CONVEX C Language Reference Manual

Document No. 720-001230-001

First Edition
May 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX C Language Reference Manual
Order No. DSW-088
Preliminary Edition

Printed in the United States of America

Revision Information for
CONVEX C Language Reference Manual

Edition	Document No.	Description
First	720-001230-001	Released with CONVEX C V4.0, May 1990. First release of the manual.

Table of Contents

1 Compatibility Modes
Description of Four Modes 1-1
Compiling 1-2
Single Mode Compilation Examples 1-2
Mixed Compatibility Modes 1-3

2 Data Types and Representations
Integral Types 2-1
Floating-Point Types 2-5
Pointer 2-8
void 2-8
union 2-9
struct Data Type and Representation 2-10
Array Data Type and Representation 2-11

3 Mixed-Language Programming
C Function Calls 3-1
Accessing FORTRAN Routines From C 3-5
Accessing Ada Routines from C 3-9

4 Input and Output
File Input and Output Concepts 4-1
Program Input and Output 4-3
Program Input and Output Examples 4-3

5 Runtime Library
Functions Versus Function-like Macros 5-2
Calling Runtime Functions 5-3
assert.h 5-3
ctype.h 5-4
errno.h 5-5
float.h 5-6
limits.h 5-8
locale.h 5-10
math.h 5-11
setjmp.h 5-14
signal.h 5-15
stdarg.h 5-18
stddef.h 5-18
stdio.h 5-19
stdlib.h 5-25
string.h 5-29
time.h 5-32

6 The Preprocessor
Preprocessor Directives 6-1

7 The asm Statement
Assembly-Language Statements 7-1

Appendices

A Compiler Directives A-1
Information Directives A-2
Control Directives A-4
B Reporting Problems B-1
Technical Assistance Center B-1
The contact Utility B-1
Prerequisites B-1
Tips on Using the contact Utility B-3
Using the contact Utility B-4

List of Tables

1-1 Compatibility Modes 1-1
2-1 Integral Type Bit Length 2-1
2-2 Integral Ranges 2-2
2-3 long long data type range 2-3
2-4 Floating-Point Bit Length 2-5
2-5 Native and IEEE Floating-Point Ranges 2-5
2-6 Native and IEEE Floating-Point Ranges 2-5
2-7 long float Range: Native and IEEE 2-8
3-1 FORTRAN and C Declarations 3-6
5-1 Compatibility Modes 5-1
5-2 Math Function Return Values 5-13
5-3 errno values of fgetpos, fsetpos and ftell 5-23
A-1 Restrictions on Directive Use A-1
A-2 Maximum Strip-Mine Lengths A-7

List of Figures

2-1 char Representation 2-2
2-2 short int Representation 2-2
2-3 int Representation 2-3
2-4 long long Representation 2-4
2-5 Single-Precision Floating Representation 2-6
2-6 Double-Precision Floating Representation 2-7
2-7 Pointer Representation 2-8
2-8 Character Data Representation 2-12
2-9 Character String Representation 2-13
3-1 Top of the Runtime Stack 3-2
3-2 Stack Layout 3-3

Preface

Purpose and Audience

This manual is a reference for CONVEX C. It is a *preliminary* edition because it does *not* contain a definition of the C language. While chapters 1, 4, 5, and 6 are suitable for inexperienced C programmers, the remaining chapters may be useful only to experienced programmers.

This manual is intended for people who are familiar with:

- *CONVEX C User's Guide*
- *CONVEX ANSI C Concepts*

These documents provide useful background material.

Organization

This manual is organized as follows:

- Chapter 1, "Compatibility Modes," provides a discussion of the four compatibility modes of the compiler, what their uses are, and how to mix them.
- Chapter 2, "Data Types and Representations," lists data types and their representations.
- Chapter 3, "Mixed Language Programming," describes the method that CONVEX C uses to call C functions and the interaction of CONVEX C with CONVEX FORTRAN and CONVEX Ada.
- Chapter 4, "Input and Output," discusses some of the functions that provide input and output for a program.
- Chapter 5, "Runtime Libraries," provides a brief summary of the functions in the ANSI C header files.
- Chapter 6, "The Preprocessor." describes directives the C preprocessor recognizes.
- Chapter 7, "The asm Statement," describes the syntax of the `asm` statement, a CONVEX extension that inserts assembly-language statements in a C program.
- Appendix A describes the compiler optimization directives.
- Appendix B provides instructions for reporting software and documentation problems to the CONVEX Technical Assistance Center (TAC).

Notational Conventions

The following conventions are used in this document:

- Words enclosed in rounded rectangles are keyboard keys that you press. For example, CTRL-g denotes the carriage return key. Words separated by a hyphen and enclosed in rounded rectangles indicate two keys that you must press simultaneously. For example, indicates that you must press the
  key while simultaneously pressing the keyboard CTRL-X character key.
- The word "enter" in a phrase such as "enter a command" means that you type the command and press the carriage return key. In contrast, the word "type" (for example, "type a line of text") means that you do not press the carriage return key.
- *Italics* designate user-suplied variables in a command-line example, introduce new terms of great importance, identify variables in mathematical equations, and indicate titles of documents.
- `Constant-width font` is used for input and output. This includes: command names and options, system calls, data structures and types, directives, program statements, display examples, printout examples, and error messages returned.
- **Bold font** is used to clearly identify user input in examples.
- Within command sequences:
  - Square brackets ([]) indicate optional input.
  - Curly brackets ({}) designate mandatory input, which must be one of two or more possible options. These options are separated by the pipe symbol (|).
  - Horizontal ellipsis (...) shows repitition of the preceding item(s).

    Consider the following example:

    COMMAND *input_file* [...]{a | b}[*output_file*]

    where COMMAND must be typed as it appears; *input_file* indicates a file name that must be supplied by you; the horizontal ellipsis in brackets indicates that additional input file names may be supplied; either `a` or `b` must be supplied; and *output_file* indicates an optional file name.
- References to the *ConvexOS Programmer's Reference* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

Associated Documents

Using CONVEX C successfully may require information not specific to topics described herein or not within the scope of this document.

The following documents are provided by CONVEX Computer Corporation to help you with CONVEX C:

- *CONVEX C User's Guide* describes how to compile programs with the CONVEX C compiler.
- *CONVEX ANSI C Concepts* is a high level introduction to the ANSI C standardized programming language. It also explains the best approach to convert applications to CONVEX C.
- *CONVEX C Optimization Guide* is a reference to methods that can be used to optimize a program.
- *CONVEX C Quick Reference* provides quick access to function prototypes, compiler directives, compiler options, and language features.
- *ConvexOS Programmer's Reference* is the 'standard reference for the ConvexOS operating system.

For more information on the C language, refer to the following books:

- *American National Standard for Information Systems -- Programming Language C.* Document Number: X3J11/90-013.
- *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele, Jr., Prentice-Hall, Inc., 1987.
- *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

Ordering Documentation

To order CONVEX documentation, complete the CONVEX Documentation and Subscription Service Order Form enclosed in the Documentation Catalog included with this manual.

To receive a specific edition of a manual, contact the local CONVEX sales office or call the Technical Assistance Center (TAC).

Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

Within the continental U.S.	1(800)952-0379.
From locations in Alaska, Hawaii, and Canada	1(214)497-4379.
From all other locations	contact the nearest CONVEX office.

Reader's Forum

If you want to mail your comments to us, please use the form at the end of this manual and list the document page number with your questions and comments.

Chapter 1
Compatibility Modes

This chapter describes compatibility modes of the compiler and how to compile source code in each mode. Examples are provided. Finally, the reason for combining compatibility modes is given and the method of combining them is explained.

Description of Four Modes

CONVEX C provides four compatibility modes as shown in Table 1-1.

Table 1-1: Compatibility Modes

Mode	Language	Default Functions
Extended	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	ANSI C	ANSI C, POSIX
Strict	ANSI C	ANSI C
Backward-compatible	non-ANSI C	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first of the POSIX standards to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at the source-code level.

The four modes differ in two areas: language specification and library functions. For example, an application that is compiled in the strict mode uses only ANSI C language features. The only library functions that are automatically linked into a program are those in the ANSI C library.

One language feature that can be used in the extended mode but not in the strict or conforming modes is the `asm` statement. This is a CONVEX extension that directs the compiler to insert in-line assembly-language statements into object code. Another CONVEX extension that is not permitted in either the conforming or strict modes is the data type `long long int`. This data type is a 64-bit integer. The extended mode is the default compatibility mode of the compiler. It sacrifices some portability when it uses CONVEX specific language features and library functions.

The major difference between the conforming mode and the strict mode is the functions that can be automatically linked into the application program: the conforming mode can link in POSIX functions while the strict mode can only link in ANSI C functions. These two modes use the same language specification because POSIX only defines an interface.

The backward-compatible mode differs considerably from the other three modes with respect to language features and functions. It does not recognize new ANSI C keywords. Further, it does not use ANSI C or POSIX functions. The only benefit of using this mode is that it is closely compatible with previous CONVEX C compilers.

Compiling

Specifying a compatibility mode with the compiler is straightforward; the suitable compiler option is included on the command line. The following examples demonstrate compiling the source file applic.c in each of the four modes:

extended	`cc applic.c`
conforming	`cc -std -D_POSIX_SOURCE applic.c`
strict	`cc -str applic.c`
backward-compatible	`cc -pcc applic.c`

The macro constant _POSIX_SOURCE defined on the command line for the conforming mode provides access to POSIX function prototypes in ANSI C header files.

Applications that conform to the POSIX standard must define this constant on the first line of every source file. Applications that do not need to conform to the POSIX standard may define it on the command line, as in the example. This constant is not automatically defined in the conforming mode because conforming POSIX applications **must** define it in the source code.

Another macro constant that has a similar purpose is _CONVEX_SOURCE. This macro provides access to the CONVEX function prototypes in the ANSI C header files. Both _POSIX_SOURCE and _CONVEX_SOURCE are automatically defined when a source file is compiled in the default mode. The _CONVEX_SOURCE macro may be used **only** if the _POSIX_SOURCE macro is also defined.

Single Mode Compilation Examples

Examples contained in this section demonstrate how to compile a program for each of the four modes of the compiler. For all the following examples, assume that the `CCOPTIONS` environment variable is null. The following example compiles an application for the extended mode:

```
cc applic.c
```

Because _CONVEX_SOURCE and _POSIX_SOURCE are defined automatically, function prototypes for CONVEX and POSIX functions are accessible in the ANSI C header files.

Consider the following example:

```
cc -D_POSIX_SOURCE -std applic.c
```

This command line compiles a program for the conforming mode of the compiler. No CONVEX extensions are used; only ANSI C language features are permitted. The `-std` compiler option indicates that only the POSIX and ANSI C system functions are automatically linked into the executable program.

The following example compiles an application that strictly conforms to the ANSI C specification:

```
cc -str applic.c
```

The following command line compiles a program in the backward-compatible mode of the compiler:

```
cc -pcc applic.c
```

Use of either the _POSIX_SOURCE macro or the _CONVEX_SOURCE macro in the backward-compatible mode causes compilation to halt because these macros provide access to constructs that are not recognized by the backward-compatible mode of the compiler. Several compiler options are incompatible with this mode. Refer to *CONVEX C User's Guide*, Chapter 2, "Compiler Fundamentals" for more information.

Mixed Compatibility Modes

Desirable features of each ANSI C mode are:

Extended mode	CONVEX extensions to the language.
Extended mode	functions that use CONVEX hardware.
Conforming mode	POSIX functions.
Strict mode	requires strict adherence to the ANSI C standard.

These, and other features, can be combined to tailor an application.

Two command lines are required to compile and link an application in a mixed compatibility mode. The first command line translates the application into object code, specifying the language features and providing access to information in the appropriate header files. The second command line indicates which libraries are linked into the executable program.

Three language specifications can be used:

- Strict ANSI C.
- ANSI C with CONVEX extensions.
- Backward-compatible C.

These language specifications are obtained by compiling an application with the strict, extended, and backward-compatible compatibility modes, respectively.

Similarly, there are four library systems:

- ANSI C functions.
- ANSI C and POSIX functions.
- ANSI C, POSIX, and CONVEX functions.
- Backward-compatible functions.

Linking with the strict, conforming, extended, and backward-compatible compatibility modes, respectively, provides access to these library systems.

For example, you might want to develop a maximally portable CONVEX program. Such a program uses functions that are specific to CONVEX hardware, but is strict in its interpretation of the language. Use the following command lines:

```
cc -D_POSIX_SOURCE -D_CONVEX_SOURCE -str -c applic.c
cc applic.o
```

The first command line provides function prototypes to CONVEX functions, while maintaining strict interpretation of the language. The _CONVEX_SOURCE macro may be used only if the _POSIX_SOURCE macro is defined, *even if no POSIX functions are used.* The second command line specifies that the ANSI C, POSIX, and CONVEX extension library system is used. Thus it is possible to tailor the compatibility modes by using two command lines.

Chapter 2
Data Types and Representations

This chapter describes data types that are supported by the CONVEX C compiler. These data types are:

- Character
- Integer
- Enumeration
- Floating-point
- Pointer
- Void
- Union
- Structure
- Array.

Data types that the compiler supports can be divided into three categories: those that are required by ANSI C, those that are CONVEX extensions, and those that are accepted in the backward-compatible mode of the compiler. Refer to Chapter 1, "Compatibility Modes," for more information on compatibility modes.

Each discussion of a data type includes a description and a representation; examples clarify certain situations. A few data types that are not accepted by an ANSI C compiler are noted. For each internal data representation, numbers at the top of the figure represent bit numbers; numbers at the bottom represent byte offsets. The least significant bit in the data representation is numbered 0.

Integral Types

char and int

The ANSI C compiler supports two integral data types: `char` and `int`. The bit length of each data type is shown in Table 2-1.

Table 2-1: Integral Type Bit Length

Data Type	Length
`char`	8-bit integer
`short int` or `short`	16-bit integer
`long int` or `int` or `long`	32-bit integer

By default, each data type is assumed to be a signed value. The range of numbers that these data types can represent is listed in Table 2-2.

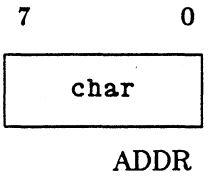
Table 2-2: Integral Ranges

Data Type	Signed Range	Unsigned Range
`char`	-128 ... 127	0 ... 255
`short int` or `short`	-32768 ... 32767	0 ... 65535
`long int`, `int`, or `long`	-2147483648 ... 2147483647	0 ... 4294967295

As ranges for the `char` data type imply, integer arithmetic can be performed on characters. The `char` representation contains a single character in the execution character set, ASCII.

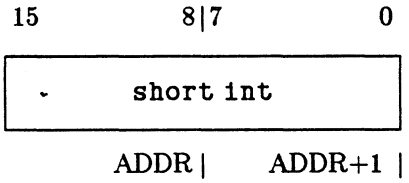
Figure 2-1 illustrates the internal representation of the `char` data type.

Figure 2–1: char Representation



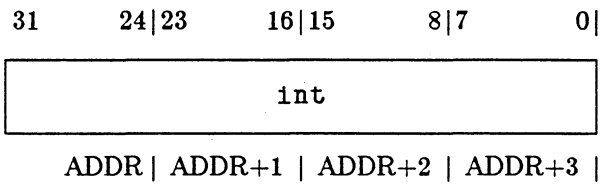
The internal representation of the `short int` data type is shown in Figure 2-3.

Figure 2–2: short int Representation



A 32-bit integer variable is declared as `int` (or `long int`) and may be unsigned or signed. Figure 2-3 shows the `int` data representation.

Figure 2–3: int Representation



`long long int`

This data type is an integer type that is implemented in 64 bits. The range for the `long long` data type is shown in Table 2-3.

Table 2-3: long long data type range

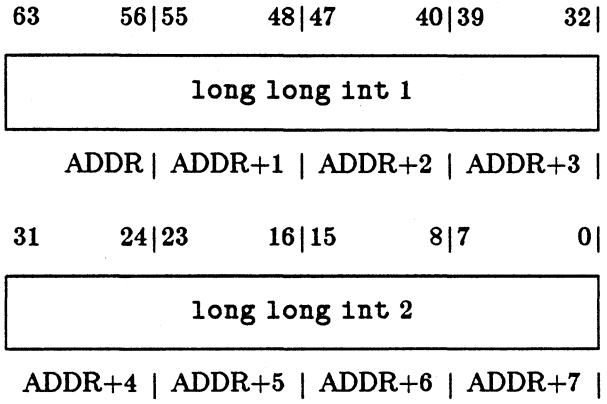
Format	long long or long long int
signed	-9223372036854775808 ... 9223372036854775807
unsigned	0 ... 18446744073709551615

Note

The `long long` data type is a CONVEX extension to the ANSI C standard. It must not be used by programs that must be ported to other compilers. This data type is accepted in the backward-compatible mode of the compiler. For more information on the compatibility modes of the compiler, refer to Chapter 1, "Compatibility Modes."

Figure 2-4 shows the `long long` data representation.

Figure 2–4: long long Representation



In the backward-compatible mode `long long` integers cannot be passed as arguments to functions that do not expect to receive them. In general, `char` and `short` values can be passed to routines that expect `int` or `long` values. CONVEX C converts 8- and 16-bit quantities to a 32-bit format before pushing them onto the runtime stack. Because 64-bit values are not truncated, improper stack alignment results when `long long int` values are passed to routines that expect `int` arguments.

Enumerated Types

An enumerated data type is a user-defined integral data type. Declare enumerated scalar data as `enum`. For example, you might declare the enumerated data type `color` as:

```
enum color { red, blue, green } hue;
```

In this example, the variable `hue` could be only one of the values (`red`, `blue`, or `green`) at any given time.

Internally, `enum` values are stored as integer representations. By default, the first enumerated value (`red` in the example above) is stored with the ordinal value of zero. Subsequent enumerated values are represented by sequential integer values. In the example shown above, the value of `blue` is 1 and `green` is 2.

Default ordinal values are overridden when they are followed by an equal sign and a new ordinal (for example: `enum color {red=10, blue=20,green=30}`). Because the `enum` data type is implemented as a `signed int`, the range of ordinal values available for use are the same as those used by a `signed int` in Table 2-2.

The data representation for the `enum` data type is the same as that for the `signed int` data type, which is shown in Figure 2-2.

Floating-Point Types

The ANSI C compiler supports three floating-point data types: `float`, `double`, and `long double`. The bit length of each of these data types is listed in Table 2-4.

