS Guide.
- The system supplies the date and time. There are both a system convention procedure that formats the system date and time according to a convention and a template procedure in which the program supplies the format.

The `cnv_displaymodel` procedure can be called to display to the user the format in which the date or time must be entered.

For example, use the `cnv_systemdatetime` procedure to display the system date and time according to the convention and language desired. If the ASeriesNative

convention and the English language is designated, the date and time is displayed as follows:

```
9:25  AM  Monday,  July 4, 1988
```

If you designate the FranceListing convention and the French language, the same date and time is displayed as follows:

```
9h25,  lundi  4 juillet 1988
```

Numerics and Currencies

The CENTRALSUPPORT library procedures can be called to inquire about numeric symbols or to format currency amounts. All numeric or currency symbols can be retrieved with a CENTRALSUPPORT library call. Monetary amounts in real number form can be formatted according to different conventions.

Page Size Formatting Features

The `cnv_formsize` CENTRALSUPPORT procedure enables the retrieval of lines-per-page and characters-per-line values for a specified convention.

For example, the Netherlands convention definition specifies 70 lines as the page length and 82 characters as the page width, while the Zimbabwe convention definition specifies 66 lines as the page length and 132 lines as the page width.

Summary of CENTRALSUPPORT Library Procedures

The CENTRALSUPPORT library procedures are integer-valued functions. The procedures return values in the output parameters and the function result value. The function result value should be checked to determine if an error has occurred.

The CENTRALSUPPORT library procedures are called by application programs and system software. An application program can call the CENTRALSUPPORT library procedures to perform the following tasks:

- Check data against ccsversion data classes
- Translate data using ccsversion mapping tables
- Compare data
- Translate characters using ccsversion escapement
- Determine the default settings on the host computer
- Identify and validate character sets and ccsversions
- Obtain information about a coded character set or a ccsversion
- Obtain default characteristics for forms and printers
- Obtain information about the convention contents

- Add, modify, and delete conventions
- Format date and time
- Format monetary and numeric data

Functional Grouping of CENTRALSUPPORT Library Procedures

The CENTRALSUPPORT library procedures are described here according to the tasks the procedures perform and the purpose of each procedure.

Identifying the Available Coded Character Sets and Ccsversions

The following table describes the available coded character sets and ccsversions:

Coded Character Sets and Ccsversions	Description
ccsvsn_names_nums	ccsvsn_names_nums returns the names and numbers of all coded character sets or all ccsversions available on the host computer. The names and numbers are listed in two arrays. These arrays are ordered so that the names in the names array correspond to the numbers in the numbers array.
centralstatus	centralstatus obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and conventions. It also returns the number of the default ccsversion.
validate_name_return_num	validate_name_return_num verifies that a designated coded character set or ccsversion name is valid on the host computer. If the character set or ccsversion is valid, the procedure returns the corresponding number.
validate_num_return_name	validate_num_return_name verifies that the designated coded character set or ccsversion number is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the name.

Obtaining Coded Character Set and Ccsversion Information

ccsinfo

ccsinfo provides basic information about a designated coded character set, including the number of bits per character (8, 16, or mixed 8 and 16) , the coding format (for example, ISO, Doublebyte, or UCSBMP) and the space character.

vsninfo

vsninfo returns the following information for a designated ccsversion:

- The number of the base character set to which the ccsversion applies
- The escapement information
- The space characters used for the ccsversion
- The array sizes required by the other ccsversion elements

Mapping Data from One Coded Character Set to Another

ccstoccs_trans_table

ccstoccs_trans_table returns a translation table used to translate data appearing in one designated character set to another designated character set. The result is a one-to-one mapping of the characters. Note that ccstoccs_trans_table **cannot** be used to obtain a translation table for multibyte character set conversions.

ccstoccs_trans_text

ccstoccs_trans_text translates data (text) from one character set to another character set by using a translation table. Characters are translated using a one-to-one mapping between the two character sets. Note that ccstoccs_trans_text **cannot** be used to map multibyte character sets.

ccstoccs_trans_text_complex

ccstoccs_trans_text_complex is an enhanced procedure that translates data (text) from one single- or multibyte character set to another single- or multibyte character set by using a translation table. Characters are translated by using a one-to-one mapping between the two character sets.

inspect_text_using_tset

inspect_text_using_tset compares the input text to the truthset returned by the vsntruthset procedure to determine whether all the characters in the text are in the truthset. This procedure can be used to determine if characters are in one of the following data classes:

- Alphabetic
- Numeric
- Lowercase
- Spaces
- Presentation
- Uppercase

trans_text_using_ttable

`trans_text_using_ttable` uses the translation table obtained by using the `vsntranstable` procedure to translate data. The type of table used determines the type of translation done. All translations are one-to-one mappings of the characters. This procedure can be used to perform the following types of translations:

- Lowercase to uppercase characters
- Uppercase to lowercase characters
- The digits 0 through 9 to alternative digits
- Alternative digits to the digits 0 through 9
- The escapement direction of each character

Processing Data According to a Ccsversion

`vsninspect_text`

`vsninspect_text` compares the input text to a designated ccsversion truthset to determine whether the characters in the text are in the truthset. This procedure can be used to determine if characters are in one of the following data classes:

- Alphabetic
- Numeric
- Lowercase
- Spaces
- Presentation
- Uppercase

`vsntranstable`

`vsntranstable` returns a translation table for a designated ccsversion. The type of translation table requested depends on the task to be performed.

Translation tables can be requested to perform the following tasks:

- Translate lowercase letters to uppercase letters
- Translate uppercase letters to lowercase letters
- Translate any digits 0 through 9 to any alternative digits (that is, one-to-one mapping of 0 through 9 to another representation for those digits)
- Translate alternative digits to 0 through 9
- Determine the escapement direction for each characters

vsntrans_text

vsntrans_text translates data using a designated ccsversion translation table. The type of table used determines the type of translation done. All translation is a one-to-one mapping of the characters.

This procedure can be used to perform the following types of translations:

- Lowercase to uppercase characters
- Uppercase to lowercase characters
- The digits 0 through 9 to alternative digits
- Alternative digits to the digits 0 through 9
- The escapement direction of each character

vsntruthset

vsntruthset returns a truthset for the designated ccsversion. The truthset contains the characters in a given data class for the ccsversion and are available for the following data classes:

- Alphabetic
- Numeric
- Lowercase
- USpace
- Presentation
- Manipulating
- Uppercase

Comparing and Sorting Text

compare_text_using_order_info

compare_text_using_order_info compares two strings by using the ordering information returned by the vsnordering_info procedure. One of the following comparison methods can be chosen:

- Equivalent comparison

This is based on the ordering sequence values of the characters

- Logical comparison

This is based on the ordering sequence values and the priority sequence values of the characters

vsncompare_text

vsncompare_text compares text where the first record can be less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the second. One of the following comparison methods is used for a designated ccsversion:

- Binary comparison

This is based on the binary code values of the characters

- Equivalent comparison

This is based on the OSVs of the characters

- Logical comparison

This is based on the OSVs and PSVs of the characters that return the ordering information for the input text.

vsngetorderingfor_one_text

vsngetorderingfor_one_text returns the ordering information for the input text. The ordering information determines how the input text is collated, including the ordering and priority sequence values of the characters and any substitution of characters made when the input text is sorted.

One of the following types of ordering information can be chosen:

- Equivalent ordering information

  This information comprises the ordering sequence values only.

- Logical ordering information

  This information comprises the ordering sequence values followed by the priority sequence values

vsnordering_info

vsnordering_info returns the ordering information for a designated ccsversion. The ordering information determines the way in which data is collated for the ccsversion. It includes the ordering and priority sequence values of the characters and any substitution of characters made when the designated ccsversion ordering is applied to a string of text.

Positioning Characters

vsnescapement

vsnescapement rearranges the input text according to the escapement rules of the ccsversion.

Determining the Available Natural Languages

mcp_bound_languages

mcp_bound_languages returns the names of the languages that are currently bound to the MCP.

Accessing CENTRALSUPPORT Library Messages

get_cs_msg

get_cs_msg returns the text of a CENTRALSUPPORT message associated with the designated error number.

Identifying the Available Convention Definitions

centralstatus

centralstatus obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and conventions. It also returns the number of the default ccsversion.

cnv_names

cnv_names returns the names of the conventions available on the host computer.

cnv_validatename

cnv_validatename indicates whether a designated convention name is currently defined on the host computer.

Obtaining Information About Conventions

cnv_info

cnv_info returns a description of all the format templates defined in a designated convention.

cnv_symbols

cnv_symbols returns the numeric and monetary symbols defined for a designated convention. The symbols in the convention are

- Numeric positive symbol
- Numeric negative symbol
- Numeric thousands separator symbol

- Numeric decimal symbol
- Numeric left enclosure symbol
- Numeric right enclosure symbol
- Numeric grouping specifications
- International currency notation
- Monetary positive symbol
- Monetary negative symbol
- Currency symbol
- Monetary thousands separator symbol
- Monetary decimal symbol
- Monetary left enclosure symbol
- Monetary right enclosure symbol
- Monetary grouping specifications

cnv_template

cnv_template returns the requested format template for a designated convention set. The template can be obtained for the following:

- Long date format
- Short date format
- Numeric date format
- Long time format
- Numeric time format
- Monetary format
- Numeric format

Formatting Dates According to a Convention

cnv_convertdate

cnv_convertdate converts a formatted numeric date passed as a parameter to the procedure to the format YYYYMMDD. The numeric date must be formatted according to the numeric date format template defined in the designated convention.

cnv_displaymodel

cnv_displaymodel returns either the date or time display model defined for a designated convention. The component designators for the model are translated to the designated language.

cnv_formatdate

cnv_formatdate formats a numeric date passed as a parameter to the procedure according to a designated convention and language. The date can be formatted using the long, short, or numeric date format defined in the convention.

cnv_formatdatetmp

cnv_formatdatetmp formats a numeric date passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

cnv_systemdatetime

cnv_systemdatetime returns the system date and/or time formatted according to the designated convention and language. The following types of templates can be chosen:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

cnv_systemdatetimetmp

cnv_systemdatetimetmp returns the system date and/or time in the designated language, formatted according to a template passed as a parameter to this procedure.

Formatting Times According to a Convention

cnv_converttime

cnv_converttime converts a formatted numeric time passed as a parameter to the procedure to the format HHMMSS. The numeric time must be formatted according to the numeric time format template defined in the designated convention.

cnv_displaymodel

cnv_displaymodel returns either the date or time display model defined for a designated convention. The component designators for the model are translated to the designated language.

cnv_formattime

cnv_formattime formats a time passed as a parameter to the procedure according to a designated convention and language. The time can be formatted using the long or numeric time format defined in the convention.

cnv_formattimetmp

cnv_formattimetmp formats a time passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

cnv_systemdatetime

cnv_systemdatetime returns the system date and/or time formatted according to the designated convention and language. The following types of templates can be chosen:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

cnv_systemdatetimetmp

cnv_systemdatetimetmp returns the system date and/or time in the designated language, formatted according to a template passed as a parameter to this procedure.

Formatting Numeric Data According to a Convention

cnv_convertnumeric

cnv_convertnumeric converts a string containing digits and numeric symbols to a real number.

Formatting Monetary Data According to a Convention

cnv_convertcurrency

cnv_convertcurrency converts a string containing digits and monetary symbols to a real number.

cnv_currencyedit_double

cnv_currencyedit_double formats a monetary value passed as a parameter to the procedure according to the monetary editing format template defined in the designated convention.

cnv_currencyedittmp_double

cnv_currencyedittmp_double formats a monetary value passed as a parameter to the procedure according to a template also passed as a parameter.

Determining the Default Page Length and Width

cnv_formsize

cnv_formsize returns the lines-per-page and characters-per-line values from a designated convention for formatting printer output.

Adding, Modifying, and Deleting Conventions

cnv_add

cnv_add adds a new convention to the SYSTEM/CONVENTIONS file. The new definition goes into effect immediately.

cnv_delete

cnv_delete deletes an existing convention from the SYSTEM/CONVENTIONS file. The convention is deleted after the next halt/load. Only conventions that have been created by a user can be deleted. System-supplied conventions cannot be deleted.

cnv_modify

cnv_modify modifies an existing convention in the SYSTEM/CONVENTIONS file. The modified set becomes effective after the next halt/load. Only conventions that have been created by a user can be modified. System-supplied conventions cannot be modified.

Library Calls

You can access the procedures in the CENTRALSUPPORT library within a C program by using the following steps:

1. Use the `#include` directive to include the file *SYMBOL/INTL/C/PROPERTIES and declare the CENTRALSUPPORT library entry points, such as:

```
#include "*SYMBOL/INTL/C/PROPERTIES"    \
               ( byfunction="CENTRALSUPPORT", \
                 intname   ="CENTRALSUPPORT"  )
```

2. Call the procedure

An example of the call syntax necessary to invoke the CENTRALSUPPORT library is provided in the description of each procedure in this section.

Refer to Appendix A, “Interface to the Library Facility,” in this manual for details on calling ALGOL procedures from a C program.

Parameter Categories

The CENTRALSUPPORT library procedures return output parameters and procedure result values. The parameter types are further described here.

Input Parameters

In many cases, the input parameter requires the program to supply the ccsversion name or number, the language name, or the convention name. This information can be obtained in the following ways:

- The MLS Guide describes all the ccsversions, languages, and conventions that are provided. However, the system on which the program is running may have only a subset of these. There can also be customized conventions that are not listed in the MLS Guide. These can be identified by the next three options.

- A system administrator, a privileged user, or any person allowed to use the system console can use MARC menus and screens to list the options that exist on the system. The MLS Guide provides the instructions needed to obtain information about ccsversions, languages, or convention definitions.

- Procedures can be called in the CENTRALSUPPORT library that can return this information. If an application is being written to be used on another system, these library procedures can be used to verify that the ccsversion, language, or convention specified by the user is valid on the system.

- The SYSTEMOPTIONS command can be used. For more information on the SYSTEMOPTIONS command, refer to the *System Commands Operations Reference Manual*.

For any procedure that accepts a ccsversion number as an input parameter, specify a -2 (or the macro name `CS_VSN_NOT_SPECIFIEDV`) as input to indicate that the system default value should be used. For any procedure that accepts a ccsversion name as an

input parameter, specify all blanks or all zeros as inputs to indicate that the system default value should be used. For any procedure that accepts a language or convention name as an input parameter, specify all blanks or zeros as inputs to indicate that the task attribute should be used. If the task attribute is not available, the CENTRALSUPPORT library searches down the hierarchy until a usable value is found.

Input Parameters with Type Values

Many of the CENTRALSUPPORT procedures have either an input parameter or an output parameter, which indicate the type of information to be applied or returned by the procedure. The values in these parameters are referred to as type values and are common across all procedures of the library.

For example, the `typ` parameter is used in a number of procedures to indicate the type of date or time formatting used. The type value indicates the type of format used. For example, a value of 3 indicates the long time format.

The file SYMBOL/INTL/C/PROPERTIES defines C macros corresponding to each of the valid type values. Use these macros rather than the actual values; for example, rather than using the value 3, the long time format may be designated with the macro `CS_LTIMEV`.

Output Parameters

These parameters are filled in by the procedure. For example, the `dest` parameter of the `ccstoccs_trans_text` procedure contains the translated text produced by the procedure.

Return Value

All the library procedures return a value as the procedure result, indicating whether an error occurred during the execution of the procedure. In general, a returned value of `CS_DATAOKV` or `CS_FALSEV` means that no error occurred and any other value means an error occurred. However, the `cnv_validatename` and `vsncompare_text` procedures are exceptions to this rule. For these procedures, the returned value can be 0 (zero), 1, or another value. A returned value of 0 (zero) means that no error occurred and the condition is FALSE. A returned value of 1 means that no error occurred and the condition is TRUE. Any other value means that an error occurred.

Each procedure lists the values that can be returned by that procedure; these values are explained at the end of this section. The results can be used to call the `get_cs_msg` procedure and to display the error that occurred or error routines can be coded to handle the possible errors.

The INCLUDE file contains `#defines` that map each integer result into a macro identifier to be used by the program.

Procedure Descriptions

The following is an alphabetical listing of the procedures residing in the CENTRALSUPPORT library that a C program can access. Each description includes a function prototype, a general overview of the procedure, an example showing how to call the procedure, and an explanation of the parameters used in the example.

Refer to the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation* for more extensive examples of the use of these procedures.

ccsinfo

Prototype

```
int ccsinfo( int ccs_num, float ( *ccs_ary) [ ], int ccs_ary_size);
```

This procedure provides basic information about a designated coded character set, including the number of bits per character (8, 16, or mixed 8 and 16) and the coding format (such as ISO, STDEBCDIC, or DoubleByte).

`ccsinfo` can be used, for example, to determine the code format for a file that has its INTMODE set to a character set name. Refer to the MLS Guide for a list of coded character sets available.

The `ccsinfo` procedure can be called as follows:

```
rslt = ccsinfo ( ccs_num, &ccs_ary, ccs_ary_size );
```

`ccs_num` is an integer passed to the procedure, containing the number of the coded character set about which information is requested.

`ccs_ary` is a float array returned by the procedure, containing the following information about a coded character set from the character set and ccsversion file:

```
Location            Information

Word [0]            Number of bits per character
                               8 bits = 1
                              16 bits = 2
           mixed (multibyte) 8/16 bits = 3

Word [1]            Repetoire number (not used by new coded character sets)

Word [2]            Encoding number  (not used by new coded character sets)

Word [3]            Code format
                                ISO = 1
                     ASERIESEBCDIC = 2
                         STDEBCDIC = 3
                              BTOS = 4
                                PC = 5
                          EBCDICMB = 6
                             ISOMB = 7
                              PCMB = 8
                             EUCMB = 9
                            2200MB = 10
                        DoubleByte = 13
                             UCSBMP = 14
                           UCSBMPNT = 15

Word [4]            A full word of space characters
```

```
                                        % If CCS is 8-bit, space
                                        % characters are 8-bit.  If CCS
                                        % 16-bit, space characters are
                                        % 16-bit.  If CCS is mixed,
                                        % space characters are 8-bit and
                                        % subCCS records also identify
                                        % 8- or 16-bit space characters.

Word [5]+           Extension facilities or subCCS records
                                        % Extension facilities are defined
                                        & when the code format is ISO
                                        & (Word [3]=1).
                                        & SubCCS records are defined for
                                        & mixed character sets (Word [0]=3).
```

Extension Facilities Format

```
Word [5].[47:08]     ISO extension facilities
                                        % 0 means NONE
                                        % 1 means UNISYS01
                                        % 255 means non-UNISYS01

     If word [5].[47:08] = 0 (NONE) the remainder of word 5 is empty.

     If word [5].[47:08] = 1 (UNISYS01) then

        Word [5].[39:01]   G1 set
                                        % 0 means undefined G1 set
                                        % 1 means defined G1 set

           If word [5].[39:01] = 1 (defined) then

              Word [5].[38:09]  Number of characters in G1 set (94 or 96)

              Word [5].[29:01]  Registration
                                        % 0 means unregistered
                                        % 1 means registered
              If word [5].[28:01] = 1 (registered) then

                   Word [5].[28:13]  Not used

                   Word [5].[15:16]  Number of escape sequence characters
                                     in the escape sequence (starting at
                                     word 6)

     If word [5].[47:08] = 255 (non-UNISYS01) then

        Word [5].[39:24]  Not Used

        Word [5].[15:16]  Number of escape sequence characters in the
                          escape sequence (starting at word 6)
```

SubCCS Records Format

```
Word [5]     Number of subCCS records (each record is 4 words long)

Word [6]     SubCCS library number

Word [7]     Character size of subCCS
                          %  8 bits = 1
                          % 16 bits = 2

Word [8]     Invocation of subCCS
                          %      None = 0
                          %   EUC SS2 = 1
                          %   EUC SS3 = 2
                          % EBCDIC SB = 3
                          %  ASCII SB = 4
                          %   2200 LS = 5

Words 6-9 are repeated starting at word10 for every subCCS which is a part of t
his coded character set.  That is, word 5 is followed by four words of informat
ion for every subCCS record.  The number of subCCS records is specified in word
 5.
```

Notes:

- *Extension facilities are the escape sequences in an ISO environment that are used to specify the code sets (such as C0, C1, G0, G1) from which ISO builds coded character sets. The escape sequences are defined in ISO standard 2022, titled "Information Processing -- ISO 7-Bit and 8-Bit Coded Character Sets".*

- *A mixed, multibyte coded character set requires at least one 8-bit CCS, at least one 16-bit CCS and a set of subCCS records to define characteristics of each CCS in the set.*

The recommended array size without escape sequences or subCCS information is five words. In all other cases, the recommended size is 30 words.

`ccs_ary_size` is an integer containing the number of elements in `ccs_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the ccsinfo procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `ccsinfo` procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV                         CS_FAULTV
CS_FILE_ACCESS_ERRORV              CS_SOFTERRV
```

ccstoccs_trans_table

Prototype

```
int ccstoccs_trans_table( int ccsnumfrom, int ccsnumto,
                          float (*ttable_ary) [ ],int ttable_ary_size );
```

This procedure returns an ALGOL-type translation table used to translate data appearing in one designated coded character set to another designated coded character set. The result is a one-to-one mapping of the characters.

When an application program performs a high volume of translations, this procedure can be used in combination with the `trans_text_using_ttable` procedure instead of calling the `ccstoccs_trans_text` procedure. The translation table from `ccstoccs_trans_table` can be passed as a parameter to `trans_text_using_ttable`. Because the table is stored in a program variable, it does not have to be retrieved each time it is used.

The `ccstoccs_trans_table` function can be called as follows:

```
rslt = ccstoccs_trans_table ( ccsnumfrom, ccsnumto,
                              &ttable_ary, ttable_ary_size );
```

`ccsnumfrom` is an integer passed to the procedure, containing the number of the coded character set from which translation is to occur.

`ccsnumto` is an integer passed to the procedure, containing the number of the coded character set to which translation is to occur.

`ttable_ary` is a float array in which the translation table is returned. The recommended size is 64 words.

`ttable_ary_size` is an integer containing the number of elements in `ttable_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `ccstoccs_trans_table` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `ccstoccs_trans_table` procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_DATA_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                    CS_FILE_ACCESS_ERRORV
CS_SOFTERRV                  CS_COMPLEX_TRAN_REQV
```

ccstoccs_trans_text

Prototype

```
int ccstoccs_trans_text( int ccsnumfrom, int ccsnumto,
                         char (*source) [ ], int source_size,
                         int source_start, char (*dest) [ ], int dest_size,
                         int dest_start,int_trans_len );
```

The ccstoccs_trans_text procedure translates data from one coded character set specified in the ccsnumfrom parameter to the coded character set specified in the ccsnumto parameter by using a translation table. Characters are translated using a one-to-one mapping between two character sets.

For example, a program may translate text in the LATIN1EBCDIC coded character set to the LATIN1ISO coded character set. Refer to the MLS Guide for a list of the coded character set numbers that are available as inputs to this procedure.

Mapping tables are not available for every combination of coded character set pairs. If the requested mapping is not available through this procedure or through the ccstoccs_trans_text_complex procedure, the procedure returns the value CS_DATA_NOT_FOUNDV. If the requested mapping is only available through the ccstoccs_trans_text_complex procedure, the procedure returns the value CS_COMPLEX_TRAN_REQV.

The ccstoccs_trans_text function can be called as follows:

```
rslt = ccstoccs_trans_text ( ccsnumfrom, ccsnumto,
                             &source, source_size, source_start,
                             &dest, dest_size, dest_start, trans_len );
```

ccsnumfrom is an integer passed to the procedure, containing the number of the coded character set associated with source data.

ccsnumto is an integer passed to the procedure, containing the number of the coded character set to which translation occurs. The destination text is in this coded character set.

source is a character array passed to the procedure, containing the text to translate.

source_size is an integer containing the number of elements in source.

source_start is an integer passed to the procedure specifying the offset (0 relative) where the translation starts.

dest is a character array in which the translated text is returned.

dest_size is an integer containing the number of elements in dest.

dest_start is an integer passed to the procedure, specifying the byte offset (0 relative) where the output text is stored in the dest array.

trans_len is an integer passed to the procedure, specifying the number of characters in source to translate, beginning at source_start.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the ccstoccs_trans_text procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the ccstoccs_trans_text procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_DATA_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                    CS_FILE_ACCESS_ERRORV
CS_NO_NUM_FOUNDV             CS_SOFTERRV
```

ccstoccs_trans_text_complex

Prototype

```
int ccstoccs_trans_text_complex( int ccsnumfrom, int ccsnumto,
                                 char (*source)[], int source_size,
                                 int *source_inx, int source_bytes,
                                 char (*dest)[], int dest_size,
                                 int *dest_inx, int dest_bytes,
                                 float (*state)[],
                                 int option )
```

The ccstoccs_trans_text_complex procedure translates data from one coded character set specified in the ccsnumfrom parameter to the coded character set specified in the ccsnumto parameter by using a translation table. Characters are translated using a one-to-one mapping between two character sets. This procedure is more flexible than the ccstoccs_trans_text procedure in that it handles both single byte and multibyte coded character sets.

For example, a program may translate text in the 16-bit JBIS8 coded character set to the 16-bit JSBIS7 coded character set. Refer to the MLS Guide for a list of the coded character set numbers that are available as inputs to this procedure.

Mapping tables are not available for every combination of coded character set pairs. If the requested mapping is not available, the procedure returns the value CS_DATA_NOT_FOUNDV.

The ccstoccs_trans_text_complex procedure can be called as follows:

```
rslt = ccstoccs_trans_text_complex ( ccsnumfrom, ccsnumto,
                                     &source,
                                     sizeof(source)/sizeof(source[0]),
                                     &source_inx, source_bytes,
                                     &dest,
                                     sizeof(dest)/sizeof(dest[0]),
                                     &dest_inx, dest_bytes,
                                     &state,
                                     sizeof(state)/sizeof(state[0]),
                                     CS_OPT_INITIAL_COMPLETEV);
```

ccsnumfrom is an integer passed to the procedure, containing the number of the coded character set associated with source data.

ccsnumto is an integer passed to the procedure, containing the number of the coded character set to which translation occurs. The destination text is in this coded character set.

source is a character array passed to the procedure, containing the text to translate. The maximum size is 64,000 words.

source_size is an integer containing the number of elements in the source array.

source_inx is an integer passed to the procedure specifying the offset (0 relative) where the translation starts.

source_bytes is an integer passed to the procedure specifying the number of bytes to map (beginning at source_inx).

dest is a character array in which the translated text is returned.

dest_size is an integer containing the number of elements in the dest array. A value that is twice the size of source_size always works.

dest_inx is an integer specifying the byte offset (0 relative) where the output text is stored in the dest array.

dest_bytes is an integer specifying the number of bytes (beginning at dest_inx) that can be filled with destination characters.

state is a real array that is ten words long. It is owned by (but not touched by) the caller. It is structured as follows:

```
Word 0      Indices to CENTRALSUPPORT tables
Word 1      Status information from previous operation
Word 2      Partial source character
Word 3      Special case optimization information
Words 4-9   Reserved
```

state_size is an integer that defines the size of the state array. It must be greater than or equal to 10.

option is an integer that indicates whether this is a continuation call. Possible values and their meaning are:

Value	Message
0	CS_OPT_INITIAL_COMPLETEV
1	CS_OPT_TAILV
2	CS_OPT_INITIAL_HEADV
3	CS_OPT_MIDDLEV
4	Same as 0.
5	CS_OPT_COMPLETEV
6	Same as 2.
7	CS_OPT_HEADV

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the ccstoccs_trans_text_complex procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible error messages returned by the ccstoccs_trans_text_complex procedure are:

```
CS_ARRAY_TOO_SMALLV              CS_FILE_ACCESS_ERORV
CS_BAD_DATA_LENV                 CS_INCOMPLETE_CHARV
CS_BAD_ARRAY_DESCRIPTIONV        CS_INCOMPLETEDATAV
CS_DATA_NOT_FOUNDV               CS_SOFTERRV
CS_FAULTEDV
```

ccsvsn_names_nums

Prototype

```
int ccsvsn_names_nums( int ccsvsn_type, int *total, char (*names_ary) [ ],
                       int names_ary_size, int (*nums_ary) [ ],
                       int nums_ary_size );
```

This procedure returns a list of coded character set names and numbers or a list of ccsversion names and numbers that are available on the system. The type of list can be specified with the ccsvsn_type parameter.

This procedure may be used to create a menu that lists the ccsversions from which to choose on the host computer. It may also be used to verify that the ccsversion used by a program is available on the host computer.

The ccsvsn_names_nums function can be called as follows:

```
rslt = ccsvsn_names_nums ( ccsvsn_type, &total, &names_ary,
                           names_ary_size, &nums_ary, nums_ary_size );
```

ccsvsn_type is an integer passed to the procedure. It allows specification of either of the following two types of information returned in the output parameters:

Value	Macro Name	Meaning
0	CS_CHARACTER_SETV	Return the names and numbers of the character sets.
1	CS_CCSVERSIONV	Return the names and numbers of the ccsversions.

total is an integer returned by the procedure that contains the number of character set or ccsversion entries that exist.

names_ary is a character array returned by the procedure. Each name corresponds to the name of a coded character set or ccsversion defined on the host. Each name takes 17 EBCDIC characters. Refer to the MLS Guide for the names of all the character sets

and ccsversions. The minimum recommended array size is 17 times the number of coded character sets or ccsversions on the system.

names_ary_size is an integer containing the number of elements in names_ary.

nums_ary is an array of integers returned by the procedure, containing the character set or ccsversion numbers. nums_ary contains all the character set or ccsversion numbers defined on the host. Each element in nums_ary corresponds to a 17-character name in names_ary. Refer to the MLS Guide for all the numbers for the character sets and ccsversions.

nums_ary_size is an integer containing the number of elements in nums_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the ccsvsn_names_nums procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the ccsvsn_names_nums procedure are:

```
CS_ARRAY_TOO_SMALLV              CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_TYPE_CODEV                CS_DATAOKV
CS_FAULTV                        CS_SOFTERRV
```

centralstatus

Prototype

```
int centralstatus( char (*sys_info) [ ], int sys_info_size,
                   int (*control_info) [ ], int control_info_size );
```

This procedure returns the name and number of the system default ccsversion, the name of the system default convention, and the name of the system default language. Information about the level of the CENTRALSUPPORT library is also returned.

This procedure may be used to provide a means for application users to inquire about the default settings on the host computer.

The centralstatus function can be called as follows:

```
rslt = centralstatus ( &sys_info,  sys_info_size,
                       &control_info,  control_info_size );
```

sys_info is a character array returned by the procedure, containing the system default ccsversion, language, and convention name in that order. Each name is 17 characters long. Names shorter than 17 characters are padded on the right with blanks. The recommended array size is 51 characters.

sys_info_size is an integer containing the number of elements in sys_info.

control_info is an integer array returned by the procedure, containing the following information:

Location	Information
Word[0]	System default ccsversion number
Word[1] through [7]	Reserved

The recommended array size is eight words.

control_info_size is an integer containing the number of elements in control_info.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the centralstatus procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the centralstatus procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV                         CS_FAULTV
CS_FILE_ACCESS_ERRORV              CS_SOFTERRV
```

cnv_add

Prototype

```
int cnv_add( char (*cnv_name) [ ], int cnv_name_size, float (*add_ary) [ ],
             int add_ary_size );
```

The cnv_add procedure adds a new convention to the SYSTEM/CONVENTIONS file. The new definition goes into effect immediately.

