CONVEX ANSI C Concepts

Document No. 720-003130-000

First Edition
May 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX ANSI C Concepts
Order No. DSW-083
First Edition

Printed in the United States of America

Revision Information for
CONVEX ANSI C Concepts

Edition	Document No.	Description
First	720-003130-000	Released with CONVEX C V4.0, April 1990. First release of the manual.

Table of Contents

Appendices

List of Tables

Preface

Purpose and Audience

This document defines terms and concepts to help you understand and use the CONVEX C compiler. It also describes the changes that ANSI C imposes on existing C applications and the process to convert those applications for use with CONVEX C.

This document is intended for those who want to understand CONVEX C V4.0, an ANSI C compiler. Some familiarity with the C language is required.

This document may be used to understand the reasons for converting to the ANSI C standard as well as some of the problems that may be encountered in any conversion process.

Organization

This manual is organized as follows:

- Chapter 1, "Introduction to ANSI C," provides a brief history of C and some features of ANSI C.
- Chapter 2, "CONVEX C Compilers," differentiates between each of the CONVEX C compilers.
- Chapter 3, "Converting to CONVEX C V4.0," describes steps required to convert applications to CONVEX C V4.0.
- Chapter 4, "Implementation-Defined Features," lists the features of CONVEX C that may render it incompatible with other ANSI C compilers.
- Appendix A, "Extended Mode Differences," describes differences between CONVEX C V4.0's extended mode and other CONVEX C compilers.
- Appendix B, "Incompatibilities of Common C," lists incompatibilities between the Common C compiler and the backward-compatible mode of CONVEX C.
- Appendix C, "Reporting Problems," provides instructions for reporting software or 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-R 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 this software successfully may require information not specific to the tasks described herein or not within the scope of this guide.

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

- *CONVEX C User's Guide* describes how to compile programs with the CONVEX C compiler.
- *CONVEX C Language Reference Manual* describes some topics of the C language. One chapter describes ANSI C features that are specific to the CONVEX compiler, and another chapter details the differences between the Common C compiler and the backward-compatible mode of CONVEX C V4.0.
- *CONVEX C Optimization Guide* presents a step-by-step method of program optimization. Background information is included and forms a foundation for concepts presented throughout the document.
- *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.
- *CONVEX UNIX Tutorial Papers* is a collection of previously published papers that provides instruction in document preparation, programming, text editing, supporting tools and languages, system maintenance, and system implementation.
- *ConvexOS Utilities User's Guide* provides detailed information on the CONVEX Assembler, Loader, text editors, and `adb` debugger.

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.

In some situations, the current edition may not be desired. 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 wish 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. Thank you.

Chapter 1
Introduction to ANSI C

This chapter contains an overview of the C language. The first section gives a short history of C and the second section details some new features provided by ANSI C.

ANSI C History

The C language was created by Dennis Ritchie for use on the UNIX operating system. It is a general-purpose programming language that contains many of the features of other high-level languages such as user-defined data types. It generates compact code that is comparable to hand-coded assembly-language programs.

Programs that use only high-level features of C are easier to port than programs that use assembly-language features. However, restriction to high-level language features does not guarantee that an application can be easily ported to another computer system. The original definition of C does not define all aspects of the language.

Computer vendors provide compiler extensions to take advantage of computer architecture and increase C application performance or reduce development time. So, even though an application uses only the high-level features of C, its use of vendor enhancements ensures that the program is incompatible with another computer vendor's compiler.

Programs that are not portable require modifications to run on other computer systems. There is a productivity tradeoff between fine-tuning an application for each computer architecture, and creating a program that runs on many different systems. The former executes faster while the latter executes in more environments.

The American National Standards Institute (ANSI) began discussions about a standard C language which encompassed the runtime libraries as well as the language itself. The impact on libraries is important because they differed most among vendors. Goals of the committee were to ease maintenance, increase efficiency, and increase the reliability of C applications. The committee made tradeoffs between backward compatibility and requirements that ANSI C be independent of operating systems. Consequently, conversion to ANSI C may require some changes in existing applications.

ANSI C Features

This section details some beneficial aspects of ANSI C, including:

- Preprocessor enhancements.
- Improved diagnostics.
- Increased efficiency.

Preprocessor Enhancements

Several enhancements have been added to the preprocessor. Two new operators may be used in definition of `#define` macros. One is #, the operator that converts a macro parameter to a string. A second is the pasting operator, ##, that concatenates two tokens.

For example, macro expansion of:

```
#define PRINT_NUM(x) printf("Number = " #x "\n")
PRINT_NUM(5);
```

results in:

```
printf("Number = " "5" "\n");
```

Similarly, macro expansion of:

```
#define PRINT_VAR(x) printf("var" #x " = %d\n", var##x)
PRINT_VAR(5);
```

results in:

```
printf("var5 = %d\n", var5);
```

Another addition is the `#elif` preprocessor directive. This functions like an `else if` construct in C. This directive makes conditional compilation code clearer. For example, the code:

```
#if exp1
 ...
#else
#if exp2
 ...
#endif
#endif
```

is equivalent to:

```
#if exp1
 ...
#elif exp2
 ...
#endif
```

Improved Diagnostics

The ANSI C standard has several additional features that improve the compiler's ability to detect errors. These include:

- Function prototypes.
- Stricter type checking.

Function Prototypes

Function prototypes are templates that specify data types of function parameters and the return type of a function. The compiler uses prototypes to verify that data of the correct type is passed to a function, and to perform type conversions on the arguments when necessary.