Table 2-4: Floating-Point Bit Length

Data Type	Length
`float`	32-bit floating-point
`double` and `long double`	64-bit floating-point

Floating-point data can be represented in either CONVEX native format or IEEE format. To process floating-point data in IEEE mode, you must specify the correct option for the compiler (Refer to *CONVEX C User's Guide*, Chapter 2 "Compiler Fundamentals.") and your machine must have IEEE support hardware.

Note

CONVEX hardware and runtime libraries support only the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

The range of values provided with each format is shown in Tables 2-5 and 2-6.

Table 2-5: Native and IEEE Floating-Point Ranges

Format	float
Native	$2.9387359 \times 10^{-39}$ ... $1.7014117 \times 10^{+38}$
IEEE	$1.1754944 \times 10^{-38}$ ... $3.4028235 \times 10^{+38}$

Table 2-6: Native and IEEE Floating-Point Ranges

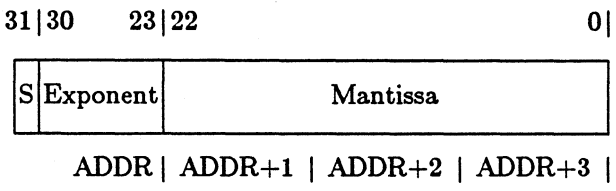
Format	double and long double
Native	$5.562684646268003 \times 10^{-309}$ ... $8.988465674311584 \times 10^{+307}$
IEEE	$2.225073858507201 \times 10^{-308}$ ... $1.797693134862317 \times 10^{+308}$

Floating-Point Representation: float

Single-precision (32-bit) floating-point variables are declared with the `float` keyword.

Positioning of the sign, exponent, and mantissa apply to native and IEEE formats; the particulars of each format are described in Figure 2-5.

Figure 2–5: Single-Precision Floating Representation



Single-Precision Native

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 128; that is, 128 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the left of the implicit 1 bit.

Single-Precision IEEE

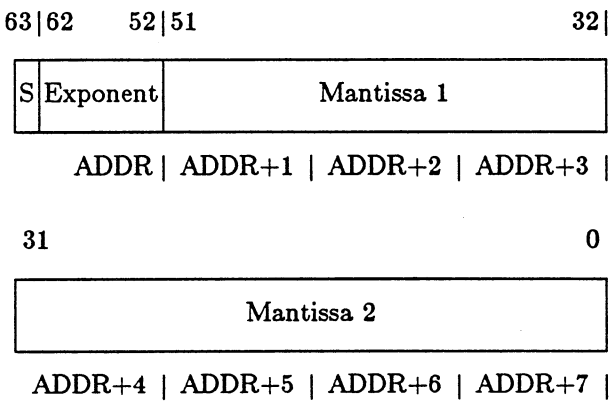
In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the right of the implicit 1 bit.

Floating-Point Representation: double, long double

Double-precision (64-bit) floating-point variables are declared with the `double` or `long double` keyword and can be represented in either native format or IEEE format. To process floating-point data in IEEE mode, the machine must have IEEE support hardware.

Figure 2-6 shows the internal representation of double-precision floating-point data. The position of the sign, exponent, and mantissa apply to native and IEEE formats; particulars of each format are described in Figure 2-6.

Figure 2–6: Double-Precision Floating Representation



Double-Precision Native

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1024; that is, 1024 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the left of the implicit 1 bit.

Double-Precision IEEE

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the right of the implicit 1 bit.

Floating-Point Type: long float

> **Note** The `long float` data type is a floating-point type supported only by the backward-compatible mode of CONVEX C. It should not be used by programs that will be ported to other computer systems.

This type is synonymous with the `double` and `long double` data types. The range for this type is detailed in Table 2-7.

Table 2-7: long float Range: Native and IEEE

Format	long float
Native	$5.562684646268003x10^{-309}$ ... $8.988465674311584x10^{+307}$
IEEE	$2.225073858507201x10^{-308}$ ... $1.797693134862317x10^{+308}$

Pointer

A pointer is a variable that contains a 32-bit address. An asterisk is used to declare a pointer. For example, the declaration

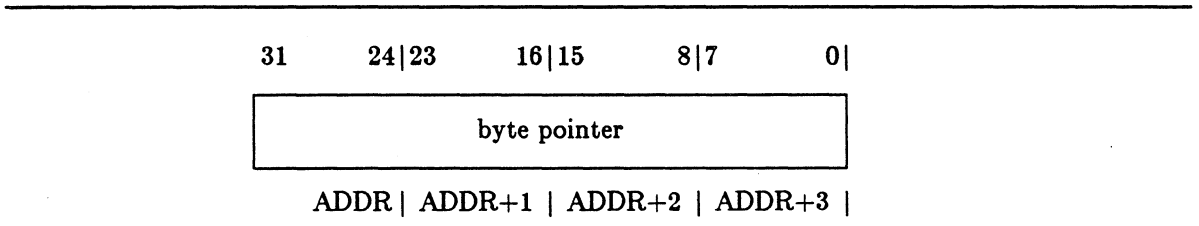
```
char *cp;
```

designates a pointer named `cp` that may be assigned the address of a `char` variable. All pointers defined in CONVEX C refer to the location of a byte in memory. Pointers have the same range of possible values as the `unsigned int` data type. Not all possible unsigned integer values, however, may be used as valid pointers.

While it is not an error for a pointer variable to contain the address of an invalid memory location, it is an error for the program to attempt to access the contents of the address to which such a pointer refers. Also, it is an error for a program to manipulate the address of a null (0) pointer.

The data representation of a pointer is shown in Figure 2-7. Word-aligned pointers have zeros as the two least-significant bit positions; halfword-aligned pointers have a zero in the least-significant bit position. Aligned addresses usually result in faster program execution because data with aligned addresses can be encached in high-speed memory by CONVEX hardware. The compiler attempts to keep addresses properly aligned.

Figure 2–7: Pointer Representation



void

The `void` data type has no values and performs no operations. This data type specifies the return type of a function that returns no value and can also be used to discard a return value. This type has no data representation.

The `void *` data type is a generic data pointer that can be used to point to any data type. It is frequently used in function prototypes when the actual return type or parameter is unknown. For example, the function prototype of malloc is:

```
extern void *malloc(size_t);
```

To obtain a pointer to a `char` object, the return type must be cast:

```
char *cp;
cp = (char *) malloc(4);
```

The representation of the `void *` data type is the same as that for a pointer shown in Figure 2-7.

union

The `union` data type permits different data types to be assigned the same storage location. For example, a 32-bit integer can occupy the same memory location as a 32-bit floating-point number. The programmer must use the correct interpretation is used program.

One common use of the `union` data type is to provide access to individual bytes in a variable. For example:

```
union {
    unsigned short word_a;
    struct {
        unsigned char hi_byte;
        unsigned char lo_byte;
    } a;
} reg;
```

High and low bytes may be manipulated as individual quantities. If `reg.a.lo_byte` is 255, adding one to its value does not affect the high byte. In contrast, adding one to the value of reg.word_a changes the value of both the high and the low bytes.

The representation of the `union` data type depends on the size of the largest member in the union. If the largest member of a union type is a `long int`, the `union` type occupies 4 bytes of storage. The address of each member of the `union` is the same as the address of the `union` type identifier.

struct Data Type and Representation

A structure is a collection of heterogenous data items that are grouped together under a single name.

ANSI C allows the assignment of one structure to another via the "=" operator. Given the declarations

```
struct employee {
    char name [40];
    int age;
    char sex;
    };
struct employee new_emp = {"john smith", 31, 'm'},
        old_emp;
```

`old_emp` can be assigned the values from `new_emp` with the single assignment statement

```
old_emp = new_emp;
```

rather than the three statements

```
strcpy( &old_emp.name, &new_emp.name );
old_emp.age = new_emp.age;
old_emp.sex = new_emp.sex;
```

ANSI C also supports the use of structures in function calls and returns. Structures are passed to functions by value. The entire structure is pushed onto the stack. If `new_emp` is passed to a function update_emp, 48 bytes are pushed on the stack. The extra three bytes maintain stack alignment for performance reasons.

Functions can also be declared to return structure values. For example,

```
struct employee update_emp( struct employee );
new_emp = update_emp( old_emp );
```

The alignment of members within a structure depends on the data types of the members. That is, an `int` member does not cross a 32-bit aligned boundary. This does not imply that all `int` members are aligned on 32-bit boundaries. For example, two 16-bit `int` members fit in the same 32-bit package.

Boundaries for members within structures are the same as the alignment values for variables on the runtime stack.

Structure Padding

The padding and alignment for members of structures are as follows:

- `char`, `unsigned char`, and `signed char` are aligned on byte boundaries.
- `short`, `unsigned short`, and `signed short` are aligned on even byte boundaries.
- `int`, `unsigned int`, and `signed int` are aligned on 4-byte boundaries.
- `long`, `unsigned long`, and `signed long` are aligned on 4-byte boundaries.
- `float` is aligned on 4-byte boundaries.
- `double` and `long double` are aligned on 4-byte boundaries.
- The alignment of arrays is dictated by the alignment of each element in the array.
- Structures and unions are aligned as required by the most restrictive member.
- Bit fields are allocated in blocks of size `int`. They begin at the high-order location of storage. Bit fields that span an `int` boundary are placed in the following `int` storage location.

Array Data Type and Representation

An array is an aggregation of data in which all items are one data type. The items are arranged in contiguous memory locations.

Arrays have between 1 and 14 dimensions. For example

```
int four_dim [4][5][6][7];
```

is a declaration of a four-dimensional array that contains 840 items. Arrays are stored in row-major order. All arrays are zero based; the first element of array `A` is `A[0]`.

Array Addresses

Like more primitive data types, the address of an individual element in an array is specified using the & operator. For example, the address of the third item in a single dimensioned array named `eigen` is:

```
&eigen[2];
```

Consequently, the base address of the array can be specified as `&eigen[0]`. However, the base address of the array may also be specified as `eigen` or `&eigen`. When arrays are passed to a function, it is the array name that specifies the array:

```
int some_array[12];          /* array declaration */
void use_array( int [] );    /* function prototype */

use_array( some_array );     /* pass the array */
```

C does not check boundary-values on arrays; thus, ensure that you do not make illegal array references in `use_array`.

Pointers to Arrays

As noted in the previous section, the name of the array is a pointer to the base address of an array. Unlike a pointer, the address of the array cannot be modified. For example, the statement

```
array_name += 5;
```

is illegal. The name of an array can never be the recipient of an assignment.

However, modifying a pointer to an array is legal. If a pointer is declared as pointing to the same type of data that is contained in the array, that pointer may point to individual elements in the array and assignments may be made to that pointer. If the pointer is declared to be a different type from the elements in the array, arithmetic operations with the pointer will probably not produce the desired results.

String Representation

A special case of an array representation is the *string* data type. For example,

```
char filename[] = "main.c";
```

defines an array of `char` that contains the filename `main.c`. Each byte in the array contains an ASCII character code. The NULL byte is automatically appended to the string. The NULL byte indicates the end of the string; it is expected by many system functions which have parameters that are strings.

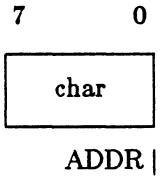
If a string is created by hand, as in:

```
char filename[7];
filename[0] = 'm'; filename[1] = 'a'; filename[2] = 'i';
filename[3] = 'n'; filename[4] = '.'; filename[5] = 'c';
filename[6] = NULL;
```

the NULL character *must* be appended to the string by hand. This character is automatically appended only when arrays are initialized to a string or in some functions such as strcat and strcpy.

Figure 2-8 shows the representation of this data type.

Figure 2–8: Character Data Representation

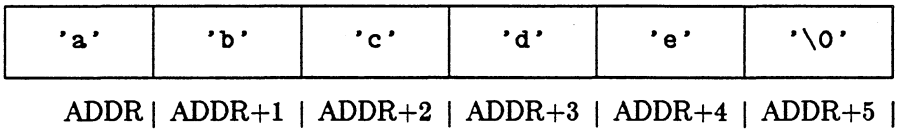


Single-character constants in C are delimited by apostrophes. ASCII codes are specified as `char` variables when you place the one- to three-digit octal number representing the desired character code preceded by a backslash (\ ) character between apostrophes, as in `'\64'`.

Arrays of character data are stored in ascending memory addresses, regardless of 32-bit word

boundaries. Figure 2-9 shows the configuration of the string "abcde" in memory.

Figure 2–9: Character String Representation



Chapter 3
Mixed-Language Programming

This chapter explains how to access routines written in other languages. Conventions used to call routines written in C are defined. Then, accessing routines in other languages based on the C model of function-calling is discussed.

C Function Calls

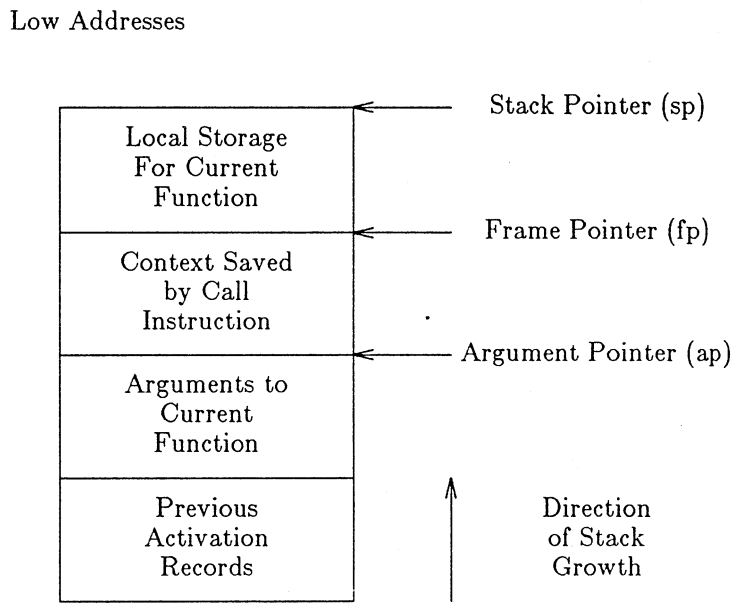
When C function calls are executed, the current state of several hardware registers used by the calling function must be preserved. These registers are the processor status word, the frame pointer (fp), and the argument pointer. Contents of these registers are pushed on the runtime stack as a part of an activation record. The only register that the called function **must** preserve is the frame pointer register.

The following sections discuss this process in more detail.

Function Stack Layout

Figure 3-1 shows the top of the runtime stack. The stack pointer (sp) register contains the address of the topmost location on the runtime stack. The fp register contains the address of the last frame pushed on the runtime stack by a `call` or `calls` assembly-language instruction. The argument pointer (ap) register contains the address of the arguments to the current function.

Figure 3–1: Top of the Runtime Stack



Standard Calling Sequence

A function call requires the following steps:

1. Push values of arguments to the function on the runtime stack in reverse order.
2. Update the ap register. The updated register should point to the first argument in the argument list. The first argument in the list is the last one pushed.
3. Push an additional word. This word must contain the number of arguments passed.
4. Call the function with a `calls` assembly-language instruction.

A `calls` instruction places a stack frame on the runtime stack. The stack frame contains current values of the program counter (pc), the processor status word (psw), the frame pointer (fp), and the argument pointer (ap). The fp is set equal to the sp and the sp is updated to point to the new top of stack.

Conventions that apply to function calls are:

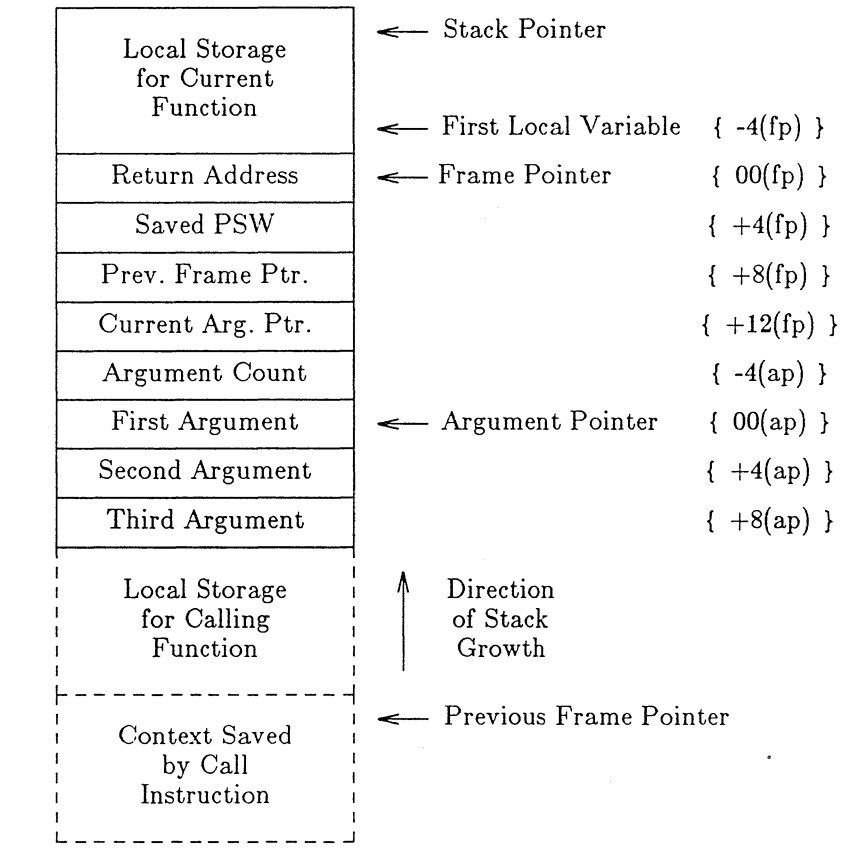
1. The called function can allocate storage for local variables on top of the runtime stack. No stack references in CONVEX C code are made relative to the top of the runtime stack. Storage allocated on the stack by a called function is automatically deallocated when the function returns.
2. The called function need not preserve the contents of any register except the fp. The called function uses the current value of the ap to access arguments passed to the function by its parent.

3. The fp points to the context block that contains the return address, the saved machine registers, and the function arguments, that are pushed on the stack by the caller. The called function references the local storage it has allocated on the runtime stack by negative offsets from the fp.

4. The called function references arguments as positive offsets from the ap. The word with an address of -4(ap) contains the number of arguments passed to the function.

Figure 3-2 shows the layout of the stack as seen by a function after it has been called and after it has allocated some storage for local variables on top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 3–2: Stack Layout



Called functions return by placing a return value in register s0, then executing the `rtn` instruction.

When the `rtn` instruction is executed, automatic storage allocated by the called function is automatically deallocated. This instruction also restores the processor status word register and the frame pointer to their previous states and then returns control to the location immediately following the `calls` instruction that called the function.