The cnv_add function can be called as follows:

```
rslt = cnv_add ( &cnv_name, cnv_name_size, &add_ary, add_ary_size );
```

cnv_name is a character array passed to the procedure, containing the name of the convention to be added to the SYSTEM/CONVENTIONS file.

cnv_name_size is an integer containing the number of elements in cnv_name.

add_ary is a float array passed to the procedure, containing the convention definition to be added to the SYSTEM/CONVENTIONS file. For the procedure to work, all fields in add_ary must contain data or an appropriate error result is returned. Data in add_ary is passed in fields as follows:

Field Meaning	Offset
Maximum integer digits	0
Maximum decimal digits	1
Maximum international decimal digits	2
International currency notation	3

Field Meaning	Offset
Long date template	5
Short date template	13
Numeric date template	21
Long time template	25
Numeric time template	33
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

The international currency notation field contains the international currency symbol defined for the convention. The international currency symbol is surrounded by a pair of matching delimiters that are not part of the symbol. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency symbol for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

`add_ary_size` is an integer containing the number of elements in `add_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_add` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_add` procedure are:

```
CS_BAD_ALTFRACDIGITSV              CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_CPLV                        CS_BAD_DATA_LENV
CS_BAD_FRACDIGITSV                 CS_BAD_LDATETEMPV
CS_BAD_LPPV                        CS_BAD_LTIMETEMPV
CS_BAD_MAXDIGITSV                  CS_BAD_MONTEMPV
CS_BAD_NDATETEMPV                  CS_BAD_NTIMETEMPV
CS_BAD_NUMTEMP                     CS_BAD_SDATETEMPV
CS_CNV_EXISTS_ERRV                 CS_CNVFILE_NOTPRESENTV
CS_CONVENTION_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                          CS_FILE_ACCESS_ERRORV
CS_SOFTERRV
```

cnv_convertcurrency

Prototype

```
int cnv_convertcurrency( int vsnnum, char (*cc_ary) [ ], int cc_ary_size,
                         char (*cnv_name) [ ], int cnv_name_size,
                         float *amt );
```

This procedure converts a string containing digits and monetary symbols to a real number.

The cnv_convertcurrency function can be called as follows:

```
rslt = cnv_convertcurrency ( vsnnum, &cc_ary, cc_ary_size,
                             &cnv_name, cnv_name_size, &amt );
```

vsnnum is an integer passed to the procedure, containing the number of the ccsversion that was in effect when the input string was created. The ccsversion contains the rules for translation of text. Refer to the MLS Guide for a list of the ccsversion numbers. If the input string contains alternate digits, then they are translated to their corresponding 0-9 numbers. The values allowed for vsnnum and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

cc_ary is a character array that contains the input string of digits and monetary symbols.

cc_ary_size is an integer containing the number of elements in cc_ary.

cnv_name is a character array that is passed to the procedure, containing the name of the convention used to format the input string. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used.

cnv_name_size is an integer containing the number of elements in cnv_name.

amt is a float number that is returned by the procedure, containing the converted number.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_convertcurrency procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_convertcurrency procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV                   CS_BAD_INPUTVALV
CS_CONVENTION_NOT_FOUNDV           CS_DATA_NOT_FOUNDV
CS_DATAOKV                         CS_FAULTV
CS_FILE_ACCESS_ERRORV              CS_SOFTERRV
```

cnv_convertdate

Prototype

```
int cnv_convertdate( int vsnnum, char (*cd_ary) [ ], int cd_ary_size,
                     char (*cnv_name) [ ], int cnv_name_size,
                     char (*date_ary) [ ], int date_ary_size );
```

This procedure converts a formatted numeric date passed as a parameter to the procedure to the format YYYYMMDD. The numeric date must be formatted according to the numeric date format template defined in the designated convention.

The `cnv_convertdate` function can be called as follows:

```
rslt = cnv_convertdate ( vsnnum, &cd_ary, cd_ary_size, &cnv_name,
                         cnv_name_size, &date_ary, date_ary_size );
```

`vsnnum` is an integer passed to the procedure, containing the ccsversion number that was in effect when date was formatted. If the input string contains alternate digits, then they are translated to their corresponding 0-9 numbers. The values allowed for `vsn_num` and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system ccsversion is not available, an error is returned.

`cd_ary` is a character array passed to the procedure, containing the formatted date.

`cd_ary_size` is an integer containing the number of elements in `cd_ary`.

`cnv_name` is a character array passed to the procedure, containing the name of the convention from which the date formatting templates are retrieved.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

`date_ary` is a character array returned by the procedure, containing the converted date in the form YYYYMMDD.

`date_ary_size` is an integer containing the number of elements in `date_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_convertdate` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_convertdate` procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV                   CS_BADDATEINPUTV
CS_BAD_TEMPCHARV                   CS_BAD_TYPE_CODEV
CS_CONVENTION_NOT_FOUNDV           CS_DATA_NOT_FOUNDV
CS_DATAOKV                         CS_FAULTV
CS_FILE_ACCESS_ERRORV              CS_SOFTERRV
```

cnv_convertnumeric

Prototype

```
int cnv_convertnumeric( int vsnnum, char (*cc_ary) [ ], int cc_ary_size,
                        char (*cnv_name) [ ], int cnv_name_size,
                        float *amt );
```

This procedure converts a string containing digits and numeric symbols to a float number.

The `cnv_convertnumeric` function can be called as follows:

```
rslt = cnv_convertnumeric ( vsnnum, &cc_ary, cc_ary_size,
                            &cnv_name, cnv_name_size, &amt );
```

`vsnnum` is an integer that is passed to the procedure, containing the number of the ccsversion that was in effect when the input string was created. If the input string contains alternate digits, then they are translated to their corresponding 0-9 numbers. The values allowed for `vsn_num` and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

`cc_ary` is a character array that is passed to the procedure, containing the string of digits and numeric symbols to be converted.

`cc_ary_size` is an integer containing the number of elements in `cc_ary`.

`cnv_name` is a character array that is passed to the procedure, containing the name of the convention used to format the input string.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

`amt` is a result returned by the procedure, containing the converted float number.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_convertnumeric` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_convertnumeric procedure are:

CS_ARRAY_TOO_SMALLV	CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV	CS_BAD_INPUTVALV
CS_CONVENTION_NOT_FOUNDV	CS_DATA_NOT_FOUNDV
CS_DATAOKV	CS_FAULTV
CS_FILE_ACCESS_ERRORV	CS_SOFTERRV

cnv_converttime

Prototype

```
int cnv_converttime( int vsnnum, char (*ct_ary) [ ], int ct_ary_size,
                     char (*cnv_name) [ ], int cnv_name_size,
                     char (*time_ary) [ ], int time_ary_size );
```

This procedure converts a formatted numeric time passed as a parameter to the procedure to the format HHMMSS. The numeric time must be formatted according to the numeric time format template defined in the designated convention.

The cnv_converttime function can be called as follows:

```
rslt = cnv_converttime ( vsnnum, &ct_ary, ct_ary_size, &cnv_name,
                         cnv_name_size, &time_ary, time_ary_size );
```

vsnnum is an integer passed to the procedure, containing the ccsversion number that was in effect when the time was formatted. If the input string contains alternate digits, then they are translated to their corresponding 0-9 numbers. The values allowed for vsn_num and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

ct_ary is a character array passed to the procedure, containing the formatted time.

ct_ary_size is an integer containing the number of elements in ct_ary.

cnv_name is a character array passed to the procedure, containing the name of the convention that was used to format the input time.

cnv_name_size is an integer containing the number of elements in cnv_name.

time_ary is a character array returned by the procedure, containing the converted time in the form HHMMSS.

time_ary_size is an integer containing the number of elements in time_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_converttime procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be

found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_converttime procedure are:

```
CS_ARRAY_TOO_SMALLV            CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV               CS_BADTIMEINPUTV
CS_CONVENTION_NOT_FOUNDV       CS_DATA_NOT_FOUNDV
CS_DATAOKV                     CS_FAULTV
CS_FILE_ACCESS_ERRORV          CS_SOFTERRV
```

cnv_currencyedit_double

Prototype

```
int cnv_currencyedit_double ( long double amt,
                              int precision,
                              char (*cnv_name) [ ],
                              int cnv_name_size, int *ce_len,
                              char (*ce_ary) [ ],
                              int ce_ary_size );
```

This procedure converts a double-precision integer that represents a currency value to a formatter monetary string. The editing performed by the procedure is done according to the convention specified. Refer to the MLS Guide for all the conventions and the type of currency formatting associated with each convention.

The procdure may be used to print a report with the numeric and currency formats for the Costa Rican convention (such as CRC 89.99) or for the Norwegian convention (such as NKR 89.99).

The cnv_currencyedit_double function can be called as follows:

```
rslt = cnv_currencyedit_double ( amt, precision, &cnv_name,
                                 cnv_name_size, &ce_len, &ce_ary,
                                 ce_ary_size );
```

amt is a double-precision integer passed to the procedure, containing the value of the currency.

precision is an integer passed to the procedure. It specifies the number of digits in AMT to be placed after the decimal symbol.

cnv_name is a character array passed to the procedure, containing the name of the convention used to edit the currency value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

ce_len is an integer returned to the procedure, containing the number of formatted characters in ce_ary.

ce_ary is a character array returned by the procedure, containing the formatted string.

ce_ary_size is an integer containing the number of elements in ce_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_currencyedit_double procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_currencyedit_double procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV       CS_BAD_DATA_LENV
CS_CONVENTION_NOT_FOUNDV        CS_DATAOKV
CS_FAULTV                       CS_SOFTERRV
CS_BAD_AMTV                     CS_BAD_PRECISIONV
```

cnv_currencyedittmp_double

Prototype

```
int cnv_currencyedittmp_double ( long double amt,
                                 int precision,
                                 char (*tmp_ary) [ ],
                                 int tmp_ary_size,
                                 char (*cnv_name) [ ],
                                 int cnv_name_size, int *ce_len,
                                 char (*ce_ary [ ],
                                 int ce_ary_size );
```

This procedure receives a double-precision integer and formats it to represent a currency value according to the template specified in the temp_ary parameter. ce_ary contains the formatted currency value. Refer to the *ClearPath Enterprise Servers MultiLingual System Administration, Operations, and Programming Guide*.

The cnv_currencyedittmp_double function can be called as follows:

```
rslt = cnv_currencyedittmp_double ( amt, precision, &tmp_ary,
                                    tmp_ary_size, &cnv_name,
                                    cnv_name_size, &ce_len,
                                    &ce_ary, ce_ary_size );
```

amt is a double-precision integer passed to the procedure, containing the value of the currency.

precision is an integer passed to the procedure. It specifies the number of digits in AMT to be placed after the decimal symbol.

tmp_ary is a character array, passed to the procedure, containing the formatting template used to format the currency value.

tmp_ary_size is an integer containing the number of elements in tmp_ary.

cnv_name is a character array passed to the procedure, containing the name of the convention to be used. When a caller-supplied monetary template (in tmp_ary) contains one or more control characters in simple form (control character without a symbol definition enclosed in square brackets ([ ]) following it), symbols associated with those control characters are retrieved from the monetary template in the convention specified by cnv_name.

For example, if a caller-supplied monetary template is *!N[-]CT[,:0,3]D[.]#!* and cnv_name contains "Sweden", then the local currency symbol is retrieved from the monetary template in Sweden convention and the amount 12345.67 is edited into a formatted string as *Kr.12,134.67*.

cnv_name_size is an integer containing the number of elements in cnv_name.

ce_len is an integer returned by the procedure, containing the number of formatted characters in ce_ary.

ce_ary is a character array returned by the procedure, containing the formatted currency value.

ce_ary_size is an integer containing the number of elements in ce_ary.

rslt is an integer returned by the procedure, indicating whether or not an error occurred during the cnv_currencyedittmp_double procedure. The meanings of the result values and messages are explained at the end of this section.

Possible messages returned by cnv_currencyedittmp_double are:

```
CS_BAD_ARRAY_DESCRIPTIONV          CS_BAD_DATA_LENV
CS_DATAOKV                         CS_FAULTV
CS_SOFTERRV                        CS_BAD_AMTV
CS_BAD_PRECISIONV
```

cnv_delete

Prototype

```
int cnv_delete( char (*cnv_name) [ ], int cnv_name_size );
```

This procedure deletes an existing convention from the SYSTEM/CONVENTIONS file. The convention is deleted after the next halt/load.

Note that this procedure can be used to delete only conventions that have been created by the user; system-supplied convention sets cannot be deleted.

The cnv_delete function can be called as follows:

```
rslt = cnv_delete ( &cnv_name, cnv_name_size );
```

cnv_name is a character array passed to the procedure, containing the name of the convention to be deleted from the conventions file.

cnv_name_size is an integer containing the number of elements in cnv_name.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_delete procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_delete procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV          CS_BAD_DATA_LENV
CS_CNVFILE_NOTPRESENTV             CS_CNV_NOTAVAILV
CS_DATAOKV                         CS_FAULTV
CS_FILE_ACCESS_ERRORV              CS_SOFTERRV
```

cnv_displaymodel

Prototype

```
int cnv_displaymodel( int typ, char (*cnv_name) [ ], int cnv_name_size,
                      char (*lang_name) [ ], int lang_name_size,
                      int *dm_len, char (*dm_ary) [ ], int dm_ary_size );
```

This procedure returns a display model for either the numeric date or the numeric time, whichever one is requested in the typ parameter. A display model is a format that can be displayed to show the form of the requested input. For example, YYDDMM is a display model that shows the form in which the date must be entered. The procedure creates the display model according to the convention and language specified.

The cnv_displaymodel function can be called as follows:

```
rslt = cnv_displaymodel ( typ, &cnv_name, cnv_name_size, &lang_name,
                          lang_name_size, &dm_len, &dm_ary
                          dm_ary_size );
```

typ is an integer passed to the procedure. typ indicates whether the display model is for the numeric date or for the numeric time.

Value	Macro Name	Meaning
0	CS_DATE_DISPLAYMODELV	The display model is for a numeric date.
1	CS_TIME_DISPLAYMODELV	The display model is for a numeric time.

cnv_name is a character array passed to the procedure, containing the name of the convention from which the date or time model is retrieved. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

lang_name is a character array passed to the procedure, containing the language name. It contains the language name to be used in formatting the date or time. If this

parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information on determining the valid language names and an explanation of the hierarchy.

`lang_name_size` is an integer containing the number of elements in `lang_name`.

`dm_len` is an integer returned by the procedure that contains the length of the requested display model.

`dm_ary` is a character array returned by the procedure, containing the requested display model.

`dm_ary_size` is an integer containing the number of elements in `dm_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_displaymodel` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_displaymodel` procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV                   CS_BAD_TYPE_CODEV
CS_CONVENTION_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                          CS_LANGUAGE_NOT_FOUNDV
CS_SOFTERRV
```

cnv_formatdate

Prototype

```
int cnv_formatdate( int typ, char (*date_ary) [ ], int date_ary_size,
                    char (*cnv_name) [ ], int cnv_name_size,
                    char (*lang_name) [ ], int lang_name_size, int *fd_len,
                    char (*fd_ary) [ ], int fd_ary_size );
```

This procedure formats a date, specified in `date_ary`, according to the convention specified in `cnv_name`. The formatted date is returned in `fd_ary` in the language specified by `lang_name`.

The `cnv_formatdate` function can be called as follows:

```
rslt = cnv_formatdate ( typ, &date_ary, date_ary_size, &cnv_name,
                        cnv_name_size, &lang_name, lang_name_size,
                        &fd_len, &fd_ary, fd_ary_size );
```

`typ` is an integer passed to the procedure, indicating one of the following three formats is to be used when the date is returned:

Column Heading	Macro Name	Column Heading
0	CS_LDATEV	Use the long date format

Column Heading	Macro Name	Column Heading
1	CS_SDATEV	Use the short date format
2	CS_NDATEV	Use the numeric date format

`date_ary` is a character array passed to the procedure, containing the date to be formatted. The date must be left justified and in the form YYYYMMDD. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

`date_ary_size` is an integer containing the number of elements in `date_ary`.

`cnv_name` is a character array passed to the procedure, containing the name of the convention used to edit the date value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

`lang_name` is a character array passed to the procedure, containing the language used in formatting the date. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language used. Refer to the MLS Guide for information about determining the valid language names on the system and an explanation of the hierarchy.

`lang_name_size` is an integer containing the number of elements in `lang_name`.

`fd_len` is an integer returned by the procedure, containing the length of the formatted date.

`fd_ary` is a character array returned by the procedure that contains the formatted date.

`fd_ary_size` is an integer containing the number of elements in `fd_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_formatdate` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_formatdate` procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV             CS_BADDATEINPUTV
CS_BAD_TYPE_CODEV            CS_CONVENTION_NOT_FOUNDV
CS_DATAOKV                   CS_FAULTV
CS_FIELD_TRUNCATEDV          CS_LANGUAGE_NOT_FOUNDV
CS_SOFTERRV
```

cnv_formatdatetmp

Prototype

```
int cnv_formatdatetmp( char (*date_ary) [ ], int date_ary_size,
                       char (*tmp_ary) [ ], int tmp_ary_size,
                       char (*lang_name) [ ], int lang_name_size,
                       int *fd_len, char (*fd_ary) [ ], int fd_ary_size );
```

This procedure formats a date, specified in `date_ary`, according to a template, specified in `tmp_ary`. The formatted date is returned in the language specified in `lang_name`.

The `cnv_formatdatetmp` function can be called as follows:

```
rslt = cnv_formatdatetmp ( &date_ary, date_ary_size, &tmp_ary,
                           tmp_ary_size, &lang_name, lang_name_size,
                           &fd_len, &fd_ary, fd_ary_size );
```

`date_ary` is a character array passed to the procedure, containing the date to be formatted. The date must be in the form YYYYMMDD. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

`date_ary_size` is an integer containing the number of elements in `date_ary`.

`tmp_ary` is a character array passed to the procedure, containing the template used to format the date. The template must use the control characters described in the MLS Guide.

`tmp_ary_size` is an integer containing the number of elements in `tmp_ary`.

`lang_name` is a character array passed to the procedure, containing the language to be used in formatting the date. Refer to the MLS Guide for information about determining the valid language names on the system on which the user program is running.

`lang_name_size` is an integer containing the number of elements in `lang_name`.

`fd_len` is an integer returned by the procedure, containing the length of the formatted date.

`fd_ary` is a character array returned by the procedure, containing the formatted date.

`fd_ary_size` is an integer containing the number of elements in `fd_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_formatdatetmp` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_formatdatetmp procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV             CS_BADDATEINPUTV
CS_BAD_TEMPCHARV             CS_BAD_TEMPLENV
CS_DATAOKV                   CS_FAULTV
CS_FIELD_TRUNCATEDV          CS_LANGUAGE_NOT_FOUNDV
CS_SOFTERRV
```

cnv_formattime

Prototype

```
int cnv_formattime( int typ, char (*time_ary) [ ], int time_ary_size,
                    char (*cnv_name) [ ], int cnv_name_size,
                    char (*lang_name) [ ], int lang_name_size, int *ft_len,
                    char (*ft_ary) [ ], int ft_ary_size );
```

This procedure formats a user-supplied time in the form designated by the typ parameter. The typ parameter identifies the kind of template to be retrieved from the named convention and used to format time. The formatted time is returned in the user-specified language.

The cnv_formattime function can be called as follows:

```
rslt = cnv_formattime ( typ, &time_ary, time_ary_size,  &cnv_name,
                        cnv_name_size, &lang_name, lang_name_size,
                        &ft_len, &ft_ary, ft_ary_size );
```

typ is an integer passed to the procedure, indicating one of the following formats is used when the time is returned:

Value	Macro Name	Meaning
3	CS_LTIMEV	Use the long time format.
4	CS_NTIMEV	Use the numeric time format.

time_ary is a character array passed to the procedure, containing the time to be formatted. The array is left-aligned and in the form HHMMSS. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

time_ary_size is an integer containing the number of elements in time_ary.

cnv_name is a character array passed to the procedure, containing the name of the convention used to edit the time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

lang_name is a character array passed to the procedure, containing the language used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language used. Refer to the MLS Guide for

information about determining the valid language names on the system being used and the explanation of the hierarchy.

lang_name_size is an integer containing the number of elements in lang_name.

ft_len is an integer returned by the procedure that contains the length of the formatted time.

ft_ary is a character array returned by the procedure, containing the time value formatted according to the specified convention and language.

ft_ary_size is an integer containing the number of elements in ft_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_formattime procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_formattime procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV                   CS_BAD_TEMPCHARV
CS_BADTIMEINPUTV                   CS_BAD_TYPE_CODEV
CS_CONVENTION_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                          CS_FIELD_TRUNCATEDV
CS_LANGUAGE_NOT_FOUNDV             CS_SOFTERRV
```

cnv_formattimetmp

Prototype

```
int cnv_formattimetmp( char (*time_ary) [ ], int time_ary_size,
                       char (*tmp_ary) [ ], int tmp_ary_size,
                       char (*lang_name) [ ], int lang_name_size,
                       int *ft_len, char (*ft_ary) [ ], int ft_ary_size );
```

This procedure formats a time value specified in time_ary, according to a template specified by tmp_ary. The formatted time is returned in the language specified in lang_name.

This procedure may be used with a caller-supplied template to format the time, rather than one of the existing templates.

The cnv_formattimetmp function can be called as follows:

```
rslt = cnv_formattimetmp ( &time_ary, time_ary_size, &tmp_ary,
                           tmp_ary_size, &lang_name, lang_name_size,
                           &ft_len, &ft_ary, ft_ary_size );
```

time_ary is a character array passed to the procedure, containing the time to be formatted in the form HHMMSSPPPP. The partial seconds field, PPPP, is optional. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

time_ary_size is an integer containing the number of elements in time_ary.

tmp_ary is a character array passed to the procedure in which the template used to format the time is specified. Refer to the MLS Guide for information about creating a template.

tmp_ary_size is an integer containing the number of elements in tmp_ary.

lang_name is a character array passed to the procedure, containing the language used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

lang_name_size is an integer containing the number of elements in lang_name.

ft_len is an integer returned by the procedure, containing the length of the formatted time.

ft_ary is a character array returned by the procedure, containing the time value formatted according to the specified template and language.

ft_ary_size is an integer containing the number of elements in ft_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_formattimetmp procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_formattimetmp procedure are:

```
CS_ARRAY_TOO_SMALLV      CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV         CS_BAD_TEMPCHARV
CS_BAD_TEMPLENV          CS_BADTIMEINPUTV
CS_DATAOKV               CS_FAULTV
CS_FIELD_TRUNCATED       CS_LANGUAGE_NOT_FOUNDV
CS_SOFTERRV
```

cnv_formsize

Prototype

```
int cnv_formsize( char (*cnv_name) [ ], int cnv_name_size,
                  int *lpp, int *cpl );
```

This procedure returns the lines-per-page and characters-per-line values from the specified convention. Each convention provides values for these items used with printed output.

The cnv_formsize function can be called as follows:

```
rslt = cnv_formsize ( &cnv_name, cnv_name_size, &lpp, &cpl );
```

cnv_name is a character array passed to the procedure, containing the name of the convention used to specify the printer form sizes. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

lpp is an integer returned by the procedure, containing the number of lines allowed per page for the specified convention.

cpl is an integer returned by the procedure, containing the number of characters allowed per line for the specified convention.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_formsize procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_formsize procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV          CS_CONVENTION_NOT_FOUNDV
CS_DATAOKV                         CS_FAULTV
CS_SOFTERRV
```

cnv_info

Prototype

```
int cnv_info( int typ, char (*cnv_name) [ ], int cnvname_size,
              int *info_len, float (*info_ary) [ ], int info_ary_size );
```

This procedure returns a description of the specified convention. The definition can be retrieved from memory or from the *SYSTEM/CONVENTIONS file.

The cnv_info function can be called as follows:

```
rslt = cnv_info ( typ, &cnv_name, cnv_name_size, &info_len,
                  &info_ary, info_ary_size );
```

typ is an integer passed to the procedure, indicating the source of the convention information returned. This information may represent either the convention information currently in memory or the information stored in the file *SYSTEM/CONVENTIONS. The following are valid values for typ:

Value	Macro Name	Meaning
0	CS_CNVMEM_INFOV	Convention info from memory
1	CS_CNVFILE_INFOV	Convention info from *SYSTEM/CONVENTIONS

Two copies of convention information reside on the system. One copy is in memory and is stored in the CENTRALSUPPORT library during initialization. The other copy is on disk and is stored in the *SYSTEM/CONVENTIONS file.

cnv_name is a character array that contains the name of the convention set for which a definition is requested. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

info_len is an integer returned by the procedure that contains the number of characters in info_ary.

info_ary is a float array returned by the procedure that contains the definition of the requested convention. Data stored in info_ary is stored in fields as follows:

Field Meaning	Offset
Maximum integer digits	0
Maximum decimal digits	1
Maximum International decimal digits	2
International currency notation	3
Long date template	5
Short date template	13
Numeric date template	21
Long time template	25
Numeric time template	33
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

The international currency notation field contains the international currency symbol defined for the convention. The international currency symbol is surrounded by a pair of matching delimiters that are not part of the symbol. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency symbol for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

info_ary_size is an integer containing the number of elements in info_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_info procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_info procedure are:

```
CS_ARRAY_TOO_SMALLV        CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV           CS_CONVENTION_NOT_FOUNDV
```

```
CS_DATAOKV                CS_FAULTV
CS_INCOMPLETE_DATAV       CS_SOFTERRV
```

cnv_modify

Prototype

```
int cnv_modify( char (*cnv_name) [ ], int cnv_name_size, float *modmask,
                float (*mod_ary) [ ], int mod_ary_size );
```

This procedure modifies an existing convention in the SYSTEM/CONVENTIONS file. The modified set becomes effective after the next halt/load.

Note that this procedure can be used to modify only conventions that have been created by a user; the system-supplied conventions cannot be modified.

The `cnv_modify` function can be called as follows:

```
rslt = cnv_modify ( &cnv_name, cnv_name_size, &modmask,
                    &mod_ary, mod_ary_size );
```

`cnv_name` is a character array that is passed to the procedure, containing the name of the convention to be modified. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

`modmask` is a float number that is passed to the procedure. It provides a mask that indicates the fields modified in the specified convention definition.

The bits in `modmask` are in the range 0 through 12. Each bit is associated with a unique field in `mod_ary`. If the bit is equal to 1, then the data in the corresponding field in `mod_ary` is validated and stored in the designated convention definition. Bit 0 in `modmask` corresponds to the first field in `mod_ary`, bit 1 to field 2, and so on.

`mod_ary` is a float array passed to the procedure, containing the data used to modify the convention. Data in `mod_ary` is passed in fields as follows:

Field Meaning	Offset
Maximum integer digits	0
Maximum decimal digits	1
Maximum international decimal digits	2
International currency notation	3
Long date template	5
Short date template	13
Numeric time template	33
Long time template	25
Numeric time template	33

Field Meaning	Offset
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

The international currency notation field contains the international currency symbol defined for the convention. The international currency symbol is surrounded by a pair of matching delimiters that are not part of the symbol. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency symbol for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

`mod_ary_size` is an integer containing the number of elements in `mod_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_modify` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_modify` procedure are:

```
CS_BAD_ALTFRACDIGITSV        CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_CPLV                  CS_BAD_DATA_LENV
CS_BAD_FRACDIGITSV           CS_BAD_LDATETEMPV
CS_BAD_LPPV                  CS_BAD_LTIMETEMPV
CS_BAD_MAXDIGITSV            CS_BAD_NDATETEMPV
CS_BAD_NTIMETEMPV            CS_BAD_MONTEMPV
CS_BAD_NUMTEMPV              CS_BAD_SDATETEMPV
CS_CONVENTION_NOT_FOUNDV     CS_CNVFILE_NOTPRESENTV
CS_CNV_NOTAVAILV             CS_DATAOKV
CS_FAULTV                    CS_FILE_ACCESS_ERRORV
CS_SOFTERRV
```

cnv_names

Prototype

```
int cnv_names( int *total, char (*names_ary) [ ], int names_ary_size );
```

This procedure returns a list of convention names and the total number of conventions that are available on the system, including any conventions that are user-defined.

The `cnv_names` function can be called as follows:

```
rslt = cnv_names ( &total, &names_ary, names_ary_size );
```

`total` is an integer returned by the procedure, containing the total number of conventions that reside on the system being used.

names_ary is a character array returned by the procedure, containing the convention definition names. Each name can be up to 17 characters long and is left-aligned in the field. Any parts of the array that are not used are filled with blanks.

names_ary_size is an integer containing the number of elements in names_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_names procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_names procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_DATAOKV
CS_FAULTV
```

cnv_symbols

Prototype

```
int cnv_symbols( char (*cnv_name) [ ], int cnv_name_size, int *total,
                 int (*symlen_ary) [ ], int symlen_ary_size,
                 char (*sym_ary) [ ], int sym_ary_size );
```

This procedure returns a list of numeric and monetary symbols defined for a specified convention.

symlen_ary and sym_ary are parallel arrays. Each entry in symlen_ary specifies the length of the symbol (in characters) of the corresponding entry in sym_ary. If an entry in symlen_ary is zero, it indicates that the symbol is not defined and the corresponding entry in sym_ary is filled with blanks. If an entry in symlen_ary is not zero, but the corresponding entry in sym_ary is all blanks, then the number of blanks specified by the symlen_ary entry constitutes the symbol.

The monetary and numeric symbols defined in the monetary and numeric templates for a convention are returned in fixed-length fields in symlen_ary. Each field is 12 bytes long, except where noted.