For example, compilers that do not accept ANSI C only recognize function declarations that do not have formal parameters:

```
double haar();
```

The function prototype of this function is:

```
double haar(int group[]);
```

All functions in a program do not require a prototype. If a function lacks a function prototype, one is implicitly defined when the compiler encounters the function definition. This function definition may be written in the ANSI C style, or in the old style that specifies the formal parameters outside of the parentheses. All succeeding references to this function must match this implicit prototype definition. If there are discrepancies, the compiler generates diagnostic messages.

Stricter Type Checking

Stricter type checking helps prevent errors resulting from improper data type use. The compiler will generate diagnostic messages when it detects the assignment of one data type to another without explicit casting. ANSI C has specified more closely which types are compatible and how they interact to permit more errors to be detected. Stricter type casting is one such improvement.

Increased Efficiency

Several features of ANSI C allow generation of more efficient code:

- Function prototypes.
- Arithmetic conversions.
- `const` and `volatile` data types.
- Aliasing.

Function Prototypes

In non-ANSI C, the `float` data type was promoted to `double` before being passed as an argument to a function. This conversion introduced additional overhead in function calls.

ANSI C does not perform this conversion if a function prototype exists.

Arithmetic Conversions

ANSI C does not require arithmetic operations on `float` data types to be converted to `double`. This eliminates unnecessary conversions. If additional precision is required, explicit casting must be used.

const and volatile Data Types

ANSI C defines two type qualifiers that affect optimizations the compiler can perform: `const` and `volatile`. Qualifiers are used in a declaration to modify the interpretation of the identifier being declared. For example, the declaration

```
const int x = 10;
```

indicates that the value contained in `x` may never be modified.

This has several implications for the compiler. First, `x` may be located in read-only memory. Second, the compiler can assume that no values will be assigned to `x` after it has been initialized. This permits the compiler to perform additional optimizations.

The `volatile` qualifier indicates that the value of its identifier may change at any time. The qualifier can inhibit optimization, so it should be used with care.

Aliasing

An *alias* is an alternate name for some object. Aliasing occurs in a program whenever two or more names point to the same object. A potential alias occurs when the compiler cannot tell whether two memory locations, which have separate names, are distinct.

The compiler can use two aliasing algorithms to compile a program. The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`.

The second aliasing algorithm is used only when the `-alias worst` compiler option is selected. It permits no such assumptions to be made; `int` pointers can point to objects of type `float`. This aliasing algorithm increases the number of potential aliases that may occur, decreasing the ability of the compiler to optimize a program.

For more information on the `-alias worst` compiler option, refer to the *CONVEX C Optimization Guide.*

Chapter 2
CONVEX C Compilers

This chapter describes CONVEX C V4.0. Particular attention is paid to the relationship between this compiler and previous CONVEX C compilers. This close relationship is important because it eases the transition from those previous compilers to ANSI C. There are two sections:

- CONVEX C compilers.
- Compatible modes.

The first section details the compilers that were available on CONVEX computers prior to the release of CONVEX C V4.0. The second section describes the four compatible modes of CONVEX C V4.0.

CONVEX C compilers

Prior to the release of CONVEX C V4.0, there were two C compilers that were available on CONVEX computers:

- Common C compiler (`cc`)
- CONVEX C V3.0 (`vc`)

The Common C compiler, formerly called the Portable C compiler, was available on many UNIX systems; it was enhanced to take advantage of CONVEX hardware. It performs very few optimizations. CONVEX computers that had this compiler prior to the release of CONVEX C V4.0 will retain it, but it is not supported. It can be invoked with the `/bin/pcc` command name.

CONVEX C V3.0 is an optional product that is no longer supported. It is capable of vector and parallel optimizations. Sites that have a licensed copy of this compiler can access it as the backup C compiler.

The CONVEX C V4.0 compiler replaces these two compilers, which are no longer supported. There are two releases of this compiler: one that is capable of only scalar optimizations and another that can perform vector and parallel optimizations. The former is available on all CONVEX computer systems, while the latter is an optional product available only to licensed sites. Because both releases of this compiler are invoked with the `cc` command name, only one may be installed.

Compatibility Modes

The CONVEX C V4.0 compiler conforms with "ANSI -- Programming Language C," but it is also closely related to the two compilers that it is replacing. Four modes enable it to support both the ANSI C standard and the obsolete compilers:

- Strict.
- Conforming.
- Extended.
- Backward-compatible.

The strict mode of the compiler accepts applications that use only the ANSI C language features and functions. Applications that are compiled in the strict mode should port easily to other ANSI C compilers. Potential incompatibilities are listed in Chapter 4, "Implementation-Defined Features."

The conforming mode conforms to the language specification of ANSI C and also provides access to POSIX functions. The POSIX standard explicitly defines the system functions that a C program may use to interact with the ConvexOS environment. POSIX has no impact on the language accepted by the compiler. This mode exists so that applications may be developed on a POSIX-compliant operating system with an ANSI C compiler in an environment conforming the ANSI C Standard and POSIX IEEE Std. 1003.1-1988.

The extended mode accepts programs written in ANSI C with a few minor language extensions. These CONVEX extensions can be used to tailor an application to the CONVEX architecture, improving performance at the expense of portability to other architectures. Some of the functions that are in CONVEX C V3.0 are available as CONVEX extensions in this compatibility mode. This is the default mode of the compiler.

The extended mode also provides access to POSIX functions. The POSIX libraries contain some of the functions that are in CONVEX C V3.0.

The backward-compatible mode is compatible with the Common C compiler and CONVEX C V3.0. It accepts none of the new features of ANSI C; its use requires the least number of changes to existing C applications. This should be the first mode used to compile non-ANSI C programs as they are ported to CONVEX C V4.0.