After control returns, the stack pointer register points to the location that contains the pushed argument count. The parent function adds a positive number offset to the value in the stack pointer to remove the argument count and any pushed arguments. The value added is the total number of bytes pushed before the call. Finally, before the parent can access any of its own arguments, it must reload its own argument pointer register from the current frame on the stack. This value is at 12(fp).

Assembly Code Generated for Standard Calls

The following example shows a section of CONVEX assembly-language code used for a function call. Generally, C functions are called as shown in the example.

Example:

```
        psh.w   lastarg         ; value of rightmost argument
        psh.w   otherargs       ; value of arguments in
        psh.w   otherargs       ; reverse order
        psh.w   firstarg        ; value of first argument
        mov     sp,ap           ; arg pointer points at first arg
        pshea   #argcount       ; push count of # of args passed
        calls   _child          ; use 'calls' to call function
        add.w   #bytecount,sp   ; remove bytes pushed for args
        ld.w    12(fp),ap       ; reload our argument pointer
```

The code shown below is used in the called child function:

```
        .globl  _child          ; called function
_child:
        sub.w   #1o             ; assuming 32-bit size
        ld.w    (-4)ap,s1       ; load count of the args passed
        st.w    s0,-4(fp)       ; first local int is at -4(fp)
        sub.w   s0,s0           ; value returned in s0
        rtn                     ; return to caller
```

Standard Function Names

Function names and global variables produced by the compiler in object code are limited to 32 characters. An underscore character is prefixed to each global variable. Include this character when using the assembly-language debugger or when writing assembly language functions to be called by C functions.

Standard Function Arguments and Return Values

CONVEX C passes arguments to functions by value rather than by reference; that is, the value of the argument, rather than its address is passed. However, arrays are passed by reference. To pass arguments by reference, use either pointer data types or the & address operator.

When structures are passed by value, all elements of the structure are pushed onto the runtime stack before the call.

Results returned from functions are returned in scalar register s0. ANSI C permits structures to

be returned from functions. Functions that return structures as results place the function result in a data area and return a pointer to that area in s0.

Accessing FORTRAN Routines From C

This section provides information required to call FORTRAN functions from C:

- Translation of FORTRAN function names to C function names.
- C data types equivalent to FORTRAN data types.
- FORTRAN parameter passing protocol.
- FORTRAN protocol for function return values.

Function Names

If you are calling a FORTRAN program, you must follow FORTRAN naming conventions. FORTRAN variables, arrays, and functions have symbolic names that must begin with a letter and may be followed by letters (A-Z), digits (0-9), underscores (_), or dollar signs ($) up to a maximum length of 40 characters. C source code should not use upper case letters because FORTRAN identifiers with external linkage use lower case letters only.

The FORTRAN compiler appends an underscore to FORTRAN subroutine and function names. Because the C compiler does not append this character use the full function or subroutine name in C. For example, a subroutine in FORTRAN named ADDEM would be accessed from C using the name `addem_`.

Data Representations

Table 3-1 shows the corresponding FORTRAN and C declarations that may be used as function parameters. In FORTRAN, all variables declared as INTEGER, LOGICAL, or REAL (without *) default to the same amount of memory, 32 bits. This memory size can be changed with a compiler option.

Table 3-1: FORTRAN and C Declarations

FORTRAN	C
	typedef unsigned long long int UINT;
LOGICAL*1 x	signed char x;
LOGICAL*2 x	signed short int x;
LOGICAL*4 x	signed long int x;
LOGICAL*8 x	signed long long int x;
INTEGER*1 x	signed char x;
INTEGER*2 x	signed short int x;
INTEGER*4 x	signed long int x;
INTEGER*8 x	signed long long int x;
REAL*4 x	float x;
REAL*8 x	double x;
REAL*16 x	struct { UINT sign:1; UINT exp:15; UINT umant:48; UINT lmant; } x;
COMPLEX*8 x	struct {float real, imaginary;} x;
COMPLEX*16 x	struct {double r, i;} x;
CHARACTER*6 x	signed char x[6];

The logical true value in FORTRAN (`.TRUE.`) consists of 1's in each bit position while the logical false value (`.FALSE.`) consists of 0's in each bit position.

Both languages use two's complement to represent integers. Both languages also use either CONVEX native or IEEE format to represent floating-point numbers. To use IEEE format, your machine must have IEEE support hardware.

When passing floating-point data to FORTRAN subroutines, use the same floating-point format (native or IEEE) in the calling routine and the called routine. VECLIB, a collection of optimized FORTRAN-callable numerical subprograms, can automatically detect the floating-point format being used. (Refer to *VECLIB User's Guide* for more information on this software package.)

Parameter Passing

Parameters may be passed to FORTRAN routines by reference only. This means that the contents of a variable are accessible in a FORTRAN routine only if the address of the variable is passed to the routine. This is simple for the primitive data types: prefix the & operator in front of the variable name. Pointers require the pointer name; arrays are automatically passed by reference.

Because FORTRAN has no data type equivalent to C `struct` data types, the only time that a structure should be passed to a FORTRAN routine is when the receiving FORTRAN variable is the COMPLEX*8 or COMPLEX*16 data type. To avoid alignment difficulties, do not use other structure data types.

For example, consider the following synopsis for a FORTRAN routine:

```
INTEGER add1, add2, sum
ADDEM( add1, add2, sum )
```

This subroutine adds two numbers and returns the result in the variable named `sum`. The correct method to call this subroutine from C is:

```
int add1, add2, sum;
addem_( &add1, &add2, &sum );
```

Note the underscore following the FORTRAN function name. The FORTRAN compiler

automatically appends this character to all FORTRAN functions.

Because CHARACTER data types are equivalent to C arrays, they are passed by including the name of the array in the parameter list. However, the length of the character array must also be appended as a long integer to the actual parameter list:

FORTRAN:

```
CHARACTER*4 name
CHARACTER*10 other
PROC_NAME( name, other )
```

C:

```
char string[4];
char other[10];
proc_name_( string, other, 4L, 10L );
```

Caution should be used in passing arrays with multiple dimensions to FORTRAN because that language stores arrays in column major order, while C stores arrays in row major order.

Passing complex data types to FORTRAN requires the use of a C `struct` data type shown in Table 3-1. An example that passes complex numbers is:

FORTRAN:

```
COMPLEX*8 add1, add2, sum
ADDEM( add1, add2, sum )
```

C:

```
struct { float r, i; } add1, add2, sum;
addem_( &add1, &add2, &sum );
```

It is possible to pass a `struct` data type because the alignment of the real and imaginary parts in FORTRAN memory coincide with two `float` data types in C.

FORTRAN Routines

FORTRAN functions are equivalent to C functions. FORTRAN subroutines are equivalent to C functions that return the `void` data type. FORTRAN functions that do not return COMPLEX*8, COMPLEX*16, or CHARACTER data types may be declared in the normal fashion, but these three data types require that an additional parameter be added to the parameter list of a function.

For example, the function

```
INTEGER add1, add2
INTEGER ADDEM( add1, add2 )
```

requires the following function prototype for ANSI C

```
extern int addem_( int *add1, int *add2 );
```

and the following function prototype for C programs compiled in the backward-compatible mode of CONVEX C:

```
extern int addem_( );
```

The C keyword `extern` indicates that the object code for `addem_` is located in another compilation unit. The parameter list for this function is empty because the backward-compatible mode of the C compiler does not recognize function prototypes.

FORTRAN functions return complex data types (`COMPLEX` and `CHARACTER`) in an implied first argument before the list of parameters declared by the FORTRAN function. The first argument should be a pointer to a structure that contains the function result when the call returns.

For example, if a function returns a `COMPLEX*16` data type

```
COMPLEX*16 add1, add2
COMPLEX*16 ADDEM( add1, add2 )
```

C requires a void function prototype:

```
struct d_complex { double dr, di; };
extern void addem_( struct d_complex *sum, struct d_complex *add1,
                                                  struct d_complex *add2 );
```

The second argument in the C function prototype corresponds to the first argument in the FORTRAN function declaration.

Even though the original FORTRAN function returns a value, the C function is void because the value returned by the FORTRAN function is accessed as a parameter in the C function prototype. The function prototype for a program that is compiled using the backward-compatible mode of CONVEX C is:

```
extern void addem_( );
```

Because the compiler performs no function-parameter type-checking in this mode, only the return type of the function must be changed.

FORTRAN functions returning a CHARACTER data type called from a C source file require a C function prototype that returns a void type with two additional parameters in the parameter list. The format of the parameter list in this case is:

function_name(string address, string length, other parameters)

For example, the FORTRAN source code

```
CHARACTER*15 GET_ERROR( )
```

requires the following ANSI C function prototype:

```
extern void get_error_( char error[], long int charlen );
```

C source code used to call this FORTRAN function is:

```
char error_type[15];
get_error_( error_type, 15L );
```

Note that an *L* is appended to the number in the function call because that number **must** be a `long int`.

FORTRAN Input and Output

FORTRAN routines that perform I/O **must not** be called from an application that is initialized as a C program. The methods that the two languages use to perform I/O are incompatible; mixing the I/O of both languages causes undefined behavior.

Accessing Ada Routines from C

Ada routines cannot be called from a C application. CONVEX C does not currently support access to CONVEX Ada routines. For information on calling C routines from an Ada application, see the CONVEX Ada documentation.

Chapter 4
Input and Output

This chapter describes methods that a program uses to receive input and generate output. First, basic concepts of input and output functions are explained, then an example is presented. Functions used for input and output are defined briefly; refer to the man page of a function for additional information.

File Input and Output Concepts

This section of the chapter defines concepts used in file I/O. These concepts are generalized later to include program I/O.

File Manipulation Paradigm

Accessing a file consists of three phases:

- Opening access to the file.
- Performing input and/or output on the file.
- Closing access to the file.

When a file is opened, a data structure of type FILE is created that is used to transfer data to or from a file. Many routines use this data structure to transfer data between files and a program. After the file is processed, access to it is closed and all buffers containing data are flushed. If this final step is not performed, data may be lost.

A file can be opened with the fopen function. This function returns a pointer to a FILE data structure. For example:

```
#include <stdio.h>

FILE *fp;
fp = fopen("some_file", "r" );
```

opens the file `some_file` to read data. The include file, stdio.h, contains standard input and output function prototypes. The FILE data type declares a file pointer that is used by other input/output routines.

After opening the file, the fprintf function (and many other functions) can be used to write text to the file. For example,

```
#include <stdio.h>

FILE *fp = fopen( "output.data", "w" );
if( fp == NULL ){
    fprintf( stderr, "Unable to open file: output.data\n");
    exit(1);
}
fprintf( fp, "This is some sample text." );
fclose( fp );
```

The data structure, pointed to by `fp`, contains status information about the file, such as whether

an error occurred when the file was last accessed. All information contained in the data structure is accessible by library functions. One such function is ferror. For example,

```
#include <stdio.h>

/* assume fp is already associated with a file */
fprintf( fp, "This is some sample text.\n");
if( ferror( fp ) )
    fprintf( stderr, "An error occurred when writing to fp.\n");
```

This function checks for an error when the file associated with structure `fp` was accessed. The error condition exists until it is cleared with the clearerr function. Many other functions can be used to manipulate I/O files.

Although `FILE` is defined in the stdio.h header file, its contents must be accessed only with standard functions, not as a `struct`.

Finally, the fclose function closes access to and flushes the buffers of a file.

File Types and Access Modes

ANSI C defines two file types: binary and text. On a CONVEX computer system, these two file types are identical. You do not need to be concerned with these two types unless you intend to port the program to another computer system. If that is the case, use the binary or text types as required by the other computer system.

The access mode for a file is defined when access to a file is opened. The three basic modes of operation for files are reading, writing, and appending. Reading consists of examining the contents of a file, without modifying them. Writing involves modifying the contents of a file. Appending data to a file means adding data at the end of a file without changing the original data. Other modes are created by combining these basic modes. For example, the read/write mode permits data in a file to be read as well as written. The method used to obtain each of the access modes is defined explicitly in fopen(3).

Files can be accessed by a program only if the program has access permission for the file. For example, if the file only has read-only permission, an error occurs if the program tries to open that file for writing, appending, or reading and writing. For more information on the permissions of a file, refer to chmod(1).

System Functions and ANSI Functions

Two groups of functions are provided with CONVEX C: system I/O functions and ANSI I/O functions. The ANSI I/O functions use the system I/O functions to perform more complicated tasks.

The system I/O functions, which are rudimentary in the actions that they perform, provide basic functions of file I/O. They are:

open, read, write, close, creat and lseek.

In contrast, there are many more ANSI C I/O functions. Some of these functions are used to manipulate the size of a file buffer to enhance performance, while others are used to obtain specific pieces of data in a file. For example, the fscanf function can extract a number from the middle of a line of text.

Thus, the group of I/O functions used in a program depends on the sophistication required to manipulate data files. Also, system I/O functions limit portability of a program because they are not available in the ANSI C standard.

Program Input and Output

Program input and output can be performed on devices as well as files. Every C program has three devices that are automatically available: `stdin`, `stdout`, and `stderr`. These devices are used for standard input (keyboard), standard output (display), and standard error output (display), respectively. Unlike files, it is not necessary to open and close access to these devices.

Access these devices using ANSI C functions only. For example, to print a sentence on a display, use the following command:

```
#include <stdio.h>

fprintf( stdout, "This sentence will be printed out on the display.\n");
```

Many other devices can be accessed from a program. Names of these devices are contained in the directory /dev on ConvexOS. Refer to the *CONVEX UNIX Primer* for more information on these devices.

ConvexOS supports redirection of input and output between files and C programs. For example,

```
% myprog < myprog.in
```

forces all input for the executable file `myprog` to come from `myprog.in`. In this case, the device `stdin` is associated with the file `myprog.in` instead of the keyboard. Similar associations occur when files are redirected by piping, which permits the output of one program to be used as the input of another program. Refer to *ConvexOS Programmer's Reference* for more information on this technique.

Another useful ANSI I/O function is freopen. This function reassigns I/O from a device to a file. For example, the program fragment

```
#include <stdio.h>

freopen( "stderr.file", "w", stderr );
```

forces all succeeding output to the device `stderr` to be stored in the file named `stderr.file`. This function is useful in reassociating `stderr` with a particular error file.

Program Input and Output Examples

An example of the use of ANSI C input and output functions includes a short description of the program followed by the program source code. A line-by-line description of the program follows the example.

Line numbers in the program are provided for reference only; they are not part of the source code.

Reading and Writing Data Structures

This program shows one way data structures can be transferred between a file and a program. It reads some information from a file into an array of structures, then appends it to the file in reverse order. Key routines that make the I/O of data structures possible are fread and fwrite.

```
1       #include <stdio.h>
2       #define NUM_PHONES 5
3       int main()
4       {
5               FILE *fp;
6               int i,j;
7               struct {
8                       char name[8];
9                       char phone[7];
10              }
11              phone_array[NUM_PHONES];
12
13              if( (fp = fopen( "phones", "a+")) == NULL) {
14                      fprintf( stderr, "Unable to open file: phones\n ");
15                      exit(1);
16              }
17              if( fseek( fp, 0L, SEEK_SET ) != 0 ) {
18                      fprintf( stderr, "error seeking origin of file\n" );
19                      exit(1);
20              }
21              j = fread( (void *)phone_array, sizeof( phone_array ),
22                              1, fp );
23
24              if( j != 1 ) {
25                      fprintf( stderr, "Error reading phone array\n" );
26                      exit(1);
27              }
28
29              if( fseek( fp, 0L, SEEK_END ) != 0) {
30                      fprintf( stderr, "error seeking end of file\n" );
31                      exit(1);
32              }
33
34              for( j=0, i=NUM_PHONES-1; i>=0; i-- )
35                      j += fwrite((void *)&phone_array[i],
36                              sizeof(phone_array[i]), 1, fp );
37
38              if( j != NUM_PHONES ) {
39                      fprintf( stderr, "Error writing phone array\n");
40                      exit(1);
41              }
42              fclose( fp );
43              return(0);
44      }
```

1 The first line includes necessary functions and data structures for input and output. The stdio.h header file includes the definition of the FILE structure.

2 Declare macro constant NUM_PHONES.

3 Start the definition of the main function.

5 fp is a data structure that retains information on a file.

6 Declare two integers i and j.

7-10 These lines define the data structure that associates a phone number with a name.

11 phone_array is the array of data structures to be used for input and output in this program.

13 This line attempts to open the file that contains names and phone numbers. The access mode is a+ because the file will initially be read, then data will be appended to the file. This access mode does not permit destruction of any data. If the file cannot be opened, perhaps due to incorrect access permissions, NULL is returned. If NULL is detected, the program halts.

17 The fseek function in this line positions the file position pointer to the beginning of the file. This is required because the append mode positions this pointer to the end of the file when access to the file is opened. This function returns a zero if it is successful; otherwise, a nonzero return value indicates an error occurred. Consequently, the program verifies that no error occurred.

21-22 The function on these two lines reads in the five sets of phone numbers. fread requires four parameters: the address of a structure that receives the data, the number of bytes for each structure, the number of times the structure is read, and the stream pointer. The fread function returns 1 if the read was successful.

24 This conditional ensures that fread reads the required data.

29 The fseek function in this line returns the file position pointer to the end of the file. Again, the return value is verified to ensure that no error occurred.

34-36 A for loop is required to write the data structures because they are appended to the contents of the file in reverse order. Consequently, data structures must be written one at a time. Every time the fwrite function is called, the first parameter contains the address of each data structure in the array.

38 Like the fread function, the fwrite function returns the number of elements sent to the designated file. This line verifies that all structures elements are written and prints an error otherwise.

42 This line flushes the buffer associated with the fp data structure, and closes the connection with the file.

fread and fwrite transfer blocks of data of known size. All they require is a pointer to the data structure and its size. Thus, I/O is fast. But, there are disadvantages to this as well. For example, the data structure must be a constant size. In the previous program, the name field was limited to 8 characters. Strings of variable length cause problems; they must be read using other more explicit methods that do not use a storage area with a fixed size. One such function is fgets.

Data files created by the functions fread and fwrite are not portable to other computer systems. The cause of this nonportability is the structure padding referred to in Chapter 2, "Data Types and Representations." Structure padding occurs when the compiler aligns data types on boundaries in memory. This can cause blank spaces to appear inside a data structure. These blank spaces accompany the data structure when it is written out using fwrite. For example, computer A may insert one blank byte between the name array and the phone array in the previous program, while computer B may insert no spaces. When computer B attempts to read computer A's output, data is lost.

Consequently, these functions should not be used when data files are to be transferred to other computer systems.

Chapter 5
Runtime Library

This chapter provides a brief overview of functions that are available in the C libraries provided with CONVEX C; ANSI C, POSIX, and CONVEX functions are included. The information is organized by header files.