See the file *SYMBOL/INTL/C/PROPERTIES for macro name definitions corresponding to the sym_ary and sym_len_ary field offsets.

The following table shows the symbols that are returned in sym_ary and the offset of the field in which the symbol is returned for the monetary symbols:

Monetary Symbol	Offset in Bytes
International currency	0
Currency	12
Thousands separator	24
Decimal	36
Positive sign	48

Monetary Symbol	Offset in Bytes
Negative sign	60
Left enclosure	72
Right enclosure	84
Monetary grouping	168

The following table shows the symbols that are returned in `sym_ary` and the offset of the field in which the symbol is returned for the numeric symbols:

Numeric Symbol	Offset in Bytes
Thousands separator	96
Decimal	108
Positive sign	120
Negative sign	132
Left enclosure	144
Right enclosure	156
Numeric grouping	192

The monetary and numeric groupings each occupy two adjacent fields (24 bytes) in `sym_ary`. The monetary and numeric groupings, when present, are character strings consisting of unsigned integers separated by commas. The integers specify the number of digits in each group and appear exactly as they are declared in the monetary and numeric templates, including embedded commas.

The following table shows the index into the `symlen_array`, which contains the symbol lengths for the monetary and numeric symbols.

Contains Length of	Offset
International currency symbol	0
Currency symbol	1
Monetary thousands separator	2
Monetary decimal symbol	3
Monetary positive symbol	4
Monetary negative symbol	5
Monetary left enclosure symbol	6
Monetary right enclosure symbol	7
Numeric thousands separator	8
Numeric positive symbol	10
Numeric negative symbol	11

Contains Length of	Offset
Numeric left enclosure symbol	12
Numeric right enclosure symbol	13
Monetary grouping	14
Numeric grouping	15

The cnv_symbols function can be called as follows:

```
rslt = cnv_symbols ( &cnv_name, cnv_name_size,  &total, &symlen_ary,
                     symlen_ary_size, &sym_ary, sym_ary_size );
```

cnv_name is a character array passed to the procedure, containing the name of the convention used to retrieve the monetary and numeric symbols. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

total is an integer returned by the procedure, containing the total number of symbols returned.

symlen_ary is an int array returned by the procedure, containing the lengths of all symbols being returned in sym_ary.

symlen_ary_size is an integer containing the number of elements in symlen_ary.

sym_ary is a character array returned by the procedure. Each field of the array contains a symbol defined in the monetary and numeric template for the specified convention. The corresponding entry in symlen_ary contains the length of each symbol.

sym_ary_size is an integer containing the number of elements in sym_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_symbols procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_symbols procedure are:

```
CS_ARRAY_TOO_SMALLV      CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV         CS_CONVENTION_NOT_FOUNDV
CS_DATAOKV               CS_FAULTV
CS_INCOMPLETE_DATAV      CS_SOFTERRV
```

cnv_systemdatetime

Prototype

```
int cnv_systemdatetime( int typ, char (*cnv_name) [ ], int cnv_name_size,
                        char (*lang_name) [ ], int lang_name_size,
                        int *date_len, int *sdt_len, char (*sdt_ary) [ ],
                        int sdt_ary_size );
```

This procedure obtains the current system date and time and formats them according to the template retrieved from the specified convention. Date and time components are translated to the natural language designated in `lang_name`. The system computes both the date and time from the result of a single TIME(6) function call. Therefore, the possibility that the date and time are split across midnight does not exist.

The `cnv_systemdatetime` function can be called as follows:

```
rslt = cnv_systemdatetime ( typ, &cnv_name, cnv_name_size, &lang_name,
                            lang_name_size, &date_len, &sdt_len,
                            &sdt_ary, sdt_ary_size );
```

`typ` is an integer passed to the procedure, indicating one of the following formats is used when the date is returned:

Value	Macro Name	Meaning
0	CS_LDATEV	Use the long date format.
1	CS_SDATEV	Use the short date format.
2	CS_NDATEV	Use the numeric date format.
3	CS_LTIMEV	Use the long time format.
4	CS_NTIMEV	Use the numeric time format.
5	CS_LDATELTIMEV	Use the long date and long time format.
6	CS_LDATENTIMEV	Use the long date and numeric time format.
7	CS_SDATELTIMEV	Use the short date and long time format.
8	CS_SDATENTIMEV	Use the short date and numeric time format.
9	CS_NDATELTIMEV	Use the numeric date and long time format.
10	CS_NDATENTIMEV	Use the numeric date and numeric time format.

`cnv_name` is a character array passed to the procedure, containing the name of the convention used to edit the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

lang_name is a character array passed to the procedure, containing the language used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

lang_name_size is an integer containing the number of elements in lang_name.

date_len is an integer returned by the procedure specifying the length of the formatted date portion of the date and/or time. If this parameter is zero, there is no formatted date in the output. If both date and time are presented in the output, the formatted date is separated from the formatted time by a blank. The extra character is reflected in the length of the formatted date.

sdt_len is an integer returned by the procedure that specifies the total length of the formatted date and/or time being returned.

sdt_ary is a character array returned by the procedure that contains the formatted date and/or time. The recommended size for this array is 96 characters.

sdt_ary_size is an integer containing the number of elements in sdt_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the cnv_systemdatetime procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_systemdatetime procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_TYPE_CODEV            CS_CONVENTION_NOT_FOUNDV
CS_DATAOKV                   CS_FAULTV
CS_FIELD_TRUNCATEDV          CS_INCOMPLETE_DATAV
CS_LANGUAGE_NOT_FOUNDV       CS_SOFTERRV
```

cnv_systemdatetimetmp

Prototype

```
int cnv_systemdatetimetmp( char (*tmp_ary) [ ], int tmp_ary_size,
                           char (*lang_name) [ ], int lang_name_size,
                           int *dtemp_len, int *date_len, int *sdt_len,
                           char (*sdt_ary) [ ], int sdt_ary_size );
```

This procedure obtains the system date and time and formats them according to a template and language supplied by the program. The system obtains the date and time from a single TIME(6) function call to avoid the possibility of splitting the date and time across a day boundary.

The cnv_systemdatetimetmp function can be called as follows:

```
rslt = cnv_systemdatetimetmp ( &tmp_ary, tmp_ary_size, &lang_name,
                               lang_name_size, &dtemp_len, &date_len,
                               &sdt_len, &sdt_ary, sdt_ary_size );
```

`tmp_ary` is a character array passed to the procedure, containing the template, left justified. If both date and time templates are present, the date template must appear first.

`tmp_ary_size` is an integer containing the number of elements in `tmp_ary`.

`lang_name` is a character array passed to the procedure, containing the language used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

`lang_name_size` is an integer containing the number of elements in `lang_name`.

`dtemp_len` is an integer passed to the procedure, specifying the length of the date template in `tmp_ary`. If `dtemp_len` is zero, there is no date template in `tmp_ary`. If both a date and time template are specified, then the date template must appear first in `tmp_ary`.

`date_len` is an integer returned by the procedure that contains the length of the date portion of the formatted date and time. This parameter is zero if there is no formatted date in the output. If both the date and time are present in the output, the formatted date is separated from the formatted time by a blank. The extra character is reflected in the length of the formatted date.

`sdt_len` is an integer returned by the procedure that contains the length of the formatted date and time.

`sdt_ary` is a character array returned by the procedure, containing the formatted date and/or time.

`sdt_ary_size` is an integer containing the number of elements in `sdt_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_systemdatetimetmp` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `cnv_systemdatetimetmp` procedure are:

```
CS_ARRAY_TOO_SMALLV       CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV          CS_BAD_TEMPCHARV
CS_DATAOKV                CS_FAULTV
CS_FIELD_TRUNCATEDV       CS_LANGUAGE_NOT_FOUNDV
CS_SOFTERRV
```

cnv_template

Prototype

```
int cnv_template( int typ, char (*cnv_name) [ ], int cnvname_size,
                  int *tmp_len, char (*tmp_ary) [ ], int tmp_ary_size );
```

This procedure returns the requested type of formatting template retrieved from the convention that is specified in `cnv_name`.

This procedure may be used to improve the performance of programs. By retrieving and storing a template to be used in many places, the performance of a program can be improved by eliminating the calls to the CENTRALSUPPORT library.

The `cnv_template` function can be called as follows:

```
rslt = cnv_template ( typ, &cnv_name, cnv_name_size, &tmp_len,
                      &tmp_ary, tmp_ary_size );
```

`typ` is an integer passed to the procedure, specifying the type of template to be returned.

This parameter can have the following values:

Value	Macro Name	Template to be Retrieved
0	CS_LONGDATE_TEMPV	Long date
1	CS_SHORTDATE_TEMPV	Short date
2	CS_NUMDATE_TEMPV	Numeric date
3	CS_LONGTIME_TEMPV	Long time
4	CS_LONGTIME_TEMPV	Numeric time
5	CS_NUMTIME_TEMPV	Monetary template
6	CS_MONETARY_TEMPV	Numeric template

`cnv_name` is a character array passed to the procedure that contains the name of the convention that is specified. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

`cnv_name_size` is an integer containing the number of elements in `cnv_name`.

`tmp_len` is an integer returned by the procedure that contains the number of characters in `tmp_ary`.

`tmp_ary` is a character array returned by the procedure that contains the requested template.

`tmp_ary_size` is an integer containing the number of elements in `tmp_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `cnv_template` procedure. Any value other than `CS_DATAOKV`

indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the cnv_template procedure are:

```
CS_ARRAY_TOO_SMALLV             CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV                CS_BAD_TYPE_CODEV
CS_CONVENTION_NOT_FOUNDV        CS_DATAOKV
CS_FAULTV                       CS_SOFTERRV
```

cnv_validatename

Prototype

```
int cnv_validatename( char (*cnv_name) [ ], int cnv_name_size );
```

This procedure returns a value in rslt that indicates whether or not the convention name specified in cnv_name is currently defined on the system.

This procedure may be used to ensure that a convention used as an input parameter exists on the system on which the program is running.

The cnv_validatename function can be called as follows:

```
rslt = cnv_validatename ( &cnv_name, cnv_name_size );
```

cnv_name is a character array passed to the procedure, containing the name of the convention that is checked. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

cnv_name_size is an integer containing the number of elements in cnv_name.

rslt is an integer that is returned as the value of the procedure, indicating the validity of the designated convention. The possible values are:

Value	Macro Name	Meaning
1	CS_DATAOKV	The convention name is valid.
0	CS_FALSEV	The convention name is not valid.

compare_text_using_order_info

Prototype

```
int compare_text_using_order_info( char (*text1) [ ], int text1_size,
                                   int text1_start, char (*text2) [ ],
                                   int text2_size, int text2_start,
                                   int compare_len, int rltn,
                                   int ord_type, float (*order_ary) [ ],
                                   int order_ary_size );
```

This procedure compares two strings using the ordering information returned by `vsnordering_info`. One of the following methods of comparison can be chosen:

- Equivalent comparison

  This is based on the ordering sequence values of characters

- Logical comparison

  This is based on the ordering sequence values and the priority sequence values of characters.

The `compare_text_using_order_info` function can be called as follows:

```
rslt = compare_text_using_order_info ( &text1, text1_size, text1_start,
                                       &text2, text2_size, text2_start,
                                       compare_len, rltn, ord_type,
                                       &order_ary, order_ary_size );
```

`text1` is a character array containing the first string to be compared.

`text1_size` is an integer containing the number of elements in `text1`.

`text1_start` is an integer containing the starting offset in array `text1` of the string to be compared.

`text2` is a character array containing the second string to be compared.

`text2_size` is an integer containing the number of elements in `text2`.

`text2_start` is an integer containing the starting offset in array `text2` of the string to be compared.

`compare_len` is an integer passed to the procedure, designating the number of characters to be compared. If `compare_len` is larger than the number of characters in the strings, then the procedure may be comparing invalid data. The value of `compare_len` should not exceed the bounds of either the `text1` array or the `text2` array. The strings should be of equal size or padded with blanks up to `compare_len`. If all pairs of characters compare equally through the last pair, the strings are considered equal. The first pair of unequal characters to be encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering is considered to be the greater string. If, due to substitution, the strings become of unequal size, the comparison proceeds as if the shorter string had been expanded by blanks on the right to make it equal in size to the longer string.

`rltn` is an integer passed to the procedure, designating what compare relation is to be checked, as follows:

Parameter Value	Macro Name	What the Comparison Is to Determine
1	CS_CMPLSSV	Is `text1` less than `text2`?

Parameter Value	Macro Name	What the Comparison Is to Determine
2	CS_CMPLEQV	Is text1 less than or equal to text2?
3	CS_CMPEQLV	Is text1 equal to text2?
4	CS_CMPGTRV	Is text1 greater than text2?
5	CS_CMPGEQV	Is text1 not equal to text2?

ord_type is an integer passed to the procedure, specifying the type of comparison to be performed, as follows:

Parameter Value	Macro Name	Description
1	CS_EQUIVALENTV	Equivalent comparison
2	CS_LOGICALV	Logical comparison

order_ary is a float array passed to the procedure, containing the ordering information used to compare the strings. Ordering information should be obtained using the vsnordering_info procedure and used as input to this procedure. The recommended size of this array is 256 words.

order_ary_size is an integer containing the number of elements in order_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the compare_text_using_order_info procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the compare_text_using_order_info procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV      CS_BAD_DATA_LENV
CS_BAD_TEXT_PARAMV             CS_BAD_TYPE_CODEV
CS_DATAOKV                     CS_FAULTV
CS_SOFTERRV
```

get_cs_msg

Prototype

```
int get_cs_msg( int msg_num, char (*lang) [ ], int lang_size,
                char (*msg) [ ], int msg_size, int *msg_len );
```

This procedure returns error message text specified by the error number in num and in the language designated in lang_name. The desired length of the return message can be specified in the msg_len parameter. If the returned text is shorter than the length specified, the procedure pads the remaining portion of the record with blanks.

An entire message consists of the following three parts:

- The *header*, which comprises the first 80 characters of the message returned in the msg parameter. The text in the header provides the error number and a concise text description.

- The *short description*, which comprises the second 80 characters of the message returned in the msg parameter. If space is a consideration, the description of the error can be limited to the header and short description.

- The *long description* which comprises the remaining characters of the message returned in the msg parameter. The long description provides a complete explanation of the error that was returned.

Part or all of the message text can be returned. Note that the header part starts at msg[0], the short description at msg[80], and the long description at msg[160]. For example, if msg_len is specified to be equal to 200 characters, then msg contains the header message padded with blanks to 80, if necessary, followed by the short description padded with blanks to 160, if necessary, followed by 40 characters of the long description.

The requested message length should be at least 80 characters, equal to one line of text. Anything less results in an incomplete message. Recommended message lengths are 80, 160, or 999. The value of 999 causes the entire message to be returned.

This procedure may be used to retrieve the text of an error message so that it can be displayed by a program.

The get_cs_msg function can be called as follows:

```
rslt = get_cs_msg ( msg_num, &lang, lang_size, &msg, msg_size,  &msg_len );
```

msg_num is an integer passed to the procedure, specifying an error that was returned by a library procedure. The error numbers and their meanings are listed at the end of this appendix.

lang is a character array passed to the procedure, specifying the language used for the message text. If this parameter contains all blanks or zeros, the procedure uses the default language hierarchy to determine the language used. Refer to the MLS Guide for details about determining the valid language names on the system and for the explanation of the hierarchy.

lang_size is an integer containing the number of elements in lang.

msg is a character array returned by the procedure, containing the message text associated with the error number specified.

msg_size is an integer containing the number of elements in msg.

msg_len is an integer passed by reference to the procedure, specifying the maximum length of the message returned. If msg_len is equal to zero, one line of text (80 characters) is returned. If msg_len is between 1 and 79, then only a partial message is returned. msg_len should not be greater than the size of the msg array. Recommended

values for msg_len are 80, 160, or a large number, such as 999, that returns all of the message. Upon completion, the actual length of the message is returned in msg_len.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the get_cs_msg procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the get_cs_msg procedure are:

```
CS_ARRAY_TOO_SMALLV                CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV                         CS_NO_NAME_FOUNDV
CS_SOFTERRV
```

inspect_text_using_tset

Prototype

```
int inspect_text_using_tset( char (*text) [ ], int text_size,
                             int text_start, float (*tset_ary) [ ],
                             int tset_ary_size, int inspect_len,
                             int flag, int *scanned_chars );
```

This procedure compares the input text to an ALGOL-type truthset returned by the vsntruthset procedure to determine whether all the characters in the text are in the truthset.

The inspect_text_using_tset function can be called as follows:

```
rslt = inspect_text_using_tset ( &text, text_size, text_start,
                                 &tset_ary, tset_ary_size, inspect_len,
                                 flag, &scanned_chars );
```

text is a character array passed to the procedure, containing the text to be inspected.

text_size is an integer containing the number of elements in text.

text_start is an integer passed to the procedure, designating the starting offset, relative to 0, at which the search is to begin.

tset_ary is a float array passed to the procedure, containing the truthset array. The recommended tset_ary length is eight words. The truthset array should be obtained using the vsn_truthset procedure and used as input to this procedure.

tset_ary_size is an integer containing the number of elements in tset_ary.

inspect_len is an integer passed to the procedure, specifying the number of characters to be searched, beginning at text_start.

flag is an integer passed to the procedure, designating whether the procedure searches for a character that is either in the truthset or not in the truthset, as follows:

Parameter Value	Macro Name	Effect
0	CS_NOTINTSETV	Procedure searches until a character is found that is not in the truthset.
1	CS_INTSETV	Procedure searches until a character is found that is in the truthset.

`scanned_chars` is an integer returned by the procedure, containing the number of characters scanned when the criteria defined in the parameter `flag` was met.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `inspect_text_using_tset` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `inspect_text_using_tset` procedure are:

```
CS_DATAOKV                    CS_FAULTV
CS_SOFTERRV                   CS_BAD_DATALENV
CS_BAD_ARRAY_DESCRIPTIONV     CS_BAD_FLAGV
```

mcp_bound_languages

Prototype

```
int mcp_bound_languages( int *total, char (*languages_ary) [ ],
                         int *languages_ary_size );
```

This procedure returns the names of the languages that are currently bound to the MCP. For information about binding a language to the operating system, refer to the MLS Guide.

The `mcp_bound_languages` function can be called as follows:

```
rslt = mcp_bound_languages ( &total, &languages_ary,
                             &languages_ary_size );
```

`total` is an integer returned by the procedure, containing the total number of languages bound to the MCP.

`languages_ary` is a character array returned by the procedure, containing the names of the languages bound to the MCP. The maximum length of each name is 17 characters, and the names are left justified. For any name that is less than 17 characters, the field is filled on the right with blanks. The recommended array size is 340 characters, which holds 20 names.

`languages_ary_size` is an integer containing the number of elements in `languages_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `mcp_bound_languages` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages

can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the mcp_bound_languages procedure are:

```
CS_ARRAY_TOO_SMALLV      CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV               CS_FAULTV
CS_SOFTERRV
```

trans_text_using_ttable

Prototype

```
int trans_text_using_ttable( char (*source) [ ], int source_size,
                             int source_start, char (*dest) [ ],
                             int dest_size, int dest_start,
                             float (*ttable_ary) [ ],
                             int ttable_ary_size, int trans_len );
```

This procedure uses the translation table obtained using the vsntranstable procedure to translate data. The type of table used determines the type of translation done. All translation is a one-to-one mapping of the characters.

The trans_text_using_ttable function can be called as follows:

```
rslt = trans_text_using_ttable ( &source, source_size, source_start,
                                 &dest, dest_size, dest_start,
                                 &ttable_ary, ttable_ary_size,
                                 trans_len );
```

source is a character array passed to the procedure, containing the text to be translated.

source_size is an integer containing the number of elements in source.

source_start is an integer passed to the procedure. It defines the starting offset, relative to 0, at which translation is to begin.

dest is an array returned by the procedure, containing the location of the translated text. This array should normally be defined as the size of the source array.

dest_size is an integer containing the number of elements in dest.

dest_start is an integer passed to the procedure, containing the starting offset of the location where translated text is to be placed.

ttable_ary is an array passed to the procedure, containing the translation table array. The recommended size of this array is 64 words.

ttable_ary_size is an integer containing the number of elements in ttable_ary.

trans_len is an integer that contains the number of characters to be translated.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the trans_text_using_ttable procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the trans_text_using_ttable procedure are:

```
CS_ARRAY_TOO_SMALLV     CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV        CS_DATAOKV
CS_FAULTV               CS_SOFTERRV
```

validate_name_return_num

Prototype

```
int validate_name_return_num( int ccsvsn_type, char (*name_ary) [ ],
                              int name_ary_size, int *num );
```

The validate_name_return_num procedure checks a coded character set or ccsversion name to determine if it resides on the host. ccsvsn_type is used to designate whether a coded character set name or a ccsversion name is to be checked and the name itself is supplied in name_ary. The procedure returns the number of the given character set or ccsversion in num.

This procedure may be used to obtain the ccsversion number needed as a parameter in other CENTRALSUPPORT library procedures.

The validate_name_return_num function can be called as follows:

```
rslt = validate_name_return_num ( ccsvsn_type, &name_ary, name_ary_size,
                                  &num );
```

ccsvsn_type is an integer passed to the procedure, indicating whether the name specified in name_ary is a character set name or a ccsversion name. The allowable values are as follows:

Value	Macro Name	Meaning
0	CS_CHARACTER_SETV	Validate name and return a character set number.
1	CS_CCSVERSIONV	Validate name and return a ccsversion number.

name_ary is a character array passed to the procedure, containing the coded character set or ccsversion name for which a number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks, the procedure uses the hierarchy to determine the ccsversion or character set used. If there is no system default, the procedure returns an error in rslt. Maximum array size is 17 characters.

name_ary_size is an integer containing the number of elements in name_ary.

num is an integer returned by the procedure, containing the specified character set or ccsversion number.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the validate_name_return_num procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the validate_name_return_num procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV    CS_BAD_DATA_LENV
CS_BAD_TYPE_CODEV            CS_DATAOKV
CS_FAULTV                    CS_NO_NAME_FOUNDV
CS_SOFTERRV
```

validate_num_return_name

Prototype

```
int validate_num_return_name( int ccsvsn_type, int num,
                              char (*name_ary) [ ], int name_ary_size );
```

The validate_num_return_name procedure checks the number of the coded character set or ccsversion to determine if it resides on the host. ccsvsn_type is used to designate whether a coded character set name or a ccsversion name is to be checked and the name itself is supplied in name_ary. The procedure returns the name of the given coded character set or ccsversion number. Refer to the MLS Guide for the list of numbers for coded character sets and ccsversions.

This procedure may be used, for example, to display to the application user the name of the coded character set or the ccsversion being used.

The validate_num_return_name function can be called as follows:

```
rslt := validate_num_return_name ( ccsvsn_type, num, &name_ary,
                                   name_ary_size );
```

ccsvsn_type is an integer that indicates whether the number specified in num is a character set number or a ccsversion number. The following are the valid values:

Value	Macro Name	Meaning
0	CS_CHARACTER_SETV	Validate number and return a character set name.
1	CS_CCSVERSIONV	Validate number and return a ccsversion name.

num is an integer passed to the procedure, containing the number for the character set or ccsversion for which the name is being requested. The value CS_VSN_NOT_SPECIFIEDV indicates that the hierarchy should be used to determine the version number. Refer to the MLS Guide for more information about the hierarchy.

name_ary is a character array returned by the procedure, containing the character set or ccsversion name. The maximum length of the name and recommended array size is 17 characters.

name_ary_size is an integer containing the number of elements in name_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the validate_num_return_name procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the validate_num_return_name procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_TYPE_CODEV            CS_DATAOKV
CS_FAULTV                    CS_NO_NUM_FOUNDV
CS_SOFTERRV
```

vsncompare_text

Prototype

```
int vsncompare_text( int vsn_num,
                     char (*text1) [ ], int text1_size, int text1_start,
                     char (*text2) [ ], int text2_size, int text2_start,
                     int compare_len, int rltn, int ord_type );
```

This procedure compares two strings, text1 and text2, using a binary, equivalent, or logical comparison, specified in ord_type. The starting position for the comparison is specified for each string along with the type of comparison to be performed.

A binary comparison is based on the binary code values of the characters. An equivalent comparison is based on the ordering sequence values (OSVs) of the characters. A logical comparison is based on the OSVs plus the priority sequence values (PSVs). These OSVs and PSVs are retrieved from the *SYSTEM/CCSFILE based on the ccsversion.

The vsncompare_text function can be called as follows:

```
rslt = vsncompare_text ( vsn_num, &text1, text1_size, text1_start,
                         &text2, text2_size, text2_start, compare_len,
                         rltn, ord_type);
```

vsn_num is an integer passed to the procedure, containing the number of the ccsversion that is used to compare text1 and text2. The following shows the valid values:

- If the number is greater than or equal to zero, then the number designates a ccsversion.
- If the number is CS_VSN_NOT_SPECIFIEDV, the procedure uses the system default ccsversion. If the system default ccsversion is not available, the procedure returns an error in rslt.

text1 is a character array passed to the procedure, containing the first text to be compared.

text1_size is an integer containing the number of elements in text1.

text1_start is an integer passed to the procedure, containing the byte offset (0 relative) of text1 where the comparison starts.

text2 is a character array passed to the procedure, containing the second text to be compared.

text2_size is an integer containing the number of elements in text2.

text2_start is an integer passed to the procedure, containing the byte offset (0 relative) of text2 where the comparison starts.

compare_len is an integer passed to the procedure, designating the number of characters to compare. If compare_len is larger than the number of characters in the strings, then the procedure may be comparing invalid data. The value of compare_len should not exceed the bounds of either the text1 array or the text2 array. The strings should be of equal size or padded with blanks up to compare_len. If all pairs of characters compare equally through the last pair, the strings are considered equal. The first pair of unequal characters to be encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering is considered to be the greater string. If, due to substitution, the strings become of unequal size, the comparison proceeds as if the shorter string had been expanded by blanks on the right to make it equal in size to the longer string.

rltn is an integer passed to the procedure, designating what compare relation is to be checked. The valid values are the following:

Value	Macro Name	Meaning
0	CS_CMPLSSV	text1 is less than text2
1	CS_CMPLEQV	text1 is less than or equal to text2
2	CS_CMPEQLV	text1 is equal to text2
3	CS_CMPGTRV	text1 is greater than text2
4	CS_CMPGEQV	text1 is greater than or equal to text2
5	CS_CMPNEQV	text1 is not equal to text2

ord_type is an integer passed to the procedure, indicating the type of comparison to be done by the procedure. The following are the valid values:

Value	Macro Name	Meaning
0	CS_BINARYV	Perform a binary comparison
1	CS_EQUIVALENTV	Perform an equivalent comparison
2	CS_LOGICALV	Perform a logical comparison

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the vsncompare_text procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be

found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the vsncompare_text procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV        CS_BAD_DATA_LENV
CS_BAD_FLAGV                     CS_BAD_TYPE_CODEV
CS_DATAOKV                       CS_FALSEV
CS_FAULTV                        CS_FILE_ACCESS_ERRORV
CS_NO_NAME_FOUNDV                CS_SOFTERRV
```

vsnescapement

Prototype

```
int vsnescapement( int vsn_num, char (*source) [ ], int source_size,
                   int source_start, char (*dest) [ ], int dest_size,
                   int *trans_len );
```

This procedure takes the input text and rearranges it according to the escapement rules of the ccsversion. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the starting position for the text is the leftmost position in the dest parameter. If the character advance direction is negative, then the starting position for the text is the rightmost position in the dest parameter. From that point on, the character advance direction value and the character escapement direction values, in combination, control where each character should be placed in relation to the previous character.

The vsnescapement function can be called as follows:

```
rslt = vsnescapement ( vsn_num, &source, source_size, source_start,
                       &dest, dest_size, &trans_len );
```

vsn_num is an integer passed to the procedure, specifying the ccsversion to be used. The ccsversion contains the escapement rules. The following are the values allowed for vsn_num:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the MLS Guide.
CS_VSN_NOT_SPECIFIEDV	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

source is a character array passed to the procedure, containing the text to be arranged according to the escapement rules.

source_size is an integer containing the number of elements in source.

source_start is an integer passed to the procedure, specifying the starting position in source for the arranging of text by escapement rules to begin.

dest is a character array returned by the procedure, containing the resulting text after the escapement rules have been applied.

dest_size is an integer containing the number of elements in dest.

trans_len is an integer passed to the procedure, specifying the number of characters to arrange according to the escapement rules.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the vsnescapement procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the vsnescapement procedure are:

```
CS_BAD_ARRAY_DESCRIPTIONV       CS_BAD_DATA_LENV
CS_DATAOKV                      CS_FAULTV
CS_FILE_ACCESS_ERRORV           CS_NO_NUM_FOUNDV
CS_SOFTERRV
```

vsngetorderingfor_one_text

Prototype

```
int vsngetorderingfor_one_text(int vsn_num, char (*source) [ ],
                               int source_size, int source_start,
                               int itext_len, char (*dest) [ ],
                               int dest_size,
                               int dest_start, int max_osvs,
                               int total_storage, int ord_type );
```

This procedure returns the ordering information for the input text, which determines how the input text is collated. It includes the ordering and priority sequence values of the characters and any substitution of characters made when the input text is sorted. One of the following types of ordering information can be chosen:

- Equivalent ordering information

  This comprises only the ordering sequence values (OSVs)

- Logical ordering information

  This comprises the OSVs followed by the priority sequence values (PSVs)

The vsngetorderingfor_one_text function can be called as follows:

```
rslt = vsngetorderingfor_one_text ( vsn_num, &source, source_size,
                                    source_start, itext_len, &dest,
                                    dest_size, dest_start, max_osvs,
                                    total_storage, ord_type );
```

vsn_num is an integer passed to the procedure, containing the number of the ccsversion that is used.

source is a character array passed to the procedure, containing the text for which the ordering information is requested.

source_size is an integer containing the number of elements in source.

source_start is an integer passed to the procedure, containing the offset of the location where the source text is to begin.

itext_len is an integer passed to the procedure, containing the length of the source text.

dest is a character array returned the procedure, containing the ordering information for the source text.

dest_size is an integer containing the number of elements in dest.

dest_start is an integer designating the starting offset at which the result values are to be placed.

max_osvs is an integer passed to the procedure, designating the maximum number of bytes used to store the ordering sequence values.

The value of max_osvs should be at least the length of the input text, but it may need to be greater to allow for substitution. The maximum substitution length is three bytes; therefore, to allow for substitution for every character, the value of max_osvs is as follows:

```
<length of source text in bytes> * 3
```

If the number of ordering sequence values returned is less than max_osvs, then the array is padded with the OSV for blank.

total_storage is an integer passed by the procedure, defining the maximum number of bytes needed to store the complete ordering information for the text. If equivalent ordering information is requested, total_storage is equal to max_osvs. If logical ordering information is requested, space must be provided for the four-bit priority values in addition to the space allowed for the OSVs. Each OSV has one PSV and one byte can hold two PSVs. Therefore, the space allowed for PSVs is (max_osvs +1)/2 and the value of total_storage is as follows:

```
max_osvs + ((max_osvs + 1)/2)
```

When the ordering information is returned by the procedure, all the OSVs are listed first, followed by all the PSVs.

ord_type is an integer passed to the procedure, indicating the type of ordering information requested, as follows:

Parameter Value	Macro Name	Type of Information Returned
1	CS_EQUIVALENTV	Equivalent ordering information
2	CS_LOGICALV	Logical ordering information

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the vsngetorderingfor_one_text procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the vsngetorderingfor_one_text procedure are:

```
CS_ARRAY_TOO_SMALLV      CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV         CS_BAD_TEXT_PARAMV
CS_BAD_TYPE_CODEV        CS_DATAOKV
CS_FAULTV                CS_FILE_ACCESS_ERRORV
CS_NO_NUM_FOUNDV         CS_SOFTERRV
```

vsninfo

Prototype

```
int vsninfo( int vsn_num, float (*vsn_ary) [ ], int vsn_ary_size );
```

This procedure returns the following information for a designated ccsversion:

- The number of the base character set to which the ccsversion applies
- The escapement information
- The space characters used for the ccsversion
- The array sizes required by the ccsversion translation tables and sets

The vsninfo function can be called as follows:

```
rslt = vsninfo ( vsn_num, &vsn_ary, vsn_ary_size );
```

vsn_num is an integer passed to the procedure, containing the ccsversion number for which information is requested.

vsn_ary is a float array returned by the procedure, containing the ccsversion information, as follows:

Location	Information Contained
WORD 0	Base character set number
WORD 1	Text line orientation
WORD 2	Line advance direction
WORD 3	Character advance direction
WORD 4	Space characters. The first byte is the number of space characters; the actual characters follow.
WORD 5.[47:08]	Size of SPACES truthset

Location	Information Contained
WORD 5.[39:08]	Size of ALPHA truthset
WORD 5.[31:08]	Size of NUMERIC truthset
WORD 5.[23:08]	Size of PRESENTATION truthset
WORD 5.[15:08]	Size of LOWER truthset
WORD 5.[07:08]	Size of UPPER truthset
WORD 6.[47:08]	This location is unused
WORD 6.[39:08]	Size of ESCAPEMENT translation table
WORD 6.[31:08]	Size of LOWTOUP translation table
WORD 6.[23:08]	Size of UPTOLOW translation table
WORD 6.[15:08]	Size of NUMTOALTDIG translation table
WORD 6.[07:08]	Size of ALTDIGTONUM translation table
WORD 7.[47:08]	Size of OSV translation table
WORD 7.[39:08]	Size of PSV translation table
WORD 7.[31:08]	Size of SUBSTCHAR translation table
WORD 7.[23:08]	Size of SUBSTSEQ translation table
WORD 7.[15:08]	Size of SUBSTOSV translation table
WORD 7.[07:08]	Size of SUBSTPSV translation table

The recommended size of `vsn_ary` is eight words.

`vsn_ary_size` is an integer containing the number of elements in `vsn_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `vsninfo` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `vsninfo` procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV                   CS_FAULTV
CS_FILE_ACCESS_ERRORV        CS_NO_NUM_FOUNDV
CS_SOFTERRV
```

vsninspect_text

Prototype

```
int vsninspect_text( int vsn_num, char (*text) [ ], int text_size,
                     int text_start, int inspect_len, int tset_type,
                     int flag, int *scanned_chars );
```

This procedure searches a specified text for characters in or not in a requested truthset. The `scanned_chars` parameter is an integer that represents the number of characters that had been scanned when the criteria specified in the `flag` parameter were met. If `scanned_chars` is equal to `inspect_len`, then all the characters were searched, but none met the criteria. Otherwise, adding the `text_start` value to the `rslt` value gives the location of the character, from the start of the array, that met the search criteria.