Applications that were compiled with CONVEX C V3.0 will require few changes; the backward-compatible mode matches the V3.0 compiler closely. Applications that were compiled with the Common C compiler will require modifications; the differences between the backward-compatible mode and the Common C compiler are listed in Appendix B, "Incompatibilities of Common C."

Procedures for converting applications compiled with either CONVEX C V3.0 or the Common C compiler to one of the compatibility modes are in Chapter 3, "Converting to CONVEX C V4.0."

Chapter 3
Converting to CONVEX C V4.0

This chapter details the steps required to convert programs to the backward-compatible or the extended mode of the CONVEX C V4.0 compiler. These steps pertain to programs that were compiled with CONVEX C V3.0 or the Common C compiler.

Several sources of information in this guide are useful in this conversion process:

- Chapter 4, "Implementation-Defined Features."
- Appendix A, "Extended Mode Differences."
- Appendix B, "Incompatibilities of Common C."

These sources list problems that may be encountered during the conversion process.

This chapter divides the conversion process into two parts. The first part details the steps necessary to port an application to the backward-compatible mode of CONVEX C V4.0; the second part details the procedure to port an application from the backward-compatible mode to the extended mode of CONVEX C V4.0.

Porting to Backward-Compatible Mode

Three steps are required to port applications compiled with CONVEX C V3.0 compiler or the Common C compiler to the backward-compatible mode of CONVEX C V4.0:

1. Compile under ConvexOS.
2. Remove lint errors.
3. Recompile in backward-compatible mode.

Step 1: Compile under ConvexOS

The first step requires the application to be compiled under ConvexOS V8.0 or later. If the application was originally compiled using the Common C compiler, it is necessary to use the `pcc` command because CONVEX C V4.0 is now invoked with the `cc` command line. CONVEX C V3.0 remains on the CONVEX systems on which it was previously installed as the backup C compiler. Any problems found in this step are probably caused by a lack of compatibility between ConvexOS and the environment in which the application was previously compiled and executed.

Step 2: Remove lint errors

The second step removes system-dependent code and code that may produce errors. This increases the portability of the application to CONVEX C V4.0. The suggested command line for this step is:

```
lint -c -h *.c
```

where

`*.c`	represents names of source files.
`-c`	detects errors that occur in casting.
`-h`	applies heuristic tests to discover errors.

Removing system-dependent code reduces chances for an error when CONVEX C V4.0 is used, or when libraries that are linked with the program are changed.

Step 3: Recompile in Backward-Compatible Mode

The third step exposes errors caused by differences between the compiler used in the first step and the CONVEX C V4.0 compiler. The command line for this step is:

```
cc -pcc *.c
```

where

`*.c`	represents names of source files.
`-pcc`	directs the compiler to execute in backward-compatible mode.

Some changes that may be required for programs that were compiled with the Common C compiler are listed in Appendix B, "Incompatibilities of Common C."

Porting to Extended Mode

After an application has been ported to the backward-compatible mode of CONVEX C V4.0, it can be ported to the extended mode.

As in the conversion to the backward-compatible mode, there are two steps:

1. Link with extended libraries.
2. Compile in extended mode.

Step 1: Link with extended libraries

Linking with ANSI C libraries removes the dependence on the old system libraries. Command lines for this step are:

```
cc -pcc -c *.c
cc *.o
```

where

`*.c`	represents names of source files.
`*.o`	represents names of object files.
`-c`	prevents the object files from being linked
`-pcc`	directs the compiler to execute in backward-compatible mode.

The first command line compiles source files with the backward-compatible mode of the compiler. The second command line links resulting object files with functions available in the extended mode of the compiler.

Failures at this stage are likely to be caused by differences between standard library functions and backward-compatible library functions, or by the application's dependence on undocumented behavior of old library routines.

Incompatibilities between backward-compatible mode libraries and ANSI C libraries are listed in Appendix A, "Extended Mode Differences."

Step 2: Compile in Extended Mode

The extended mode of the ANSI C compiler provides the ANSI C features, POSIX functions, and CONVEX extensions. The command line used at this step is:

```
cc *.c
```

where

`*.c`	represents names of source files.

Failures at this step are probably caused by differences between ANSI C and the definition of C accepted by the backward-compatible mode of the compiler. These language differences are documented in Appendix A.

Chapter 4
Implementation-Defined Features

This chapter states the implementation-defined features of the ANSI C compatibility mode of CONVEX C. Using these features may cause problems when a program is being transferred to or from another computer system. When such a transfer takes place, the implementation-defined features of the two compilers must be compared to determine what changes are required in the program.

Implementation-defined features in this chapter are organized according to the ANSI X3J11/90-013 document "American National Standard for Information Systems -- Programming Language C," Appendix A.6.3, "Portability Issues: Implementation-defined Behavior."

All these features pertain only to the strict compatibility mode of CONVEX C, except where noted.

Translation

- A file that contains one or more syntax errors will generate a message in one of the following forms when it is being compiled:

  cc: Error/Warning on line *line* of *file*: *message*
  cpp: *file line*: *message*

  where

cc	indicates a compiler warning or error.
cpp	indicates a preprocessor error.
file	is the name of the file that contains the error.
line	is the line on which the error was detected.
message	is the warning or error message that the syntax error generates.

Environment

- The main function has three arguments: argc, argv, and envp. argc contains the number of arguments on the program command line, argv is an array of strings each of which is an argument on the command line, and envp is a pointer to an array of strings that constitute the environment of the program. The last array element in argv and envp is a NULL pointer.

- The interactive devices available to an executable program are contained in the /dev directory. The files have the names tty*, pty*, and console.

Identifiers

- Every character in an identifier with *internal linkage* (one that is not visible in another compilation unit) is significant.
- An identifier with *external linkage* (one that is used in another compilation unit) has 41 significant initial characters.
- Characters in an identifier with external linkage are case sensitive.