Under each header file, the functions are listed with a short description and some common macro definitions.

The header files discussed in this chapter are:

- assert.h
- ctype.h
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h

Each section describes the ANSI C functions, associated structures, identifiers, and macros. Two sections describe the CONVEX and POSIX extensions that are accessible with these ANSI C header files. The compatibility mode of the compiler determines which extensions are available.

CONVEX C provides four compatibility modes as shown in Table 5-1.

Table 5-1: Compatibility Modes

Mode	Language	Default Functions
Extended	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	ANSI C	ANSI C, POSIX
Strict	ANSI C	ANSI C
Backward-compatible	non-ANSI C	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first of the POSIX standards to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at the source-code level.

By avoiding non-standard features of an operating system, an application has a greater chance of being ported to another computer system without major modifications.

Functions Versus Function-like Macros

Function-like macros are very similar to functions; a function-like macro is called in the same manner as a function. There are several differences:

- Function-like macros increase the size of the code.
- Function-like macros are faster; there is no function overhead such as register saving.
- Functions inhibit optimization.
- Contents of functions have local scope.
- Functions have an address.

Function-like macros are most suitable for routines that are used often but require little code; such routines include character input and output functions. Several of the header files, ctype.h and stdio.h in particular, include both functions and function-like macros for some of the routines, permitting the programmer to choose which version should be used.

Function-like macros are always defined after the function prototype is declared. Two ways to access the function instead of the macro are:

- Undefine the function-like macro using `#undef`.
- Surround the function name with parentheses.

For example, in the header file ctype.h, isalpha is declared as a function prototype and defined as a function-like macro. Two ways of calling the function are

```
#include <ctype.h>

#undef isalpha
int ch = isalpha('A');
```

and

```
#include <ctype.h>

int ch = (isalpha)('A');
```

The macro is accessed as usual:

```
#include <ctype.h>

int ch = isalpha('A');
```

Thus, it is possible to select either a function or a function-like macro when a choice exists.

Functions that have function-like macros are identified in the following sections.

Calling Runtime Functions

There are two ways to access a library function:

- Include the associated header file.
- Implicitly declare the function if no types from the header file are required.

Each method has its advantages and disadvantages.

Including a header file is easy and portable. Any data types, structures, and macros that are required to use the function are automatically declared and defined. When an application is ported to another system, changes to data structures in header files probably do not require the source code to be changed.

For example, use memmove

```
#include <string.h>

char arra[10];

memmove( arra, &arra[6], 4 );
```

to replace the last four elements of `arra` with the first four elements.

In the absence of a function prototype, a function is implicitly declared when it is encountered. Integral promotions are performed on each integral argument (that is, `char` is promoted to `int`, and arguments that have type `float` are promoted to type `double`).

Although the CONVEX functions in the following descriptions are written with a function prototype, only the return type of the function is specified in the header files. This permits the functions to be called with a variable number of parameters. It also means that the formal parameters are declared implicitly.

In conclusion, header files are not required to access a runtime function, but their inclusion is easy and assists in maintaining the program and is required by the language in many cases.

assert.h

This header file contains a function-like macro that halts a program if its argument is not true. This macro is useful in the development of an application, to ensure that certain conditions exist.

ANSI C

The function-like macro is:

void assert(expression)
Aborts program if `expression` is false and `NDEBUG` is not defined. Writes file name and line number of the file with the error.

CONVEX Extension

void _assert(expression)
Equivalent to the assert function.

ctype.h

This header file contains character handling functions that are used for classifying character-coded integer values by table lookup. Each function returns a nonzero value for true or zero for false.

ANSI C

Functions declared are:

int isalnum(int ch)
Returns true for any letter or digit.

int isalpha(int ch)
Returns true for any letter.

int iscntrl(int ch)
Returns true for any control character. Control characters are nonprinting characters.

int isdigit(int ch)
Returns true for any decimal-digit character.

int isgraph(int ch)
Returns true for any printing character except space (' ').

int islower(int ch)
Returns true for any lower case letter.

int isprint(int ch)
Returns true for any printing character including space (' ').

int ispunct(int ch)
Returns true for any punctuation character. These characters are defined as all printing characters except spaces, digits, or letters.

int isspace(int ch)
Returns true for any space, tab, carriage return, newline, vertical tab, or form feed.

int isupper(int ch)
Returns true for any upper case letter.

int isxdigit(int ch)
Returns true for any hexadecimal digit.

int tolower(int ch)
Returns the corresponding lower case letter when the argument is an upper case letter. If the argument is not an upper case letter, it returns the argument unchanged.

int toupper(int ch)
Returns the corresponding upper case letter when the argument is a lower case letter. If the argument is not a lower case letter, it returns the argument unchanged.

Functions also declared as function-like macros are:

- isalnum
- isalpha
- isalpha
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit

CONVEX Extensions

Function-like macros are:

int isascii(int ch)
Returns true for any ASCII character.

int toascii(int ch)
Returns the argument with all bits that are not part of the standard ASCII character turned off.

int _tolower(int ch)
Returns the same data as tolower, except it has a restricted domain and runs faster. If the argument to _tolower is not an upper case letter, its result is undefined.

int _toupper(int ch)
Returns the same data as toupper, except it has a restricted domain and runs faster. If the argument to _toupper is not a lower case letter, its result is undefined.

errno.h

This header file contains the definition of many macro constants for error conditions.

ANSI C

Two macros defined are:

EDOM
Contains the value that indicates a domain error.

ERANGE
Contains the value that indicates a range error.

An external identifier declared is:

errno
Contains a value indicating the most recent error. This identifier is accessed globally by several library routines.

float.h

This header file defines macro constants that indicate characteristics of floating-point data types.

ANSI C

Macros defined are:

FLT_ROUNDS
Rounding mode for floating-point addition. CONVEX computers round to the nearest representable value.

FLT_RADIX
Base number system for exponent representation.

FLT_MANT_DIG
Number of digits in the floating-point mantissa of the `float` type. The base is the value of `FLT_RADIX`.

DBL_MANT_DIG
Number of digits in the floating-point mantissa of the `double` type. The base is the value of `FLT_RADIX`.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa of the `long double` type. The base is the value of `FLT_RADIX`.

FLT_DIG
Number of decimal digits in a number of type `float` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

DBL_DIG
Number of decimal digits in a number of type `double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

LDBL_DIG
Number of decimal digits in a number of type `long double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

FLT_MIN_EXP
Smallest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `float`.

DBL_MIN_EXP
Smallest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `double`.

LDBL_MIN_EXP
Smallest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `long double`.

FLT_MAX_EXP
Largest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `float`.

DBL_MAX_EXP
Largest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `double`.

LDBL_MAX_EXP
Largest integer z such that `FLT_RADIX`$^{(z-1)}$ is a normalized floating-point number of type `long double`.

FLT_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `float`.

DBL_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `double`.

LDBL_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `long double`.

FLT_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `float`.

DBL_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `double`.

LDBL_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `long double`.

FLT_MAX
Largest representable number of type `float`.

DBL_MAX
Largest representable number of type `double`.

LDBL_MAX
Largest representable number of type `long double`.

FLT_EPSILON
Smallest number z of type `float` such that $1.0 + z \neq 1.0$.

DBL_EPSILON
Smallest number z of type `double` such that $1.0 + z \neq 1.0$.

LDBL_EPSILON
Smallest number z of type `long double` such that $1.0 + z \neq 1.0$.

FLT_MIN
Smallest normalized positive number of type `float`.

DBL_MIN
Smallest normalized positive number of type `double`.

LDBL_MIN
Smallest normalized positive number of type `long double`.

limits.h

This header file defines macro constants that indicate sizes of integral data types.

ANSI C

Macros defined are:

CHAR_BIT
Number of bits in `char` data type.

SCHAR_MIN
Minimum value for `signed char` data type.

SCHAR_MAX
Maximum value for `signed char` data type.

UCHAR_MAX
Maximum value for `unsigned char` data type.

CHAR_MIN
Minimum value for an object of type `char`.

CHAR_MAX
Maximum value for an object of type `char`.

MB_LEN_MAX
Maximum number of bytes in a multibyte character.

SHRT_MIN
Minimum value for `short int` data type.

SHRT_MAX
Maximum value for `short int` data type.

USHRT_MAX
Maximum value for `unsigned short int` data type.

INT_MIN
Minimum value for `int` data type.

INT_MAX
Maximum value for `int` data type.

UINT_MAX
Maximum value for `unsigned int` data type.

LONG_MIN
Minimum value for `long int` data type.

LONG_MAX
Maximum value for `long int` data type.

ULONG_MAX
Maximum value for `unsigned long int` data type.

POSIX Extensions

Macros defined are:

_POSIX_ARG_MAX
Total number of bytes, including environment data, for one of the exec functions.

_POSIX_CHILD_MAX
For each real user ID, number of simultaneous processes.

_POSIX_LINK_MAX
A file's link count value.

_POSIX_MAX_CANON
A terminal canonical input queue's size in bytes.

_POSIX_MAX_INPUT
The buffer size for a terminal input queue in bytes.

_POSIX_NAME_MAX
Character length of a file name.

_POSIX_NGROUPS_MAX
For each process: number of simultaneous supplementary group IDs.

_POSIX_OPEN_MAX
Number of files that can be open for one process simultaneously.

_POSIX_PATH_MAX
Character length of a pathname.

_POSIX_PIPE_BUF
Number of bytes that can be written to a pipe without being interrupted.

The following macro constants are undefined (with `#undef`) by this header file:

- CHILD_MAX
- LINK_MAX
- MAX_INPUT
- MAX_CANON
- NAME_MAX

locale.h

Functions in this header file tailor the international output environment for functions that control character handling, string collation, date and time formatting, and numeric editing. This header file enables the output of a program to be customized for a specific nationality.

Only the "C" locale is implemented in CONVEX C.

ANSI C

The structure defined is:

lconv
Contains the characters that can be changed in each locale.

The macros defined in this file represent categories that are used by the setlocale function. They are:

LC_ALL
A composite of all the other categories.

LC_COLLATE
This category impacts the comparison function used by the functions strcoll and strxfrm.

LC_CTYPE
This category impacts the following functions:

- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit

The function-like macros of the same name are not impacted by this category. Therefore, it is necessary to undefine a function-like macro before the function that is affected by LC_CTYPE can be used.

LC_MONETARY
This category impacts information returned by the localeconv function regarding monetary characters.

LC_NUMERIC
This category impacts the decimal point character used by formatted input and output functions, string conversion functions, and non-monetary formatting information returned by the localeconv function.

LC_TIME
This category impacts the strftime function.

Functions declared in this header file are:

struct lconv *localeconv(void)
Returns a pointer to a structure of current locale information. The structure is in the static storage class and cannot be modified by the application.

char *setlocale(int category, const char *locale)
Changes or queries a program's current locale. `category` is a category macro defined in this header file. The `locale` argument may three values:

`""`
sets the category to the locale of the minimal C translation.
`"C"`
sets the category to the locale of the minimal C translation.
`NULL`
queries the system for the current locale of the specified category.

math.h

This header file declares math functions.

Math function errors are grouped into two categories: domain errors (`EDOM`) and range errors (`ERANGE`). Domain errors occur when the size of an argument causes significant inaccuracy in the function result. When a domain error occurs, the value of the macro EDOM is stored in `errno`, an external identifier declared in errno.h.

Range errors result when the computed value cannot be represented within the machine's precision. When a range error occurs, the value of the macro ERANGE is stored in `errno`.

ANSI C

The macro defined is:

HUGE_VAL
Largest positive double precision value. It is not a constant expression and cannot be evaluated by the preprocessor.

Functions declared in this header file are listed below. If the result of the function overflows, the function returns `HUGE_VAL`; the sign is the same as that of the correct function result except, the tan function. The function returns zero when an underflow occurs. `errno` is set to the value of `ERANGE` only when an overflow occurs.

ANSI C math functions are:

double acos(double x)
Returns arccosine of the argument that is in the range [-1,+1].

double asin(double x)
Returns arcsine. The argument must be in the range [-1,+1].

double atan(double x)
Returns arctangent.

double atan2(double numer, double denom)
Returns arctangent of `numer/denom`. Both arguments may not be zero.

double ceil(double x)
Returns smallest integer not less than the argument, $\lceil x \rceil$.

double cos(double x)
Returns cosine of the argument that is measured in radians.

double cosh(double x)
Returns hyperbolic cosine.

double exp(double x)
Returns exponential, e^x.

double fabs(double x)
Returns absolute value, $|x|$.

double floor(double x)
Returns largest integer not greater than the argument, $\lfloor x \rfloor$.

double fmod(double numer, double denom)
Returns floating-point remainder of `numer/denom`.

double frexp(double number, int *pow)
Returns `f`, where $\frac{1}{2} \leq f$ or $f = 0$ and `number` $= f \times 2^{(*pow)}$. If `number` = 0, `*pow` and `f` are zero.

double ldexp(double x, int pow)
Returns $x \times 2^{pow}$.

double log(double x)
Returns natural logarithm of `x`.

double log10(double x)
Returns base-10 logarithm of `x`.

double modf(double number, double *i)
Splits `number` into a fraction, `f`, and an integer `*i`, where `f + (*i) = number`. The function returns `f`.

double pow(double x, double pow)
Returns x^{pow}.

double sin(double x)
Returns sine of the argument that is measured in radians.

double sinh(double x)
Returns hyperbolic sine.

double sqrt(double x)
Returns positive square root of the argument, $|\sqrt{x}|$.

double tan(double x)
Returns tangent of the argument that is measured in radians.

double tanh(double x)
Returns hyperbolic tangent.

Table 5-2 lists values returned by math functions when a domain error occurs.

Table 5-2: Math Function Return Values

Function	Domain Error Return Value
acos	acos(1)
asin	asin(1)
atan	pi/2
atan2	pi/2
cos	cos(0)
cosh	HUGE_VAL
sin	sin(0)
sinh	-HUGE_VAL
log	log(|x|)
log10	log10(|x|)
pow	pow(|x|)
sqrt	sqrt(|x|)

CONVEX Extensions

When a range or domain error occurs for the following functions, the global integer `errno` is set to the value of a math error. All the math errors are defined in errno.h. These errors are more specific than EDOM and ERANGE.

double atof(const char * str)
Returns the `double` representation of the first number contained in the string pointed to by `str`. This ANSI C function is declared in this file for convenience; the ANSI C standard places this function in stdlib.h.

double cabs(struct { double x, y;} z)
Returns z's Euclidean length.

double dcvtid(double x)
Converts native mode input, x, into IEEE floating-point mode.

double gamma(double x)
Returns the log gamma of x.

double hypot(double x, double y)
Returns the Euclidean distance of x and y.

double idcvtd(double x)
Converts IEEE mode input, x, into native floating-point mode.

int ipow(int x, int pow)
Returns x^{pow}.

float ircvtr(float x)
Converts IEEE mode input, **x**, into native floating-point mode.

double j0(double x)
Bessel functions of the first kind (j1, order 0).

double j1(double x)
Bessel functions of the first kind (j1, order 1).

double jn(int n, double x)
Bessel functions of the first kind (j1, order n).

double lpow(long long x, long long pow)
Returns x^{pow}.

float rcvtir(float x)
Converts native mode input, **x**, into IEEE floating-point mode.

double y0(double x)
Bessel functions of the second kind (y1, order 0).

double y1(double x)
Bessel functions of the second kind (y1, order 1).

double yn(int n, double x)
Bessel functions of the second kind (y1, order n).

For descriptions of these functions, refer to their man pages.

setjmp.h

This header file declares two functions that can be used instead of the normal function call and return paradigm.

ANSI C

The type defined is:

jmp_buf
Declares an identifier that retains the context of the calling environment. It is used by the two functions declared in this header file.

Functions declared are:

int setjmp(jmp_buf buffer)
Save the calling environment context in `buffer`.

void longjmp(jmp_buf buffer, int retval)
Restore the environment context saved in `buffer`. Program execution resumes with the statement following the setjmp function associated with `buffer`. the setjmp function returns `retval` if it is nonzero; if `retval = 0`, the setjmp function returns 1.

POSIX Extensions

The type defined is:

sigjmp_buf
Declares a location in which to store the calling environment information.

Functions declared are:

int sigsetjmp(sigjmp_buf env, int savemask)
Save the calling environment in `env` and if `savemask` is nonzero, save the process's current signal mask as part of the calling environment.

void siglongjmp(sigjmp_buf, int)
Restore the calling environment saved in `env` and if the signal mask is saved in the calling environment, restore that as well.

signal.h

This header file declares the functions that control the behavior of a program when signals occur. It also defines macro constants that represent various signals.

ANSI C

The type defined is:

sig_atomic_t
No interrupts are recognized when a value is assigned to an object of this type. This type is useful for communication between the program and its signal handlers.

The functions are:

void (*signal(int sig, void (*func)(int)))(int)
Define the behavior of the program when a particular signal is received.

int raise(int sig)
Sends a signal to the executing program.

A function-like macro is:

int raise(sig)

CONVEX Extensions

Refer to the header file for a list of the signals. Basic categories are:

- Traps.
- Floating-point exceptions.
- Bus errors.
- Segment errors.

Structures defined are:

sigaction
An aggregate of a signal handler, a bit mask, and some flags.

sigvec
Overlaid by struct sigaction.

sigcontext
Contains information pushed on the stack when a signal is delivered. It is made available to the handler to allow it to properly restore state if a non-standard exit is performed.

A function-like macro is:

int sigmask(number)
Creates a mask for the bit indicated by `number`. For example, the value of sigmask(5) is 16.

A function is:

int (*signal(int sig, void (*func)(int sig, int subcode, struct sigcontext *scp)))(int)
Defines the behavior of the program when a particular signal is received. The difference between this function and the ANSI C signal function is that the function func accepts three parameters, while the ANSI C signal handler accepts only one parameter.

POSIX Extensions

The structures defined are:

sigaction
An aggregate of a signal handler, a bit mask, and some flags.

sigvec
Overlaid by struct sigaction.

Three macros defined are used as the first argument to the sigprocmask macro. They determine how the signal set of a process is changed.

SIG_BLOCK
Resulting signal set is a combination of the current signal set and the signal set pointed to by another argument to the function.

SIG_UNBLOCK
Resulting signal set is a intersection of the current signal set and the complement of the signal set pointed to by another argument to the function.

SIG_SETMASK
Resulting signal set is the signal set pointed to by another argument to the function.