The `vsninspect_text` function can be called as follows:

```
rslt = vsninspect_text ( vsn_num, &text, text_size, text_start,
                         inspect_len, tset_type, flag, &scanned_chars );
```

`vsn_num` is an integer passed to the procedure, specifying the ccsversion to be used. The ccsversion contains the rules for applying a truthset. The following are the values allowed for `vsn_num`:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the MLS Guide.
CS_VSN_NOT_SPECIFIEDV	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

`text` is a character array passed to the procedure. The array is then searched using the requested truthset.

`text_size` is an integer containing the number of elements in `text`.

`text_start` is an integer passed to the procedure, containing the byte offset (0 relative) in `text` where the search starts.

`inspect_len` is an integer passed to the procedure, specifying the number of characters to be searched beginning at `text_start`. In other words, it specifies the maximum length of the search.

`tset_type` is an integer passed to the procedure, indicating the type of truthset used for the search. The following are the values allowed for `tset_type` and their meanings:

Value	Macro Name	Meaning
12	CS_ALPHAV	Alphabetic truthset. It identifies the alphabetic characters contained in the specified ccsversion.

Value	Macro Name	Meaning
13	CS_NUMERICSV	Numeric truthset. It identifies the numeric digits contained in the specified ccsversion.
14	CS_PRESENTATIONV	Graphics truthset. It identifies the characters in the ccsversion that are graphic.
15	CS_SPACESV	Spaces truthset. It identifies the characters that represent spaces in the specified ccsversion.
16	CS_LOWERCASEV	Lowercase truthset. It identifies the characters in the ccsversion that are lowercase alphabetic.
17	CS_UPPERCASEV	Uppercase truthset. It identifies the characters in the ccsversion that are uppercase alphabetic.

Refer to the MLS Guide for more information about truthsets and their meanings.

`flag` is an integer passed to the procedure, indicating the type of search to perform. The values allowed for `flag` and their meanings are as follows:

Value	Macro Name	Meaning
0	CS_NOTINTSETV	Search text until a character is found that is not in the requested truthset.
1	CS_INTSETV	Search text until a character is found that is in the requested truthset.

`scanned_chars` is an integer returned by the procedure, containing the number of characters that had been scanned when the search criteria were met.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `vsninspect_text` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `vsninspect_text` procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_FLAGV                 CS_BAD_TYPE_CODEV
CS_DATA_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                    CS_FILE_ACCESS_ERRORV
CS_NO_NUM_FOUNDV             CS_SOFTERRV
```

vsnordering_info

Prototype

```
int vsnordering_info( int vsn_num, float (*order_ary) [ ],
                      int order_ary_size );
```

This procedure returns the ordering information for a designated ccsversion. The ordering information determines the way in which data is collated for the ccsversion. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the designated ccsversion ordering is applied to a string of text.

The table obtained with this procedure may be used with the `compare_text_using_order_info` procedure. Using this combination of procedures in place of a general call to the `vsncompare_text` procedure can improve the performance of an application program that performs a high volume of translations.

The `vsnordering_info` function can be called as follows:

```
rslt = vsnordering_info ( vsn_num, &order_ary, order_ary_size );
```

`vsn_num` is an integer passed to the procedure, containing the number of the ccsversion for which ordering information is requested.

`order_ary` is an array returned from the procedure, containing the ordering information for the ccsversion. The first two words of the array give size information in words. Word 1 is not used at this time. Word 0 contains the following information:

Word	Offset	Information Contained
`Word 0`	`[47:08]`	Size of ordering ttable
`Word 0`	`[39:08]`	Size of priority ttable
`Word 0`	`[31:08]`	Size of substitution characters truthset
`Word 0`	`[23:08]`	Size of substitution sequences array
`Word 0`	`[15:08]`	Size of substitution ordering array
`Word 0`	`[07:08]`	Size of substitution priority array

`order_ary` also contains components in the following order:

1. Ordering translation table
2. Priority translation table
3. Substitution characters truthset
4. Substitution sequences array
5. Substitution ordering array
6. Substitution priority array

The recommended size of `order_ary` is 256 words.

`order_ary_size` is an integer containing the number of elements in `order_ary`.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the vsnordering_info procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the vsnordering_info procedure are:

```
CS_ARRAY_TOO_SMALLV      CS_BAD_ARRAY_DESCRIPTIONV
CS_DATAOKV               CS_DATA_NOT_FOUNDV
CS_FAULTV                CS_FILE_ACCESS_ERRORV
CS_NO_NUM_FOUNDV         CS_SOFTERRV
```

vsntranstable

Prototype

```
int vsntranstable( int vsn_num, int ttable_type, float (*ttable_ary) [ ],
                   int ttable_ary_size );
```

This procedure returns a translation table for a designated ccsversion. The type of translation table requested depends on the task to be performed.

Translation tables can be used to perform the following tasks:

- Translate lowercase letters to uppercase letters.
- Translate uppercase letters to lowercase letters.
- Translate any digits 0 through 9 to any alternative digits (that is, one-to-one mapping of 0 through 9 to another representation for those digits).
- Translate alternative digits to 0 through 9.
- Determine the escapement direction for each character.

Refer to the MLS Guide for more information about translation tables.

The translation table from vsntranstable can be retained in a user array so that it does not have to be retrieved each time it is used. This translation table can then be passed as a parameter to trans_text_using_ttable.

The vsntranstable function can be called as follows:

```
rslt = vsntranstable ( vsn_num, ttable_type, &ttable_ary
                       ttable_ary_size );
```

vsn_num is an integer passed to the procedure, designating the number of the ccsversion from which the translation table is retrieved.

ttable_type is an integer passed to the procedure, designating the type of translation table to be returned, as follows:

Parameter Value	Macro Name	Translation Task
5	CS_NUMTOALTDIGV	Numeric characters to alternative digits
6	CS_ALTDIGTONUMV	Alternative digits to numeric characters
7	CS_LOWTOUPCASEV	Lowercase characters to uppercase characters.
8	CS_UPTOLOWCASEV	Uppercase characters to lowercase characters
9	CS_ESCMENTPERCHARV	Escapement direction for each character

`ttable_ary` is an array returned from the procedure, containing the translation table. The size of this array must be at least 64 words.

`ttable_ary_size` is an integer containing the number of elements in `ttable_ary`.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `vsntranstable` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `vsntranstable` procedure are:

```
CS_ARRAY_TOO_SMALLV             CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_TYPE_CODEV               CS_DATAOKV
CS_DATA_NOT_FOUNDV              CS_FAULTV
CS_FILE_ACCESS_ERRORV           CS_NO_NUM_FOUNDV
CS_SOFTERRV
```

vsntrans_text

Prototype

```
int vsntrans_text( int vsn_num, char (*source) [ ], int source_size,
                   int source_start, char (*dest) [ ], int dest_size,
                   int dest_start, int trans_len, int ttable_type );
```

This procedure applies a translation table, specified by `ttable_type`, to the source text and places the result into `dest`. The same array can be used for both the source and destination text.

The following are the five translation table types available:

- Numeric to alternate digits
- Alternate digits to numeric
- Lowercase to uppercase
- Uppercase to lowercase

- Escapement direction for each character

The vsntrans_text function can be called as follows:

```
rslt = vsntrans_text ( vsn_num, &source, source_size, source_start,
                       &dest, dest_size, dest_start, trans_len,
                       ttable_type );
```

vsn_num is an integer passed to the procedure, containing the number of the ccsversion used. The ccsversion contains the rules for translation of text. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for vsn_num and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

source is a character array passed to the procedure, containing the data to translate.

source_size is an integer containing the number of elements in source.

source_start is an integer passed to the procedure, containing the byte offset (0 relative) of source where translation starts.

dest is a character array returned by the procedure, containing the translated text.

dest_size is an integer containing the number of elements in dest.

dest_start is an integer returned by the procedure, indicating the starting offset location where translated text should be placed.

trans_len is an integer passed to the procedure, containing the number of characters to translate, beginning at source_start.

ttable_type is an integer that contains the type of translation requested. The allowed values for ttable_type and their meanings are as follows:

Value	Macro Name	Meaning
5	CS_NUMTOALTDIGV	Translate numbers 0 through 9 to alternate digits specified in the ccsversion.
6	CS_ALTDIGTONUMV	Translate alternate digits to numbers 0 through 9.
7	CS_LOWTOUPCASEV	Translate all characters from lowercase to uppercase.
8	CS_UPTOLOWCASEV	Translate all characters from uppercase to lowercase.

Value	Macro Name	Meaning
9	CS_ESCMENTPERCHARV	Translate a character to its escapement value.

`rslt` is an integer that is returned as the value of the procedure, indicating whether an error occurred during the `vsntrans_text` procedure. Any value other than `CS_DATAOKV` indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the `vsntrans_text` procedure are:

```
CS_ARRAY_TOO_SMALLV          CS_BAD_ARRAY_DESCRIPTIONV
CS_BAD_DATA_LENV             CS_BAD_TYPE_CODEV
CS_DATA_NOT_FOUNDV           CS_DATAOKV
CS_FAULTV                    CS_FILE_ACCESS_ERRORV
CS_NO_NUM_FOUNDV             CS_SOFTERRV
```

vsntruthset

Prototype

```
int vsntruthset( int vsn_num, int tset_type, float (*tset_ary) [ ],
                 int tset_ary_size );
```

This procedure returns a truthset for the designated ccsversion. The truthset contains the characters in a given data class for the ccsversion and are available for the following data classes:

- Alphabetic
- Numeric
- Presentation (These are the characters that can be displayed or printed on a presentation device.)
- Spaces
- Lowercase
- Uppercase

The truthset table from `vsntruthset` can be retained in a user array so that it does not have to be retrieved each time it is to be used. This truthset table can then be passed as a parameter to `inspect_text_using_tset`.

The `vsntruthset` function can be called as follows:

```
rslt = vsntruthset ( vsn_num, tset_type, &tset_ary, tset_ary_size );
```

`vsn_num` is an integer passed to the procedure, containing the number of the ccsversion from which the truthset is retrieved. The ccsversion contains the rules for translation of text. Refer to the MLS Guide for a list of ccsversion numbers.

The values allowed for vsn_num and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
CS_VSN_NOT_SPECIFIEDV	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

tset_type is an integer passed to the procedure, indicating the type of truthset to be returned, as follows:

Parameter Value	Macro Name	Truthset Returned
12	CS_ALPHAV	Alphabetic truthset
13	CS_NUMERICV	Numeric truthset
14	CS_PRESENTATIONV	Graphics truthset
15	CS_SPACESV	Spaces truthset
16	CS_LOWERCASEV	Lowercase truthset
17	CS_UPPERCASEV	Uppercase truthset

tset_ary is an array returned from the procedure, containing the truthset table. The recommended size of this array is eight words.

tset_ary_size is an integer containing the number of elements in tset_ary.

rslt is an integer that is returned as the value of the procedure, indicating whether an error occurred during the vsntruthset procedure. Any value other than CS_DATAOKV indicates an error. An explanation of the error result values and messages can be found at the end of this section. Check the procedure result whenever this procedure is used.

Possible messages returned by the vsntruthset procedure are:

```
CS_DATAOKV                    CS_FAULTV
CS_ARRAY_TOO_SMALLV           CS_BAD_ARRAY_DESCRIPTIONV
CS_NO_NUM_FOUNDV              CS_BAD_TYPE_CODEV
```

Explanation of Error Values

All of the procedures in the CENTRALSUPPORT library return integer results to indicate the success or failure of the procedure.

The range 1000 through 1999 consists of error messages caused by a software error.

Values from 2000 through 2999 contain error messages in which the caller passed invalid data to a procedure, but the CENTRALSUPPORT library was able to return some valid data.

Values from 3000 through 3999 contain error messages in which the caller passed invalid data to a CENTRALSUPPORT procedure and the CENTRALSUPPORT library was unable to return any valid data.

Values from 4000 through 4999 contain error messages in which the caller passed data for which the CENTRALSUPPORT library could find no return information. The CENTRALSUPPORT library completed the request, but no data was returned.

Internationalization Errors

Table E-1 lists the possible error results, their values, and their meaning.

Table E–1. Possible Internationalization Errors

Result	Value	Meaning
CS_ARRAY_TOO_SMALLV	3001	The array size is smaller than the length of the data it is supposed to contain.
CS_BAD_ALT_FRAC_DIGITSV	3017	The “international fractional digits” value is either missing or out of range.
CS_BAD_ARRAY_DESCRIPTIONV	3000	A parameter was incorrectly specified as less than zero.
CS_BAD_CPLV	3028	The “characters per line” value is either missing or it is out of range.
CS_BAD_DATA_LENV	3002	The length is not valid or the offset + length is greater than the size of at least one array.
CS_BAD_DATEINPUTV	3012	The input date contains illegal characters.
CS_BAD_DATESEPARATORV	3044	The date components are separated by an invalid character.
CS_BAD_DAYOFYEAR	3058	The day of year value is outside the valid range. Acceptable values are in the range 1 through 365 (1 through 366 for a leap year).
CS_BAD_DAYV	3048	The day value is outside of the valid range. Acceptable values for months January through December are as follows: 1..31, 1..28 (1..29 in a leap year), 1..31, 1..30, 1..31, 1..30, 1..31, 1..31, 1..30, 1..31, 1..30, and 1..31, respectively.
CS_BAD_FLAGV	3007	The flag specified is out of acceptable range.
CS_BAD_FRACDIGITSV	3016	The “fractional digits” value is either missing or out of range.
CS_BAD_HOURV	3051	The hour value is outside of the valid range for the 24-hour clock. Acceptable values are in the range 0 through 23.

Table E–1. Possible Internationalization Errors

Result	Value	Meaning
CS_BAD_INPUTVALV	3035	The input value did not contain digits or an expected symbol was missing.
CS_BAD_LDATETEMPV	3018	The long date template is either missing or contains invalid information.
CS_BAD_LPPV	3027	The “lines per page” value is either missing or it is out of range.
CS_BAD_LTIMETEMPV	3021	The long time template is either missing or it contains invalid information.
CS_BAD_MAXDIGITSV	3015	The “maximum digits” value is either missing or out of range.
CS_BAD_MINDIGITSV	3032	The “minimum digits” value is either missing or out of range.
CS_BAD_MINUTEV	3053	The minute value is outside the valid range. Acceptable values are in the range 0 through 59.
CS_BAD_MONTEMPV	3023	The monetary template is either missing or it contains invalid information.
CS_BAD_MONTHV	3047	The month value is outside of the valid range. Acceptable values are in the range 1 through 12.
CS_BAD_NDATETEMPV	3020	The numeric date template is either missing or it contains invalid information.
CS_BAD_NTIMETEMPV	3022	The numeric time template is either missing or contains invalid information.
CS_BAD_NUMTEMPV	3024	The numeric template is either missing or it contains invalid information.
CS_BAD_PRECISIONV	3038	The PRECISION parameter value is out of range.
CS_BAD_PSECONDV	3055	The partial second value contains invalid characters.
CS_BAD_SDATETEMPV	3019	The short date template is either missing or it contains invalid information.
CS_BAD_SECONDV	3054	The second value is outside of the valid range. Acceptable values are in the range 0 through 59.

Table E–1. Possible Internationalization Errors

Result	Value	Meaning
CS_BAD_TEMPCHARV	3011	An invalid control character was detected in the template.
CS_BAD_TEMPLENV	3030	An invalid template length value was encountered.
CS_BAD_TEXT_PARAMV	3008	For at least one text item the space allocated for OSVs in the output or the total storage allocated in the output is not large enough to hold the necessary OSVs and/or PSVs.
CS_BAD_TYPE_CODEV	3006	The type code specified is out of acceptable range.
CS_BAD_YEARV	3046	A nonzero value is required for the year component.
CS_BAD_YEARYYV	3045	The year component exceeds 2 digits.
CS_BAD12HOURV	3052	The hour value is outside of the valid range for 12-hour clock. Acceptable values are in the range 1 through 12.
CS_BADTIMEINPUTV	3013	The input time contains illegal characters.
CS_BADTIMESEPARATORV	3050	Time components are separated by an invalid character.
CS_CNV_EXISTS_ERRV	3014	An attempt was made to add a new convention with the name of an existing convention.
CS_CNV_NOTAVAILV	3036	The specified convention does not exist and cannot be retrieved, modified, or deleted.
CS_CNVFILE_NOTPRESENTV	3037	Convention file unavailable.
CS_COMPLEX_TRAN_REQV	4004	No data could be returned because complex mapping is required. This mapping is available by calling the CcsToCcs_Trans_Text_Complex procedure.
CS_CONVENTION_NOT_FOUNDV	2002	The data is not in the requested convention; it is in MYSELF.CONVENTION or the SYSTEM CONVENTION.
CS_DATA_NOT_FOUNDV	4002	The requested data was not found.
CS_DATAOKV	1	No errors; TRUE.

Table E–1. Possible Internationalization Errors

Result	Value	Meaning
CS_DEL_PERMANENT_CNV_ERRV	3040	The named convention is permanent and cannot be deleted from the SYSTEM/CONVENTIONS file.
CS_FALSEV	0	No errors; FALSE.
CS_FAULTV	1001	An unexpected fault occurred during procedure execution. This error may occur if array parameters passed to the procedure are not of the required length.
CS_FIELD_TRUNCATEDV	2003	The date or time component was too long and was truncated.
CS_FILE_ACCESS_ERRORV	1000	An error occurred while accessing the CCSFILE or the CONVENTIONS file.
CS_INCOMPLETE_CHARV	2005	An attempt was made to map one CCS to another and the source data ended in an incomplete multibyte character.
CS_INCOMPLETE_DATAV	2004	There was not enough space in the output array; therefore, only part of the date was returned.
CS_LANGUAGE_NOT_FOUNDV	2001	The data is not in the requested language; it is in MYSELF.LANGUAGE or the SYSTEM LANGUAGE.
CS_MISSING_DATE_COMPONENTV	3059	The day of year value cannot be calculated because a date component (year, month, or day) is missing.
CS_MISSING_MONTH_COMPONENTV	3057	The month value is required but missing.
CS_MISSING_RBRACKETV	3033	A right bracket ( ]) is required to end a “t” control character symbol definition list.
CS_MISSING_TCCOLONV	3034	An expected colon (:) is missing from the “t” control character in a monetary or numeric template.
CS_MUTUAL_EXCLUSIVEV	3031	A mutually exclusive combination of control characters was encountered in a monetary or numeric template.
CS_NO_CNVNAMEV	3039	A required convention name was not provided.
CS_NO_DATEINPUTV	3049	An input date is required but missing.

Table E–1. Possible Internationalization Errors

Result	Value	Meaning
CS_NO_MATCH_FOUNDV	4003	No match could be found in the code format specified for the character set or ccsversion number.
CS_NO_NAME_FOUNDV	3004	The requested name was not found.
CS_NO_NUM_FOUNDV	3003	The requested number was not found.
CS_NO_TIME_INPUTV	3056	An input time is required but missing.
CS_REQSYMBOLV	3029	A required symbol in either a monetary or numeric template is missing.
CS_SOFTERRV	1002	A software error was detected. There is an error in the procedure implementation.

Appendix F
The ASCII and EBCDIC Character Sets

The ASCII character set is listed in Table F–1.

Table F–1. ASCII Character Set

Decimal	Octal	Hex	Char	Meaning
0	000	00	NUL	Null
1	001	01	SOH	Start of Heading
2	002	02	STX	Start of Text
3	003	03	ETX	End of Text
4	004	04	EOT	End of Transmission
5	005	05	ENQ	Enquiry
6	006	06	ACK	Acknowledge
7	007	07	BEL	Bell
8	010	08	BS	Backspace
9	011	09	HT	Horizontal Tab
10	012	0A	LF	Line Feed (newline)
11	013	0B	VT	Vertical Tab
12	014	0C	FF	Form Feed
13	015	0D	CR	Carriage Return
14	016	0E	SO	Shift Out
15	017	0F	SI	Shift In
16	020	10	DLE	Data Link Escape
17	021	11	DC1	Device Control 1
18	022	12	DC2	Device Control 2
19	023	13	DC3	Device Control 3
20	024	14	DC4	Device Control 4
21	025	15	NAK	Negative Acknowledge
22	026	16	SYN	Synchronous Idle
23	027	17	ETB	End of Transmission Block

Table F–1. ASCII Character Set

Decimal	Octal	Hex	Char	Meaning
24	030	18	CAN	Cancel
25	031	19	EM	End of Medium
26	032	1A	SUBC	Substitute
27	033	1B	ESC	Escape
28	034	1C	FS	File Separator
29	035	1D	GS	Group Separator
30	036	1E	RS	Record Separator
31	037	1F	US	Unit Separator
32	040	20	SP	Space
33	041	21	!	Exclamation
34	042	22	"	Quote
35	043	23	#	Number Sign
36	044	24	$	Dollar Sign
37	045	25	&	Percent
38	046	26	&	Ampersand
39	047	27	'	Apostrophe
40	050	28	(	Left Parenthesis
41	051	29	)	Right Parenthesis
42	052	2A	*	Asterisk
43	053	2B	+	Plus
44	054	2C	,	Comma
45	055	2D	-	Minus (Hyphen)
46	056	2E	.	Period
47	057	2F	/	Slash
48	060	30	0	Zero
49	061	31	1	One
50	062	32	2	Two
51	063	33	3	Three
52	064	34	4	Four
53	065	35	5	Five
54	066	36	6	Six
55	067	37	7	Seven
56	070	38	8	Eight

Table F–1. ASCII Character Set

Decimal	Octal	Hex	Char	Meaning
57	071	39	9	Nine
58	072	3A	:	Colon
59	073	3B	;	Semicolon
60	074	3C	<	Less Than
61	075	3D	=	Equal
62	076	3E	>	Greater Than
63	077	3F	?	Question Mark
64	100	40	@	At Sign
65	101	41	A	Uppercase A
66	102	42	B	Uppercase B
67	103	43	C	Uppercase C
68	104	44	D	Uppercase D
69	105	45	E	Uppercase E
70	106	46	F	Uppercase F
71	107	47	G	Uppercase G
72	110	48	H	Uppercase H
73	111	49	I	Uppercase I
74	112	4A	L	Uppercase J
75	113	4B	K	Uppercase K
76	114	4C	L	Uppercase L
77	115	4D	M	Uppercase M
78	116	4E	N	Uppercase N
79	117	4F	O	Uppercase O
80	120	50	P	Uppercase P
81	121	51	Q	Uppercase Q
82	122	52	R	Uppercase R
83	123	53	S	Uppercase S
84	124	54	T	Uppercase T
85	125	55	U	Uppercase U
86	126	56	V	Uppercase V
87	127	57	W	Uppercase W
88	130	58	X	Uppercase X
89	131	59	Y	Uppercase Y

Table F–1. ASCII Character Set

Decimal	Octal	Hex	Char	Meaning
90	132	5A	Z	Uppercase Z
91	133	5B	[	Left Bracket
92	134	5C	\	Backslash
93	135	5D	]	Right Bracket
94	136	5E	^	Not Sign
95	137	5F	_	Underscore
96	140	60	`	Grave Accent
97	141	61	a	Lowercase a
98	142	62	b	Lowercase b
99	143	63	c	Lowercase c
100	144	64	d	Lowercase d
101	145	65	e	Lowercase e
102	146	66	f	Lowercase f
103	147	67	g	Lowercase g
104	150	68	h	Lowercase h
105	151	69	i	Lowercase i
106	152	6A	j	Lowercase j
107	153	6B	k	Lowercase k
108	154	6C	l	Lowercase l
109	155	6D	m	Lowercase m
110	156	6E	n	Lowercase n
111	157	6F	o	Lowercase o
112	160	70	p	Lowercase p
113	161	71	q	Lowercase q
114	162	72	r	Lowercase r
115	163	73	s	Lowercase s
116	164	74	t	Lowercase t
117	165	75	u	Lowercase u
118	166	76	v	Lowercase v
119	167	77	w	Lowercase w
120	170	78	x	Lowercase x
121	171	79	y	Lowercase y
122	172	7A	z	Lowercase z

Table F–1. ASCII Character Set

Decimal	Octal	Hex	Char	Meaning
123	173	7B	{	Left Brace
124	174	7C	|	Vertical Bar
125	175	7D	}	Right Brace
126	176	7E	~	Tilde
127	177	7F	DEL	Delete

The EBCDIC character set is listed in Table F–2.

Table F–2. EBCDIC Character Set

Decimal	Octal	Hex	Char	Meaning
0	000	00	NUL	Null
1	001	01	SOH	Start of Heading
2	002	02	STX	Start of Text
3	003	03	ETX	End of Text
5	005	05	HT	Horizontal Tab
7	007	07	DEL	Delete
11	013	0B	VT	Vertical Tab
12	014	0C	FF	Form Feed
13	015	0D	CR	Carriage Return
14	016	0E	SO	Shift Out
15	117	0F	SI	Shift In
16	020	10	DLE	Data Link Escape
17	021	11	DC1	Device Control 1
18	022	12	DC2	Device Control 2
19	023	13	DC3	Device Control 3
22	026	16	BS	Backspace
24	030	18	CAN	Cancel
25	031	19	EM	End of Medium
28	034	1C	FS	File Separator
29	035	1D	GS	Group Separator
30	036	1E	RS	Record Separator
31	037	1F	US	Unit Separator
37	045	25	LF	Line Feed (newline)

Table F–2. EBCDIC Character Set

Decimal	Octal	Hex	Char	Meaning
38	046	26	ETB	End of Transmission Block
39	047	27	ESC	Escape
45	055	2D	ENQ	Enquiry
46	056	2E	ACK	Acknowledge
47	057	2F	BEL	Bell
50	062	32	SYN	Synchronous Idle
55	067	37	EOT	End of Transmission
60	074	3C	DC4	Device Control 4
61	075	3D	NAK	Negative Acknowledge
63	177	3F	SUB	Substitute
64	100	40	SP	Space
74	112	4A	[	Left Bracket
75	113	4B	.	Period
76	114	4C	<	Less Than
77	115	4D	(	Left Parenthesis
78	116	4E	+	Plus
79	117	4F	!	Exclamation Point
80	120	50	&	Ampersand
90	132	5A	]	Right Bracket
91	133	5B	$	Dollar Sign
92	134	5C	*	Asterisk
93	135	5D	)	Right Parenthesis
94	136	5E	;	Semicolon
95	137	5F	^	Not Sign
96	140	60	-	Minus (Hyphen)
97	141	61	/	Slash
106	152	6A	|	Vertical Bar
107	153	6B	,	Comma
108	154	6C	%	Percent
109	155	6D	_	Underscore
110	156	6E	>	Greater Than
111	157	6F	?	Question Mark
121	171	79	`	Grave Accent

Table F–2. EBCDIC Character Set

Decimal	Octal	Hex	Char	Meaning
122	172	7A	:	Colon
123	173	7B	#	Number Sign
124	174	7C	@	At Sign
125	175	7D	'	Apostrophe
126	176	7E	=	Equal
127	177	7F	"	Quote
129	201	81	a	Lowercase a
130	202	82	b	Lowercase b
131	203	83	c	Lowercase c
132	204	84	d	Lowercase d
133	205	85	e	Lowercase e
134	206	86	f	Lowercase f
135	207	87	g	Lowercase g
136	210	88	h	Lowercase h
137	211	89	i	Lowercase i
145	221	91	j	Lowercase j
146	222	92	k	Lowercase k
147	223	93	l	Lowercase l
148	224	94	m	Lowercase m
149	225	95	n	Lowercase n
150	226	96	o	Lowercase o
151	227	97	p	Lowercase p
152	230	98	q	Lowercase q
153	231	99	r	Lowercase r
161	241	A1	~	Tilde
162	242	A2	s	Lowercase s
163	243	A3	t	Lowercase t
164	244	A4	u	Lowercase u
165	245	A5	v	Lowercase v
166	246	A6	w	Lowercase w
167	247	A7	x	Lowercase x
168	250	A8	y	Lowercase y
169	251	A9	z	Lowercase z

Table F–2. EBCDIC Character Set

Decimal	Octal	Hex	Char	Meaning
192	300	C0	{	Left Brace
193	301	C1	A	Uppercase A
194	302	C2	B	Uppercase B
195	303	C3	C	Uppercase C
196	304	C4	D	Uppercase D
197	305	C5	E	Uppercase E
198	306	C6	F	Uppercase F
199	307	C7	G	Uppercase G
200	310	C8	H	Uppercase H
201	311	C9	I	Uppercase I
208	320	D0	}	Right Brace
209	321	D1	J	Uppercase J
210	322	D2	K	Uppercase K
211	323	D3	L	Uppercase L
212	324	D4	M	Uppercase M
213	325	D5	N	Uppercase N
214	326	D6	O	Uppercase O
215	327	D7	P	Uppercase P
216	330	D8	Q	Uppercase Q
217	331	D9	R	Uppercase R
224	340	E0	\	Backslash
226	342	E2	S	Uppercase S
227	343	E3	T	Uppercase T
228	344	E4	U	Uppercase U
229	345	E5	V	Uppercase V
230	346	E6	W	Uppercase W
231	347	E7	X	Uppercase X
232	350	E8	Y	Uppercase Y
233	351	E9	Z	Uppercase Z
240	360	F0	0	Zero
241	361	F1	1	One
242	362	F2	2	Two
243	363	F3	3	Three

Table F–2. EBCDIC Character Set

Decimal	Octal	Hex	Char	Meaning
244	364	F4	4	Four
245	365	F5	5	Five
246	366	F6	6	Six
247	367	F7	7	Seven
248	370	F8	8	Eight
249	371	F9	9	Nine

Appendix G
Syntax Summary

Lexical Grammar

Tokens

token:
    constant
    identifier
    keyword
    operator
    punctuator
    string-literal

Keywords

keyword: one of

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

Identifiers

identifier:
    identifier digit
    identifier nondigit
    nondigit

digit: one of
    0 1 2 3 4 5 6 7 8 9

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

Constants

constants:
character-constant
enumeration-constant
floating-point constant
integer-constant

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

c-char:
any character in the source character set, except the single-quote ('), backslash (\), or newline character
escape-sequence

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

decimal-constant:
nonzero-digit
decimal-constant digit

digit-sequence:
digit
digit-sequence digit

enumeration-constant:
identifier

escape-sequence: one of
\ *escape-sequence-character*
\ $octal\text{-}digit_{opt}$ $octal\text{-}digit_{opt}$ *octal-digit*
\x *hexadecimal-code*

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

exponent-part:
e $sign_{opt}$ *digit-sequence*
E $sign_{opt}$ *digit-sequence*

floating-point constant:
fractional-constant $exponent\text{-}part_{opt}$ $floating\text{-}suffix_{opt}$
digit-sequence exponent-part $floating\text{-}suffix_{opt}$

floating-suffix: one of
f l F L

fractional-constant:
digit-sequence$_{opt}$ *. digit-sequence*
digit-sequence .

hexadecimal-code: one of
hexadecimal-code$_{opt}$ *hexadecimal-digit*

hexadecimal-constant:
Ox hexadecimal-digit
OX hexadecimal-digit
hexadecimal-constant hexadecimal-digit

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

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

long-suffix: one of
l L

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

octal-constant:
0
octal-constant octal-digit

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

sign: one of
+ -

unsigned-suffix: one of
u U

String Literals

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

s-char:
any character in the source character set, except the double-quote ("), backslash (\), or newline character
escape-sequence

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

Operators

operator: one of