Characters

- Source and execution character sets are ASCII characters.
- No multibyte character encodings are implemented in this version of CONVEX C.
- There are eight bits in a character in the execution character set.
- Mapping of members of the source character set to members of the execution character set is one to one.
- The value of character constants containing undefined escape sequences will be the same value as the character constant without the back slash. For example, '\c' is the same as 'c'. Also, the values of '~', '@', '$', and any control characters between single quotes will be the ASCII value for that character.
- If a character constant contains more than one character, its value is obtained using the following pseudocode:

```
int value = 0;
while( more characters )
    value = ( value << 8 ) + ( value of next character );
```

Because an integer occupies four bytes of memory, the value of a character constant that has more than four characters will consist of only the last four characters. An example of character representations is contained in Table 4-1.

Table 4-1: Character Constant Representation

Constant	Representation
'a'	0x61
'abc'	0x616263
'abcdef'	0x63646566

The result is considered a signed value. In the extended mode, because `value` can be type `long long int`, the character constant can hold up to eight characters.

- One locale is available with CONVEX C: `"C"`. Multibyte characters are not implemented.
- An unqualified `char` type has the same range of values as a `signed char` type.

Integers

- Integers are represented in two's complement form. Ranges for the integral types are enumerated in the *CONVEX C Language Reference Manual*, Chapter 2, "Data Types and Representations."

- Converting an integer to a shorter signed integer is the same as transferring the low order N bits of the source integer, where N is equal to 8 * sizeof(destination). Converting from an unsigned quantity to a signed quantity of the same length does not change the representation; the most significant bit of the unsigned quantity is interpreted as the sign bit. Some of these conversions are shown in Table 4-2.

Table 4-2: Integer Conversion

int	cast to short
-2147483647	1
65535	-1
unsigned short	**cast to short**
65535	-1
32768	-32768

- Performing a bitwise operation on a signed integer has the same result as a bitwise operation on an unsigned integer of the same value.

- The sign of the remainder on integer division is the same as the sign of the numerator.

- A right shift of a negative-valued signed integral type results in a positive value; sign replication does not occur.

Floating-Point

- The data representations and ranges of the floating-point numbers are described in the *CONVEX C Language Reference Manual*, Chapter 2, "Data Types and Representations."

- When an integral number is converted to a floating-point number that cannot exactly represent the original number, it is rounded to the nearest value that can be represented.

- When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest value that can be represented.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array is `size_t`, defined as an `unsigned int`.
- Pointers and integers can be cast to each other without any changes to the underlying bit pattern.
- The type of integer required to hold the difference between two pointers to elements of the same array is `ptrdiff_t` defined as an `int`.

Registers

- The `register` storage class has no effect on register usage except in the presence of `asm` statements.

Structures, unions, enumerations, and bit-fields

- If a member of a union object is accessed after a value has been stored in a different member of the object, the value of the member depends the types of the two members.

  For example, given:

```
union {
    short snum;
    int  inum;
} num;
```

  the `short` representation overlays the third and fourth bytes of the `int` representation because members in a union have the same address. Consequently, the first byte of a `short` overlays the third byte of an `int`.

  Similarly, if a union object contains two members, a `float` and a `char`, the `char` overlays the exponent and sign bit of the `float`. Refer to the *CONVEX C Language Reference Manual*, Chapter 2 for the representations of data types.

- 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.

- An unqualified `int` bit field is treated as an `unsigned int` bit field.
- The order of allocation of bit fields within an `int` is from the most-significant bit position to the least significant bit position.
- A bit field cannot straddle a word (32-bit) boundary, but can straddle a byte (8-bit) or half-word (16-bit) boundary.
- An enumerated data type is represented as a `signed int`.

Qualifiers

- Any object that has a `volatile`-qualified type is accessed when its value is used or modified at optimization level `-no`.

Declarators

- The number of declarators that may modify an arithmetic, structure, or union type is at least 12.

Statements

- The maximum number of `case` values in a `switch` statement is 257.

Preprocessing Directives

- Character constants in conditional inclusion directives are unsigned.
- The method used for locating a file in a `#include` directive depends on its delimiters. A file name in a `#include` directive is delimited by double quotes or angle brackets.

  If the file name is delimited by double quotes, directories specified by the `-I` option on the compiler command line are searched. If this option is not used, the directory in which the compilation takes place is searched, followed by the system directory /usr/include.

  Alternatively, if the file name is delimited by angle brackets, directories specified by the `-I` option of the compiler command line are searched. If the file is not found, the system directory /usr/include is examined.

- As indicated in the previous item, source files in the `#include` directive may be delimited by double quotes.
- Source file character sequences are mapped to the ASCII character set.
- When a `#pragma` directive is encountered, it is ignored because no `#pragma` directives are recognized in CONVEX C V4.0.
- The system translation of __DATE__ and __TIME__ are always available.

Library functions

- The `NULL` macro expands to the integer value 0.
- The diagnostic message displayed by the assert function is of the form:

  `Assertion failed:` *<integer expression>*`, file` *<name>*`, line` *<number>*

  After assert prints the diagnostic message, the abort function is executed which results in an IOT trap.
- The information in Table 4-3 indicates the sets of characters that cause the functions isalnum, isalpha, iscntrl, islower, isprint, and isupper to return a true value.

Table 4-3: Characters Checked by ctype.h Functions

Function	Returns True Value For
isalnum	any letter or digit
isalpha	any letter
iscntrl	delete character or ASCII character with value < 32
islower	any lower case letter
isprint	ASCII code 32 through 126, inclusive
isupper	any upper case letter

- Table 4-4 indicates the values returned by math functions when a domain error occurs.