Functions declared are:

int kill(pid_t pid, int sig)
Sends signal `sig` to the process specified by `pid`. Returns 0 if the function is successful, otherwise it returns -1.

int sigaction(int sig, struct sigaction *act, struct sigaction *oact)
Customizes or examines the action associated with a particular signal. `act` is a pointer to the new action; `oact` is used to store the old action if it is not NULL. If `act` is NULL, the signal handling is not changed. Returns 0 if the function is successful; otherwise, -1. Returns 0 if the function is successful; otherwise, -1.

int sigaddset(sigset_t *set, int signo)
Adds the individual signal specified by the value of `signo` to the signal set pointed to by `set`. Returns 0 if the function is successful; otherwise, -1.

int sigdelset(sigset_t *set, int signo)
Removes the individual signal specified by the value of `signo` from the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise; -1 is returned.

int sigemptyset(sigset_t *set)
Excludes all POSIX signals from the initialization of the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise -1 is returned.

int sigfillset(sigset_t *set)
Includes all POSIX signals in the initialization of the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise -1 is returned.

int sigismember(sigset_t *set, int signo)
Tests whether the set pointed to by `set` contains the signal specified by the value of `signo`. Returns 1 if the signal is a member of the set, and 0 if it is not. If an error occurs, -1 is returned.

int sigpending(sigset_t *set)
`set` is the storage location for the set of signals that are blocked from delivery and pending for the calling process. Returns 0 if it successful, and -1 if there is an error.

int sigprocmask(int how, sigset_t *set, sigset_t *oset)
Examines or changes the calling process's signal mask. `how` is one of the three macros described previously; `set` lists the modifications required as specified by `how`; and oset is the resulting signal set. If `set` is NULL no modifications take place and `oset` contains the current signal set. Returns 0 if function is successful; -1 otherwise.

int sigsuspend(sigset_t *sigmask)
The set of signals pointed to by `sigmask` replaces a process's current signal mask. The process is suspended until it receives a signal that terminates it or a signal that forces it to execute a signal-catching function. Returns a -1 if an error occurs.

stdarg.h

The contents of this file are used to access parameters of functions that may have a variable number of arguments.

ANSI C

The type defined is:

va_list
Declares a variable argument list structure.

Function-like macros are:

type ***va_arg(va_list list,** *type*)
Returns the next parameter in the argument list. The second argument is the type of the next parameter.

void va_end(va_list list)
Releases access to the structure defined by `va_list`.

void va_start(va_list list, <LastParam>)
initializes the data structure `list`. `<LastParam>` is the last nonvariable parameter in the function argument list.

stddef.h

ANSI C

Types defined are:

ptrdiff_t
Type resulting from the subtraction of two pointers.

size_t
Type returned by the sizeof macro operator.

wchar_t
Data type for wide characters.

The macro defined is:

NULL
Null pointer constant

Function-like macro is:

size_t offsetof(structure, field)
Offset, in bytes, of a structure member from the base address of its structure.

stdio.h

This header file declares types, macros, and functions that are used for input and output.

ANSI C

Macros defined are:

BUFSIZ
Buffer size for setbuf.

EOF
Value returned by functions that indicates the end of a file.

FILENAME_MAX
Maximum number of characters permitted in a file name.

FOPEN_MAX
Minimum number of files that can be open simultaneously.

_IOFBF
Parameter of setvbuf function that indicates full buffering for input/output.

_IOLBF
Parameter of setvbuf function that indicates line buffering for input/output.

_IONBF
Parameter of setvbuf function that indicates no buffering for input/output.

L_tmpnam
Maximum length of a file name that is returned by the tmpnam function.

NULL
Null pointer constant.

SEEK_CUR
Parameter of the fseek function that sets the file reference point at the beginning of the file.

SEEK_END
Parameter of the fseek function that sets the file reference point at the end of the file.

SEEK_SET
Parameter of the fseek function that sets the file reference point at the current file position.

stderr
File pointer for standard error stream.

stdin
File pointer for standard input stream.

stdout
File pointer for standard output stream.

TMP_MAX
Minimum number of unique file names that can be produced by the tmpnam function.

The types defined are:

FILE
Object that records all information needed to access a file.

fpos_t
Object that retains all information required to specify uniquely a file position.

size_t
Type returned by the sizeof macro operator.

Operations On Files

int remove(const char *filename)
Discards the link between a file name and its file contents. Returns a nonzero value when an error occurs.

int rename(const char *old, const char *new)
Associates a new file name with a file. The link between `*old` and the contents of the file is removed. Returns a nonzero when an error occurs.

FILE *tmpfile(void)
Creates a temporary binary file. When the program terminates the file is removed automatically. NULL is returned if an error occurs.

char *tmpnam(char *file_name)
Returns a valid file name that is not the same as an existing file name.

File Access Functions

int fclose(FILE *file)
Flushes the buffer associated with `file`, then closes it. Returns EOF if an error occurs.

int fflush(FILE *file)
Flushes the buffer associated with `file`. Returns EOF if a write error occurs.

FILE *fopen(const char *filename, const char *mode)
Initiates access to a file. `mode` determines the type of access. Returns a pointer to the file structure or NULL if an error occurs.

FILE *freopen(const char *filename, const char *mode, FILE *fp)
Associates a new file with an existing file pointer. Any file already associated with the stream is closed. Access to the file is specified by `mode`. Returns NULL if an error occurs.

void setbuf(FILE *fp, char *buffer)
Establishes full buffering for a stream with `buffer` or causes input and output with a file pointer, `fp`, to be unbuffered.

int setvbuf(FILE *fp, char *buffer, int buf_type, size_t size)
Tailors the type of buffering associated with a file pointer to fully buffered, line buffered, or unbuffered. The third parameter selects the buffer type. It can have one of three values: _IOFBF, _IOLBF, or _IONBF. It also permits a user-defined buffer, specified by `buffer` and `size`. Returns a nonzero value if an error occurs.

Formatted Input/Output Functions

int fprintf(FILE *fp, const char *format, ...)
Formats text, as specified by `format` and writes the text to the file pointed to by `fp`. Returns the number of characters written.

int fscanf(FILE *fp, const char *format, ...)
Reads input from the file pointed to by `fp` using `format` and places the input data in the addresses specified by additional arguments. Returns `EOF` if no data is read; otherwise, it returns the number of conversions that were performed.

int printf(const char *format, ...)
Performs the same function as fprintf but writes to `stdout`.

int scanf(const char *format, ...)
Performs the same function as fscanf but reads from `stdin`.

int sprintf(char *array, const char *format, ...)
Performs the same function as fprintf, except the formatted text is written into the array pointed to by `array`. The null character is appended to the formatted text.

int sscanf(const char *array, const char *format, ...)
Performs the same function as fscanf, except the input is obtained from the array pointed to by `array`.

int vfprintf(FILE *stream, const char *format, va_list arg)
Performs the same function as fprintf. `arg` specifies the variable argument list. `arg` must be initialized with va_start before vfprintf is called. The header file stdarg.h must be included when this function is used.

int vprintf(const char *format, va_list arg)
Performs the same function as printf. `arg` specifies the variable argument list. `arg` must be initialized with va_start before vprintf is called. The header file stdarg.h must be included when this function is used.

int vsprintf(char *arra, const char *format, va_list arg)
Performs the same function as sprintf. `arg` specifies the variable argument list. `arg` must be initialized with va_start before vsprintf is called. The header file stdarg.h must be included when this function is used.

Character Input and Output Functions

int fgetc(FILE *fp)
Returns the next character from the file pointed to by `fp`. If the end of the file is encountered or an error occurs, EOF is returned.

char *fgets(char *str, int n, FILE *fp)
Reads `n`-1 characters, or up to a newline character, from the file pointed to by `fp` into the string pointed to by `str`.

int fputc(int ch, FILE *fp)
Outputs a character to the file pointed to by `fp`. fputc performs the same function as putc, except that fputc is a function, whereas putc is a macro.

int fputs(const char *str, FILE *fp)
Copies the null-terminated string `str` to the file pointed to by `fp`. Returns EOF if a write error occurs.

int getc(FILE *fp)
Returns the next character from the file pointed to by `fp`, or EOF if the end of the file is encountered or an error occurs.

int getchar(void)
Returns the next character from `stdin`, or EOF if the end of the file is encountered or an error occurs.

char *gets(char *str)
Reads a string from the standard input `stdin`. Reads up to a newline character or the end of the file. A null pointer is returned if an error occurs.

int putc(int ch, FILE *fp)
Outputs a character to the file pointed to by `fp`. Returns EOF if a write error occurs.

int putchar(int ch)
Outputs a character to the standard output, `stdout`. Returns EOF if a write error occurs.

int puts(const char *str)
Writes the null-terminated string pointed to by `str` to the standard output, `stdout`, and appends a newline character. Returns EOF if an error occurs.

int ungetc(int ch, FILE *fp)
Pushes character, `ch`, back into an input buffer associated with the file pointer `fp`. Returns EOF if an error occurs.

Direct Input and Output Functions

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)
Reads a block of data from the file associated with `fp`. `ptr` is the base address of the array for the data; `nmemb` is the number of elements in the array; and `size` is the number of bytes required for each element in the array. Returns the number of array elements actually read.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
Writes a block of data to the file associated with `fp`. `ptr` is the base address of the data to write; `nmemb` is the number of items to write; and `size` is the number of bytes that represents each item. Returns the number of items actually written.

File Positioning Functions

int fgetpos(FILE *fp, fpos_t *pos)
`pos` is set equal to the current file position indicator of the file pointed to by `fp`. If an error occurs, a nonzero value is returned and `errno` is set equal to one of the values in Table 5-3.

Table 5-3: errno values of fgetpos, fsetpos and ftell

Value Returned	Error Condition
EBADF	file is not open
ESPIPE	file name is associated with a pipe or a socket
EINVAL	file position is negative

int fseek(FILE *fp, long int offset, int ref_pt)
Sets the file position of the file pointed to by `fp` to the position indicated by `ref_pt` and `offset`. `ref_pt` can be one of three values: SEEK_CUR, the current file position; SEEK_SET, the beginning of the file; and SEEK_END, the end of the file. `offset` is the number of bytes from the reference point. Returns a nonzero value when a domain error occurs.

int fsetpos(FILE *fp, const fpos_t *pos)
Sets file position of file pointed to by `fp` equal to value stored in `pos` that originated in a prior call to fgetpos. Sets the file position indicator for `stream` according to the value of the object pointed to by `pos`, which is be a value obtained from an earlier call to fgetpos on the same file.

long ftell(FILE *fp)
Returns the offset of the current file position of the file pointed to by `fp` from the beginning of the file. The value is measured in bytes. If an error occurs, it returns -1L and sets `errno` equal to a value listed in Table 5-2.

void rewind(FILE *fp)
Sets the file position indicator to the beginning of the file pointed to by `fp`.

Error-handling Functions

void clearerr(FILE *fp)
clears end-of-file and error indicators for the file associated with `fp`.

int feof(FILE *fp)
Returns a nonzero value if the end-of-file indicator for the file associated with `fp` is set.

int ferror(FILE *fp)
Returns a nonzero value if the error indicator of the file associated with `fp` is set.

void perror(const char *str)
Writes the error message associated with the value stored in `errno` to `stderr`. This message is preceded by the string pointed to by `str`. Tables 7-8 through 7-15 list the error messages associated with each valid value of `errno`.

Routines in the following list are available as function-like macros and as functions:

- getc
- getchar
- putc
- putchar
- feof
- ferror

CONVEX Extensions

The function declared is:

FILE *popen(char *str, char *mode)
Creates a pipe between the calling process and the command to be executed, which is pointed to by `str`. `mode` can be `"r"` for reading or `"w"` for writing. The value returned is a pointer that can be used to write to the standard input of the command or read from its standard output. If an error occurs, `NULL` is returned.

int pclose(FILE *pp)
Waits for the process associated with the pipe pointer, `pp`, to terminate and returns the exit status of the command. Returns -1 on a domain error.

POSIX Extensions

Two macros are defined:

L_cuserid
Maximum length of the controlling user ID including the terminating null byte.

L_ctermid
Maximum length of the controlling terminal ID including the terminating null byte.

The following functions are declared:

int *fdopen(int fildes, const char *type)
Associates a file with a file descriptor.

int fileno(const FILE *fp)
Returns the integer file descriptor associated with `fp`. This is also available as a function-like macro.

char *cuserid(char *str)
Returns a name associated with the effective user ID of the process.

stdlib.h

This header file contains declarations for some general utilities.

ANSI C

The macros defined are:

EXIT_FAILURE
Argument of `exit` that indicates abnormal termination status.

EXIT_SUCCESS
Argument of `exit` that indicates normal termination status.

NULL
Null pointer constant

MB_CUR_MAX
Number of bytes required to represent a multibyte character in the current locale.

RAND_MAX
The maximum integer returned by rand.

The types defined in this file are:

size_t
Result type of the sizeof operator.

wchar_t
Data type for wide characters.

div_t
Return type of the div function.

ldiv_t
Return type of the ldiv function.

String Conversion Functions

double atof(const char *nptr)
Returns the `double` representation of the initial portion of the string pointed to by `nptr`.

int atoi(const char *nptr)
Returns the `int` representation of the initial portion of the string pointed to by `nptr`.

int atol(const char *nptr)
Returns the `long int` representation of the initial portion of the string pointed to by `nptr`.

double strtod(const char *nptr, char **endptr)
Returns the `double` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. Returns HUGE_VAL and sets `errno` to ERANGE if an overflow occurs. Returns 0 and sets `errno` to ERANGE if an underflow occurs.

long strtol(const char *nptr, char **endptr, int base)
Returns the `long int` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. If a range error occurs it sets `errno` to ERANGE and returns LONG_MIN if the sign of the answer is negative and LONG_MAX if the sign is positive.

unsigned long strtoul(const char *nptr, char **endptr, int base)
Returns the `unsigned long int` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. If a range error occurs, `errno` is set to ERANGE and ULONG_MAX is returned.

Pseudo-random Sequence Generation Functions

int rand(void)
Returns a nonnegative integer less than or equal to `RAND_MAX`.

void srand(unsigned int seed)
Reseeds the sequence of pseudorandom integers.

Memory Management Functions

void *calloc(size_t nmemb, size_t size)
Allocates a block of memory for `nmemb` objects, each of size `size`.

void free(void *pm)
Deallocates the memory pointed to by `pm`.

void *malloc(size_t size)
Returns a pointer to a memory block that has `size` bytes. Returns a null pointer if sufficient memory is not available.

void *realloc(void *ptr, size_t size)
Changes the memory size originally allocated to `ptr`. The new memory size requirement is specified by `size`. Returns the new memory pointer or a null pointer if an error occurs.

Communication with the Environment Functions

void abort(void)
Causes program to terminate abnormally if the signal SIGABRT is not cleared by a signal handler. Does not return program execution to its caller.

int atexit(void (*func)(void))
The function pointed to by `func` is registered. It is called without arguments at normal program termination.

void exit(int status)
Causes program to terminate normally.

char *getenv(const char *name)
Searches environment list for a string of the form `name=`*value*. Returns a pointer to the string *value* if such a string is present. If such a string is not present, getenv returns the NULL value.

int system(const char *string)
Issues a shell command. The current process waits until the shell has completes the command indicated by `string`, then returns the exit status of the shell.

Searching and Sorting Functions

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))
Searches for `key` in the array pointed to by `base` using the function `compare`. Each array element has `size` bytes. Returns a pointer to the matching element of the array if it exists; otherwise it returns a null pointer.

void qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))
Provides a quick-sort utility. `base` is a pointer to the base of the data; `nmemb` is the number of elements; `size` is the width of an element in bytes; `compare` is the name of the routine that compares two elements in the array.

Integer Arithmetic Functions

int abs(int j)
Returns the absolute value of `j`.

div_t div(int numer, int denom)
Returns the quotient and remainder of the arguments. The structure returned has two members, `quot` and `rem`.

long labs(long j)
Returns the `long` representation of the absolute value of `j`.

ldiv_t ldiv(long numer, long denom)
Returns the `long` representation of the quotient and remainder of the arguments. The return value has two members, `quot` and `rem`.

Multibyte Character Functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. If `*str` is `NULL` in any of the following functions, `0` is returned indicating that no state-dependent encodings exist.

int mblen(const char *str, size_t n)
Returns the number of bytes, up to `n`, of the multibyte character pointed to by `str`. This is always 1 unless `n = 0` or `*str == NULL`.

int mbtowc(wchar_t *pwc, const char *str, size_t n)
Because all multibyte characters are length 1, the first character pointed to by `*str` is copied to `*pwc` and 1 is returned. However, if `n = 0`, no conversion is performed and `-1` is returned.

int wctomb(char *str, wchar_t wchar)
Converts the multibyte character code stored in the object `wchar` into a multibyte character whose base address is `str`. Returns 1 because all multibyte characters are of length 1.

Multibyte String Functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. Conversions between multibyte strings and wide characters are trivial because no special encodings are supplied by CONVEX. If `*str` is `NULL` in any of the following functions, `0` is returned indicating that no state dependent encodings exist.

size_t mbstowcs(wchar_t *pwcs, const char *str, size_t n)
Copies `n` bytes, or until `NULL` is encountered, of the multibyte string pointed to by `*str` to the wide character string pointed to by `*pwcs`. Returns the number of multibyte characters copied.

size_t wcstombs(char *str, const wchar_t *wcs, size_t n)
Copies an array of `n` or fewer multibyte character encodings into a multibyte character string. Returns one less than the number of characters copied.

string.h

String-handling functions are used to manipulate character arrays and blocks of memory.

ANSI C

One macro is defined:

NULL
Null pointer constant.

One type is defined:

size_t
Type returned by the sizeof macro operator.

Copying Functions

void *memcpy(void *str1, const void *str2, size_t n)
Copies the array of type `char` pointed to by `str2` into the array pointed to by `str1`. `n` or fewer elements are copied. If the arrays overlap, the behavior is undefined. Returns `str1`.

void *memmove(void *str1, const void *str2, size_t n)
Copies the array of type `char` pointed to by `str2` into the array pointed to by `str1`. `n` or fewer elements are copied. The arrays may overlap. Returns `str1`.

char *strcpy(char *str1, const char *str2)
Copies the null-terminated string pointed to by `str2` into the string pointed to by `str1`. If the strings overlap, the behavior is undefined. Returns `str1`.

char *strncpy(char *str1, const char *str2, size_t n)
Copies exactly `n` characters from string pointed to by `str2` into the string pointed to by `str1`, truncating or null-padding if necessary. If the strings overlap, the behavior is undefined. Returns `str1`.