```
[ ]  ( )  . ->

++  --  &  *  +  -  ~  !  sizeof

/  %  <<  >>  <  >  <=  >=

==  !=  ^  |  &&  ||  ?  :

=  *=  /=  %=  +=  -=  <<=  >>=

&=  ^=  |=  ,  #  ##
```

Punctuators

punctuator: one of

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

Phrase Structure Grammar

Expressions

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

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

argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*

```
assignment-expression:
       conditional-expression
       unary-expression assignment-operator assignment-expression

assignment-operator:
        =  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=

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

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

constant-expression:
       conditional-expression

equality-expression:
       relational-expression
       equality-expression == relational-expression
       equality-expression != relational-expression

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

expression:
       assignment-expression
       expression , assignment-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

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

postfix-expression:
       primary-expression
```

```
        postfix-expression [ expression ]
        postfix-expression ( argument-expression-listopt )
        postfix-expression . identifier
        postfix-expression -> identifier
        postfix-expression ++
        postfix-expression --

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

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

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

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

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

Declarations

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
fct-specifier declaration-specifiers$_{opt}$

declarator:
pointeropt direct-declarator

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

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

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

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

enum-specifier:
enum identifieropt { *enumeration-list* }
enum identifier

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

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

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

init-declarator:
declarator
declarator = *initializer*

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

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

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

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

pointer
**type-specifier-list*$_{opt}$
**type-specifier-list*$_{opt}$ *pointer*

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

fct-specifier:
inline

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

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

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

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

struct-or-union:
struct
union

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

typedef-name:
identifier

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

type-specifier:
char
const
double
enum-specifier
__far
float
int
long
__near
short
signed
struct-or-union-specifier
typedef-name
unsigned
void
volatile

type-specifier-list:
type-specifier
type-specifier-list *type-specifier*

Statements

statement:
compound-statement
control-statement
expression-statement
iteration-statement
jump-statement
labeled-statement

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

control-statement:
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
switch (*expression*) *statement*

```
declaration-list:
        declaration
        declaration-list declaration

expression-statement:
        expression_opt ;

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

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

labeled-statement:
        identifier: statement
        case constant-expression : statement
        default : state ment

statement-list:
        statement
        statement-list statement
```

External Definitions

```
external-definition:
        function-definition
        declaration
        linkage-specification

file:
        external-definition
        file external-definition

function-body:
        declaration-list_opt compound-statement

function-definition:
        declaration-specifiers_opt declarator function-body

linkage-specification:
        extern string-literal { declaration-list }
        extern string-literal declaration
```

Preprocessing Directives

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

elif-group:
#elif constant-expression newline group$_{opt}$

elif-groups:
elif-group
elif-groups elif-group

else-group:
#else *newline group*$_{opt}$

endif-line:
#endif *newline*

group:
group-part
group group-part

group-part:
pp-tokens$_{opt}$ *newline*
if-section
control-line

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

h-char:
any character in the source character set, except the newline
character and >

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

if-group:
#if *constant-expression newline group*$_{opt}$

```
        #ifdef    identifier newline group_opt
        #ifndef   identifier newline group_opt

if-section:
        if-group  elif-groups_opt else-group_opt endif-line

lparen:
        the left-parenthesis character without preceding whitespace

newline:
        the newline character

pp-number:
        digit
        . digit
        pp-number digit
        pp-number non-digit
        pp-number   e sign
        pp-number   E sign
        pp-number  .

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

#pragma binder_match = (<string1>, <string2>)

preprocessing-file:
        group_opt

preprocessing-token:
        character-constant
        header-name /* only within the #include directive */
        identifier
        keyword
        operator
        pp-number
        punctuator
        string-literal
        each non-whitespace character that cannot be one of the above

q-char:
  any character in the source character set, except the newline character
      and "

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

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

library-linkage:
(*library-attributes*)

library-attributes:
bytitle = *string-literal* , intname = *string-literal*
byfunction = *string-literal* , intname = *string-literal*
byinitiator , intname = *string-literal*

Header Summary

<alloc.h>—Memory Management

```
_far
_HEAPBADBLK
_HEAPBADHDR
_HEAPBADPTR
_HEAPBADSIZE
_HEAPEMPTY
_HEAPEND
_HEAPFULL
_heapinfo
_HEAPNILPTR
_HEAPOK
_HEAPSHMSEG
_near
NULL
size_t

int _fheapchk(void);
int _fheapset(char  ch);
int _fheapwalk(struct _heapinfo *info);
int FP_OFF(void _far *ptr);
int FP_SEG(void _far *ptr);
int _heapchk(void);
int _heapset(char ch);
int _heapwalk(struct _heapinfo *info);
int _nheapchk(void);
int _nheapset(char ch);
int _nheapwalk(struct _heapinfo *info);
size_t _fheapavl(void);
size_t _fheapmax(void);
size_t _fheapsize(void _far *ptr);
size_t _heapavl(void);
size_t _heapmax(void);
size_t _heapsize(void *ptr);
size_t _nheapavl(void);
size_t _nheapmax(void);
size_t _nheapsize(void _near *ptr);
```

```
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void _far *_fcalloc(size_t nmemb, size_t size);
void _far *fmalloc(size_t size);
void _far *_frealloc(void _far *ptr, size_t size);
void _far *MK_FP(int seg, int off);
void _far *_scalloc(size_t nmemb, size_t size);
void _far *_smalloc(size_t size);
void _far *_srealloc(void _far *ptr, size_t size);
void _ffree(void _far *ptr);
void _near *_ncalloc(size_t nmemb, size_t size);
void _near *_nmalloc(size_t size);
void _near *_nrealloc(void _near *ptr, size_t size);
void _nfree(void _near * ptr);
void _sfree(void _far * ptr);
```

<assert.h>—Diagnostics

```
NDEBUG

void assert(int expression);
```

<ctype.h>—Character Handling

```
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);
```

<errno.h>—Errors

```
E2BIG
_E2BIG
EACCES
_EACCES
EAGAIN
_EAGAIN
EASSERT
_EASSERT
EATTRERR
_EATTRERR
EATTRLISTERR
_EATTRLISTERR
EATTRO
_EATTRO
EBADF
_EBADF
EBADMERGEINPUTS
_EBADMERGEINPUTS
EBADMSG
_EBADMSG
_EBADSIG
EBADSORTRECLEN
_EBADSORTRECLEN
EBUSY
_EBUSY
ECANCELED
_ECANCELED
ECHILD
_ECHILD
EDATAERR
_EDATAERR
EDEADLK
_EDEADLK
EDOM
_EDOM
EENDOFFILEERR
_EENDOFFILEERR
EEXIST
_EEXIST
EFAULT
_EFAULT
EFBIG
_EFBIG
EFILECLOSEERR
_EFILECLOSEERR
EFILENOTAVAIL
_EFILENOTAVAIL
EFILEOPENERR
_EFILEOPENERR
```

```
EFILEPOSREQ
_EFILEPOSREO
EFILERO
_EFILERO
EFILEWO
_EFILEWO
EFTELLTOOLARGE
_EFTELLTOOLARGE
EHEAPERR
_EHEAPERR
EHEAPFULL
_EHEAPFULL
EIDRM
_EIDRM
EINPROGRESS
_EINPROGRESS
EINTR
_EINTR
EINVAL
_EINVAL
EINVALSORTVER
_EINVALSORTVER
EINVLDATTR
_EINVLDATTR
EINVLDATTRVAL
_EINVLDATTRVAL
EINVLDMODE
_EINVLDMODE
EINVLDNAME
_EINVLDNAME
EIO
_EIO
EIOERR
_EIOERR
_EIOLOGIC
EISDIR
_EISDIR
ELOOP
_ELOOP
EMFILE
_EMFILE
EMLINK
_EMLINK
EMSGSIZE
_EMSGSIZE
ENAMETOOLONG
_ENAMETOOLONG
ENFILE
_ENFILE
ENODEV
_ENODEV
```

```
ENOENT
_ENOENT
ENOEXEC
_ENOEXEC
ENOFILEPOS
_ENOFILEPOS
ENOHOST
_ENOHOST
ENOLCK
_ENOLCK
ENOMEM
_ENOMEM
ENOMSG
_ENOMSG
ENOSORTRESTART
_ENOSORTRESTART
ENOSPC
_ENOSPC
ENOSYS
_ENOSYS
ENOTBLK
_ENOTBLK
ENOTDIR
_ENOTDIR
ENOTEMPTY
_ENOTEMPTY
ENOTSUP
_ENOTSUP
ENOTTY
_ENOTTY
ENXIO
_ENXIO
EOK
_EOK
EPARITYERR
_EPARITYERR
EPERM
_EPERM
EPIPE
_EPIPE
ERANGE
_ERANGE
EROFS
_EROFS
_ESIGNALERR
_ESIGNALERR
ESPIPE
_ESPIPE
ESRCH
_ESRCH
ETIME
```

```
_ETIME
ETXTBSY
_ETXTBSY
EXDEV
_EXDEV
```

<fcntl.h>—File Control

```
F_DUPFD
F_GETFD
F_GETFL
F_GETLK
F_RDLCK
F_SETFD
F_SETFL
F_SETLK
F_SETLKW
F_UNLCK
F_WRLCK
FD_CLOEXEC
O_ACCMODE
O_APPEND
_O_ATTRIBUTES
_O_BINARY
O_CREAT
O_EXCL
O_NONBLOCK
O_RDONLY
O_RDWR
_O_TEMPORARY
O_TRUNC
O_WRONLY
struct flock

int creat(const char *path, mode_t mode);
int fcntl(int fildes, int cmd, ...);
int open(const char *path, int oflag, ...);
```

<float.h>—Floating-Point Characteristics

```
DBL_DIG
DBL_EPSILON
DBL_MANT_DIG
DBL_MAX
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN
DBL_MIN_10_EXP
DBL_MIN_EXP
FLT_DIG
FLT_EPSILON
```

```
FLT_MANT_DIG
FLT_MAX
FLT_MAX_10_EXP
FLT_MAX_EXP
FLT_MIN
FLT_MIN_10_EXP
FLT_MIN_EXP
FLT_RADIX
FLT_ROUNDS
LDBL_DIG
LDBL_EPSILON
LDBL_MANT_DIG
LDBL_MAX
LDBL_MAX_10_EXP
LDBL_MAX_EXP
LDBL_MIN
LDBL_MIN_10_EXP
LDBL_MIN_EXP
```

<grp.h>—Group Structure

```
struct group

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

<iso646.h>—Operator Synonyms

```
and
and_eq
bitand
bitor
compl
not
not_eq
or
or_eq
xor
xor_eq
```

<limits.h>—Limits of Integral Types

```
ARG_MAX
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHILD_MAX
INT_MAX
INT_MIN
LONG_MAX
LONG_MIN
```

```
MB_LEN_MAX
NAME_MAX
NGROUPS_MAX
OPEN_MAX
PATH_MAX
PIPE_BUF
_POSIX_ARG_MAX
_POSIX_CHILD_MAX
_POSIX_NAME_MAX
_POSIX_NGROUPS_MAX
_POSIX_OPEN_MAX
_POSIX_PATH_MAX
_POSIX_PIPE_BUF
_POSIX_SEM_NSEM_MAX
_POSIX_SEM_VALUE_MAX
_POSIX_SSIZE_MAX
_POSIX_STREAM_MAX
_POSIX_TZNAME_MAX
SCHAR_MAX
SCHAR_MIN
SEM_NSEMS_MAX
SEM_VALUE_MAX
SHRT_MAX
SHRT_MIN
SSIZE_MAX
STREAM_MAX
TZNAME_MAX
UCHAR_MAX
UINT_MAX
ULONG_MAX
USHRT_MAX
```

<locale.h>—Localization

```
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);
```

<math.h>—Mathematics

```
HUGE_VAL

double acos(double x);
```

```
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double ceil(double x);
double cos(double x);
double cosh(double x);
double exp(double x);
double fabs(double x);
double floor(double x);
double fmod(double y, 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 sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
```

<pwd.h>—Password Structure

```
struct passwd

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

<semaphore.h>—POSIX Semaphores

```
sem_t

int sem_close(sem_t *sem);
int sem_destroy(sem_t *sem);
int sem_getvalue(sem_t *sem, unsigned int *sval);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_unlink(const char *name);
int sem_wait(sem_t *sem);
sem_t *sem_open(const char *name, int *oflag, ...);
```

<setjmp.h>—Nonlocal Jumps

```
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savemask);
jmp_buf
sigjmp_buf
```

```
void longjmp(jmp_buf env, int val);
void siglongjmp (sigjmp_buf env, int val);
```

<siginfo.h>—Signal Generation Information

```
FPE_FLTDIV
FPE_FLTINV
FPE_FLTOVF
FPE_FLTRES
FPE_FLTSUB
FPE_FLTUND
FPE_INTDIV
FPE_INTOVF
ILL_BADSTK
ILL_COPROC
ILL_ILLADR
ILL_ILLOPC
ILL_ILLOPN
ILL_ILLTRP
ILL_PRVOPC
ILL_PRVREG
SEGV_ACCERR
SEGV_INVINX
SEGV_MAPERR
SEGV_STRPRO
SI_ASYNCIO
SI_CHLD
SI_HARDWARE
SI_MSGQ
SI_QUEUE
SI_TIMER
SI_USER
struct siginfo_t
```

<signal.h>—Signal Handling

```
pid_t
sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGALRM
SIGBUS
SIGCHLD
SIGCONT
SIGEMT
SIGFPE
SIGHUP
SIGILL
SIGINT
```

```
SIGKILL
SIGPIPE
SIGPOLL
SIGPWR
SIGQUIT
SIGSEGV
sigset_t
SIGSTOP
SIGSYS
SIGTERM
SIGTRAP
SIGTSTP
SIGTTIN
SIGTTOU
SIGUSR1
SIGUSR2
SIGWINCH
struct sigaction

int kill(pid_t pid, int sig);
int raise(int sig);
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sighold(int sig);
int sigignore(int sig);
int sigismember(const sigset_t *set, int signo);
int sigpause(int sig);
int sigpending(sigset_t *set);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int sigpush(void);
int sigrelse(int sig);
int sigsuspend(const sigset_t *sigmask);
void(*signal(int sig, void (*func)(int)))(int);
void(*sigset (int sig, void (*disp)(int))) (int);
```

<sort.h>—Sort and Merge

```
maxnumofmergeinputs
_maxnumofsortkeys
NULL
_restart_allowed
_restart_cant_until_input
_restart_err_rec
_restart_none
_restart_previous
_sort_ascending
_sort_comp_func
```

```
_sort_comp_key
_sort_descending
_sort_file
_sort_func
_sort_type_alpha
_sort_type_binary
_sort_type_double
_sort_type_invalid
_sort_type_lsep_sign
_sort_type_lsign
_sort_type_lsign_hex
_sort_type_num
_sort_type_real
_sort_type_rsep_sign
_sort_type_rsign
_sort_type_rsign_hex
_sort_type_unsign_hex
void merge(struct c_info *merge_struct, ... );
void sort(struct _c_info *sort_struct, ... );
void trans(char *outputbuf, char *inputbuf,
           int len, float *ttable_ary);
void ttable(char *ttable_descr, float *ttable_ary);
```

<stdarg.h>—Variable Arguments

```
type va_arg(va_list ap, type);
va_list
void va_start(va_list ap, argN);

void va_end(va_list ap);
```

<stddef.h>—Common Definitions

```
NULL
offsetof(type, member-designator)
ptrdiff_t
size_t
wchar_t
```

<stdio.h>—Input/Output

```
BUFSIZ
EOF
FILE
FILENAME_MAX
FOPEN_MAX
fpos_t
_IOFBF
_IOLBF
_IONBF
L_ctermid
```

```
L_cuserid
L_tmpnam
NULL
SEEK_CUR
SEEK_END
SEEK_SET
size_t
stderr
stdin
stdout
TMP_MAX

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
char *tmpnam(char *s);
FILE *fopen(const char *filename, const char *type);
FILE *freopen(const char *filename,
          const char *type, FILE *stream);
FILE *tmpfile(void);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fileno(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
int getc(FILE *stream);
int getchar(void);
int printf(const char *format, ...);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int remove(const char *filename);
int rename(const char *old, const char *new);
int scanf(const char *format, ...);
int setvbuf(FILE *stream, char *buf int type, size_t size);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int ungetc(int c, FILE *stream);
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);
            size_t nelem, FILE *stream);
```

```
long ftell(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);
size_t fwrite(const void *ptr, size_t size,
void clearerr(FILE *stream);
void perror(const char *s);
void rewind(FILE *stream);
void setbuf(FILE *stream, char *buf);
```

<stdlib.h>—General Utilities

```
div_t
EXIT_FAILURE
EXIT_SUCCESS
ldiv_t
MB_CUR_MAX
NULL
RAND_MAX
size_t
wchar_t

char *getenv(const char *name);
div_t div(int number, int denom);
double atof(const char *nptr);
double strtod(const char *nptr, char **endptr);
int abs(int i);
int atexit(void (*func)(void));
int atoi(const char *nptr);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pw, const char *s, size_t n);
int rand(void);
int system(const char *string);
int wctomb(char *s, wchar_t wchar);
ldiv_t ldiv(long int number, long int denom);
long atol(const char *nptr);
long int labs(long int i);
long strtol(const char *nptr, char **endptr int base);
size_t mbstowcs(wchar_t *pwcs, const char *, size_t n);
size_t wcstombs(char *, const wchar_t *pwcs, size_t );
unsigned long strtoul(const char *nptr,
                  char **endptr, int base;
void abort(void);
void exit(int status);
void free(void *ptr);
void qsort(void *base, size_t nel, size_t keysize
                   int(*compar)(const void *, const void *));
void srand(unsigned int seed);
void *bsearch(const void *key, const void *base,
               size_t nel, size_t keysize,
               int(*compar)(const void *, const void *));
void *calloc(size_ nmemb, size_t size);
void *malloc(size_t size);
void putenv(const char *string);
void *realloc(void *ptr, size_t size);
```

<string.h>—String Handling

```
NULL
size_t
```

```
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c);
char *strcpy(char *s1, const char *s2);
char *strerror(int errnum);
char *strncat(char *s1, const char *s2, size_t n);
char *strnchr(const char *s, int c, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
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 strcspn(const char *s1, const char *s2);
size_t strlen(const char *s);
size_t strspn(const char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
void *memchr(const void *s, int c size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
```

<sys/ipc.h>—Interprocess Communication Access Structure

```
IPC_CREAT
IPC_EXCL
IPC_NOWAIT
IPC_PRIVATE
IPC_RMID
IPC_SET
IPC_STAT
struct ipc_perm
```

<sys/sem.h>—X/Open Semaphores

```
GETALL
GETNCNT
GETPID
GETVAL
GETZCNT
SEM_UNDO
SETALL
SETVAL
struct sem
struct sembuf
struct semid_ds

int semctl(int semid, int semnum, int cmd, . . . );
```

```
int semget(key_t key, int nsems, int semflg);
int semop(int semid, struct sembuf *sops, size_t nsops);
```

<sys/shm.h>—Shared Memory Facility

```
SHM_R
SHM_RDONLY
SHM_RND
SHM_W
shmatt_t
SHMLBA
struct shmid_ds

int shmctl(int shmid, int cmd, struct_ds *buf);
int shmdt(const void *shmaddr);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

<sys/stat.h>—Data Returned by the stat Function

```
S_IFBLK
S_IFCHR
S_IFDIR
S_IFIFO
S_IFLNK
S_IFMT
S_IFREG
_S_IGUARDUSR
S_IRGRP
S_IROTH
S_IRUSR
S_IRWXG
S_IRWXO
S_IRWXU
S_ISBLK
S_ISCHR
S_ISDIR
S_ISFIFO
S_ISGID
S_ISLNK
S_ISREG
S_ISUID
_S_IUSEGUARD
S_IWGRP
S_IWOTH
S_IWUSR
S_IXGRP
S_IXOTH
S_IXUSR
```

```
struct _MCPstat
struct stat

int chmod(const char *path, mode_t mode);
int fstat(int fildes, struct stat *buf);
int _MCPfstat(int fildes, struct _MCPstat *buf);
int _MCPstat(const char *path, struct _MCPstat *buf);
int mkfifo(const char *path, mode_t mode);
int stat(const char *path, struct stat *buf);
mode_t umask(mode_t cmask);
```

<sys/times.h>—Processor Times

```
clock_t
struct tms

clock_t times (struct tms *buffer);
```

<sys/types.h>—Primitive System Data Types

```
clock_t
dev_t
gid_t
id_t
ino_t
key_t
mode_t
nlink_t
off_t
P_MYID
pid_t
size_t
time_t
uid_t
ushort
```

<sys/utsname.h>—System Name Structure

```
SYS_NMLM
struct utsname

int uname(struct utsname *name);
```

<sys/wait.h>—Declarations for Waiting

```
WCONTINUED
WEXITSTATUS
WIFCONTINUED
WIFEXITED
WIFSIGNALED
```

```
WIFSTOPPED
WNOHANG
WSTOPPED
WSTOPSIG
WTERMSIG
WUNTRACED

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

<time.h>—Date and Time

```
clock_t
CLOCKS_PER_SEC
extern char *tzname [];
extern int daylight;
extern long int timezone;
NULL
size_t
struct tm
time_t

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
clock_t clock(void);
double difftime(time_t time2, time_t time1);
size_t strftime(char *s, size_t maxsize,
       const char *format, const struct tm *timeptr);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
void tzset (void);
```

<unistd.h>—Symbolic Constants and Miscellaneous Functions

```
F_OK
NULL
_PC_CHOWN_RESTRICTED
_PC_NAME_MAX
_PC_NO_TRUNC
_PC_PATH_MAX
_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED
_POSIX_NO_TRUNC
_POSIX_SAVED_IDS
_POSIX_SEMAPHORE
R_OK
_SC_ADDRESS_MAX
_SC_ARG_MAX
_SC_CHILD_MAX
_SC_CLK_TCK
_SC_NGROUPS_MAX
_SC_OPEN_MAX
_SC_PAGESIZE
_SC_SAVED_IDS
_SC_STREAM_MAX
_SC_TZNAME_MAX
SEEK_CUR
SEEK_END
SEEK_SET
STDERR_FILENO
STDIN_FILENO
STDOUT_FILENO
W_OK
X_OK

char *getcwd(char *buf, size_t size);
char *getlogin(void);
gid_t getegid(void);
gid_t getgid(void);
int access(const char *path, int amode);
int chdir(const char *path);
int chown(const char *path, uid_t owner, gid_t group);
int close(int fildes);
int dup(int fildes);
int dup2(int fildes, int fildes2);
int execl(const char *path, const char *arg0, .../*, (char *) */);
int execle(const char *path,
           const char *arg0, .../*, (char *)0, char *const envp[]*/);
int execlp(const char *file, const char *arg0, .../*, char *)0 */);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

```
int execvp(const char *file, char *const argv[]);
int getgroups(int gidsetsize, gid_t grouplist[]);
int pause(void);
int setgid(gid_t gid);
int setpgid(pid_t pid, pid_t pgid);
int setuid(uid_t uid);
long pipe(int fildes[2]);
long int fpathconf(int fildes, int name);
long int pathconf(int fildes, int name);
long int sysconf(int name);
off_t lseek(int fildes, off_t offset, int whence);
pid_t fork(void);
pid_t getpgrp(void);
pid_t getpid(void);
pid_t getppid(void);
pid_t setsid(void);
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
uid_t geteuid(void);
uid_t getuid(void);
unsigned int alarm(unsigned int sec);
unsigned int sleep(unsigned int sec);
void _exit(int status);
```

Appendix H
Normal Compiler Output Messages

The error messages contained in this appendix are normal compiler output messages that occur when the C compiler detects an error in program code. These errors are listed in alphabetic order, with special characters, symbols, and numbers preceding the usual alphabetic sequence. At the end of each error or warning message, the file title of an included file that the message is issued from may appear between "[" and "]". The file title is added if the error or warning message is part of the included file, and only appears in the error file and remote file.

For a list of abnormal compiler output error messages, see Appendix J, "Abnormal Compiler Output Messages," in this manual.

`$FARHEAP must be set to #include <sys/shm.h>.`

The <sys/shm.h> header has been included and the $FARHEAP compiler control option is not set. The <sys/shm.h> header is required to invoke the X/Open defined shared memory feature. This feature also requires that the $FARHEAP compiler control option be set.

`$FARHEAP(INSTALLMEMORY) must be reset with $FARHEAP (SET ONE).`

If $FARHEAP(ONE) is set, $FARHEAP(INSTALLMEMORY) must be reset. Either reset $FARHEAP(ONE) or reset $FARHEAP(INSTALLMEMORY).

`$FARHEAP(RESIZEMEMORY) must be rest with $FARHEAP (SET ONE).`

If $FARHEAP(ONE) is set, $FARHEAP(RESIZEMEMORY) must be reset. Either reset $FARHEAP(ONE) or reset $FARHEAP(RESIZEMEMORY).

`$MEMORY_MODEL must be TINY or SMALL with $FARHEAP (SET ONE).`

If $FARHEAP(ONE) is set, $MEMORY_MODEL must be set to TINY or SMALL, not LARGE or HUGE. Either reset $FARHEAP(ONE) or change $MEMORY_MODEL to TINY or SMALL.

`$MERGE option may only occur once.`

The MERGE compiler control option may only be specified once for any given compile.

`$ option name is limited to 31 characters.`

A user defined compiler control option may be a maximum of 31 characters. The extra characters are truncated.

`$ option not allowed inside INITIALCCI file.`

The compiler control option in error is not allowed inside an INITIALCCI file.