Table 4-4: 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
log	log(|x|)
log10	log10(|x|)
pow	pow(|x|)
sin	sin(0)
sinh	-HUGE_VAL
sqrt	sqrt(|x|)

HUGE_VAL is defined in the math.h header file. It is the largest floating-point value. It may not be representable as a `float`.

- The mathematics functions do not set the integer expression `errno` to the macro ERANGE on underflow errors.
- When the fmod function has a second argument of zero, a domain error occurs and a zero is returned by the function.
- Signals that can be used by the signal function are listed in Table 4-5.
- Table 4-5 also lists semantics for each signal recognized by the signal function.

Table 4-5: Available Signals and their Semantics

Signal	Semantics
SIGABRT	abort
SIGALRM	alarm clock
SIGBUS	bus error
SIGCHLD	to parent on child stop or exit
SIGCONT	continue a stopped process
SIGEMT	EMT instruction and sigabrt
SIGFPE	floating point exception
SIGHUP	hangup
SIGILL	illegal instruction
SIGINT	interrupt
SIGIO	IO possible signal
SIGIOT	IOT instruction
SIGKILL	kill (cannot be caught or ignored)
SIGLOST	resource lost
SIGPIPE	write on a pipe with no process to read it
SIGPOLL	IO possible signal
SIGPROF	profiling time alarm
SIGQUIT	quit
SIGSEGV	segmentation violation
SIGSTOP	sendable stop not from tty
SIGSYS	bad argument to system call
SIGTERM	software termination signal from kill
SIGTRAP	trace trap
SIGTSTP	stop signal from tty
SIGTTIN	background read attempted from control terminal
SIGTTOU	background write attempted to control terminal
SIGURG	urgent condition on IO channel
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGVTALRM	virtual time alarm
SIGWINCH	window changed
SIGXCPU	cpu time limit exceeded
SIGXFSZ	file size limit exceeded

- The default action for each of the signals is listed in Table 4-6.

Table 4-6: Default Actions for Signals

Signal	Default Action
SIGABRT	terminate program and dump core image
SIGALRM	terminate program
SIGBUS	terminate program and dump core image
SIGCHLD	discard signal
SIGCONT	discard signal
SIGEMT	terminate program and dump core image
SIGFPE	terminate program and dump core image
SIGHUP	terminate program
SIGILL	terminate program and dump core image
SIGINT	terminate program
SIGIO	discard signal
SIGIOT	terminate program and dump core image
SIGKILL	terminate program
SIGLOST	terminate program and dump core image
SIGPIPE	terminate program
SIGPOLL	discard signal
SIGPROF	terminate program
SIGQUIT	terminate program and dump core image
SIGSEGV	terminate program and dump core image
SIGSTOP	suspend process
SIGSYS	terminate program and dump core image
SIGTERM	terminate program
SIGTRAP	terminate program and dump core image
SIGTSTP	suspend process
SIGTTIN	suspend process
SIGTTOU	suspend process
SIGURG	discard signal
SIGUSR1	terminate program
SIGUSR2	terminate program
SIGVTALRM	terminate program
SIGWINCH	suspend process
SIGXCPU	terminate program
SIGXFSZ	terminate program

- After a signal occurs, its signal is blocked.
- The default handling is not reset if the `SIGILL` signal is received by a handler specified to the signal function.
- The last line of a text stream does not require a terminating new-line character.
- Space characters that are written out to a text stream immediately before a new-line character are not eliminated.

- No null characters are appended to the data written to a binary stream.
- When a file is opened in the append mode, the file position indicator is initially positioned at the end of the file.
- When a write occurs on a text stream, the associated file is not truncated beyond that point.
- CONVEX C supports unbuffered, fully buffered, and line buffered file input and output. The `stdin`, `stdout`, and `stderr` streams are line buffered.

  Two functions affect buffering of a file: setbuf and setvbuf. These functions replace the file input and output buffers that are automatically allocated by an application with an array. If the second argument of these functions is `NULL`, the input and output is unbuffered.
- It is possible to have zero-length files.
- A file name may consist of 1 to 14 characters. The characters may not include the *slash* or the null character. File names . and .. have special meanings.
- If the remove function is executed on a file that is still open, you can continue to read and write to that file until the file is explicitly closed. At that time, the file ceases to exist.
- If an attempt is made to rename a file to a file that already exists, the existing destination file is erased.
- The %p conversion specification in the fprintf function is used to display the address of a pointer. The address displayed is a hexadecimal number that does not have a `0x` prefix.
- When the conversion specification in the fscanf function is %p, a hexadecimal number is input. This number must not have a `0x` prefix.
- The '-' character in the scanlist of a `%[` conversion specifier has no special meaning: that is, it simply matches a '-' character in the input regardless of where the '-' appears in the scanlist.
- The values to which the macro `errno` is set by the fgetpos or ftell function on failure are listed in Table 4-7.

Table 4-7: errno values of fgetpos and ftell

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

- Tables 4-8 through 4-15 list the messages generated by the perror and strerror functions.