Concatenation Functions

char *strcat(char *str1, const char *str2)
Concatenates the string pointed to by `str2` at the end of the string pointed to by `str1`. The null-terminating character of `str1` is overwritten by the first character of `str2`. If the strings overlap, the behavior is undefined. Returns `str1`.

char *strncat(char *str1, const char *str2, size_t n)
Concatenates `n` or fewer characters of the string pointed to by `str2` at the end of the string pointed to by `str1`. The null-terminating character of `str1` is overwritten by the first character of `str2`. If the strings overlap, the behavior is undefined. Returns `str1`.

Comparison Functions

int memcmp(const void *buf1, const void *buf2, size_t n)
Compares the first `n` bytes of `buf1` and `buf2`. Each byte is considered to be an `unsigned char`. Returns:

- a positive integer if `buf1` is greater than `buf2`.
- zero if the two buffers are the same.
- a negative integer if `buf1` is less than `buf2`.

int strcmp(const char *str1, const char *str2)
Compares characters in two strings and returns:

- a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
- zero if the two strings are the same.
- a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

int strcoll(const char *str1, const char *str2)
Compares characters in two strings. The comparison function is affected by the current locale category LC_COLLATE. Returns:

- a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
- zero if the two strings are the same.
- a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

int strncmp(const char *str1, const char *str2, size_t n)
Compares the first `n` characters of two strings and returns:

- a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
- zero if the first `n` characters of the two strings are the same.
- a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

size_t strxfrm(char *str1, const char *str2, size_t n)
Transforms sufficient characters in the string pointed to by `str2` such that the string pointed to by `str1` will have `n` or fewer characters. The transformation does not change the expected result of the strcoll function. Returns the number of characters in the transformed string.

Search Functions

void *memchr(const void *str, int ch, size_t n)
Returns a pointer to the first location of the character `ch` (interpreted as `unsigned char`) in the first `n` characters of the string pointed to by `str`. If no character is found, it returns NULL.

char *strchr(const char *str, int ch)
Returns a pointer to the first location of the character `ch` (interpreted as `char`) in the string pointed to by `str`. If no character is found, it returns NULL.

size_t strcspn(const char *str1, const char *nset)
Returns the length of the largest initial substring of the string pointed to by `str1` consisting only of characters not contained in the string pointed to by `nset`.

char *strpbrk(const char *str1, const char *set)
Searches for characters listed in the string `set` in the string pointed to by `str1`. Returns a pointer to such a character or NULL if a matching character is not found.

char *strrchr(const char *str, int ch)
Returns a pointer to the last location of the character `ch` (interpreted as `char`) in the string pointed to by `str`. If no character is found, it returns NULL.

size_t strspn(const char *str1, const char *set)
Returns the length of the largest initial substring of the string pointed to by `str1` consisting only of characters contained in the string pointed to by `set`.

char *strstr(const char *str1, const char *pattern)
Returns a pointer to the first substring that matches the string pointed to by `pattern` in the string pointed to by `str`. Returns NULL if no such substring is found.

char *strtok(char *str, const char *delimiters)
Divides the string pointed to by `str` into tokens delimited by the characters in the string pointed to by `delimiters`. The first function call returns a pointer to the first token in `str`. Delimiters are overwritten by the NULL character. Subsequent calls to *strtok* must have NULL as the first argument; a pointer to the next token is returned. If a token is not found, the return value is NULL.

Miscellaneous Functions

void *memset(void *buf, int ch, size_t n)
Initializes the first `n` characters of the array pointed to by `buf` with the character `ch`.

char *strerror(int errno)
Returns a pointer to the error message associated with the error number `errno`. A list of the error messages associated with each value of `errno` is contained in Tables 7-8 through 7-15 This is also available as a function-like macro.

size_t strlen(const char *str)
Returns the number of characters in the string pointed to by `str`.

CONVEX Extensions

Functions declared are:

char *index(char *string, int ch)
Returns a pointer to the first location of the character `ch` (interpreted as `char`) in the string pointed to by `string`. If no character is found, it returns NULL.

char *rindex(char *string, int ch)
Returns a pointer to the last location of the character `ch` (interpreted as `char`) in the string pointed to by `string`. If no character is found, it returns NULL.

time.h

ANSI C

Macros defined in this header file are:

NULL
Null pointer constant.

CLOCKS_PER_SEC
The number of times the system clock iterates for each second.

Types defined in this header file are:

size_t
Type returned by the sizeof macro operator.

clock_t
The type returned by the clock function.

time_t
The type returned by the time function.

One structure is defined;

tm
Members of this structure are:

tm_sec
Seconds after the minute.

tm_min
Minutes after the hour.

tm_hour
Hours since midnight.

tm_mday
Day of the month.

tm_mon
Months since January.

tm_year
Years since 1990.

tm_wday
Days since Sunday.

tm_yday
Days since January 1.

tm_isdst
Daylight Saving Time flag: positive if DST is in effect; 0 if DST is not in effect; negative if DST status is unknown.

Time Manipulation Functions

clock_t clock(void)
Returns the execution time for the current process.

double difftime(time_t time1, time_t time0)
Returns the difference in seconds between two times. This is also available as a function-like macro.

time_t mktime(struct tm *timeptr)
Converts the time in `tm` representation to time in `time_t` representation. Returns the converted value or -1 if the conversion cannot be performed.

time_t time(time_t *timer)
Returns the current time in a `time_t` representation. If `timer` is not NULL, the value returned is also stored in `timer`.

Time Conversion Functions

char *asctime(const struct tm *timeptr)
Converts a time in the `tm` representation into a string the form

```
Wed Apr 25 23:26:45 1962\n\0
```

Returns a pointer to the string.

char *ctime(const time_t *timer)
Converts into ASCII a time pointed to by `timer`.

struct tm *gmtime(const time_t *timer)

struct tm *localtime(const time_t *timer)
Converts time returned by the time function, pointed to by `timer`, to a structure containing the broken-down time, correcting for time zone and possible daylight savings time.

size_t strftime(char *str, size_t n, const char *format, const struct tm *time)
Converts `time` into a string pointed to by `str` using the format specified by `format`. The number of characters produced cannot be greater than `n`. If `n` is exceeded, the contents of the string are undefined and 0 is returned; otherwise, the number of characters produced is returned.

POSIX Extensions

One macro is defined:

CLK_TCK
Number of clock ticks for each second.

One external identifier is declared:

tzname
An array of time zone names.

Chapter 6
The Preprocessor

Macros may be defined to avoid embedding numbers in source code. For example, if some functions use π, it is better to define a macro with a meaningful name that contains the value for π rather than to use the number 3.14 throughout the source file. Thus, if greater precision is needed, the value for π need be changed in only one place in the source file.

The preprocessor also supports function-like macros. These macros are shorthand for one or more lines of source code that use one or more parameters. For example, if a set of three statements is used frequently throughout a source file, a macro may be defined that represents the three lines. Then, whenever the three lines are needed, only the macro must be written. This is explained in more detail in the sections below.

Another feature of the preprocessor is its support for conditional compilation. Conditional compilation is a technique used to restrict the source code to be compiled. For example, it may be used to remove parts of the code that produce debugging information that is not needed in the completed application. Conditional compilation also makes it easier to port source code between computer systems.

Preprocessor Directives

Preprocessor directives are indicated by a # symbol which may appear anywhere on a line as long as it is preceded only by white space and C comments. The directive must follow the # symbol on the same line with only blanks, horizontal tabs, or C comments separating the two. The directive remains in effect for only the line that it is on; there are no multiline directives.

Preprocessor directives may have zero or more arguments.

#define

The syntax of this directive is:

```
#define name definition
```

The definition for this directive is terminated by the newline character. In its simplest form, this directive is used to define constants. For example, the following line provides a definition for π:

```
#define PI 3.14159265
```

After defining this constant, it is necessary to use only the identifier PI when the value of π is required.

When more precision is required, the definition of PI may be changed to

```
#define PI (4 * atan(1))
```

Because the definition of a name is terminated by a newline character, it is not limited to one word.

A multiword definition is:

```
#define UCHAR unsigned char
```

Thus, this type of definition replaces commonly used pieces of code.

The utility of this directive is extended by the '\' (backslash or continuation) character. This symbol permits the definition of a name to extend over multiple lines. Lines connected with the '\' character are recognized by the preprocessor as a single line. For example, the following definition is considered to be on one line:

```
#define close_files fclose( file1 );\
    fclose( file2 );\
    fclose( file3 )
```

When the word `close_files` is seen in the source file in which this definition is visible, the preprocessor replaces it with the three C statements.

Finally, this directive may define a macro function when the macro name contains parameters. For example, the previous definition may be written as:

```
#define close_files( file1, file2, file3 )\
    fclose( file1 );\
    fclose( file2 );\
    fclose( file3 )
```

This permits macro definitions to be more flexible. The macro function must have the left parenthesis immediately follow the macro name. No intervening spaces are permitted.

Even though a function-like macro looks like a C function, there are differences:

- Function calls require more overhead; therefore, they may be slower.
- Function-like macros increase the length of a source file.
- Parameters in a macro may have unexpected side effects.

An example of a macro with an unexpected side effect is:

```
#define DIV2( quotient, divisor )\
    if( divisor >= 0 && divisor < sizeof( quotient )\
        quotient >> divisor
```

(This function is correct only if `divisor` is a power of two.) This macro replaces a call to DIV2 with integral division performed by a binary shift. It may have incorrect results if its parameters are defined incorrectly:

```
num1 = 21;
num2 = 1;
result = DIV2( num1, num2++);
```

This macro function does not behave as expected because the second parameter is incremented each time it is used in the boolean expression to which `DIV2` expands.

#include

The #include statement replaces the specified line by the contents of a specified file. The format of this statement is as follows:

```
#include "filename"
```

or

```
#include <filename>
```

The first form searches for the specified file in the directory of the original source file and then in a sequence of standard places. The second form searches for the specified file in only the standard places. `#include` statements may be nested.

#undef

The syntax of this directive is:

```
#undef name
```

This directive removes the definition of a macro name by removing any association between *name* and a macro defined for it using `#define`.

Macro Operators

Two operators are useful in defining a macro. The first is #, the operator that converts a parameter to a string. This operator can be used only with formal arguments in the definition of a function-like macro. Its effect is to surround the parameter with double quotes; it converts the parameter to a string literal. If the actual parameter contains white space, only the white space embedded within text is significant, and contiguous white space is reduced to one blank. Further, if the actual parameter contains symbols that are normally preceded by a backslash in a string literal, such a symbol is inserted automatically into the resulting string literal.

For example, the following function can be used to open a file:

```
#define file_open( filename )\
    fopen( #filename, "r" )
```

This macro function could be used in the following manner:

```
FILE *fp;
fp = file_open( myfile );
```

After the preprocessor has expanded this macro, the following code exists:

```
FILE *fp;
fp = fopen( "myfile", "r" );
```

A second operator that is sometimes used in the macro definition is ##, the token pasting operator. It creates one token from two tokens. For example, it may be used to perform operations on variables that are spelled similarly:

```
#define print_var( num ) \
    printf("var" #num " = %d\n", var##num )
```

```
int var1, var2, var3, var4;
print_var( 3 );
```

After expansion, the following code exists:

```
int var1, var2, var3, var4;
printf("var" "3" " = %d\n", var3 );
```

Conditional Compilation

Many directives can be used in conditional compilation. They are similar to their counterparts in C with one primary exception: the expression that the preprocessor evaluates must compute to a constant *prior* to program execution. Consequently, symbols in an expression are limited to macro names and integers.

The syntax of the `if` directive is:

```
#if expression
```

where *expression* is a combination of expressions constructed with C operators such as `&&`, `||`, and `!`.

An operator provided by the preprocessor is `defined`, which returns `1` if its operand has an existing macro definition. Such is the case if the operand is defined on the *cc* command line with the *-D* compiler option (for more information on this option, refer to Chapter 2, "Compiler Fundamentals" in *CONVEX C User's Guide*), or if the operand was defined in the source file without being undefined prior to its use in `defined`.

If *expression* evaluates to a nonzero number, source code between `#if` and a following `#elif`, `#else`, or `#endif` directive remains in the source file.

The `#else` directive is the preprocessor counterpart to the `else` keyword; the `#elif` directive is the preprocessor counterpart to the `else if` keyword combination. The `#endif` directive terminates the `#if` directive.

For example, the following conditional code compiles different source code for different executing environments:

```
#define CONVEX              1
#define OTHER_COMPUTER      0
assert( !(CONVEX != 0 && OTHER_COMPUTER != 0) );

#if CONVEX
    long long j;
#elif OTHER_COMPUTER
    long j;
#endif
```

In this code, extensions to C are used, depending on the architecture of the execution environment. Note that the assert function is not executed until the program is run, while the conditional compilation occurs before runtime. If both `CONVEX` and `OTHER_COMPUTER` macro constants are nonzero, a diagnostic message is printed and the program halts. (Refer to assert(3) for more information on this function-like macro.)

Two other conditional compilation directives are `#ifdef` and `#ifndef`. These are the same as `#if defined( )` and `#if !defined( )`, respectively.

pragma

The `#pragma` directive is currently ignored by the CONVEX C compiler.

error

The syntax of this directive is:

```
#error preprocessor tokens
```

This directive displays an error message when the preprocessor is executing.

Instead of using the assert function in the previous example, the following code may be used:

```
#if OTHER_COMPUTER != 0 && CONVEX != 0
    #error "illegal environment"
#endif
```

This code has the advantage of producing error messages during compilation time instead of execution time.

line

The syntax of this directive is:

```
#line number [filename]
```

It associates a different line number and possibly a file name with a source file.

This directive is useful when the source file is generated by another program. For example, the origin of the source file may be a translator for another language, such as FORTRAN. This directive can be used to associate errors in the C source file with lines in the FORTRAN source file. For example, if the second line of the FORTRAN source file generated lines 100 through 150 of the C source file, before those lines,

```
#line 2 "f_source"
```

could be generated by the program that translates FORTRAN into C. In this way, errors that occur in processed files may be attributed to the source file that caused the error.

Chapter 7
The asm Statement

The `asm` statement is a CONVEX extension that is only available in the extended and backward-compatible modes. Prior versions of CONVEX C required the `-asm` compiler option to compile source files that contain `asm` statements; this option is *no longer necessary.*

Assembly-Language Statements

The CONVEX C compiler allows you to insert assembly-language statements into a C program by means of the `asm` statement. This statement has the form

```
asm ("assembly_language_instruction");
```

The compiler turns off global optimization and global register allocation when compiling source files that contain `asm` statements. Therefore, isolate functions that use the `asm` statement from functions that can be optimized.

Use the `asm` statement to embed assembly language statements in C code rather than writing an entire routine in assembly-lanuage. For instance:

```
void func(int interr[], int blast[])
{
    int i;

    ;asm ("dsi"); /* disable interrupts */
    for( i=0; i<10; i++ ){
        if( interr[i] )
            blast[i] = 1;
    }
    ;asm ("eni"); /* enable interrupts */
}
```

The extra semicolon before the `asm` statement is good coding practice that forces the directive to work in all cases—even when it follows an `if` statement. If the semicolon is omitted before the `asm` statement that directly follows an `if` statement, the embedded assembly-language statements may be in the wrong part of the conditional.

The available assembly-language instructions are described in the *CONVEX Assembly-Language User's Guide* and the *CONVEX Architecture Reference.*

APPENDIX A
Compiler Directives

A compiler directive provides information that the compiler cannot deduce or instructs the compiler to override conditions that automatically control optimization, vectorization, or parallelization. A compiler directive has the form

/*$**dir** *directive* [, *directive*]*/

The directive must begin with the characters /*$**dir**, followed by one or more of the directives in this appendix, followed by */. You may insert one or more spaces or tabs after the initial /* and before the final */. If two or more directives are contained within the /* */, they are separated by commas.

The scope of a compiler directive is the program unit in which it appears. For directives related to vectorization, the scope is the loop that immediately follows the directive in the program, excluding nested loops. Because the compiler ignores comments, you may surround directive lines by any number of comment lines. Do not, however, include other comments within the same comment delimiters as a directive; use a separate set of /* */ delimiters.

Certain combinations of directives are invalid when used within the same program unit and will cause the program unit to be rejected by the compiler. Table A-1 lists the invalid combinations.

Table A-1: Restrictions on Directive Use

The directive...	Cannot be used with...
force_parallel	scalar, select, synch_parallel, unroll
force_vector	pstrip, scalar, select, synch_parallel, unroll, vstrip
pstrip	force_vector, scalar, unroll
scalar	force_parallel, force_vector, pstrip, select, synch_parallel, vstrip
select	force_parallel, force_vector, scalar, unroll
synch_parallel	force_parallel, force_vector, scalar, unroll
unroll	force_parallel, force_vector, pstrip, select, synch_parallel, vtrip
vstrip	force_vector, scalar, unroll

Information Directives

Information directives provide information to the compiler and may or may not cause the compiler to take any action. The information directives are

- max_trips
- no_recurrence
- no_side_effects

The max_trips Directive

The max_trips directive tells the compiler that the following loop is never executed more than the specified number of times. The format of this directive is

```
max_trips (n)
```

This directive can be used to prevent strip mining, when it might otherwise be performed, by specifying a value of *n* that is less than the vector register length of 128. *n* must be an integer greater than 0.

The no_recurrence Directive

The no_recurrence directive instructs the compiler to disregard an apparent recurrence in a loop. If there is no other impediment to vectorization, the loop is vectorized. The format of this directive is

```
no_recurrence
```

When the no_recurrence directive is used, the compiler breaks the recurrence by arbitrarily removing one or more dependencies of the cycle.

Example:

If j is positive, there is no recurrence.

```
/$dir no_recurrence*/
for (i=0; i<n; i++)
  a[i] = a[i+j];
```

The compiler always processes a no_recurrence directive when the apparent recurrence involves an array element; the compiler always ignores a no_recurrence directive when the apparent recurrence involves a scalar. In the latter case, the compiler knows that a recurrence exists.

Note	Incorrect results may occur if you mistake a real recurrence for an apparent one. Always test vector results against scalar results to determine whether a recurrence is real or apparent.

The no_side_effects Directive

The `no_side_effects` directive instructs the compiler that the specified functions do not modify the value of a parameter or global variable, perform a read or write, or call another routine. The format of this directive is

`no_side_effects` (*func* [, *func*])

The parameter *func* specifies one or more user-defined functions.

This directive allows scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that is never used. The function call can be removed because the function has no side effects—it does not matter whether or not the call is made. Such optimization opportunities usually arise after other optimizations are performed and rarely occur in the original source text.

Although the directive can appear anywhere in a function definition, to be effective, it should be used before the named function is called. Use the directive if the compiler gives the advisory message "More optimization is possible if this function call has no side effects."