`"8" and "9" may not appear in an octal constant.`

An octal constant (an integer that started with 0 or a \0... in a character or string constant) was scanned that contained the digit 8 or 9. Octal constants must only contain the digits 0 through 7. A common mistake is to place a leading 0 in front of a decimal integer constant, making it an octal constant. Another mistake is to forget the "x" when forming a hexadecimal constant (`089` instead of `0x89`). For more information on integer, character, and string constants, see Section 1, "Overview of the C Language."

`",..." not permitted in this construct.`

A function that is declared with a linkage specification may not have a variable length argument list. For more information on function declarators, see Section 3, "Declarations," and for more information on linkages, see Appendix B, "Interface to the Library Facility," both in this manual.

`"inline" can be used only when defining a function.`

The storage class `inline` was used in something other than a function definition. For more information on storage class specifiers, see Section 3, "Declarations," in this manual.

`"void" type not allowed.`

Objects cannot be declared with type `void`. For more information on void type, see Section 2, "Types," in this manual.

`"<name>" is not an enumeration tag.`

When the <name> is used after the `enum` keyword, the name must first be declared as an enumeration tag. To correct this error, add the enumeration tag declaration or change the name to match an already declared enumeration tag. For more information on enumeration types, see Section 2, "Types," in this manual.

`<name> cannot be the subject of a #define directive.`

Predefined macro names cannot be the subject of a `#define` or `#undef` directive. For a complete list of the predefined macro names, refer to Section 8, "The C Preprocessor," in this manual.

`<name> cannot be the subject of a #undef directive.`

Predefined macro names cannot be the subject of a `#define` or `#undef` directive. For a complete list of the predefined macro names, refer to Section 8, "The C Preprocessor," in this manual.

`<number> actual arguments does not match <number> formals.`

The call did not supply the same number of arguments as the function was declared or defined to have. This can also occur if the function was implicitly defined or, if its declaration did not specify the number of arguments, this call had a different number than a prior call. To correct this error, the number of arguments should be made the same or the function should be declared as taking a variable number of arguments. For more information on how to declare variable number of arguments, see Section 3, "Declarations," in this manual.

`<sequence number> Invalid OPTION Mnemonic: <name>.`

The source file specifies an invalid mnemonic for the OPTION task attribute. A correct mnemonic should be specified. Refer to the *Work Flow Administration and Programming Guide* for a list of valid mnemonics for the OPTION task attribute.#include file not found.

An attempt was made to `#include` a file that could not be found. Check to be sure the file name is spelled correctly and the search path, if any, is set up correctly. This error is affected by the ANSI compiler control option. If ANSI is set, this error is generated; otherwise, the compile will wait on a "NO FILE". For more information on file inclusion, see Section 8, "The C Preprocessor," in this manual. For more information on the SEARCH compiler control option, see Section 10, "Compiler Control Options," in this manual.

`__heap_t allowed only with MEMORY_MODEL = TINY or SMALL.`

It is not possible to pass the heap to an external function if the program is compiled with a MEMORY_MODEL other than TINY or SMALL. For more information on hidden parameters, see Appendix B, "Interface to the Library Facility," and for more information on MEMORY_MODEL, see Section 10, "Compiler Control Options," both in this manual.

`A declaration must declare a name, a tag, or members of an enumeration.`

A declaration must declare something, such as a typedef, variable, or function name; a structure, union, or enumeration tag; or a name for an enumeration value. For more information on declarations, see Section 3, "Declarations," in this manual.

`A FAULT OCCURRED WHILE SCANNING THE DISPLAY FORM TITLE - <file name>.`

The file name specified is invalid. Correct the file name and recompile.

`A NAME NODE WAS EXPECTED BETWEEN OR AFTER SLASHES IN THE FILE NAME - <file nam e>.`

The file name is invalid because it contains two consecutive slashes or it ends with a slash. Correct the file name and recompile.

`A name with two leading underscores ("<name>") must not be used.`

Names that start with two underscores (__) should only be generated by the C compiler. Do not declare names that start with two underscores. For more information on identifiers, see Section 1, "Overview of the C Language," in this manual.

`A NON-ALPHANUMERIC CHARACTER WAS FOUND IN THE FAMILYNAME FOR THE FILE NAME <fi le name>.`

A family name must contain only uppercase alphanumeric characters. Correct the file name and recompile.

`A NULL QUOTED STRING IS ILLEGAL AS AN IDENTIFIER IN THE FILE NAME - <file name >.`

The file name may contain invalid characters. An identifier that represents one level of the file title or a usercode can include hyphens and underscores without quotes. If other non alphanumeric characters are included, then the identifier must be enclosed in quotes. Correct the file name and recompile.

`A RIGHT PARENTHESIS WAS EXPECTED AFTER THE USERCODE FOR THE FILE NAME - <file name>.`

An identifier that represents a usercode should be enclosed in parentheses; it may contain hyphens or underscores without quotes. If other non alphanumeric characters are included, then quotes are required. Correct the file name and recompile.

`A SLASH WAS EXPECTED BETWEEN SUCCESSIVE IDENTIFIERS IN THE FILE NAME - <file n ame>.`

The file name may contain invalid characters. An identifier that represents one level of the file title or a usercode can include hyphens and underscores without quotes. If other non alphanumeric characters are included, then quotes are required. Correct the file name and recompile.

`A specifier or type is required.`

An argument for an old style function declaration has been declared in the argument section without a type or storage class specifier. For more information on old style function declaration, see Section 7, "Functions," in this manual.

`A typedef having function type cannot be used in the definition (with body) of a function.`

In the definition of a function, the definition cannot inherit the function type from a typedef. For example, in the following code fragment, it is illegal for `f` to inherit its function type `int` from typedef `F`:

```
typedef int F(void);
F f {/* ...*/}   /* Syntax/Constraint Error */
```

This error can occur also if the wrong typedef identifier is used in declaring valid enums, structs, or unions.

`A USERCODE WAS EXPECTED AFTER THE LEFT PARENTHESIS IN THE FILE NAME - <file name>.`

An identifier that represents a usercode should be enclosed in parentheses; it may contain hyphens or underscores without quotes. If other non alphanumeric characters are included, then quotes are required. Correct the file name and recompile.

`Addressable expression expected.`

The context requires an addressable expression, but the operand is not addressable. Addressable expressions are called lvalues. The result of a function call or a cast is not an lvalue. For more information on lvalues, see Section 5, “Expressions and Operators,” in this manual.

`Addressable expression expected. Note: Objects whose address has not been taken in the source code may not be addressed in a TADS session.`

Objects that are addressed are mapped differently in memory than those that are not addressed. If an object is not addressed when a program is compiled, it may not be addressed during a TADS debug session on that program.

`An array must be of length 1 or greater.`

An array may not be declared with less than one element. For more information on array declarators, see Section 3, “Declarations,” in this manual.

`An empty declaration is non-standard.`

An empty declaration was detected. This is usually caused by having two semicolons (**;**) in a row before the first statement or by following the closing curly-brace (}) of a function body with a semicolon. For more information on declaration syntax, see Section 3, “Declarations,” in this manual.

`AN IDENTIFIER THAT CONTAINS A NON-ALPHANUMERIC CHARACTER WAS NOT ENCLOSED IN Q UOTATION MARKS FOR THE FILE NAME - <file name>.`

The file name may contain invalid characters. An identifier that represents one level of the file title or a usercode can include hyphens and underscores without quotes. If other non alphanumeric characters are included, then quotes are required. Correct the file name and recompile.

`Any macro definition removed by #undef will be unavailable for use in a TADS s ession. This warning will not be repeated for this compilation unit.`

This warning is issued by the compiler if the TADS option is enabled and a `#undef` directive is encountered.

`Application of the OPTIMIZE compiler control option may cause unexpected respo nses: statements and their breakpoints may be moved out of order or even remov ed.`

This warning is issued during a TADS debug session if the program being debugged was compiled with the OPTIMIZE option enabled.

`Argument not castable (void, struct, or union?).`

The operand to the cast operator does not have scalar type. Casting a structure, union, function, or void type is not allowed. It is sometimes possible to take the address of the operand, cast it to be a pointer to a different structure, union, or function, and then dereference the cast. For more information on cast operators, see Section 5, "Expressions and Operators," in this manual.

`Arithmetic type expected.`

One of the operands in an arithmetic expression does not have arithmetic type. For an operator description and more information on type and class of expressions, see Section 5, "Expressions and Operators," in this manual.

`Array of reference not allowed.`

The notation used to declare a reference parameter is similar to that used for declaring a pointer. It is therefore syntactically possible to declare a pointer to a reference, an array of references, a reference to a reference, or a function returning a reference. However, such declarations are not semantically meaningful and are, hence, flagged as errors. For more information on reference parameters, see Appendix A, "Interface to the Library Facility," in this manual.

`As of SSR 44.2, the macro _ASERIES_SOURCE will automatically be set to the cur rent compiler level. This affects the names included in headers, cross-bindin g, and functionality. Define _ASERIES_SOURCE to be between 410 and 422 to ret ain current behavior. See C Manual Volume 2 for more information.`

Starting in SSR 43.1, this warning is no longer generated. The macro _ASERIES_SOURCE will not be set to the current compiler level in SSR 44.2 or later releases.

`As of SSR 44.2, the macro _ASERIES_SOURCE will automatically be set to the current compiler level. This affects the names included in headers, cross-binding, and functionality. Define _ASERIES_SOURCE to 409 or less to retain current behavior. See C Manual Volume 2 for more information.`

Starting in SSR 43.1, this warning is no longer generated. The macro _ASERIES_SOURCE will not be set to the current compiler level in SSR 44.2 or later releases.

`Assignment of constant being done, not comparison.`

The test-expression of an `if` statement or the condition expression of a `while`, `do`, or `for` statement consists of only an assignment (=) with a constant value. It is more likely that a comparison (==) was desired. For more information on assignment and equality operators, see Section 5, "Expressions and Operators," in this manual.

`Assignment target is missing a qualifier of the assignment value.`

If a pointer is declared to point to a type that is qualified with either the `const` or `volatile` qualifier, the pointer may only be assigned to a similarly qualified pointer. For more information about type qualifiers, see Section 3, "Declarations," in this manual.

`Assignment to const qualified type illegal.`

The target of the assignment has a constant qualified type and cannot be assigned. The constant qualification should be removed or the assignment should be removed. For more information on `const` type qualifiers, see Section 3, "Declarations," in this manual.

`Assignment to this object is ambiguous with another access to it in the expression (<number>).`

C does not specify the order of evaluation for other operands of most operators, therefore the embedded assignment and the access to the target of the assignment can be evaluated in either order. The value of the expression can change depending on the level of optimization done by the compiler. The embedded assignment should be moved out as a separate statement or use the comma (`,`), if (`?:`), logical-and (`&&`), or logical-or (`||`) operator to control the order of evaluation. For more information on these operators and on gray code, see Section 5, "Expressions and Operators," in this manual.

The number in parentheses is used to distinguish multiple conflicts within the same expression.

`Bit field must be integral.`

A bit field is a set of contiguous bits within a word that is treated as an integer. A bit field that is not declared to have integral type is flagged as an error. For more information on bit fields, see Section 2, "Types," in this manual.

`Bit field must have type "int" (signed or unsigned).`

A bit field is a set of contiguous bits within a word that is treated as an integer. If the ANSI option is enabled, a bit field that is not declared to have a type of `signed int` or `unsigned int` is flagged as an error. For more information on bit fields, see Section 2, "Types," in this manual.

`Bitwise operation on signed type may be undefined.`

The bitwise operators shift left (<<), shift right (>>), and negation (~) may give undesirable results when performed on signed integral values. For more information on integer types, see Appendix C, "Porting C Applications from Other Systems," in this manual. For more information on the UNSIGNED and SIGNEDFIELD compiler control options, see Section 10, "Compiler Control Options," in this manual.

`Block linkage incorrect.`

An internal procedure to the compiler has received data that is incorrect. This is a compiler error.

`Bound codefiles are not TADS capable.`

The source file specified both TADS and BINDINFO, LEVEL, or LIBRARY compiler control option. This warning indicates that a TADS session can be initiated on an object file with bind information (BINDINFO, LEVEL, or LIBRARY), but it cannot be initiated on an object file generated by the BINDER.

`Break is only allowed in do, for, and while loops or switch statements.`

The `break` statement can only be used inside the statement controlled by a `do`, `for`, `while`, or `switch` statement. For more information on statements, see Section 6, "Statements," in this manual.

`Call on macro setjmp or macro sigsetjmp is too complex.`

For strict conformance to the ANSI C standard, an invocation of the `setjmp` macro may appear only in a very limited set of contexts. This message may also be affected by the compiler control options ANSI and LINT(SETJMP). The macro `sigsetjmp` has the same restrictions. For more information on compiler control options, see Section 10, "Compiler Control Options," in this manual.

`Cannot be referenced as a subordinated option.`

The current compiler control option is not a class option with subordinate sub options. It cannot be referenced as a subordinate option.

`Cannot be used within the OPTION(...) clause.`

Only user defined options can be specified with the OPTION compiler control option. They may only be Boolean options.

`Cannot use __file_t type when compiling for POSIX.`

The source file has defined `_POSIX_SOURCE` or has defined `_ASERIES_SOURCE` to a value 423 or greater. This is known as compiling code for POSIX. When you are compiling code for POSIX, files can be referenced only by their File Descriptor number (FD). The File Descriptor number, which is the `_file_no` member in the FILE structure, can be passed as a call-by-value integer. It cannot be converted into a call-by-reference FILE.

If necessary to pass a file, the FD can be passed. The ALGOL entry point must be changed to have a call-by-value integer formal parameter and to use the POSIX I/O interfaces defined in file SYMBOL/POSIX/ALGOL/PROPERTIES.

`Case label is only allowed in switch statements.`

The case label syntax can only be used inside the statement controlled by the `switch` statement. The label can either be deleted or it can be placed within a statement controlled by the `switch` statement. For more information on the `switch` statement, see Section 6, “Statements,” in this manual.

`Cast permitted to scalar types only.`

An illegal type cast was attempted. In a cast expression, both the type and the expression must have scalar type. For more information on the cast operator, refer to Section 5, “Expressions and Operators,” in this manual.

`Casting between integer and pointer is not portable.`

Casting between an integer and a pointer may produce different results on different machines. This warning indicates that the indicated cast may not be portable to another machine. For more information on the cast operator, refer to Section 5, “Expressions and Operators,” and for information on suppressing this warning, refer to the “PORT Option” in Section 10 in this manual.

`CODE VERSION MISMATCH in the following software:`

The version of the listed software does not match the version of the software currently running. One or the other is in error and unknown results could happen if the versions do not match.

`Comment open at end of file.`

A comment was opened (`/*...`), but was not closed (`...*/`) before the end of the file. The comment must be closed in the same file that opened it.

`COMPILER ERROR <sequence number> (<internal number>):`
`LIBRARY_INFO table size of <number> exceeded.`

The source file contains too many library declarations, too many procedures declared as library entry points, or too many characters in strings associated with the libraries (titles, intnames, and so forth). Either reduce the above limits or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
LIBRARY_PARAMS table size of <number> exceeded at <function name>.
```

The source file contains too many parameters in the procedures declared as library entry points. Either reduce the number of parameters or the complexity of the parameters.

```
COMPILER ERROR <sequence number> (<internal number>):
LIBRARY template array size of <number> exceeded for library at Address Couple
 = (<stack address>).
```

The source file contains too many library declarations, too many procedures declared as library entry points, or too many characters in strings associated with the libraries (titles, intnames, and so forth). Either reduce the above limits or split the source file into smaller files.

```
COMPILER ERROR <sequence number>(<internal number>):
Compiler limit exceeded: max_cStructUnionNameSpace.
```

The source file contains a structure or union declaration with too many nested struct or union specifiers. Reduce the number of nested struct or union specifiers by combining two or more struct or union specifiers.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of codeptr_LTS exceeded.
```

The result code file is to be bound, but the source file contains too many procedures passed as parameters or whose address are taken. Either reduce the number of above or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of declared_temp entries (<number>) exceeded.
```

The statement, possibly with its adjoining statements, requires too many temporaries to be compiled. Simplify the statement by moving subexpressions out as separate statements.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of formal calls exceeded.
```

The resulting code file is to be bound, but the source file contains too many calls on procedures passed as parameters or calls on pointers to procedures. Either reduce the number of such calls or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of info entries (<number>) exceeded.
```

The source file has too many identifiers to be compiled. Either reduce the number of identifiers or split the file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of label table entries exceeded.
```

The generated code has too many branches. Either reduce the complexity or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of real_set entries (<number>) exceeded.
```

The statement, possibly with its adjoining statements, requires too many temporaries to be compiled. Simplify the statement by moving subexpressions out as separate statements.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of str table entries exceeded.
```

Too many string constants exist in the source file. Reduce the number of string constants by combining string constants together or splitting the source file.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of symbol entries (<number>) exceeded.
```

The source file has too many characters in its identifiers to be compiled. Either reduce the length of the identifiers, reduce the number of identifiers, or split the file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of twig table entries exceeded.
```

The source file contains too many occurrences of variables, constants, or operators. Either reduce the number of the above items or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Maximum number of txt table entries exceeded.
```

Too many characters exist in all the string constants in source file. Reduce the number of characters by removing string constants, making the strings shorter, or splitting the source file.

```
COMPILER ERROR <sequence number> (<internal number>):
Too many library entry points (<number>) in library at Address Couple =
(<stack address>).
```

The source file contains too many procedures declared as library entry points. Either reduce the number or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Too many setjmp calls in one function.
```

The function contains too many calls on the `setjmp` macro to be compiled. Either reduce the number of calls or split the function into smaller functions.

```
COMPILER ERROR <sequence number> (<internal number>):
Too much statistic information needed.
```

The STATISTICS compiler control option was set for too many procedures and/or blocks to be compiled. Either reduce the number of procedures and/or blocks that have STATISTICS set or split the source file into smaller files.

```
COMPILER ERROR <sequence number> (<internal number>):
Type Stack Overflow -- Probably a reentrant expression tree.
```

The source file contains an expression that is too complex. The expression should be split into smaller expressions, each as a separate statement.

```
Compiler option $BINDINFO ignored because "main" is not declared.
```

The compiler option BINDINFO may only be set for C source files that contain the "`main`" function. It is normally unnecessary to specify this compiler option for C source files because the C compiler chooses the correct setting automatically.

```
Compiler option $LIBRARY ignored because "main" is declared.
```

The compiler option LIBRARY may only be set for C source files that do not contain the "`main`" function. It is normally unnecessary to specify this compiler option for C source files because the C compiler chooses the correct setting automatically.

```
Compiler option FARHEAP must be set to use "__near" or "__far" type qualifier.
```

The FARHEAP compiler control option must be set to use the type qualifiers or the macros and functions declared in the header `<alloc.h>`.

```
Concatenated string literals must have same type.
```

Adjacent string literals are concatenated by the compiler. However, they must be of the same type. For example, a normal string constant may not be concatenated with a wide string constant. For more information on string constants, refer to Section 1, "Overview of the C Language," in this manual.

`Conflicting type declaration.`

The same type qualifier (`volatile` or `const`) was given twice, the same type adjective (`short`, `long`, `signed`, `unsigned`) was given twice, or the type was given twice. The duplicate information should be deleted. For more information on type specifiers and qualifiers, refer to Section 3, "Declarations," in this manual.

`Conflicts with prior $BINDER_MATCH option with same first string but different second string.`

A $BINDER_MATCH option or #pragma binder_match was already processed that had the same first string (the "name" string) but a different second string (the "value" string). Both "value" strings must be identical or the "name" strings must be different. Refer to Section 10, "Compiler Control Options," for information on the BINDER_MATCH option.

`Constant expression expected.`

The indicated expression cannot be evaluated at compile time. Either change the expression so that it can be evaluated at compile time or initialize the variable with an assignment statement instead of with an initializer on the declaration. For more information on constant expressions, refer to Section 5, "Expressions and Operators," in this manual.

`Constant integer expression expected.`

The indicated operand is not a constant expression or does not have integral type. The operator, however, requires a constant integral expression. For more information on expression types and constant expressions, refer to Section 5, "Expressions and Operators," in this manual.

`Constant integer non-negative expression expected.`

The indicated operand is not a constant expression or does not have integral type or has a negative value. The context, however, requires a non-negative constant integral expression. For more information on expression types and constant expressions, refer to Section 5, "Expressions and Operators," in this manual.

`Constant too large.`

A numeric constant was scanned that is larger than can be represented internally. For more information about the range of C variables, refer to Section 2, "Types." If the number is a single precision floating point, use of double precision may provide enough range. For more information on floating-point types, refer to Section 2, "Types," and for more information on the DBLTOSNGL compiler control option, refer to Section 10, "Compiler Control Options," in this manual.

`Continue is only allowed in do, for, and while loops.`

The `continue` statement can only be used inside the statement controlled by a `do`, `for`, and `while` statement. For more information on the `continue` statement, refer to Section 6, "Statements," in this manual.

`Data Type of Twig is Unknown.`

An internal procedure to the compiler has received an invalid parameter. This is a compiler error.

`DELETE or VOIDT may only be used on $ cards from the CARD file.`

The DELETE or VOIDT compiler control options may only appear in the primary input file (CARD). All other occurrences are considered errors.

`Duplicate case labels are not allowed.`

The value of this case label is equal to the value of a previous case label that occurs within the same `switch` statement. All case labels for the same `switch` statement must be unique. To correct this error, change the value of either case label, delete one, or move one to another `switch` statement. For more information on the `switch` statement, refer to Section 6, "Statements," in this manual.

`Enumerated option must be compared equal (= or ==) or not equal (!= or ^= or <>) to enumerated constant.`

Inside a Boolean expression in a compiler control image, an enumerated option can be compared only for equality or inequality with one of its enumerated constants.

`Enumerated option must be compared (= or ==, != or ^= or <>, <, <=, >, >=) to integer constant or integer option.`

Inside a Boolean expression in a compiler control image, an integer option can be compared only with another integer option or an unsigned integer constant. Comparison can be for the following:

- Equality (= or ==)
- Inequality (^= or != or <>)
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

`Error in preprocessing conditional.`

An error was encountered within a preprocessing conditional, such as `#if` or `#elif`. For more information on conditional inclusion, refer to Section 8, "The C Preprocessor," in this manual.

`ERROR LIMIT OF <error limit> HAS BEEN EXCEEDED. COMPILATION WILL BE TERMINATED .`

Either the limit specified on the ERRORLIMIT compiler control card has been exceeded or the default limits set by the particular compiler has been exceeded. The limit can be increased or decreased by using the ERRORLIMIT option.

`ERROR: Maximum number of $BINDER_MATCH options exceeded.`

The number of $BINDER_MATCH or #pragma binder_match options exceeded an internal limit. Remove some of the options.

`ERROR: Maximum number of characters in $BINDER_MATCH options exceeded.`

The number of characters in the strings for $BINDER_MATCH options and the #pragma binder_match directive exceeded an internal limit. Either remove some of the options or shorten some of the strings.

`Expected "<token>".`

A syntax error was encountered in a situation where <token> was the only possible valid token that could appear at the indicated error position and the compiler was unable to make a more concrete suggestion (such as, "Missing <token>" or "indicated token ignored: expected <token>") because of additional syntax errors nearby or an unusually complex syntactic context.

`Expression not compatible with function return type.`

An attempt was made to return an expression whose type is not compatible with the return type of the function. For example, this may happen when trying to return a struct from a function whose return value is defined to be a simple type such as `int` or pointer.

`Extra comma (,) is non-standard.`

The identifier in an enumerator list was declared with a trailing comma. This message appears as an error, if the ANSI option is enabled. Please refer to "Enumeration Types" in Section 2 of this manual for a description of the correct syntax for enumeration declarations.

`Extraneous source characters ignored.`

One or more source characters were ignored in an attempt to recover from a syntax error. This error message will not be produced once the syntax error is fixed.

`Extraneous "<token>".`

A syntax error was encountered. It is likely that the indicated token is superfluous and should be removed to correct the error.

`Function and array types are not permitted as function result types.`

Functions may return a value of any type, except array or function. For more information on return values, refer to Section 7, "Functions," in this manual.

`A function argument cannot be void.`

An expression yielding a value to be passed as a function argument cannot be of type `void`.

`Function declared as: extern int <name> ().`

The function name that occurs before the parentheses was called, but it was never declared or defined. In this case, a declaration is automatically added. The function is declared with storage class `extern` and result type of `int`. For more information on implicit and explicit function declarations, refer to Section 3, "Declarations," in this manual.

`Function <name> not allowed in this context.`

Certain standard library functions may not be invoked dynamically during a TADS debug session. An attempt to invoke such a function caused this message to be displayed.

`Function "<name>" not optimized because of calls on macro setjmp or macro sigsetjmp.`

Certain special optimizations that might normally be performed when the OPTIMIZE compiler control option is set will not be done for this function because it contains calls on the `setjmp` macro or the `sigsetjmp` macro.

`Function "<name>" was declared static at line <number> and was referenced, but was never defined.`

A function that is declared `static` should be defined within the scope of the program, since such functions cannot be imported from other programs. If the ANSI option is enabled, this message appears as an error; otherwise, the message appears as a warning. For more information on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`Function "<name>" is declared static and not used.`

A static function, "<name>", has been declared and not used. For more information, see the subordinate option UNUSEDSTATICFUNCTION under "LINT Option" in Section 10, "Compiler Control Options."

`Function (or pointer to function) expected.`

The indicated expression should describe a function to call, but it does not have type function or pointer to function. If a function call was not desired, the syntax should be corrected; if needed, an expression of type function or pointer to function should be used. For more information on reading and writing complex declarators, refer to Section 3, "Declarations," in this manual.

`Function prototype required with this construct.`

A function that is declared with a linkage specification must be declared using the function prototype format. In particular, a function with no parameters must be declared with a `void` parameter list (for example, `int f (void);`) rather than an empty parameter list. For more information on function declarators, refer to Section 3, "Declarations," and for more information on linkages, refer to Appendix A, "Interface to the Library Facility," both in this manual.

`Function returning reference not allowed.`

The notation used to declare a reference parameter is similar to that used for declaring a pointer. It is therefore syntactically possible to declare a pointer to a reference, an array of references, a reference to a reference, or a function returning a reference. However, such declarations are not semantically meaningful and are, hence, flagged as errors. For more information on reference parameters, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Function type not allowed.`

A member of a structure or union cannot have type function. The type can be changed to be pointer to function, which is allowed. For more information on how to declare a pointer to a function, refer to Section 3, "Declarations," in this manual.

`Function type (supplied by typedef) cannot be qualified.`

Function types cannot be qualified by the `const` or `volatile` qualifiers. Normally when a function is declared or defined, any qualifiers present apply to the result type. When the function type is provided by a `typedef`, this cannot be done. Either the qualifiers should be removed or they should be moved inside the `typedef` definition. For more information on type specifiers and qualifiers, refer to Section 3, "Declarations," in this manual.

`GET_CONSTANT only handles logical, integer, and real types.`

An invalid parameter was passed to an internal compiler procedure during compilation. This is a compiler error. If there are any errors, correcting the errors may resolve the problem.

`GET_CONSTANT parameter not constant.`

An internal procedure to the compiler has received an invalid parameter when it was expecting a constant parameter. This is a compiler error.

`goto statements are not allowed as input during a TADS session.`

During a TADS debug session, preprocessing directives, label declarations, and `goto` statements are not allowed.

`Guessed "<keyword>".`

A syntax error was encountered. It is likely that the indicated identifier should be replaced with the suggested <keyword> since the <keyword> would be syntactically correct and the spelling of the <keyword> is close to that of the identifier.

`Identifier after #endif is non-standard.`

A `#endif` directive was followed by an identifier. This message appears as an error this message appears as an error if the ANSI option is enabled.

`Identifier required (type or typedef found).`

A function was defined using the prototype syntax. Prototype syntax is where the type of each formal argument and, optionally, the name is supplied between parentheses. When defining a function, however, the name is required and therefore, needs to be supplied.

Another possible cause is that an old-style declaration is desired, but the name of the first formal argument is the same as the name of a typedef. Since the first token inside the opening parenthesis is a type, prototype syntax is assumed. Either change the name of the formal argument or use prototype scope. For more information on function prototype, refer to Section 3, "Declarations," in this manual.

`Identifiers are limited to <number> significant characters.`

The indicated identifier exceeds the maximum number of significant characters for identifiers. All characters in the identifier name beyond this limit are ignored. Multiple identifiers that differ only with respect to characters beyond this limit will be indistinguishable. For more information on identifiers, refer to Section 1, "Overview of the C Language," in this manual.

`If a function is declared both without arguments and also as a prototype, all argument types must be default-compatible.`

A function can be declared in both the old style format and the function prototype format only if the argument types in the function prototype would not be subject to default argument promotion (namely, `char`, `short int`, and `float`). The message only appears if the ANSI option is enabled. Mixing old style and function prototype declaration for the same function is not recommended. For more information on function call, refer to Section 5, "Expressions and Operators," and for more information on defining functions, refer to Section 7, "Functions," both in this manual.

```
Illegal cast of pointer.
```

Pointers can be cast to pointer type or an integral type only and only pointers or integers can be cast to pointer type. If desired, multiple casts can be used to achieve the desired type. For more information on the cast operator, refer to Section 5, "Expressions and Operators," in this manual.

```
Illegal compatibility TARGET list or missing parenthesis. Code will be produce
d for TARGET = LEVEL0.
```

The target list specifying the secondary system is either missing a comma, a right parenthesis, or contains an invalid target identifier.

```
Illegal enumeration value.
```

The enumeration value required for this specific compiler control option is either less than one or an invalid value.

```
Illegal INCLUDE syntax.
```

The INCLUDE compiler control option is either incomplete or incorrect. Please check the INCLUDE syntax.

```
Illegal option in $ card.
```

No information was found within the parentheses used with the current option. A value was required within the empty parentheses.

```
Illegal or unrecognized $ option in expression.
```

Within an expression an illegal compiler control option was found or members of the expression are not valid.

```
Illegal TARGET value in $ card.
```

The TARGET option specified was not recognizable by the current system.

```
Illegal use of OPTION declarator.
```

The compiler control card OPTION was incorrectly specified.

```
Illegal VERSION option syntax in $ card.
```

The VERSION option syntax is incomplete or incorrect.

```
Implicit cast from integral type to pointer type.
```

An expression with integral type was implicitly cast to have pointer type. This is only a warning. If the cast is desired, an explicit cast will prevent this warning from being displayed. For more information on the cast operator, refer to Section 5, "Expressions and Operators," in this manual.

```
Implicit cast from pointer type to integral type.
```

An expression with pointer type was implicitly cast to have integral type. This is only a warning. If the cast is desired, an explicit cast will prevent this warning from being displayed. For more information on the cast operator, refer to Section 5, "Expressions and Operators," in this manual.