Table 4-8: Error Messages

errno Value	Error Message
EACCES	Permission denied
EAGAIN	No more processes
EBADF	Bad file number
EBUSY	Mount device busy
ECHILD	No children
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EINTR	Interrupted system call
EINVAL	Invalid argument
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENFILE	File table overflow
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOMEM	Insufficient free swap space
ENOSPC	No space left on device
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTTY	Not a typewriter
ENXIO	No such device or address
EPERM	Not owner
EPIPE	Broken pipe
EROFS	Read-only file system
ESPIPE	Illegal seek
ESRCH	No such process
ETXTBSY	Text file busy
EXDEV	Cross-device link
E2BIG	Arg list too long

Table 4-9: Math Error Messages

errno Value	Error Message
EDOM	Argument too large
ERANGE	Result too large

Table 4-10: Nonblocking and Interrupt IO Error Messages

errno Value	Error Message
EALREADY	Operation already in progress
EINPROGRESS	Operation now in progress
EWOULDBLOCK	Operation would block

Table 4-11: Ipc/Network Argument Error Messages

errno Value	Error Message
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Can't assign requested address
EAFNOSUPPORT	Address family not supported by protocol family
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
ENOPROTOOPT	Protocol not available
ENOTSOCK	Socket operation on non-socket
EOPNOTSUPP	Operation not supported on socket
EPFNOSUPPORT	Protocol family not supported
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	Protocol wrong type for socket
ESOCKTNOSUPPORT	Socket type not supported

Table 4-12: NFS Error Messages

errno Value	Error Message
EREMOTE	Too many levels of remote in path
ESTALE	Stale NFS file handle

Table 4-13: SystemV Record Locking Error Messages

errno Value	Error Message
EDEADLK	Deadlock situation detected/avoided
ENOLCK	No record locks available
ENOSYS	Function not implemented

Table 4-14: Ipc/Network Operational Error Messages

errno Value	Error Message
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by peer
EDQUOT	Disc quota exceeded
EHOSTDOWN	Host is down
EHOSTUNREACH	Host is unreachable
EISCONN	Socket is already connected
ELOOP	Too many levels of symbolic links
ENAMETOOLONG	File name too long
ENETDOWN	Network is down
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network is unreachable
ENOBUFS	No buffer space available
ENOTCONN	Socket is not connected
ENOTEMPTY	Directory not empty
EPROCLIM	Too many processes
EREFUSED	Connection refused
ESHUTDOWN	Can't send after socket shutdown
ETIMEDOUT	Connection timed out
ETOOMANYREFS	Too many references: can't splice
EUSERS	Too many users

Table 4-15: Tape System Error Messages

errno Value	Error Message
ETPBADFORM	Labeled tape not in proper format
ETPNOSUPPORT	Tape feature not supported
ETPTRUNC	Logical tape record would be truncated

- The functions calloc, malloc, and realloc return a unique pointer if the size requested is zero. This pointer is not NULL.
- The abort function closes open files and deletes temporary files.
- The exit function returns its argument if that argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`.
- The set of environment names used by the getenv function includes:

  `PATH, HOME, TERM, SHELL, TERMCAP, EXINIT, USER, PRINTER.`

  Other environment names may also be defined. Similarly, it is possible for an environment to have no environment names. setenv can be used to change the environment list used by getenv.

- The argument of the system function may be any valid command for the `sh` command. When system is executed, the current process waits until the shell has completed, then returns the exit status of the shell.

- The contents of the error message strings returned by the strerror function are listed in Tables 4-8 through 4-15.

APPENDIX A
Extended Mode Differences

This appendix discusses problems that may be encountered when porting programs from CONVEX C V3.0 or Common C to the extended mode of CONVEX C V4.0. This is the default compatibility mode of the compiler. Problems include:

- Language features that prevent a non-ANSI C program from being compiled.
- Changes in the semantics of the language.
- Changes in header file organization.
- Future directions in ANSI C.

Most of these changes concern translating existing code to ANSI C. Information provided can be used to assess difficulties in porting programs between systems. Another source of information is Appendix B, "Incompatibilities of Common C." This appendix states the incompatibilities between the Common C compiler and the backward-compatible mode of CONVEX C.

Changes that ANSI C imposes on existing applications can be grouped into two categories: changes that prevent compilation and changes that alter semantics of a construct without inhibiting compilation. Semantic changes are potentially more disruptive because different code may be produced without warning.

Changes That Prevent Compilation

Changes introduced by the ANSI C standard are:

- Five new keywords have been added: `const`, `enum`, `signed`, `void`, and `volatile`. These words in the wrong context will generate an error message.
- Declaring an identifier that is common between two or more compilation units must obey the following rule: only one compilation unit may contain a definition of the identifier; the remaining compilation units must declare the identifier using the `extern` storage class.
- The numerals `8` and `9` are no longer available in octal constants.
- String literals cannot be modified.
- It is not permitted to convert a pointer of any object type (other than `void`) to a pointer of a different object type without an explicit cast.
- Function pointers may not be converted to nonfunction pointers, and vice versa.
- Pointers to functions that have different parameter-type information are different types.
- `long float` is not in the ANSI C standard. Its use generates a warning in all ANSI C modes of CONVEX C.
- `extern` declarations only have block scope in ANSI C.
- Functions called with a variable number of parameters must have a function prototype.

- The `entry` and `asm` statements are not accepted in ANSI C. The `asm` statement is available as a CONVEX extension. `entry` is not a CONVEX extension.
- Accessing a nonexistent member of a structure or union is an error.
- Empty declarations are invalid except for mutually referencing `struct` and `union` structures.
- Representing `register` variables as wider types (as when `register char` is quietly changed to `register int`) is no longer permitted.
- Declaring zero-length arrays is invalid.
- No type specifiers may be added to a type that was defined using `typedef`.
- Formal parameters of a function may not be `typedef` names.
- Predefined macro names, such as __FILE__ and __LINE__, may not be redefined or undefined.

Semantic Changes

A list of semantic changes introduced by the ANSI C standard follows. These semantic changes do not cause the compiler to generate error messages; they may cause a non-ANSI C program to generate incorrect output. Most of these changes are cited in the *ANSI - Programming Language C* document.