Example 1:

A function call with no side effects is invariant with respect to a loop, provided its arguments are loop invariant and the call may be moved out of the loop.

```
/*$dir no_side_effects (F1,F2)*/
   ...
   x = y*F1(5,z) - w;
   ...
/* If the x= does not reach a use of x, the assignment */
/* statement may be removed                            */
```

Example 2:

A function call may inhibit code motion. The directive is not applicable; the user must perform the optimization at the source level. (The source must be modified to add the directive.)

```
for (i=0; i<n; i++) {

/* If f(3) has no side effects and is invariant */
/* z= can be removed from the loop, which may   */
/* make z loop invariant                        */

   z = f(3);
}
```

Equivalent code is

```
  t1 = f3(a);
  for (i=0; i<n; i++) {
      ...
      z = t1;
  }
```

Code motion moves the `z=t1`.

Control Directives

The control directives can be used for optimization, parallelization, and vectorization. The control directives are

- `begin_tasks, next_task, end_tasks`
- `scalar`
- `force_parallel, force_vector`
- `no_vector, no_parallel`
- `prefer_vector, prefer_parallel`
- `pstrip, vstrip`
- `select`
- `synch_parallel`
- `unroll`

The scope of a vectorization directive is the loop immediately following the directive; the scope does not, however, apply to loops nested therein. Some of the directives let you select loops or code sections to be parallelized rather than leaving the choice up to the compiler.

Tasking Directives

The tasking directives let you specify a group of independent tasks for parallel execution. The tasking directives are

- `begin_tasks`
- `next_task`
- `end_tasks`

The `begin_tasks` directive identifies the beginning of the task group; the `next_task` directive identifies each individual task in the group; and the `end_tasks` directive terminates the task group. The following code illustrates the use of these directives:

```
/*$dir begin_tasks*/
  statement;
  ...
/*$dir next_task*/
  statement;
  ...
/*$dir next_task*/
  statement;
  ...
/*$dir end_tasks*/
```

The preceding example is equivalent to the following loop:

```
/*$dir force_parallel*/
for (i=1; i<=3; ++i)
  switch(i) {
  case 1:
    statement;
    break;
  case 2:
    statement;
    break;
  ...
}
```

A maximum of 255 tasks can be specified between a `begin_tasks` and an `end_tasks` directive.

A directive must always be followed by a statement, even if the statement is null. If an `end_tasks` directive immediately precedes a closing brace, place a semicolon after the directive as shown in the following example.

```
for (i=0; i<100; i++)
  {
    /*$dir begin_tasks*/

    for (j=0; j<100; j++)
      ...
    /*$dir next_task*/
    for (k=0; k<100; k++)
      ...
    /*$dir next_task*/
      ...
    /*$dir end_tasks*/;
  }
```

The scalar Directive

The `scalar` directive prevents the loop that follows from being vectorized or parallelized. The format of this directive is

```
scalar
```

The body of the loop may still be vectorized or parallelized if an outer loop interchanges with the scalar loop. The `scalar` directive is useful when the iteration count of the loop is too low for the overhead involved in setting up vectorization, or when the numerical results must be the same as for a scalar loop. This directive can also be used to prevent loop interchange, which may not choose the best loop to interchange when it cannot deduce the iteration counts of the loops involved.

It is possible for the results of a vectorized loop to differ from its scalar equivalent. For example, floating-point sum and product reduction operators may give different answers because the underlying hardware does not process the operands in sequential order.

In the following example, the compiler normally interchanges the i loop with the j loop so that elements of a, b, and c are accessed contiguously. The `scalar` directive ensures that the loop of greater iteration count is retained as the innermost loop.

```
/*$dir scalar*/
int a[1000][28];
for (j=0; j<n; j++)      /* where n=28   */
  for (i=0; i<m; i++)    /* where m=1000 */
    a[i][j] = b[i][j] + c[i][j];
```

In the following example, neither iteration count is sufficient to warrant vectorizing the loops.

```
/*$dir scalar*/
for (i=0; i<n; i++)     /* where n=2 */
{
/*$dir scalar*/
for (j=0; j<m; j++)     /* where m=2 */
  a[i][j] = b[i][j] + c[i][j];
}
```

Force Directives

The force directives tell the compiler that the following loop is to be either parallelized or vectorized regardless of apparent recurrences or loop dependencies. The force directives are

- `force_parallel`
- `force_vector`

The `force_parallel` directive tells the compiler that the iterations of the following loop are independent and that the loop should be parallelized. The `force_vector` directive also implies that the iterations of the following loop are independent but tells the compiler to vectorize, rather than parallelize, the loop. If `force_parallel` and `force_vector` are specified for the same loop, the loop is vectorized and the resulting strip-mine loop is parallelized.

The `force_parallel` and `force_vector` directives ignore any dependencies between iterations that the compiler may have located. Also, even though you use these directives on a loop, you may not get the desired code transformation if the compiler cannot generate the requested code; scalar recurrences usually cause this problem.

It is possible to use a `force_vector` directive with a loop that was fully vectorized and get incorrect answers because the directive causes the compiler to ignore dependencies.

A loop can be executed in the following ways.

If you specify...	Then the loop is processed as...
scalar	serial
vector	vector but not parallel
parallel	parallel but not vector
parallel, vector	parallel outer strip and vector inner strip

Loops may be parallelized with the `force_parallel` directive regardless of whether or not they contain calls. The `force_vector` directive should only be used with fully vectorizable loops. Neither directive can be used with the scalar directive or with the `no_recurrence` directive or an error condition results.

The `force_vector` directive can only be used on innermost loops. The `force_parallel` directive may be used on any parallelizable loop that does not contain a loop preceded by the `force_parallel` directive. The `force_parallel` directive is effective only if the `-O3` compiler option is specified.

Limiting Directives

The limiting directives tell the compiler not to perform either.vectorization or parallelization on the following loop. The limiting directives are:

- `no_vector`
- `no_parallel`

The `no_vector` directive tells the compiler not to vectorize the loop that immediately follows; parallelization is not prevented.

The `no_parallel` directive tells the compiler not to parallelize the loop that immediately follows; vectorization is not prevented.

If the `no_vector` and `no_parallel` directives both precede a loop, the result is the same as if the `scalar` directive were used.

Examples:

```
/*$dir select (10,4,200)*//* Run scalar if the actual trip count is 1-4.
                            Run parallel if the trip count is 5-10.
                            Run vector if the trip count is 11-200.
                            Run parallel-vector if the trip count > 201. */

/*$dir select (0,*,*)*/   /* Run scalar if the loop has no vectorizable code. */

/*$dir select (*,*,*)*/   /* Equivalent to $dir scalar. */
```

The synch_parallel Directive

The `synch_parallel` directive tells the compiler that the following loop should be executed in parallel even though it requires synchronization that might result in less than full efficiency. The format of this directive is

```
synch_parallel
```

This directive is effective only if the `-O3` option is specified on the compiler command line.

The loop in the following example might run faster on a machine with four processors than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

```
/*$dir synch-parallel*/
for (i=1; i<=32; ++i) {
  if (a[i] < 0) {
    a[i] = a[i-1] + b[i];
    d[i] = e[i] * f[i];
  }
}
```

The unroll Directive

The `unroll` directive reduces loop overhead by replicating the body of the loop that follows. Unrolling is performed on scalar loops. The format of this directive is

```
unroll
```

To be eligible for unrolling, a loop must contain no internal branching and must have an iteration count that can be determined by the compiler. The compiler unrolls a loop completely only if its iteration count is known to be less than 5. Partial unrolling takes place after vectorization on loops that are still scalar

For this directive to take effect, optimization level `-O3` or `-O2` must be specified on the compiler command line.

Tips on Using the contact Utility

The `contact` utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a .contact file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the `contact` utility

Using a .contact File

When you invoke `contact`, it prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt. Follow these steps:

1. Create a .contact file in your home directory.

2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke `contact`, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

Aborting the Report

To abort a contact report, either press the interrupt key (usually CTRL-C) or choose the abort option when prompted by the `contact` utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named dead.report in your home directory.

Submitting the dead.report File

After you abort a contact session, the `contact` utility saves the report in a file named dead.report in your home directory. Using the `contact` command with the `-r` option automatically merges the contents of the dead.report file into the new contact session. Enter

contact -r

and `contact` finds the dead.report file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, `contact` returns to the final prompt, which asks you to review, edit, submit, or abort the report.

Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the `fg` (foreground) command lets you toggle back and forth between the `contact` utility and the shell. You cannot, however, use CTRL-Z and `fg` to toggle back and forth if you are using the Bourne shell (`sh`).

Ending a Response

The `contact` utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by `contact`:

~e	start the text editor (defined in the EDITOR environment variable)
~h	display a list of available tilde-escape sequences
~p	print the contact report to the terminal screen
~r *filename*	read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.
~~	insert a single tilde as the first character in the line

Using the contact Utility

The `contact` utility prompts for the following information:

- your name, title, phone number, and corporate name
- name and version of the product
- one-line summary of the problem
- detailed description of the problem
- priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation related to the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts.

Step 1a To invoke the `contact` utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software or documentation question. The next screen illustrates the `contact` command and the system response.

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

Step 1b If there is a .contact file in your home directory, `contact` skips the first prompt. The next screen illustrates the `contact` command and the system response when a .contact file is in your home directory.

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

Step 2 The `contact` utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the `which` (or `whence` if you use `csh`) and `vers` commands to find the version number of the product. Use the `fg` command to return to the session and enter the version number in the form X.X or X.X.X.X.

Step 3 The `contact` utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

Step 4 The `contact` utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the adb(1) or csd(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

Step 5 The `contact` utility prompts for the priority of the problem. The next screen illustrates this prompt and priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

Step 6 The `contact` utility prompts for an explanation of how to reproduce the problem. Please include the command syntax and options you used and anything else you did to make the program run.

Step 7 The `contact` utility prompts for any other pertinent comments. Please include all relevant information.

Step 8 The `contact` utility prompts for suggestions regarding documentation supporting the product. Indicate whether documentation could be revised to address the problem.

Step 9 The `contact` utility asks for names of files necessary to reproduce the problem. The next screen illustrates the `contact` prompt and sample user response.

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

> **Note** Tilde-escape sequences are not recognized in responses to this prompt. In `contact`, a tilde in this section means your home directory. This convention is based on use of the tilde for expanding file names in `csh`.

If files specified are small text files, they are automatically included in the contact report. If the files are too large to be included in this report, `contact` gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the `tar` command (refer to the tar(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

Step 10 The `contact` utility prompts you to review, edit, submit, or abort the contact report. The next screen illustrates this prompt.

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review — review the text of the contact report. You are then prompted again to select an option.

Edit — edit the text of the contact report. If you choose to edit the report, `contact` puts you in your default text editor.

Submit — sends the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the `contact` utility and returns you to the shell.

Abort — saves the text of the report in a file named dead.report in your home directory. This option exits the `contact` utility and returns you to the shell.

Index

_assert function lrm-5-3
_CONVEX_SOURCE
  use lrm-1-2, lrm-1-3
_IOFBF macro lrm-5-19
_IOLBF macro lrm-5-19
_IONBF macro lrm-5-19
_POSIX_ARG_MAX macro lrm-5-9
_POSIX_CHILD_MAX macro lrm-5-9
_POSIX_LINK_MAX macro lrm-5-9
_POSIX_MAX_CANON macro lrm-5-9
_POSIX_MAX_INPUT macro lrm-5-9
_POSIX_NAME_MAX macro lrm-5-9
_POSIX_NGROUPS_MAX macro lrm-5-9
_POSIX_OPEN_MAX macro lrm-5-9
_POSIX_PATH_MAX macro lrm-5-9
_POSIX_PIP_BUF macro lrm-5-9
_POSIX_SOURCE macro lrm-1-2, lrm-1-3
_tolower
  macro lrm-5-5
_toupper
  macro lrm-5-5
#define lrm-6-1
#include
  statement lrm-6-3
#undef, use of lrm-5-2
-asm, compiler option lrm-7-1
.contact file, skipping first prompt by using lrm-B-3

A
abort function
  description lrm-5-27
abs function lrm-5-28
acos function
  description lrm-5-11
Ada routines, calling lrm-3-9
Alaska
  technical assistance for, how to obtain lrm-viii
ANSI C
  compatibility mode features lrm-1-3
argument pointer lrm-3-1
array
  addresses lrm-2-11
  data type lrm-2-11
  dimensions lrm-2-11
asctime function lrm-5-33
asin function
  description lrm-5-11
asm keyword lrm-7-1
asm statement lrm-7-1
assembly-language statements lrm-7-1
assert function
  description lrm-5-3
assert.h contents lrm-5-3
associated documents lrm-vii
  how to order lrm-vii
atan function
  description lrm-5-11
atan2 function
  description lrm-5-12
atexit function lrm-5-27
atof function lrm-5-13, lrm-5-26
atoi function lrm-5-26
atol function lrm-5-26

B
backward-compatible mode
  example lrm-1-3
  long long int lrm-2-4
  overview lrm-1-1
begin_tasks directive lrm-A-4
bibliography lrm-vii
bsearch function lrm-5-27
BUFSIZ macro lrm-5-19

C
cabs function lrm-5-13
calling
  Ada routines lrm-3-9
  FORTRAN functions lrm-3-7
  runtime functions lrm-5-3
calling sequence
  standard lrm-3-2
calloc function
  description lrm-5-27
calls, standard function lrm-3-4
Canada
  technical assistance for, how to obtain lrm-viii
ceil function lrm-5-12
char data type
  description lrm-2-1, lrm-2-12
CHAR_BIT macro lrm-5-8
CHAR_MAX macro lrm-5-8
CHAR_MIN macro lrm-5-8
character data lrm-2-12
character input and output functions lrm-5-21
CHILD_MAX macro lrm-5-9
clearerr function lrm-5-24
CLK_TCK macro lrm-5-33
clock function lrm-5-33
clock_t type lrm-5-32
CLOCKS_PER_SEC macro lrm-5-32
communication with the environment lrm-5-27
comparison functions lrm-5-30
compatibility mode features lrm-1-3
compatibility modes
  backward-compatible lrm-1-1
  conforming lrm-1-1
  default lrm-1-1
  description lrm-1-1
  differences lrm-1-1
  examples lrm-1-2
  extended lrm-1-1
  strict lrm-1-1
compiler directives lrm-A-1
compiling
  conditional lrm-6-4

compiling (cont)
language specifications lrm-1-3
library systems lrm-1-3
multiple modes lrm-1-3
single mode lrm-1-2
compiling examples
backward-compatible mode lrm-1-3
conforming mode lrm-1-2
default mode lrm-1-2
extended mode lrm-1-2
mixed mode lrm-1-4
strict mode lrm-1-2
concatenation functions lrm-5-29
conditional compilation lrm-6-4
conforming compatibility mode
description lrm-1-1
example lrm-1-2
`contact`
aborting the report lrm-B-3, lrm-B-7
editing the report lrm-B-6
ending a response lrm-B-4
ending the report lrm-B-6
including files in the report lrm-B-6
invoking lrm-B-1, lrm-B-4
prerequisites lrm-B-1
prompts lrm-B-4
reporting problems lrm-B-1
restrictions on tilde-escape sequences lrm-B-6
reviewing the report lrm-B-6
skipping first prompt by using .contact file lrm-B-3
step-by-step discussion of prompts lrm-B-4
submitting dead.report file lrm-B-3
submitting the report lrm-B-6
suspending the report lrm-B-3
tilde-escape sequences lrm-B-4
tips on using lrm-B-3
CONVEX Extensions
assert.h lrm-5-3
ctype.h lrm-5-5
math.h lrm-5-13
signal.h lrm-5-16
stdio.h lrm-5-24
stdlib.h lrm-5-31
copying functions lrm-5-29
cos function
description lrm-5-12
cosh function
description lrm-5-12
ctime function lrm-5-33
ctype.h
contents lrm-5-4
cuserid function lrm-5-25
customer support
telephone number for lrm-viii

D

data representations
FORTRAN lrm-3-5
data type
array lrm-2-11
`char` lrm-2-1, lrm-2-12
`double` lrm-2-5
`enum` lrm-2-4
`float` lrm-2-5
`int` lrm-2-1
`long` lrm-2-1
`long double` lrm-2-5
`long float` lrm-2-7
`long int` lrm-2-1
`long long` lrm-2-3
`long long int` lrm-2-3
pointer lrm-2-8
`short` lrm-2-1
`short int` lrm-2-1
string lrm-2-12
`void` lrm-2-8
DBL_DIG macro lrm-5-6
DBL_EPSILON macro lrm-5-7
DBL_MANT_DIG macro lrm-5-6
DBL_MAX macro lrm-5-7
DBL_MAX_10_EXP macro lrm-5-7
DBL_MAX_EXP macro lrm-5-6
DBL_MIN macro lrm-5-7
DBL_MIN_10_EXP macro lrm-5-7
DBL_MIN_EXP macro lrm-5-6
dcvtid function lrm-5-13
dead.report file
submitting lrm-B-3
using `-r` option to submit lrm-B-3
default compatibility mode
description lrm-1-1
example lrm-1-2
difftime function lrm-5-33
direct input and output functions lrm-5-22
directives
compiler lrm-A-1
preprocessor lrm-6-1
div function lrm-5-28
`div_t` type lrm-5-26
documentation
ordering lrm-vii
subscription service, how to apply lrm-vii
`double`
data type lrm-2-5, lrm-2-6
double-precision floating point lrm-2-6

E

EDOM
macro lrm-5-5
use lrm-5-11
`end_tasks` directive lrm-A-4
`enum`
data type lrm-2-4
EOF macro lrm-5-19

ERANGE
  macro lrm-5-5
  use lrm-5-11
errno
  use lrm-5-13
  variable lrm-5-5
errno.h
  contents lrm-5-5
error handling functions lrm-5-24
error reporting lrm-B-1
exit function lrm-5-27
EXIT_FAILURE macro lrm-5-25
EXIT_SUCCESS macro lrm-5-25
exp function lrm-5-12
extended compatibility mode
  description lrm-1-1
  example lrm-1-2