`Incomplete type not allowed.`

The context requires that the size of the type be known. The type supplied is incomplete and so the size cannot be determined. One solution might be to use a pointer to the type instead of the type itself. Otherwise the type should be completed or the context changed. For more examples of incomplete types and how to complete them, refer to Section 2, "Types," in this manual.

`Indicated token ignored: expected "<token>".`

A syntax error was encountered. It is likely that the indicated token should be replaced with the suggested <token> to correct the error. In some cases, the suggested <token> will indicate a class of tokens instead of a specific token, for example,

`Indicated token ignored: expected "number".`

Means that a number such as 1 or 542 should be supplied, not the identifier "number".

`Initializer not permitted with this type.`

Function types and void types may not be declared with initializers. For more information on initializers, refer to Section 3, "Declarations," in this manual.

`int expression expected.`

The indicated operand does not have integral type. The operator, however, requires an integral expression. For more information on determining the type of an expression, refer to Section 5, "Expressions and Operators," in this manual.

`Integral type expected.`

One of the operands in an integral expression does not have integral type. For an operator description and more information on type and class of expressions, refer to Section 5, "Expressions and Operators," in this manual.

`Internal assertion is false.`

Unexpected results were discovered internally to the compiler. This is a compiler error.

`Invalid file title.`

An invalid file title was used in an `#include` statement. Check the syntax of the file name. For more information the `#include` directive, refer to Section 8, "The C Preprocessor," in this manual.

`Invalid index expression.`

In an expression containing the subscript operator ([ ]), one of the two operands must be an array or pointer type and the other operand must be an integral type. For more information on array subscripting, refer to Section 5, "Expressions and Operators," in this manual.

`Invalid preprocessing directive.`

The preprocessor directive given is not one of the valid preprocessor directives. For more information on preprocessor directives, refer to Section 8, "The C Preprocessor," in this manual.

`Invalid storage class for initialized variable.`

An attempt was made to use an initializer in an invalid context. For example, initializers may not be used for a `typedef` identifier. Also they may not be used for a variable that is declared `extern` within a function. For more information on defining and declaring external variables and on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`Invalid TARGET value in $ card. Code will be produced for TARGET = LEVEL0.`

The processor specified in the TARGET compiler control option is not a recognized processor. The compile is continuing with the default value of TARGET=LEVEL0.

`It is illegal to have nested parenthesized option lists.`

Multiple subordinate options or strings are not allowed within the parenthesized list.

`Label declarations are not allowed as input during a TADS session.`

During a TADS debug session, preprocessing directives, label declarations, and `goto` statements are not allowed.

`Label "<name>" does not occur in this function.`

An attempt was made to goto a label that is not defined within the function containing the `goto`. The `goto` statement can only be used to transfer control from one statement in a function to another statement in the same function. For more information on `goto` statements, refer to Section 6, "Statements," in this manual. Refer to Volume 2, "Headers and Functions," for information on the header `<setjmp.h>` and about transferring control to a point outside of the current function.

`Last argument discarded, no matching formals.`

The function being called has fewer formal arguments than the number of actual arguments supplied. The extra actual argument has no side effects so it can be safely discarded. The number of actual arguments should match the number of formal arguments or the function should be declared and defined as taking variable number of arguments. For more information on how to declare a variable number of arguments, refer to Section 3, "Declarations," in this manual.

`Last <number> arguments discarded, no matching formals.`

The function being called has fewer formal arguments than the number of actual arguments supplied. The extra actual arguments have no side effects so they can be safely discarded. The number of actual arguments should match the number of formal arguments or the function should be declared and defined as taking variable number of arguments. For more information on how to declare a variable number of arguments, refer to Section 3, "Declarations," in this manual.

`Length of continued line exceeds maximum of <number> -- line ignored.`

A source line, which may be continued onto several physical lines by terminating each physical line with a backslash, may be up to <number> characters long. A source line exceeding this maximum length is ignored by the compiler.

`Lexical error.`

A character or sequence of characters that does not form a valid C token was encountered. The compiler resumes compilation after skipping over the offending character or characters.

`Library already declared.`

An attempt was made to declare a library that has already been declared. For more information on referencing libraries, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Line <sequence number>: badly formed library name or title.`

The source file contains a library declaration where the library name or title is not a legal name or title. A legal library name is 1 to 17 letters or digits. A legal library title is a legal file title.

`Line <sequence number>: deimplemented attribute.`

The source file specified an attribute that is no longer implemented. The use of that attribute needs to be removed and possibly replaced by use of newer attributes.

`Line <sequence number>: code file is too large to bind.`

The generated code for the "outer block" is too large for Binder to handle. The outer block contains code to initialize static variables and variables with file scope and any code outside of functions. Either reduce the amount of code placed in the outer block or split the source file into smaller files.

`Line <sequence number>: the code segment has exceeded the maximum row size of the CODE file. The AREASIZE attribute of the CODE file should be increased to at least <number>.`

The source file needs to a larger AREASIZE specified for the CODE file to compile. To specify the AREASIZE, refer to either the CANDE *COMPILE* command or the WFL *COMPILE* statement.

`Line <sequence number>: the data segment has exceeded the maximum row size if the CODE file. The AREASIZE attribute of the CODE file should be increased to at least <number>.`

The source file needs a larger AREASIZE specified for the CODE file to compile. To specify the AREASIZE, refer to the CANDE *COMPILE* command or the WFL *COMPILE* statement.

`Line <sequence number>: the program has exceeded the maximum D<number> size of <number>.`

Too many variables, procedures, or constants exist in the source file. If the "D": level is 1, too much code and/or constants exist. If the "D" level is 2, too many global variables and/or procedures exist. If the "D" level is 3 or greater, too many local variables and/or procedures exist. Reduce the number or split the source file into smaller files.

`Line continued at end of file.`

The last physical line in the source file terminates with a backslash, which is intended to indicate that the source line is continued onto the next physical line. This might indicate that some lines have been inadvertently deleted from the source file.

`Line number must be between 1 and <number>.`

The integer argument to a `#line` directive must be within a specific range. If the ANSI option is enabled, the range is 1-32767; otherwise, the range is 1-99999999.

`Linkage specification differs from previous declaration.`

A previous declaration of this function contained a linkage specification that is different from the linkage specification in the current declaration. For more information on linkages, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Linkage specification only allowed at file scope.`

A function may not be declared with a linkage specification in any scope other than file scope. For more information on the scope of identifiers, refer to Section 1, "Overview of the C Language," and for more information on linkages, see Appendix B, "Interface to the Library Facility," both in this manual.

`Macro _ASERIES_SOURCE (<SSR1>) is greater than compiler level (<SSR2>). Newer release of compiler may not compile the same way. Define _ASERIES_SOURCE to <SSR2> to ensure compatibility.`

The macro _ASERIES_SOURCE controls what names are declared in headers, their values, and sometimes the semantics of the names. If _ASERIES_SOURCE specifies a larger SSR value than the compiler, and if changes occur in a future release, the compiler might not compile the program or it might compile the program incorrectly in that release. To avoid this problem, define _ASERIES_SOURCE to <SSR2>. To avoid this warning, reset the compiler control option $LINT(_ASERIES_SOURCE).

`Malformed character constant.`

An invalid backslash (\) escape sequence was encountered in a character constant.

`Missing actual macro argument for macro "<name>".`

A parameterized macro was invoked with too few actual arguments. The number of actual arguments should match the number of formal arguments in the macro definition. Also actual arguments must be non-empty token strings. For more information on macro definitions, refer to Section 8, "The C Preprocessor," in this manual.

`Missing comma ',' in $ option expression.`

A comma is the next token required but was not found.

`Missing #endif.`

An `#if` preprocessor directive was encountered that does not have a matching `#endif` directive. Check `#if`/`#endif` pairings to determine whether an extra `#if` was included or an `#endif` erroneously omitted; for example, by accidentally embedding it in a comment. For more information on conditional inclusion, refer to Section 8, "The C Preprocessor," in this manual.

`Missing left parenthesis '(' in $ option expression.`

A left parenthesis is the next token required but was not found.

`Missing right parenthesis ')' in $ option expression.`

A matching right parenthesis was not found to match the left parenthesis found in the compiler control record expression.

`Missing right parenthesis after subfile specification in INCLUDE.`

The matching right parenthesis was not found following the specification of a subfile name in an INCLUDE option. The subfile name must match a subfile name specified in a corresponding COPYBEGIN and COPYEND compiler control card.

`Missing semicolon (;) is non-standard.`

The last member of a structure or union was declared without a trailing semicolon. This message appears as an error if the ANSI option is enabled. Refer to "Structure Types" and "Union Types" in Section 2 of this manual for a description of the correct syntax for structure and union declarations.

`Missing string delimiter (" ") at the end of line.`

A string was opened (`"...`) on a line, but was not closed (`..."`) on the same line. If the string needs to be longer than one line, a trailing backslash (\) can be used to logically join the next line to the end of the current line. Or each line can contain a separate string and because they are adjacent, they will be concatenated to form a single string. For more information on string constants, refer to Section 1, "Overview of the C Language," in this manual.

`Missing "<token>".`

A syntax error was encountered. It is likely that the suggested <token> was inadvertently omitted and should be inserted before the error indicator to correct the error. In some cases, the suggested <token> will indicate a class of tokens instead of a specific token, for example,

`Missing "number".`

Means that a number such as 1 or 542 should be supplied, not the identifier "`number`".

`MORE THAN 17 CHARACTERS OCCURRED IN A NAME NODE FOR FILE NAME - <file name>.`

The name nodes of a file name are separated by slashes and should not contain more than 17 characters. Correct the file name and recompile.

`MORE THAN 12 NODES OCCURRED IN THE FILE NAME - <file name>.`

The file name is invalid because it contains more than 12 name nodes. Correct the file name and recompile.

`Must #include <sys/types.h> before <...>.`

The header indicated by ". . . " requires typedef declarations that are only in <sys/types.h>. If <sys/types.h> is **not** included in the file, you must insert the line "#include <sys/types.h>" before including this header. If <sys/types.h> **is** included in the file, but is after this header, you must move <sys/types.h> before this header.

`Name already declared = "<name>".`

The identifier <name> was already used in another declaration in the same scope. The problem can be avoided by changing either name or if possible, placing this declaration in a more nested scope. For more information on the scope of identifiers, refer to Section 1, "Overview of the C Language," in this manual.

```
Name not declared = "<name>".
```

The identifier <name> has not been declared yet or the scope of its declaration has already finished. Possibly it is misspelled, it needs to be declared, or the scope of its declaration needs to be more global. For more information on the scope of identifiers, refer to Section 1, "Overview of the C Language," in this manual.

```
Newline and quote characters are not allowed in header names.
```

A file title in a `#include` directive that is bounded by quotes (") may not contain the newline or quote character.

```
Newline, formfeed, vertical-tab characters are not allowed between the # and p
reprocessing directives.
```

Only preprocessor white space characters may occur between the hash (#) symbol and the preprocessing directive. Preprocessor white space characters include: horizontal tab, space, and comments. For more information on white space, refer to Section 1, "Overview of the C Language," in this manual and for more information on lexical elements and lexical conventions of the preprocessor, refer to Section 8, "The C Preprocessor," also in this manual.

```
NEWSOURCE sequence error.
```

The sequence of the NEW file being created is out of order. The record being processed has a sequence value less than that of the last record placed in the NEW file.

```
NO IDENTIFIERS WERE FOUND IN THE FILE NAME - <file name>.
```

The file name may contain invalid characters. An identifier that represents one level of the file title or a usercode can include hyphens and underscores without quotes. If other non alphanumeric characters are included, then quotes are required. Correct the file name and recompile.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function name
> cannot be unrolled; missing from Support Library.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated inline because it was not found in the support library. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function name
> cannot be unrolled. Cheap blocks cannot be unrolled.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated inline because it contained a “cheap block”. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function name
> cannot be unrolled. Only functions can be unrolled.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated in because it is not a normal function. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function name
> cannot be unrolled. Only Level 3 Libraries may be unrolled. This Support Lib
rary is level <number>.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain function calls be replaced by inline code. The indicated function cannot be regenerated inline because the support library is not in its normal form. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function name
> cannot be unrolled. This function has too many locals.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated inline because it has too many local declarations. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): Cannot find li
brary directory. No functions will be unrolled. This may be caused by inabilit
y to decode D2 stack building code. Stopped reading code at (<stack address>)
(#<number>).
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated inline because the support library is not in its normal form. This is a warning only and cannot be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): Compiler incon
sistency while unrolling <function name>.
```

The OPTIMIZE (SET UNRAVEL) compiler control option was set, requesting that certain functions calls be replaced by inline code. The indicated function cannot be regenerated inline because it contained unrecognized code. This is a warning only and can be ignored.

```
Non-fatal COMPILER ERROR <sequence number> (<internal number>): Length paramet
er to memcmp exceeds length of string constant parameter.
```

A call on the library function `memcmp` consists of a constant length parameter and one or more constant string parameters. The length parameter is longer than the string constant. The length is truncated to the length of the string constant. This is a warning only.

`Not a valid assignment target.`

The target of the assignment is not a valid target. Valid targets are called "lvalues". The result of a function call or a case is not an lvalue. Refer to Section 5, "Expressions and Operators," in this manual for a list of valid lvalues.

`Not allowed with $FARHEAP (SET ONE).`

The indicated operation is not allowed when $FARHEAP(ONE) is set. Either avoid that operation or reset $FARHEAP(ONE).

`Not assignment compatible.`

The type of the value of an assignment is not compatible with the type of the target or the type of the actual argument is not compatible with the type of the formal argument. For more information on assignment operators and function calls, refer to Section 5, "Expressions and Operators," in this manual.

`Not valid as a by-reference argument.`

If an external function is declared to take an argument that is passed by reference, the actual argument in the function call must be an lvalue. For more information on lvalue, refer to Section 5, "Expressions and Operators," and for more information on parameter passing, see Appendix A, "Interface to the Library Facility," both in this manual.

`Not valid as a FORTRAN argument.`

If an external function with FORTRAN linkage is declared to take an array, struct, or union parameter, the actual argument in the function call must be an lvalue. For more information on lvalue, refer to Section 5, "Expressions and Operators," and for more information on parameter passing, refer to Appendix A, "Interface to the Library Facility," both in this manual.

`Not valid as an operand in this context.`

The indicated operand is not valid in the context in which it is used. Usually this means the type is not acceptable to the operator. For more information on operators, refer to Section 1, "Overview of the C Language," in this manual.

`Number of file fragments in virtual input file exceeds maximum allowed.`

An internal compiler limit has been exceeded. The source program could be resequenced or the number of include files could be lowered to stay within the limit.

`Numbers are limited to 11 digits on $ cards.`

The maximum number of digits in a compiler control record is limited to 11 digits.

`Object "<name>" is declared and not used.`

An object, "<name>", has been declared and not used. For more information, see the subordinate option UNUSEDOBJECT under "LINT Option" in Section 10, "Compiler Control Options."

`Octal constant larger than a single character.`

An octal escape sequence was scanned in a character constant (`"\0...'`) or inside a string constant (`"\0..."`). The value of the octal constant is larger than 255, the maximum value of a character.

`Old-style argument declarations only allowed with old-style functions.`

Old-style functions are defined with only the argument names appearing between parentheses. Prototype-style functions provide type information as well as the argument names between the parentheses. Old-style argument declarations occur when the type of the arguments are provided between the parenthesized list and the curly-braces ({ }). Old-style argument declarations are not allowed with prototype-style function definitions. For more information on old style and function prototype, refer to Section 3, "Declarations," in this manual.

`On Mark 4.1, attribute lists in compiler sheet array will be deimplemented. Se e 3.9 MCP-TASKING PRI-506499 for detour.`

The compiler was initiated from a 3.8 or older CANDE or by a program that does not conform to the new rules for passing attribute lists (file, task, library, and data base equations). The new rules will be enforced starting with the 4.1 release. This warning indicates that the initiating program needs to be upgraded before the 4.1 release.

`One or more non-static functions must be defined.`

An attempt was made to compile a source file that does not contain any functions visible outside of the source file scope. This C language implementation requires that at least one non-static function must be declared within a source file.

`Only allowed for extern function declarations.`

Linkage specifications and hidden parameters may only be used in declarations of functions with storage class `extern`. For more information on linkages, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Only "extern" may be used here.`

If a function is declared within the body of another function, that function may only have the storage class `extern`; all other storage classes are invalid. For more information on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`Only function can be extern-ed from a library.`

Within a library program, the only objects that may be given the `extern` storage class are functions. These are then the entry points for the library. Data objects may not be `extern` within a library. For more information on the library facility, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Only function definitions (with body) can declare old-style arguments.`

Old-style argument declarations may only be used when defining a function with its body. They may not be used, for example, in function prototypes. For more information on old style declarations, refer to Section 7, "Functions," in this manual.

`Only one storage class may be specified.`

A declaration may contain only one storage class specifier. For more information on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`Option can appear only as first item on $ card.`

Some compiler control cards are required to be the first option on a particular record. The current option in error should be placed on a separate card.

`OPTION_NAME array has become inconsistent.`

The compiler has stored an option into one of its internal tables with an unrecognized type. This is a compiler error.

`Parameter type differs from previous declaration.`

If an external function is declared to take an argument that is passed by reference, the type of the actual argument must match the formal argument. For more information on parameter passing, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Parameterization differs from previous declaration.`

A previous declaration of this function specified a parameter list that is different in the number of types of parameters than the current declaration.

`Please be aware that during TADS sessions, optimizations used as a result of t he OPTIMIZE compiler control option may reorder or eliminate execution of cert ain statements or remove potential statement breakpoints.`

This warning is issued by the compiler if both the OPTIMIZE and TADS options are enabled.

`Pointer expression expected.`

The indicated operand does not have pointer type. The operator, however, requires a pointer expression. For more information on determining the type of an expression, refer to Section 5, "Expressions and Operators," in this manual.

`Pointer to reference not allowed.`

The notation used to declare a reference parameter is similar to that used for declaring a pointer. It is therefore syntactically possible to declare a pointer to a reference, an array of references, a reference to a reference, or a function returning a reference. However, such declarations are not semantically meaningful and are, hence, flagged as errors. For more information on reference parameters, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Poorly formed conditional inclusion directive controlling constant expression.`

The constant expression part of a conditional inclusion preprocessor directive, such as `#if` or `#elif`, does not form a valid integral constant. This expression may not contain, for example, a `sizeof` operator, a cast, or an enumeration constant. For more information on conditional inclusion, refer to Section 8, "The C Preprocessor," in this manual.

`Preprocessing directives are not allowed as input during a TADS session.`

During a TADS debug session, preprocessing directives, label declarations, and `goto` statements are not allowed.

`Previously undefined $ option.`

The compiler control option is not a recognized option and is therefore treated as a user defined option.

`Prior phrase must be last item on $ card.`

Some compiler control options must be the only option on a particular record. No other option may follow it.

`Reference to reference not allowed.`

The notation used to declare a reference parameter is similar to that used for declaring a pointer. It is therefore syntactically possible to declare a pointer to a reference, an array of references, a reference to a reference, or a function returning a reference. However, such declarations are not semantically meaningful and are, hence, flagged as errors. For more information on reference parameters, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Reference type not supported for this language.`

A parameter has been declared by-reference for a function whose specified language does not support by-reference parameter passing. For more information on parameter passing and linkages, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Result type differs from previous declaration.`

A previous declaration of this function specified a result type different from this declaration. Possibly the result type was not specified and defaulted to `int` type. Change the result type in either declaration to be the same.

`Result type must be scalar or "void".`

A function that is declared with a linkage specification must have a scalar or `void` return type; it may not, for example, return a structure. For more information on linkages, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Result type must be "void".`

A function declared with COBOL linkage must have a return type of `void`. For more information on linkages and parameter passing, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Return with expression not allowed for void functions.`

Functions that are defined to return `void` may not use any form of the return statement except the empty form:

```
return;
```

Either the function declaration needs to be changed to indicate that a value is to be returned or the expression part of the return statement must be removed. For more information on return values, refer to Section 7, "Functions," in this manual and for more information on void type, refer to Section 2, "Types," in this manual.

`Scalar type expected (not void, union, or struct).`

The expression does not have scalar type (arithmetic or pointer), but the context requires scalar type. For more information on types of expressions, refer to Section 5, "Expressions and Operators," in this manual.

`Sequence number expected.`

Integer constants representing sequence numbers in TADS commands may not be in hexadecimal.

`Statement linkage incorrect.`

An internal procedure to the compiler has received data that is incorrect. This is a compiler error.

`String assignment (=) or update (+=) operator expected after string-valued option.`

The syntax for a string valued compiler control option requires either an assignment (=) or update (+=) operator. Either the incorrect option was used or the option syntax is incorrect.

`String constant expected.`

A valid string was expected for the string valued compiler control option.

`String too long.`

1) A character array has been initialized with a string literal whose length is longer than the array's declared length. For more information on initializing arrays, refer to Section 3, "Declarations," in this manual.

2) The length of one of the strings for $BINDER_MATCH or `#pragma binder_match` exceeded an internal limit. The longest allowed string for these options is 255 characters. Shorten the string that is longer than 255 characters.

`Struct/union must declare at least one named member.`

This declaration of a structure or union only declares unnamed bit fields. At least one named member must be declared. For more information on structure types and union types, refer to Section 2, "Types," in this manual.

`Subfile specification in INCLUDE must be a string or is missing.`

The left parenthesis in the INCLUDE option specifying an include of a symbolic named region was found without the corresponding symbolic name. The symbolic name must be a string.

`The $ option <option name> is assumed to be a user defined option.`

The compiler control option is not a recognized option and is therefore treated as a user defined compiler control option.

`The # operator may be applied to macro parameters only.`

The # preprocessor operator in the macro body of a parameterized macro must be followed by a formal argument as the next preprocessing token in the macro body. For more information on the # preprocessor operator, refer to Section 8, "The C Preprocessor," in this manual.

`The ## token may not begin or end a macro definition.`

The preprocessing concatenation operator (##) must have preprocessor tokens on either side of it upon which it may do the concatenation. It may not appear at the beginning or end of a replacement list of a macro definition. For more information on the ## preprocessor operator, refer to Section 8, "The C Preprocessor," in this manual.

`The combined space required for data items declared in <function name> (<numbe r> words) exceeds the maximum available space of <number> words. Check MEMORY_ MODEL compiler control option.`

The local variables declared in the indicated function require more space than can be supported using the current MEMORY_MODEL compiler control option. Either reduce the size requirements of the local variables or, if the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

`The combined space required for data items declared in the outer block (<numbe r> words) exceeds the maximum available space of <number> words. Check MEMORY_ MODEL compiler control option.`

The global variables require more space than can be supported using the current MEMORY_MODEL compiler control option. Either reduce the size requirements of the global variables or, if the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

`The combined space required for variable <name> (<number> words) exceeds the m aximum available space of <number> words. Check MEMORY_MODEL compiler control option.`

The indicated variable is larger than can be supported using the current MEMORY_MODEL compiler control option. Either reduce the size of the variable or, if the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

`The combined space required for variable length arguments to <function name> ( <number> words) exceeds the maximum available space of <number> words. Check M EMORY_MODEL compiler control option.`

A call on the indicated function requires more space for the variable number of arguments than can be supported using the current MEMORY_MODEL compiler control option. Either reduce the size of the arguments passed as variable number of arguments or, if the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

`The compiler has faulted with fault number <number>.`

This is a compiler error indicating that the compiler has faulted with a recognized program fault.

`The cycle delta is restricted to three digits.`

The VERSION compiler control option requires the update version specified to have a cycle delta no greater than 999.

`The cycle number is restricted to three digits.`

The VERSION compiler control option requires the <cycle> number to be no greater 999.

`The ending comment delimiter token (*/) is expected to be in the same file as the opening /* token.`

The source file contains an opening comment delimiter (/*) that does not have a matching closing comment delimiter (*/). For more information on comments, refer to Section 1, "Overview of the C Language," in this manual.

`THE FILE TITLE DID NOT TERMINATE WITH A PERIOD FOR THE FILE NAME - <file name> .`

The file name should terminate with a period (.). Correct the file name and recompile.

`The function "<name>" is used but never declared.`

The function was declared with storage class `static`, but a function body for it was not defined in the file. Storage class `static` prevents a definition in another file from satisfying this declaration. Either remove the `static` keyword or provide define the function with a body within the file. For more information on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`The global identifier "main" must be declared as a function returning int with either no arguments or with the arguments (int, char **).`

The global identifier "`main`" is restricted by the ANSI standard to refer to a function that takes one of the following two forms: either "`int main(void);`" or "`int main(int, char **);`". If the identifier is intended to be the "`main`" function that is invoked at program startup, its declaration should be altered to match one of the two valid forms. Otherwise, the identifier should be renamed so as not to conflict with the name of the "`main`" function.

`The library named could not be found. The default name will be used. Library name = <library name>.`

In a string compiler control option, the specified library name is not a recognized library.

`The maximum code file size of 262,144 disk sectors (47,185,920 bytes) has been exceeded.`

The maximum allowable code file size is 262,144 disk sectors (47,185,920 bytes). If the size of the code file cannot be reduced, the compiler is unable to produce a code file. The size of a code file increases if the following compiler options have been set: BINDINFO, LINEINFO, and TADS. Resetting one or all of these compiler options may allow the compiler to produce a code file.

```
The maximum nesting level for #include files is <number>.
```

The limit of nested include files (indicated by <number>) has been exceeded. This error commonly occurs when a file `#includes` itself, either directly or through some other `#include` file.

```
The maximum number of different INCLUDE files (509) has been exceeded. INCLUDE
 ignored.
```

The internal limit within the compiler is 509 include files per program. That limit has been exceeded. The compiler will continue without doing the include.

```
The maximum number of different physical files (508) has been exceeded. MERGE
ignored.
```

The internal limit within the compiler is 509 include files per program. That limit has been exceeded. The compiler will ignore the Merge.

```
The maximum number of user defined dollar options has been exceeded.
```

The internal limit of user defined compiler control options has been exceeded. The error may be corrected by limiting the number of user defined options within the current program source.

```
The only storage class allowed for arguments is register.
```

An invalid storage class was used in the declaration of a function argument. In a function argument declaration, only `register` storage class may be specified. For more information on storage class specifier defaults, refer to Section 3, "Declarations," in this manual.