- A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered the most serious semantic change that results from the current ANSI C standard.
- The compiler may not reorder expressions that contain successive identical commutative or associative operators.
- Programs with character sequences that are trigraphs, such as `??!`, in string constants, character constants, or header names, will now be replaced by the corresponding character representation of such a trigraph. Trigraphs and characters they represent are listed in Table A-1.

Table A-1: Trigraph Representations

trigraph symbols	represents
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

- A program relying on file scope rules for external declarations may be valid under block scope rules but behave differently.
- Unsuffixed integer constants may have different types. Their type depends on the smallest integral type required to represent them.

- A constant of the form `'\078'` is valid, but now has a different meaning. It now denotes a character constant whose value is the combination of the values of `'\07'` and `'8'`.
- Because the escape sequences `'\a'` and `'\x'` have been added to ANSI C, a constant of the form `'\a'` or `'\x'` may now have different meaning.
- It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme may not behave the same as with a previous C compiler.
- To eliminate ambiguity, the following assignment operators no longer exist:

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

  Further, for the assignment operators of the form +=, no space is permitted between the two characters + and =.
- Expressions with `float` operands may now be computed at lower precision. Compilers prior to the ANSI C standard performed all floating-point operations in `double`.
- Shifting by a `long` count no longer converts the shifted operand to `long`.
- Calculations in `#if` expressions may simulate either the translation environment or the execution environment. Consequently, a program that depends on properties of one particular environment may now give different answers.
- The empty declaration

```
struct x;
```

  is no longer innocuous because it may now be used to hide a definition of `x` that exists in an outer block.
- Code that relies on a bottom-up parsing of aggregate initializers with partially elided braces will not yield the expected initialized object.
- Functions that depend on `char` or `short` parameter types being widened to `int`, or `float` to `double`, may behave differently.
- A macro that relies on formal parameter substitution within a string literal will produce different results.
- Comments in ANSI C are replaced with a blank; in Common C they are removed. If your code relies on `/**/` to paste tokens together in Common C, you must now use ## instead. ## is the paste operator in the ANSI C preprocessor.
- Function-like macros may be recursive, but they won't be recursively expanded.
- Results of floating-point operations may not be the same due to differences in casting and rounding.

Header File Changes

The following is a list of changes in organization of header files. Differences are noted when a function or macro that is present in the CONVEX C V3.0 compiler or Common C compiler is not located in the same header file in an ANSI C mode of CONVEX C V4.0.

References are made to CONVEX extensions and POSIX functions. These extensions and functions are available in the appropriate compatibility modes of CONVEX C V4.0 that are discussed in Chapter 2, "CONVEX C Compilers," of this guide.

This section does not list differences between the backward-compatible mode of CONVEX C V4.0 and the CONVEX C V3.0 compiler. These differences are noted in the Appendix B, "Incompatibilities of Common C."

Changes in organization of header files are:

ctype.h	The four function-like macros isascii, toascii, _toupper, and _tolower do not exist in the ANSI C standard, however, they do exist as CONVEX extensions.
math.h	Single-precision math functions (sfabs, ssqrt, shypot, scabs, ssin, scos, stan, sasin, sacos, satan, satan2, sexp, slog, spow, ssinh, scosh, stanh) are available only in the backward-compatible mode of the compiler. These math functions are available as function-like macros in the extended compatibility mode. Macro definitions `HUGE` and `HUGEI`, while present in the backward-compatible mode, have been replaced by the macro definition `DBL_MAX` in the ANSI C modes.
signal.h	Macro names for the signals `SIGCLD`, `SIG_CATCH`, and `SIG_HOLD` are no longer available. The function-like macro SIGNALS_IN_PROG does not exist. These four macros are available only in the backward-compatible mode of CONVEX C.
stdio.h	Functions fileno and fdopen are POSIX functions.
strings.h	This header file no longer exists. All its functions are now contained in the string.h header file.

The functions creat, close, lseek, open, read, write, and unlink are available as POSIX extensions.

Signal handlers must execute signal(*signal*,`SIG_DFL`) before they process a signal.

Future Directions

Changes to the C language that are expected to be made in the future are listed below. While some of these changes may not occur, you should be aware of them to avoid possible incompatibilities in the future. These are taken from "ANSI - Programming Language C" document.

Changes concerning the C language:

- Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolescent feature that is a concession to existing C compilers.
- Lower-case letters as escape sequences in character and string literals are reserved for future standardization.
- Using old-style function declarations that have empty parentheses will be obsolete.
- Two parameters declared with an array type (prior to their adjustment to pointer type) may not refer to the same or overlapping objects. The `-alias array_args` option of the compiler enforces a stronger version of this requirement. Its use reduces the necessity for `no_recurrence` optimization directive. Refer to *CONVEX C User's Guide* for more information on the `-alias array_args` compiler option.

Changes to library specifications that may be expected in a future standardization of ANSI C are listed below. These changes are taken from the *ANSI C - Programming Language C* document.

errno.h	Macros that begin with 'E' and a digit or 'E' and an upper-case letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the errno.h header.
ctype.h	Function names that begin with either 'is' or 'to', and a lower-case letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the ctype.h header.
locale.h	Macros that begin with 'LC_' and an upper-case letter (followed by any combination of digits, letters, and underscore) may be added to definitions in the locale.h header.
math.h	Names of all existing functions declared in the math.h header, suffixed with 'f' or 'l', may be used for corresponding functions with `float` and `long double` arguments and return values.
signal.h	Macros that begin with either 'SIG' and an upper-case letter or 'SIG_' and an upper-case letter (followed by any combination of digits, letters, and underscore) may be added to definitions in the signal.h header.
stdio.h	Lower-case letters may be added to the conversion specifiers in fprintf and fscanf. Other characters may be used in extensions.
stdlib.h	Function names that begin with 'str' and a lower-case letter (followed by any combination of digits, letters, and underscore) may be added to declarations in the stdlib.h header.
string.h	Function names that begin with 'str', 'mem', or 'wcs' and a lower-case letter (followed by any combination of digits, letters, and underscore) may be added to declarations in the string.h header.