F
fabs function lrm-5-12
fclose function lrm-5-20
fdopen function lrm-5-25
feof function lrm-5-24
ferror function lrm-5-24
fflush function lrm-5-20
fgetc function lrm-5-21
fgetpos function lrm-5-23
fgets function lrm-5-21
figure
  char representation lrm-2-2
  character data representation lrm-2-12
  character string representation lrm-2-13
  double-precision floating representation lrm-2-7
  int representation lrm-2-3
  long long representation lrm-2-4
  pointer representation lrm-2-8
  short int representation lrm-2-2
  single-precision floating representation lrm-2-6
  stack layout lrm-3-3
  top of the runtime stack lrm-3-2
FILE
  description of lrm-4-2
  type lrm-5-20
file access functions lrm-5-20
file input and output
  concepts lrm-4-1
file manipulation paradigm lrm-4-1
file positioning functions lrm-5-23
file types and access modes lrm-4-2
FILENAME_MAX macro lrm-5-19
fileno function lrm-5-25
float
  data type lrm-2-5
float.h contents lrm-5-6
floating-point
  32-bit lrm-2-5
  64-bit lrm-2-5
  data types lrm-2-5
floating-point (cont)
  IEEE format lrm-2-5
  native format lrm-2-5
floating-point range
  double lrm-2-5
  float lrm-2-5
  long double lrm-2-5
  long float lrm-2-8
floor function lrm-5-12
FLT_DIG macro lrm-5-6
FLT_EPSILON macro lrm-5-7
FLT_MANT_DIG macro lrm-5-6
FLT_MAX macro lrm-5-7
FLT_MAX_10_EXP macro lrm-5-7
FLT_MAX_EXP macro lrm-5-6
FLT_MIN macro lrm-5-7
FLT_MIN_10_EXP macro lrm-5-7
FLT_MIN_EXP macro lrm-5-6
FLT_RADIX macro lrm-5-6
FLT_ROUNDS macro lrm-5-6
fmod function
  description lrm-5-12
fopen function lrm-5-20
FOPEN_MAX macro lrm-5-19
force directives lrm-A-6
force_parallel directive lrm-A-6
force_vector directive lrm-A-6
formatted input and output functions lrm-5-21
FORTRAN
  mixing IO with C lrm-3-9
fpos_t type lrm-5-20
fprintf function
  description lrm-5-21
  example lrm-4-1
fputc function lrm-5-21
fputs function lrm-5-22
frame pointer lrm-3-1
fread function lrm-5-22
free function lrm-5-27
freopen function lrm-5-20
frexp function lrm-5-12
fscanf function
  description lrm-5-21
fseek function
  associated macros lrm-5-19
  description lrm-5-23
fsetpos function lrm-5-23
ftell function lrm-5-23
function
  arguments lrm-3-4
  stack layout lrm-3-1
function names
  lrm-3-4
  accessing function names from C lrm-3-5
function-like macros versus functions lrm-5-2
functions
  character input and output lrm-5-21
  communication with the environment lrm-5-27
  compared to function-like macros lrm-5-2
  comparison lrm-5-30

functions (cont)
concatenation lrm-5-29
copying lrm-5-29
direct input and output lrm-5-22
error-handling lrm-5-24
file access lrm-5-20
file positioning lrm-5-23
formatted input and output lrm-5-21
implicit declaration lrm-5-3
integer arithmetic lrm-5-28
memory management lrm-5-27
miscellaneous lrm-5-31
multibyte character lrm-5-28
multibyte string lrm-5-28
operations on files lrm-5-20
pseudo-random sequence generation lrm-5-26
search lrm-5-30
searching and sorting lrm-5-27
string conversion lrm-5-26
string handling lrm-5-29
time lrm-5-32
time conversion lrm-5-33
time manipulation lrm-5-33
further reference lrm-vii
fwrite function lrm-5-22

G
gamma function lrm-5-13
generic pointer lrm-2-8
getc function lrm-5-22
getchar function lrm-5-22
getenv function
description lrm-5-27
gets function lrm-5-22
gmtime function lrm-5-33

H
Hawaii
technical assistance for, how to obtain lrm-viii
header files
assert.h lrm-5-3
ctype.h lrm-5-4
errno.h lrm-5-5
float.h lrm-5-6
limits.h lrm-5-8
locale.h lrm-5-10
math.h lrm-5-11
setjmp.h lrm-5-14
signal.h lrm-5-15
stdarg.h lrm-5-18
stddef.h lrm-5-18
stdio.h lrm-5-19
stdlib.h lrm-5-25
string.h lrm-5-29
time.h lrm-5-32
use of lrm-5-3
HUGE_VAL macro lrm-5-11
hypot function lrm-5-13

I
idcvtd function lrm-5-13
IEEE format
double-precision lrm-2-7
single-precision lrm-2-6
implicit function declaration lrm-5-3
index function lrm-5-31
information directives lrm-A-2
input and output
mixing C and FORTRAN lrm-3-9
program lrm-4-3
int
data type lrm-2-1
INT_MAX macro lrm-5-8
INT_MIN macro lrm-5-8
integer arithmetic functions lrm-5-28
integer range
char lrm-2-2
enum lrm-2-4
int lrm-2-2
long lrm-2-2
long int lrm-2-2
long long lrm-2-3
long long int lrm-2-3
short lrm-2-2
short int lrm-2-2
integer representation
16-bit lrm-2-1
32-bit lrm-2-1
8-bit lrm-2-1
integer type lrm-2-1
ipow function lrm-5-13
ircvtr function lrm-5-14
isalnum function
description lrm-5-4
LC_CTYPE lrm-5-10
isalpha function
description lrm-5-4
LC_CTYPE lrm-5-10
isascii
macro lrm-5-5
iscntrl function
description lrm-5-4
LC_CTYPE lrm-5-10
isdigit function
description lrm-5-4
LC_CTYPE lrm-5-10
isgraph function
description lrm-5-4
LC_CTYPE lrm-5-10
islower function
description lrm-5-4
LC_CTYPE lrm-5-10
isprint function
description lrm-5-4
LC_CTYPE lrm-5-10
ispunct function
description lrm-5-4

ispunct function (cont)
LC_CTYPE lrm-5-10
isspace function
description lrm-5-4
LC_CTYPE lrm-5-10
isupper function
description lrm-5-4
LC_CTYPE lrm-5-10
isxdigit function
description lrm-5-4
LC_CTYPE lrm-5-10

J

j0 function lrm-5-14
j1 function lrm-5-14
jmp_buf type lrm-5-14
jn function lrm-5-14

K

kill function lrm-5-17

L

L_ctermid macro lrm-5-25
L_cuserid macro lrm-5-25
L_tmpnam macro lrm-5-19
labs function lrm-5-28
LC_ALL macro lrm-5-10
LC_COLLATE macro lrm-5-10
LC_CTYPE macro lrm-5-10
LC_MONETARY macro lrm-5-10
LC_NUMERIC macro lrm-5-10
LC_TIME macro lrm-5-10
lconv structure lrm-5-10
LDBL_DIG macro lrm-5-6
LDBL_EPSILON macro lrm-5-7
LDBL_MANT_DIG macro lrm-5-6
LDBL_MAX macro lrm-5-7
LDBL_MAX_10_EXP macro lrm-5-7
LDBL_MAX_EXP macro lrm-5-7
LDBL_MIN macro lrm-5-7
LDBL_MIN_10_EXP macro lrm-5-7
LDBL_MIN_EXP macro lrm-5-6
ldexp function lrm-5-12
ldiv function lrm-5-28
ldiv_t type lrm-5-26
limiting directives lrm-A-6
limits.h contents lrm-5-8
LINK_MAX macro lrm-5-9
locale.h
contents lrm-5-10
localeconv function
description lrm-5-11
LC_MONETARY lrm-5-10
LC_NUMERIC lrm-5-10
localtime function lrm-5-33
log function
description lrm-5-12
log10 function
description lrm-5-12
long
data type lrm-2-1
long double
data type lrm-2-5
long float
data type lrm-2-7
long int data type lrm-2-1
long long data type lrm-2-3
long long int
data type lrm-2-3
function parameter, backward-compatible mode lrm-2-4
LONG_MAX macro lrm-5-8
LONG_MIN macro lrm-5-8
longjmp function
description lrm-5-14
lpow function lrm-5-14

M

malloc function
description lrm-5-27
math domain error return values lrm-5-13
math.h
contents lrm-5-11
MAX_CANON macro lrm-5-9
MAX_INPUT macro lrm-5-9
max_trips directive lrm-A-2
MB_CUR_MAX macro lrm-5-25
MB_LEN_MAX macro lrm-5-8
mblen function lrm-5-28
mbstowcs function lrm-5-28
mbtowc function lrm-5-28
memchr function lrm-5-30
memcmp function lrm-5-30
memcpy function lrm-5-29
memmove function lrm-5-29
memory management functions lrm-5-27
memset function lrm-5-31
miscellaneous functions lrm-5-31
mixed-mode
example lrm-1-4
mktime function lrm-5-33
modf function lrm-5-12
multibyte
character functions lrm-5-28
string functions lrm-5-28
multiple mode compiling lrm-1-3

N

NAME_MAX macro lrm-5-9
native format
double-precision lrm-2-7
single-precision lrm-2-6
next_task directive lrm-A-4
no_recurrence directive lrm-A-2
no_side_effects directive lrm-A-3
note
IEEE 754 specifications lrm-2-5

note (cont)
`long float` is a CONVEX extension lrm-2-7
`long long int` is a CONVEX extension lrm-2-3
`no_recurrence` directive lrm-A-2
restrictions on tilde-escape sequences with `contact` lrm-B-6
NULL macro
use lrm-5-18, lrm-5-19, lrm-5-25, lrm-5-29, lrm-5-32

O

`offsetof` macro lrm-5-18
operations on files lrm-5-20
operators
pasting (##) lrm-6-3
stringizing (#) lrm-6-3
optimization
directives lrm-A-4
options
`-std` lrm-1-2
ordering documentation
how to lrm-vii

P

padding
structures lrm-2-11
parallelization
directives lrm-A-4
parameter passing to FORTRAN lrm-3-6
pasting operator
lrm-6-3
pclose function lrm-5-24
perror function lrm-5-24
pointer
data type lrm-2-8
generic lrm-2-8
pointers
to arrays lrm-2-12
popen function lrm-5-24
POSIX
conforming applications lrm-1-2
definition lrm-1-1, lrm-5-1
POSIX Extensions
limits.h lrm-5-9
setjmp.h lrm-5-15
signal.h lrm-5-16
stdio.h lrm-5-25
time.h lrm-5-33
pow function lrm-5-12
preference directives lrm-A-7
preprocessor
description of lrm-6-1
directives lrm-6-1
operators lrm-6-3
preprocessor directive
#define lrm-6-1
#elif lrm-6-4
preprocessor directive (cont)
#else lrm-6-4
#endif lrm-6-4
#error lrm-6-5
#if lrm-6-4
#ifdef lrm-6-4
#ifndef lrm-6-4
#line lrm-6-5
#pragma lrm-6-5
#undef lrm-6-3
printf function lrm-5-21
processor status word lrm-3-2
program counter lrm-3-2
program input and output lrm-4-3
pseudo-random sequence generation functions lrm-5-26
`pstrip` directive lrm-A-8
`ptrdiff_t` type lrm-5-18
putc function lrm-5-22
putchar function lrm-5-22
puts function lrm-5-22

Q

qsort function lrm-5-27

R

raise function lrm-5-15
rand function
return value lrm-5-26
with `RAND_MAX` lrm-5-25
`RAND_MAX` macro lrm-5-25
rcvtir function lrm-5-14
Reader's Forum lrm-viii
realloc function
description lrm-5-27
remove function
description lrm-5-20
rename function
description lrm-5-20
reporting problems lrm-viii
representation
char lrm-2-2
character data lrm-2-12
`double` lrm-2-7
`enum` lrm-2-4
`float` lrm-2-6
int lrm-2-2
long lrm-2-2
`long double` lrm-2-7
long int lrm-2-2
`long long` lrm-2-4
`long long int` lrm-2-4
pointer lrm-2-8
`short` lrm-2-2
`short int` lrm-2-2
string lrm-2-12, lrm-2-13
rewind function lrm-5-23
rindex function lrm-5-31

runtime
functions, calling lrm-5-3
library lrm-5-1
stack lrm-3-1

S

scalar, optimization directive lrm-A-5
scanf function lrm-5-21
SCHAR_MAX macro lrm-5-8
SCHAR_MIN macro lrm-5-8
search functions lrm-5-30
searching and sorting functions lrm-5-27
SEEK_CUR macro lrm-5-19
SEEK_END macro lrm-5-19
SEEK_SET macro lrm-5-19
select, optimization directive lrm-A-8
setbuf function lrm-5-20
setjmp function
description lrm-5-14
setjmp.h contents lrm-5-14
setlocale function lrm-5-11
setvbuf function
associated macros lrm-5-19
description lrm-5-20
short
data type lrm-2-1
short int data type lrm-2-1
SHRT_MAX macro lrm-5-8
SHRT_MIN macro lrm-5-8
sig_atomic_t type lrm-5-15
SIG_BLOCK macro lrm-5-16
SIG_SETMASK macro lrm-5-16
SIG_UNBLOCK macro lrm-5-16
sigaction function lrm-5-17
sigaction structure lrm-5-16
sigaddset function lrm-5-17
sigcontext structure lrm-5-16
sigdelset function lrm-5-17
sigemptyset function lrm-5-17
sigfillset function lrm-5-17
sigismember function lrm-5-17
sigjmp_buf type lrm-5-15
siglongjmp function lrm-5-15
sigmask macro lrm-5-16
signal function lrm-5-15
signal.h
contents lrm-5-15
sigpending function lrm-5-17
sigprocmask function lrm-5-17
sigprocmask, associated macros lrm-5-16
sigsetjmp function
description lrm-5-15
sigsuspend function lrm-5-17
sigvec structure lrm-5-16
sin function
description lrm-5-12
single mode compiling lrm-1-2
single-precision
IEEE format lrm-2-6
native format lrm-2-6
sinh function
description lrm-5-12
size_t type lrm-5-18, lrm-5-20, lrm-5-26, lrm-5-29, lrm-5-32
sprintf function lrm-5-21
sqrt function
description lrm-5-12
srand function lrm-5-26
sscanf function lrm-5-21
stack frame lrm-3-2
stack pointer lrm-3-1
stdarg.h contents lrm-5-18
stddef.h contents lrm-5-18
stderr device lrm-5-19
stdin device lrm-5-19
stdio.h
contents lrm-5-19
stdlib.h
contents lrm-5-25
stdout device lrm-5-19
strcat function lrm-5-29
strchr function lrm-5-30
strcmp function lrm-5-30
strcoll function lrm-5-30
strcpy function lrm-5-29
strcspn function lrm-5-31
strerror function
description lrm-5-31
strftime function
description lrm-5-33
LC_TIME lrm-5-10
strict compatibility mode
description lrm-1-1
example lrm-1-2
string concatenation operator (#) lrm-6-3
string conversion functions lrm-5-26
string handling functions lrm-5-29
string.h
contents lrm-5-29
stringizing operator (#) lrm-6-3
strip-mining
directives lrm-A-7
parallel lrm-A-7
strlen function lrm-5-31
strncat function lrm-5-29
strncmp function lrm-5-30
strncpy function lrm-5-29
strpbrk function lrm-5-31
strrchr function lrm-5-31
strspn function lrm-5-31
strstr function lrm-5-31
strtod function lrm-5-26
strtok function lrm-5-31
strtol function lrm-5-26
strtoul function lrm-5-26
struct
data type lrm-2-10
representation lrm-2-10
structure
member alignment lrm-2-10
padding lrm-2-11

strxfrm function lrm-5-30
synch_parallel directive lrm-A-9
system function
  description lrm-5-27
system functions and ANSI functions lrm-4-2

T
table
  compatibility modes lrm-1-1, lrm-5-1
  errno values of fgetpos, fsetpos, and ftell lrm-5-23
  floating-point bit length lrm-2-5
  FORTRAN and C declarations lrm-3-6
  integral ranges lrm-2-2
  integral type bit length lrm-2-1
  long float range: native and IEEE lrm-2-8
  long long data type range lrm-2-3
  math function return values lrm-5-13
  maximum strip-mine lengths lrm-A-7
  native and IEEE floating-point ranges lrm-2-5
  restrictions on optimization directives lrm-A-1
TAC (Technical Assistance Center) lrm-viii, lrm-B-1
tan function lrm-5-12
tanh function lrm-5-13
tasking directives lrm-A-4
  listed lrm-A-4
technical assistance
  obtaining lrm-viii
technical assistance center
  telephone number for lrm-viii
technical assistance center (TAC) lrm-viii, lrm-B-1
tilde-escape sequences
  restrictions on use lrm-B-6
  use in contact lrm-B-4
time
  conversion functions lrm-5-33
  functions lrm-5-32
  manipulation functions lrm-5-33
time function lrm-5-33
time_t type lrm-5-32
time.h contents lrm-5-32
tm structure lrm-5-32
TMP_MAX macro lrm-5-19
tmpfile function lrm-5-20
tmpnam function
  description lrm-5-20
  L_tmpnam lrm-5-19
  TMP_MAX lrm-5-19
toascii
  macro lrm-5-5
tolower function lrm-5-4
toupper function lrm-5-5
trouble reports lrm-viii, lrm-B-1
type conversion from FORTRAN lrm-3-5
tzname variable lrm-5-33

U
UCHAR_MAX macro lrm-5-8
UINT_MAX macro lrm-5-8
ungetc function lrm-5-22
union
  data type lrm-2-9
UNIX-to-UNIX
  Communication Protocol lrm-B-1
  copy command, uucp lrm-B-1
unroll directive lrm-A-9
USHRT_MAX macro lrm-5-8
UUCP
  connection to TAC lrm-B-1
uucp
  UNIX-to-UNIX copy command lrm-B-1

V
va_arg macro lrm-5-18
va_end macro lrm-5-18
va_list type lrm-5-18
va_start macro lrm-5-18
vectorization
  directives lrm-A-4
vers command
  using to find program version number lrm-B-2
vfprintf function lrm-5-21
void data type
  description lrm-2-8
vprintf function lrm-5-21
vsprintf function lrm-5-21
vstrip directive lrm-A-8

W
wchar_t type lrm-5-18, lrm-5-26
wcstombs function lrm-5-28
wctomb function lrm-5-28
whence command
  using to find program path name lrm-B-2
which command
  using to find program path name lrm-B-2

Y
y0 function lrm-5-14
y1 function lrm-5-14
yn function lrm-5-14

CONVEX C Language Reference Manual
720-001230-001, First Edition

Reader's Forum

Please use this form to submit comments or questions concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us continue in our efforts to generate quality customer documentation. Please list the page number for questions or comments.

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____Title_____

Company_____Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

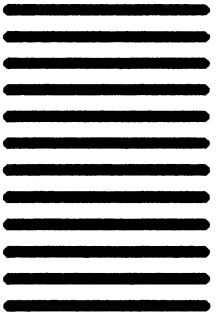
Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska, Hawaii, & Canada	1(214)497-4379
From all other locations	Contact nearest CONVEX office

Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851
USA

(Fold Here Second)

(Tape or Staple)