```
The $BYTEADDRESS option may be enabled only for MCP/AS (Extended) machines. Th
e target for this compile is not MCP/AS (Extended), therefore $BYTEADDRESS has
 been disabled.
```

This is a warning message informing you that the $BYTEADDRESS option can be enabled only for MCP/AS (Extended) systems.

`The other conflicting access to object (<number>).`

C does not specify the order of evaluation for other operands of most operators; therefore, the embedded assignment and the access to the target of the assignment can be evaluated in either order. The value of the expression can change depending on the level of optimization done by the compiler. The embedded assignment should be moved out as a separate statement or use the comma (`,`), if (`?:`), logical-and (`&&`), or logical-or (`||`) operator to control the order of evaluation. For more information these operators and on gray code, refer to Section 5, "Expressions and Operators," in this manual.

The number in parentheses () is used to distinguish multiple conflicts within the same expression.

`The patch delta is restricted to four digits.`

The VERSION compiler control option requires the update version specified to have a patch delta no greater than 9999.

`The patch number is restricted to four digits.`

The VERSION compiler control option requires the <patch> number to be no greater than 9999.

`The specifier must be repeated for each argument in a function prototype.`

An argument (other than the first argument) in a function prototype declaration was declared without a type specifier. Each argument in a function prototype must be declared with its own type specifier. Refer to Section 3, "Declarations," for more information on function prototypes.

`The symbolically named region from the $ INCLUDE card could not be allowed.`

In an INCLUDE compiler control option that is including a symbolically named region that should be delimited with COPYBEGIN and COPYEND options, the named region could not be found in the specified file.

`The two operands are not compatible.`

One of the operands to the binary or ternary operator is not compatible with the other operand or operands. For more information on operators, refer to Section 1, "Overview of the C Language," in this manual.

`The variable "<name>" exceeds maximum size of <number> bytes. Check MEMORY_MOD EL compiler control option.`

The variable that is being declared is too large. The maximum number of bytes that is legal is provided. The "Check MEMORY_MODEL..." sentence is supplied if selecting a larger memory model would increase the maximum limit. For more information on the MEMORY_MODEL compiler control option, refer to Section 10, "Compiler Control Options," in this manual.

`The variable "<name>" exceeds maximum size of <number> elements. Check MEMORY_ MODEL compiler control option.`

The variable that is being declared is too large. The maximum number of elements that is legal is provided. The "Check MEMORY_MODEL..." sentence is supplied if selecting a larger memory model would increase the maximum limit. For more information on the MEMORY_MODEL compiler control option, refer to Section 10, "Compiler Control Options," in this manual.

`The version delta is restricted to two digits.`

The VERSION compiler control option requires the update version specified to have a version delta no greater than 99.

`The version number is restricted to two digits.`

The VERSION compiler control option requires the <release> number that is sometimes referred to the <version> number to be no greater than 99.

`This declaration of "<name>" is not consistent with a previous declaration.`

This declaration of the object <name> conflicts with a previous declaration of the same object. For example, this error may occur if the same variable "`temp`" were declared in two places as "`extern int temp;`" and then "`extern double temp;`".

`This directive is not in the same source file as its opening conditional direc tive.`

Conditional directives such as `#else`, `#elif`, and `#endif` require an opening `#if` directive within the same source file. For more information on conditional inclusion, refer to Section 8, "The C Preprocessor," in this manual.

`This dollar option is a recognized option and may not be changed after the fir st statement of the program.`

Some compiler control cards are only allowed to be set or reset prior to the first record of source program text. The option is a valid option, but cannot be changed after compilation is under way.

This formal macro argument ("<name>") has already been defined for the macro "<name>".

Formal arguments to parameterized macros must each have a unique identifier. For more information on macro definitions, refer to Section 8, "The C Preprocessor," in this manual.

`This generates the token "defined" through macro replacement. This generation is not portable.`

A preprocessor token "defined" was generated through macro replacement. Use of this token as the preprocessor operator "`defined`" is not portable. The ANSI compiler control option determines whether this use generates a compiler error or just a warning. For more information on the `defined` operator, refer to Section 8, "The C Preprocessor," in this manual.

`This INCLUDE file has been altered since its earlier use.`

The alter date and alter time for the file being included does not match today's date or time from the last time it was included in this compile.

`This option cannot be used in this environment. It will be ignored.`

The current string compiler control option has no meaning in the current compiling environment and should not be specified.

`This qualifier is not allowed in this context.`

The indicated qualifier has been used illegally. Refer to the appropriate section in this manual for information on the usage of "const", "volatile", "_ _near", and "_ _far" qualifiers.

A near or far qualifier was encountered for an object with "auto" storage class, or a member of a structure or union that is not a pointer to a near or far object.

`This redefinition of the macro "<name>" is not benign.`

A macro can be redefined, but only if the list of tokens is identical to the original definition. White space tokens (blanks, comments) do not have to exactly match, but their positions in the list must match. To correct this error, the two definitions should be made the same or the macro should be `#undef` before it is redefined. For more information on undefining and redefining macros, refer to Section 8, "The C Preprocessor," in this manual.

`This storage class may not be used in this context.`

A storage class specifier was used in a declaration for which it is not appropriate; for example, declaring a global variable with the storage class `auto`. For more information on storage class specifiers, refer to Section 3, "Declarations," in this manual.

`Too few arguments supplied for macro "<name>".`

A call on the parameterized macro <name> must have the same number of parameters as the macro definition. For more information on macro definitions with parameters, refer to Section 8, "The C Preprocessor," in this manual.

`Too many arguments, limit is <number>.`

An attempt was made to declare a function with too many arguments. The internal limit for the number of arguments allowed is given in <number>. For more information on internal compiler limits, refer to Appendix B, “Internal Compiler Limits,” in this manual.

`Too many arguments supplied for macro "<name>".`

A call on the parameterized macro <name> must have the same number of parameters as the macro definition. For more information on macro definitions with parameters, refer to Section 8, “The C Preprocessor,” in this manual.

`Too many bad labels declared.`

The source file contains too many “bad” labels. A bad label is a global label that is branched to from inside a nested procedure. Either reduce the number of bad labels or split the source file into smaller files.

`Too many card image segments.`

An internal procedure to the compiler has received data that is incorrect. This is a compiler error.

`Too many characters in constant.`

A multi-character constant, such as 'ABC', may contain no more than four characters. For more information on character constants, refer to Section 1, “Overview of the C Language,” in this manual.

`Too many cross referenced identifiers.`

The internal tables of the compiler have exceeded the maximum number of cross referenced identifiers.

`Too many Linear Sequences.`

An internal procedure to the compiler has received date that is incorrect. This is a compiler error.

`Too many ordered operators (conditional-and, expression-if, and so forth).`

An internal limit of operators has been exceeded in the compiler.

`Too many values (<number> extra).`

More values were supplied between the curly braces ({ }) in the initializer than were required for the variable or for a member of the variable being declared. Possibly the total number of values supplied is correct, but they are grouped incorrectly. For more information on initializers, refer to Section 3, “Declarations,” in this manual.

`Total string-valued-$-option pool has exceeded its maximum size.`

The internal limit of string compiler control options has been exceeded. This is a compiler error. The error can be avoided by limiting the number of strings or their length.

`Type not supported for this language.`

If a function is declared with a linkage specification, the types of the function's parameters are restricted to those supported by the specified language. For more information on parameter passing, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Unexpected token: remainder of line discarded.`

A syntax error was encountered in a preprocessing directive. It is possible that the error can be corrected by removing any text beyond the error position on the line containing the directive. Note that an empty preprocessing directive is permitted, so the error position may be just beyond the hash (#). This indicates a seriously flawed directive. For more information on valid preprocessing directives, refer to Section 8, "The C Preprocessor," in this manual.

`Unexpected token: source ignored up to "<token>".`

A syntax error was encountered from which the compiler was not able to recover by suggesting a minor correction. The compiler has therefore ignored a portion of the program and has restarted compilation at <token>. Usually, a single declaration, statement, or group of statements has been ignored and may contain undetected errors. This error occurs when the syntax errors are numerous or severe.

`Unimplemented #pragma ignored.`

A `#pragma` preprocessing directive is a mechanism provided by the ANSI C standard for specifying implementation-specific extensions to the C language. The indicated pragma is not provided in this C language implementation and has been ignored by the compiler.

`Unknown linkage specification.`

A function was declared with an unknown linkage specification. For more information on valid linkages, refer to Appendix A, "Interface to the Library Facility," in this manual.

`Unrecognized escape character.`

Unrecognized escape character. A backslash (\) was scanned inside a character constant (`'\...'`) or a string constant (`"\..."`). The characters after the backslash do not form a valid escape sequence. For more information on metatoken escape-sequence, refer to Section 1, "Overview of the C Language," in this manual.

`Unrecoverable syntax errors. Compile terminated.`

A severe syntax error was encountered. The compiler was unable to find a point from which the compile could be restarted with any likelihood of producing correct or useful errors and warnings.

`Unsigned integer constant expected.`

An integer value was expected for the integer compiler control option.

`Update operator (+=) can only be used for string valued options.`

The NEW compiler control option does not require update operators in its syntax.

`Use of adjectives incompatible with this type.`

A type adjective (`short`, `long`, `signed`, `unsigned`) cannot be used with the specified type. For more information on character, integer, and floating-point types, refer to Section 2, "Types," in this manual.

`Use of adjectives incompatible with typedef.`

A type adjective (`short`, `long`, `signed`, `unsigned`) cannot be used when the type is specified using a `typedef`. Either use the type adjective when declaring the typedef or do not use a `typedef`. For more information on initializers, refer to Section 3, "Declarations," in this manual.

`Using the <option name> instead of subordinate option <class name>(<option name>).`

The <option name> is the name of both an option and a subordinate option. The value of the option is used inside of Boolean expressions in an compiler control image. The following example uses the option UNSIGNED and not the subordinate option PORT(UNSIGNED):

```
$SET PORT(CHAR2=UNSIGNED)
```

To use the value of the subordinate option, you should include the following <class name>:

```
$SET PORT(CHAR2=PORT(UNSIGNED))
```

`When used, "void" must be the only argument.`

If a formal argument to a function is declared of type `void`, `void` must be the only item between the parentheses, for example, `f(void)`. For more information on function declarators, refer Section 3, "Declarations," in this manual.

`Wide character strings not allowed here.`

A wide character string was provided in a place where only a normal character string is allowed. Change the wide character string to be a normal character string.

`Wrong tag.`

The tag is the identifier after `struct`, `union`, or `enum`. Either the identifier was not previously declared to be a tag or it was declared to be a tag for a different type (`struct`, `union`, or `enum`) than is currently being used. For more information on enumerator, structure, and union types, refer to Section 2. "Types," in this manual.

Appendix I
Abnormal Compiler Output Messages

The error messages in this appendix indicate that the C compiler has detected an unexpected logic error in its own processing. Contact your customer support representative to report the problem.

```
COMPILER ERROR <sequence number> (<internal number>):
Address Couple displacement of <number> is too large.

COMPILER ERROR <sequence number> (<internal number>):
Address Couple of (<stack address>) not legal in this context.

COMPILER ERROR <sequence number> (<internal number>):
Address function valid only for t_vars with addressed on and in_area off or
for t_consts.

COMPILER ERROR <sequence number> (<internal number>):
Argument error in entry point at (<stack address>); argument #<number>.

COMPILER ERROR <sequence number> (<internal number>):
Argument error in get_integer.

COMPILER ERROR <sequence number> (<internal number>):
Bad argument class.

COMPILER ERROR <sequence number> (<internal number>):
Bad argument kind.

COMPILER ERROR <sequence number> (<internal number>):
Bad code_array call.

COMPILER ERROR <sequence number> (<internal number>):
Bad display item.

COMPILER ERROR <sequence number> (<internal number>):
Bad element size (<number>), should be 0, 1, 2, or 4.

COMPILER ERROR <sequence number> (<internal number>):
Bad function name.

COMPILER ERROR <sequence number> (<internal number>):
Bad op for bit_set_type compare.

COMPILER ERROR <sequence number> (<internal number>):
Bad parameters for sort merge.
```

```
COMPILER ERROR <sequence number> (<internal number>):
Bad string constant.

COMPILER ERROR <sequence number> (<internal number>):
Bad TAG = <name> at (<stack address>).

COMPILER ERROR <sequence number> (<internal number>):
Bad twig for pPASCALMOD parameter.

COMPILER ERROR <sequence number> (<internal number>):
Badly formed twig in assignment statement.

COMPILER ERROR <sequence number> (internal number>):
Branch Fixup link is bad at <number>. Op code = <op code>

COMPILER ERROR <sequence number> (<internal number>):
Cannot make copy of object at (<stack address>).

COMPILER ERROR <sequence number> (<internal number>):
Case values not ordered by Language Processor.

COMPILER ERROR <sequence number> (<internal number>):
Dangling self reference.

COMPILER ERROR <sequence number> (<internal number>):
Dangling label # <number>.

COMPILER ERROR <sequence number> (<internal number>):
Did not link to your support library (<name>).

COMPILER ERROR <sequence number> (<internal number>):
Different descriptor sizes for source/destination in pARRAYCOPY.

COMPILER ERROR <sequence number> (<internal number>):
Duplicate Passed_index = <number> at (<stack address>).

COMPILER ERROR <sequence number> (<internal number>):
Error class not in 0..3.

COMPILER ERROR <sequence number> (<internal number>):
Error in handling repeat/reduce twig.

COMPILER ERROR <sequence number> (<internal number>):
Error in handling set code generation.

COMPILER ERROR <sequence number> (<internal number>):
Error in twigs for pACCEPT.

COMPILER ERROR <sequence number> (<internal number>):
Error in twigs for pTEXTOPEN.

COMPILER ERROR <sequence number> (<internal number>):
Error in twigs for pWRITESL.
```

```
COMPILER ERROR <sequence number> (<internal number>):
Error in twigs for text I/O.

COMPILER ERROR <sequence number> (<internal number>):
Error in twigs from Language Processor.

COMPILER ERROR <sequence number> (<internal number>):
First argument to ALL must be a character_type constant.

COMPILER ERROR <sequence number> (<internal number>):
Functions may not be used as procedures and vice versa.

COMPILER ERROR <sequence number> (<internal number>):
Illegal Address Couple = (<stack address>).

COMPILER ERROR <sequence number> (<internal number>):
Illegal case value.

COMPILER ERROR <sequence number> (<internal number>):
Illegal compare. Cannot compare: Left Type is <name> Right Type is <name>

COMPILER ERROR <sequence number> (<internal number>):
Illegal data initialize.

COMPILER ERROR <sequence number> (<internal number>):
Illegal Edit Operator.

COMPILER ERROR <sequence number> (<internal number>):
Illegal segment dictionary item at <number>.

COMPILER ERROR <sequence number> (<internal number>):
Improper argument for: <name>  A t_var is required, but instead <name>.

COMPILER ERROR <sequence number> (<internal number>):
Improper CELL entry at (<stack address>)

COMPILER ERROR <sequence number> (<internal number>):
Improper character intrinsic.

COMPILER ERROR <sequence number> (<internal number>):
Improper criterion.

COMPILER ERROR <sequence number> (<internal number>):
Improper file attribute or mnemonic.

COMPILER ERROR <sequence number> (<internal number>):
Improper handling of Type Stack Pointer. Bad by <number>. Operator is <name>

COMPILER ERROR <sequence number> (<internal number>):
Improper open type.

COMPILER ERROR <sequence number> (<internal number>):
Improper rebase call. Item at (<stack address>) is not type. Old TAG = <name>
New Type = <name>
```

```
COMPILER ERROR <sequence number> (<internal number>):
Improper reference argument.

COMPILER ERROR <sequence number> (<internal number>):
Improper selection expression for Perform.

COMPILER ERROR <sequence number> (<internal number>):
Improper statement kind.

COMPILER ERROR <sequence number> (<internal number>):
Improper t_pair field.

COMPILER ERROR <sequence number> (<internal number>):
Improper twig structure for vector enter.

COMPILER ERROR <sequence number> (<internal number>):
Improperly formed tree for FREEZE.

COMPILER ERROR <sequence number> (<internal number>):
Incorrect BLIND call. Type of argument is <name>  Type of BLIND is <name>

COMPILER ERROR <sequence number> (<internal number>):
Integer argument is out of range.

COMPILER ERROR <sequence number> (<internal number>):
Internal error (<number>) encountered while generating bindinfo.

COMPILER ERROR <sequence number> (<internal number>):
Internal logic error in code generation.

COMPILER ERROR <sequence number> (<internal number>):
Invalid branch label.

COMPILER ERROR <sequence number> (<internal number>):
Item at Address Couple = (<stack address>) is not a PCW.

COMPILER ERROR <sequence number> (<internal number>):
Language Processor failed to generate error message.

COMPILER ERROR <sequence number> (<internal number>):
Lex level of <number> not legal in this context.

COMPILER ERROR <sequence number> (<internal number>):
Library at (<stack address>) is already declared.

COMPILER ERROR <sequence number> (<internal number>):
Maximum address for a code segment has been exceeded.

COMPILER ERROR <sequence number> (<internal number>):
Maximum number of D2 procedures exceeded.

COMPILER ERROR <sequence number> (<internal number>):
Missing assignment in Block 1 for <variable name>.
```

```
COMPILER ERROR <sequence number> (<internal number>):
Missing End Edit Operator.

COMPILER ERROR <sequence number> (<internal number>):
NAMC in invalid context (improper TAG = <name>). Address Couple = (<stack ▯▯▯▯
address>).

COMPILER ERROR <sequence number> (<internal number>):
Op Code (<name> = <hex number>) not valid in this context.

COMPILER ERROR <sequence number> (<internal number>):
Op code or data_type not valid in this context.

COMPILER ERROR <sequence number> (<internal number>):
Param 3/4 combination invalid.

COMPILER ERROR <sequence number> (<internal number>):
Param not integer type.

COMPILER ERROR <sequence number> (<internal number>):
Param not t_var.

COMPILER ERROR <sequence number> (<internal number>):
Parameters must be declared before variables.

COMPILER ERROR <sequence number> (<internal number>):
Passed_index = <number> is too big.

COMPILER ERROR <sequence number> (<internal number>):
Poorly formed twig in field store.

COMPILER ERROR <sequence number> (<internal number>):
Reference operator is required, but instead: <name>.

COMPILER ERROR <sequence number> (<internal number>):
The compiler has requested code generation for lex level D<number> without
initializing the stack frame for D<number>.

COMPILER ERROR <sequence number> (<internal number>):
The label #<number> has been declared more than once.

COMPILER ERROR <sequence number> (<internal number>):
Trying to stretch descriptor at (<stack address>) but not descriptor.

COMPILER ERROR <sequence number> (<internal number>):
Twig for argument is wrong.

COMPILER ERROR <sequence number> (<internal number>):
Twig for value argument is bad.

COMPILER ERROR <sequence number> (<internal number>):
Unimplemented bindinfo construct (<number>).
```

```
COMPILER ERROR <sequence number> (<internal number>):
Unmatched types between Tree and Type Stack. Tree Type is <name>  ----
Stack Type is <name>.

COMPILER ERROR <sequence number> (<internal number>):
Unmatched types between Tree and Type Stack. Tree Type is <name>

Stack Type is <name>  Operator is <name>

COMPILER ERROR <sequence number> (<internal number>):
Unmatched types between Tree and Type Stack. Tree Type is <name> Stack Type is
 <name>  Variable is <variable name>

COMPILER ERROR <sequence number> (<internal number>):
VALC in invalid context (improper TAG = <name>). Address Couple = (<stack
address>).

COMPILER ERROR <sequence number> (<internal number>):
Value arrays must be passed with t_subarry or t_subpush.

COMPILER ERROR <sequence number> (<internal number>):
Word at (<stack address>) is not a data descriptor. TAG = <name>.

COMPILER ERROR <sequence number> (<internal number>):
Word at (<stack address>) is not a FIB. TAG = <name>.

COMPILER ERROR <sequence number> (<internal number>):
Word at (<stack address>) is not a PCW. TAG = <name>/

COMPILER ERROR <sequence number> (<internal number>):
Word at (<stack address>) not a library.

COMPILER ERROR <sequence number> (<internal number>):
Wrong number of parameters at call. Number in trees = <number>.
Number in argument array = <number>.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):  <function
name> cannot be unrolled. Unrecognized Primary Operator: <hex number>.

Non-fatal COMPILER ERROR <sequence number> (<internal number>): <function
name> cannot be unrolled. Unrecognized Primary Operator while scanning D3
stack building code: <hex number>.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):  This function
 has too many (<number>) arguments to be bindable.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):  <function
name> cannot be unrolled. Unrecognized Variant Operator: <hex number>.
```

Appendix J
Understanding Railroad Diagrams

This appendix explains railroad diagrams, including the following concepts:

- Paths of a railroad diagram
- Constants and variables
- Constraints

The text describes the elements of the diagrams and provides examples.

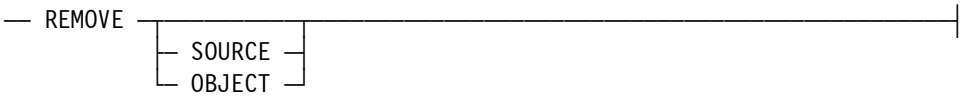
Railroad Diagram Concepts

Railroad diagrams are diagrams that show you the standards for combining words and symbols into commands and statements. These diagrams consist of a series of paths that show the allowable structures of the command or statement.

Paths

Paths show the order in which the command or statement is constructed and are represented by horizontal and vertical lines. Many commands and statements have a number of options so the railroad diagram has a number of different paths you can take.

The following example has three paths:



The three paths in the previous example show the following three possible commands:

- REMOVE
- REMOVE SOURCE
- REMOVE OBJECT

A railroad diagram is as complex as a command or statement requires. Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements.

Railroad diagrams are intended to show

- Mandatory items
- User-selected items
- Order in which the items must appear
- Number of times an item can be repeated
- Necessary punctuation

Follow the railroad diagrams to understand the correct syntax for commands and statements. The diagrams serve as quick references to the commands and statements.

The following table introduces the elements of a railroad diagram:

Table J–1. Elements of a Railroad Diagram

The diagram element...	Indicates an item that...
Constant	Must be entered in full or as a specific abbreviation
Variable	Represents data
Constraint	Controls progression through the diagram path

Constants and Variables

A constant is an item that must be entered as it appears in the diagram, either in full or as an allowable abbreviation. If a constant is partially boldfaced, you can abbreviate the constant by

- Entering only the boldfaced letters
- Entering the boldfaced letters plus any of the remaining letters

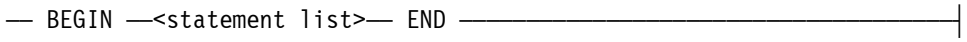
If no part of the constant is boldfaced, the constant cannot be abbreviated.

Constants are never enclosed in angle brackets (< >) and are in uppercase letters.

A variable is an item that represents data. You can replace the variable with data that meets the requirements of the particular command or statement. When replacing a variable with data, you must follow the rules defined for the particular command or statement.

In railroad diagrams, variables are enclosed in angle brackets.

In the following example, BEGIN and END are constants, whereas <statement list> is a variable. The constant BEGIN can be abbreviated since it is partially boldfaced.



Valid abbreviations for BEGIN are

- BE
- BEG
- BEGI

Constraints

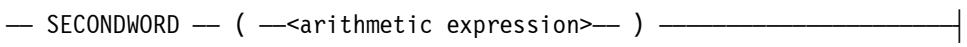
Constraints are used in a railroad diagram to control progression through the diagram. Constraints consist of symbols and unique railroad diagram line paths. They include

- Vertical bars
- Percent signs
- Right arrows
- Required items
- User-selected items
- Loops
- Bridges

A description of each item follows.

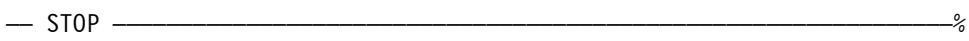
Vertical Bar

The vertical bar symbol (|) represents the end of a railroad diagram and indicates the command or statement can be followed by another command or statement.



Percent Sign

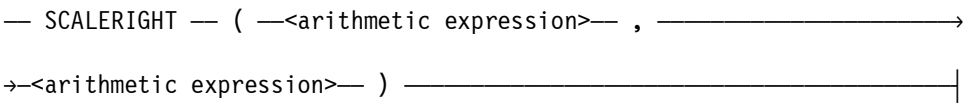
The percent sign (%) represents the end of a railroad diagram and indicates the command or statement must be on a line by itself.



Right Arrow

The right arrow symbol (>)

- Is used when the railroad diagram is too long to fit on one line and must continue on the next
- Appears at the end of the first line, and again at the beginning of the next line



Required Item

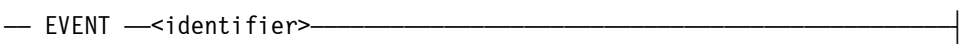
A required item can be

- A constant
- A variable
- Punctuation

If the path you are following contains a required item, you must enter the item in the command or statement; the required item cannot be omitted.

A required item appears on a horizontal line as a single entry or with other items. Required items can also exist on horizontal lines within alternate paths, or nested (lower-level) diagrams.

In the following example, the word EVENT is a required constant and <identifier> is a required variable:



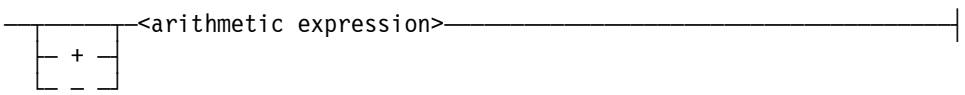
User-Selected Item

A user-selected item can be

- A constant
- A variable
- Punctuation

User-selected items appear one below the other in a vertical list. You can choose any one of the items from the list. If the list also contains an empty path (solid line) above the other items, none of the choices are required.

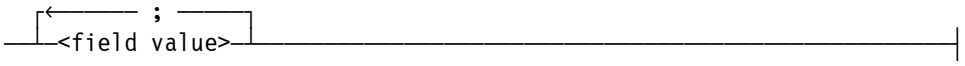
In the following railroad diagram, either the plus sign (+) or the minus sign (–) can be entered before the required variable <arithmetic expression>, or the symbols can be disregarded because the diagram also contains an empty path.



Loop

A loop represents an item or a group of items that you can repeat. A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, connected on both sides by vertical lines. The top line is a right-to-left path that contains information about repeating the loop.

Some loops include a return character. A return character is a character—often a comma (,) or semicolon (;)—that is required before each repetition of a loop. If no return character is included, the items must be separated by one or more spaces.



Bridge

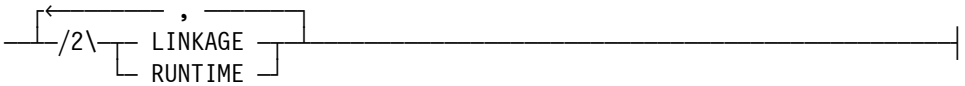
A loop can also include a bridge. A bridge is an integer enclosed in sloping lines (/ \) that

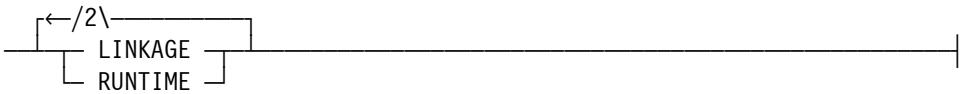
- Shows the maximum number of times the loop can be repeated
- Indicates the number of times you can cross that point in the diagram

The bridge can precede both the contents of the loop and the return character (if any) on the upper line of the loop.

Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.

In the first bridge example, you can enter LINKAGE or RUNTIME no more than two times. In the second bridge example, you can enter LINKAGE or RUNTIME no more than three times.





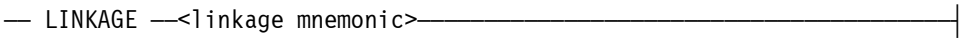
In some bridges an asterisk (*) follows the number. The asterisk means that you must cross that point in the diagram at least once. The maximum number of times that you can cross that point is indicated by the number in the bridge.

In the previous bridge example, you must enter LINKAGE at least once but no more than twice, and you can enter RUNTIME any number of times.

Following the Paths of a Railroad Diagram

The paths of a railroad diagram lead you through the command or statement from beginning to end. Some railroad diagrams have only one path; others have several alternate paths that provide choices in the commands or statements.

The following railroad diagram indicates only one path that requires the constant LINKAGE and the variable <linkage mnemonic>:



Alternate paths are provided by

- Loops
- User-selected items
- A combination of loops and user-selected items

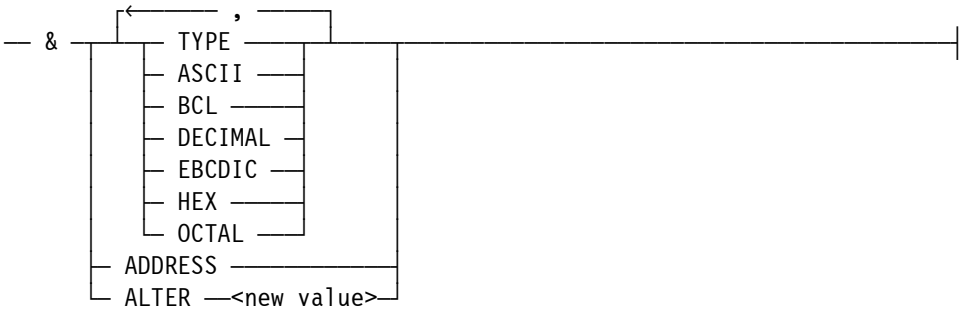
More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes

- An ampersand (&)
- Constants that are user-selected items

These constants are within a loop that can be repeated any number of times until all options have been selected.

The first alternative path requires the ampersand and the required constant ADDRESS. The second alternative path requires the ampersand followed by the required constant ALTER and the required variable <new value>.

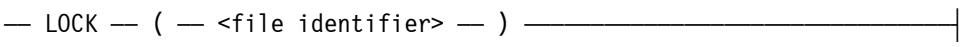


Railroad Diagram Examples with Sample Input

The following examples show five railroad diagrams and possible command and statement constructions based on the paths of these diagrams.

Example 1

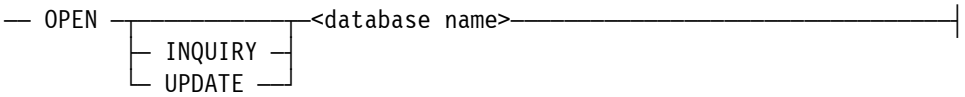
<lock statement>



Sample Input	Explanation
LOCK (FILE4)	LOCK is a constant and cannot be altered. Because no part of the word is boldfaced, the entire word must be entered. The parentheses are required punctuation, and FILE4 is a sample file identifier.

Example 2

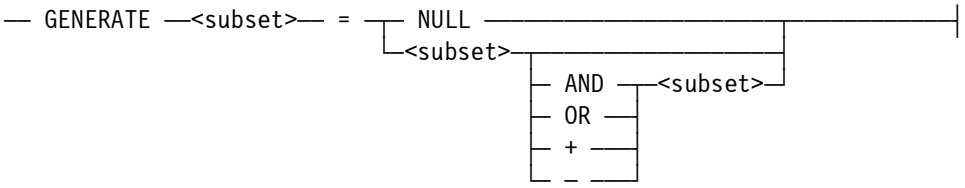
<open statement>



Sample Input	Explanation
OPEN DATABASE1	The constant OPEN is followed by the variable DATABASE1, which is a database name. The railroad diagram shows two user-selected items, INQUIRY and UPDATE. However, because an empty path (solid line) is included, these entries are not required.
OPEN INQUIRY DATABASE1	The constant OPEN is followed by the user-selected constant INQUIRY and the variable DATABASE1.
OPEN UPDATE DATABASE1	The constant OPEN is followed by the user-selected constant UPDATE and the variable DATABASE1.

Example 3

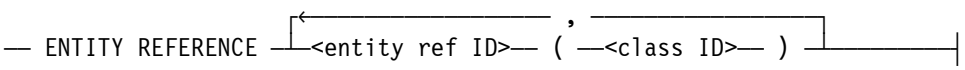
<generate statement>



Sample Input	Explanation
GENERATE Z = NULL	The GENERATE constant is followed by the variable Z, an equal sign (=), and the user-selected constant NULL.
GENERATE Z = X	The GENERATE constant is followed by the variable Z, an equal sign, and the user-selected variable X.
GENERATE Z = X AND B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the AND command (from the list of user-selected items in the nested path), and a third variable, B.
GENERATE Z = X + B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the plus sign (from the list of user-selected items in the nested path), and a third variable, B.

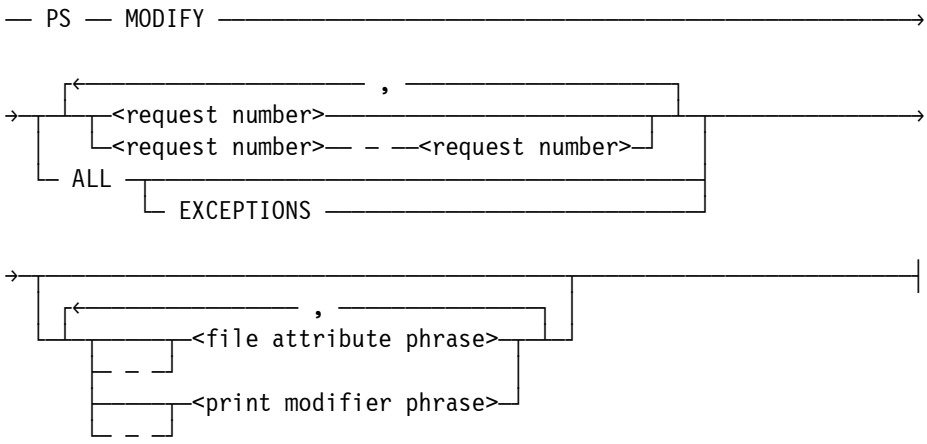
Example 4

<entity reference declaration>



Sample Input	Explanation
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR)	The required item ENTITY REFERENCE is followed by the variable ADVISOR1 and the variable INSTRUCTOR. The parentheses are required.
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR), ADVISOR2 (ASST_INSTRUCTOR)	Because the diagram contains a loop, the pair of variables can be repeated any number of times.

Example 5



Sample Input	Explanation
PS MODIFY 11159	The constants PS and MODIFY are followed by the variable 11159, which is a request number.
PS MODIFY 11159,11160,11163	Because the diagram contains a loop, the variable 11159 can be followed by a comma, the variable 11160, another comma, and the final variable 11163.
PS MOD 11159–11161 DESTINATION = "LP7"	The constants PS and MODIFY are followed by the user-selected variables 11159–11161, which are request numbers, and the user-selected variable DESTINATION = “LP7”, which is a file attribute phrase. Note that the constant MODIFY has been abbreviated to its minimum allowable form.
PS MOD ALL EXCEPTIONS	The constants PS and MODIFY are followed by the user-selected constants ALL and EXCEPTIONS.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

8600 2268–206