APPENDIX B
Incompatibilities of Common C

This appendix details the incompatibilities between the backward-compatible mode of CONVEX C V4.0 and the Common C compiler. This list of incompatibilities is provided to ease the transition to the backward-compatible mode of CONVEX C.

The incompatibilities are divided into two sections:

- Language definition.
- Command line.

Most of the incompatibilities between the two compilers result from illegal code that is accepted by the Common C compiler. Illegal C code is not accepted in any compatibility mode of CONVEX C.

Language Definition

Type Qualifiers

The Common C compiler permits type qualifiers to modify a data type created with `typedef`. For example,

```
typedef int my_int;
typedef unsigned my_int u_my_int;
```

This is illegal C code; such qualifiers are not allowed in the backward-compatible mode of CONVEX C.

static and extern Mixing

Mixing `static` and `extern` in a declaration is undefined in the C language; such declarations are not portable between compilers.

Multiple Initializers

The Common C compiler erroneously permits simple variables to have multiple initializers. For example,

```
{
extern int init1(), init2(), init3();
int x = { init1(), init2(), init3() };
}
```

When compiled with Common C, three functions are executed, with the last one assigning its return value to x. Multiple initializations are not permitted with the CONVEX C compiler in any of its compatibility modes.

Casts

The Common C compiler permits objects on the left side of an assignment operation to be cast. For example,

```
{
int x;
(int)x = 10;
}
```

This is allowed only when the cast is the same type as the variable or is a pointer to the same type as the variable. This is an error in the Common C compiler. CONVEX C detects this error.

Order of Evaluation

The order of expression evaluation used by the two compilers is not the same. This will impact numerical computations and may change the order in which side effects occur. The changes may cause a program that produced correct results with the Common C compiler to produce incorrect results with the CONVEX C compiler. The program is not a valid C program because it depends on the order of evaluation in ways not specified by the language. For example:

```
a[i++] = b[i]
```

will not produce the same results on both compilers. For example, given:

```
#include <stdio.h>

int a[2] = {0, 1};
int b[2] = {3, 4};

main(){
    int i = 0;
    a[i++] = b[i];
    printf("a[0] = %d\n", a[0] );
}
```

If compiled with `/bin/pcc file.c`, this program displays `a[0] = 3`, but if it is compiled with `cc -pcc file.c`, it displays `a[0] = 4`.

Uninitialized Variables

Applications that depend on uninitialized variables, dangling pointers, and other poor programming practices may not have the same behavior on both compilers.

Negative Bit Shifts

The Common C compiler allows bit shifts to use negative operands. For example, a right shift with a negative right operand is equivalent to a left shift with a positive right operand. Negative operands of bit shift operators generate errors in the CONVEX C compiler if the shift length is a constant.

Switch Statements with Pointers

The Common C compiler allows the expression in a `switch` statement to have a pointer value. The CONVEX C compiler reports an error if the expression does not have integral type.

Undefined Functions

The Common C compiler silently converts functions to the `extern` storage class if they are declared `static` and used in a compilation unit but not defined. The CONVEX C compiler produces a warning when it performs this conversion.

Common C Functions Returning short int or char

Undeclared functions that return a `short int` or `char` have their return value automatically widened, in the calling routine, to an `int` by the Common C compiler; this matches the default declaration of a function. CONVEX C does not perform this automatic conversion. Programs that depend on this behavior will produce erroneous results. However, `lint` is capable of finding the offending errors.

Command Line Differences

Obsolete Options

The `-t` compiler option provided by the Common C compiler is silently ignored by CONVEX C.

The semantics of the `-B` compiler option are slightly different between the two compilers.

APPENDIX C
Reporting Problems

This appendix introduces the CONVEX Technical Assistance Center (TAC) and `contact` utility. The `contact` utility is an online system for reporting problems to the TAC. To learn `contact` by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using `contact`, read through this appendix. It describes prerequisites and tips for using `contact` and the step-by-step process `contact` takes you through.

Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

The contact Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

Prerequisites

To use `contact` requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- full path name of the program or utility in question
- version number of the program or utility in question

UUCP Connection

Before using `contact`, check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The `uucp` (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

Finding the Program Path Name

To determine the full path name of the program or utility in question, use the `which` command. The next screen illustrates using the `which` command to find the full path name of the loader (`ld`) utility.

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is /bin/ld.

For more information on the `which` command, refer to the which(1) man page. You can also use the `info` online information system. Enter **info which** at the system prompt.

If you use the C shell (`csh`), you can also use the `whence` command to find the program path name. The `whence` command works like `which`, but faster.

Finding the Program Version Number

To determine the version number of the program or utility in question, use the `vers` command. The next screen illustrates using the `vers` command to find the version number of the loader (`ld`) utility. Enter **vers**, then the path name of the program or utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the `vers` command, refer to the vers(1) man page. You can also use the `info` online information system. To do so, enter **info vers** at the system prompt.

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

D

E

F

CONVEX ANSI C Concepts
720-003130-000, 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:

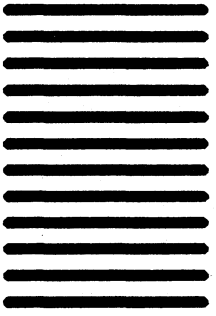
Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska, Hawaii, & Canada	1(214)497-4379
From all other locations	Contact nearest CONVEX office

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)