CONVEX Portable C Compiler
User's Guide

Document No. 720-000530-202

Fourth Edition
March 1989

CONVEX Computer Corporation
Richardson, Texas

CONVEX Portable C Compiler User's Guide
Order No. DSW-007
Fourth Edition

Printed in the United States of America

Revision Information for
CONVEX Portable C Compiler
User's Guide

Edition	Document No.	Description
Fourth	720-000530-202	Released with CONVEX UNIX V7.1, March 1989. The name of the product was changed from CONVEX C to the portable C compiler and a complete editorial revision was performed.
3.0	720-000530-201	Released with CONVEX UNIX V6.1, October 1987
2.0	720-000330-000	Released with CONVEX UNIX V3.0, January 1986
1.0	720-000330-000	Released with CONVEX UNIX V1.0, February 1985 First release of the manual.

Table of Contents

Appendices

List of Figures

Preface

This manual tells you how to use the portable C compiler and describes the enhancements that have been implemented by CONVEX Computer Corporation.

The generally accepted standard definition of the C language is contained in B.W. Kernighan and D.M. Ritchie's *The C Programming Language* (Prentice-Hall, 1978). Throughout this manual, all references to Kernighan and Ritchie or to *The C Programming Language* refer to this book.

Intended Audience

It is assumed that the reader is familiar with the C language and with the use of the UNIX operating system.

Organization

This manual consists of the following chapters and appendices:

Chapter 1 contains a general overview of the C programming language and includes a review of the language constructs.

Chapter 2 describes the commands used to compile, load, debug, and execute C programs.

Chapter 3 describes CONVEX extensions to the C language.

Chapter 4 describes the data types available in the portable C compiler.

Chapter 5 describes the processes involved in making assembly-language function calls.

Appendix A lists compiler error messages.

Appendix B tells you how to report software and documentation problems.

Associated Documents

The following documents are recommended:

- *CONVEX Architecture Handbook* - Describes the CONVEX instruction set and the architecture of CONVEX computers.
- *CONVEX Assembly Language User's Guide* - Describes the CONVEX Assembler.
- *CONVEX UNIX Programmer's Manual, Parts I and II* - Documents the UNIX operating system.

- *CONVEX adb Debugger User's Guide* - Describes how to use the *adb* debugger to debug programs at the assembly-language level.
- *CONVEX Consultant User's Guide* - Documents an optional program package that includes the CONVEX *csd* debugger, the post-mortem dump (*pmd*) utility, and the *gprof*, *bprof*, and *prof* profilers.
- *CONVEX UNIX Primer* - Contains a basic introduction to the UNIX operating system and its use.
- *CONVEX Loader User's Guide* - Describes how to use the loader for specific applications.
- *The C Programming Language*, B. Kernighan and D. Ritchie, Prentice-Hall, Inc., 1978 - Describes the C language and contains both a tutorial introduction and a reference manual.

Notational Conventions

The following conventions are used in this document:

- Mnemonics enclosed in "less than" and "greater than" signs designate ASCII nonprintable characters. For example, <CR> stands for carriage return.
- Brackets [ ] designate optional entries.
- A horizontal ellipsis . . . shows repetition of the preceding item(s).
- A vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- References to the *CONVEX UNIX Programmer's Manual* appear in the form *csd*(1), where the name of the manual page is followed by its section number enclosed in parentheses.
- *Italics* within text indicate commands, file names, or programs.
- Within command sequences and text set off from regular text, **boldface** type indicates literals. Words appearing in boldface should be typed just as they appear. *Italics* within command sequences indicate generic commands or filenames. Substitute actual commands or filenames for the italicized words.

Chapter 1
C Language Overview

The C programming language is used widely for systems and scientific programming. The primary components of the language are constructs that map well onto the architecture of modern scientific computers. C also supports address arithmetic operations.

NOTE

With the current release of software and documentation, the name of the compiler is changed from "CONVEX C" to "portable C." The command to invoke the compiler (*cc*) remains the same.

The basic C data types correspond directly to the primitive scalar data types handled by the architecture of CONVEX computers. Thus, although the portable C compiler does not directly support the vector processing capabilities of the CONVEX hardware, operations on C data types are fast and efficient.

C is not a strongly typed language; that is, the compiler permits data of a particular type to be passed to functions that have already declared the data to be of a different type. A separate utility, *lint*, provides semantic checks that allow you to avoid this and similar problems. The compiler passes arguments to functions by using the "call-by-value" approach (it sends functions the value of arguments, rather than the arguments themselves). Variable addresses are also passed, however, to achieve the effects of the FORTRAN "call-by-reference" operation.

1.1 Basic Structure of C Programs

The basic structure of C programs is as follows:

- C programs generally consist of several function subprograms. (In C, there are no strict distinctions between subroutines and functions.)
- C subprograms may return results to their callers.
- Information is passed between functions via parameters.
- C functions contain variable declarations and a block of executable statements.
- Variables declared within a function are not known to other functions.
- Global variables may be declared in a source file external to the function. The names and values of global variables are visible to all the routines in the program.
- Local variables in a function do not ordinarily retain their values between invocations of the function. The values of variables that are statically allocated, however, are preserved between subsequent executions of the function that defines them.

1.2 Data Types

The C language defines character data types, three sizes of integer data types, and floating-point data types. In addition

- C supports structured aggregate data (called *struct*) in multidimensional arrays.
- Structured variables may include bit-field definitions.
- C supports an enumerated scalar data type (called *enum*) that permits one of a set of scalar values (RED, BLUE, or GREEN, for example) to be assigned to a variable.
- C allows several variables to have their storage locations made equivalent via the *union* construct.
- C also includes user-defined data types (called *typedef*).
- C permits the creation of strongly typed pointer variables that can point to any data type, including functions.
- The values returned by C functions may be of any data type, including *struct*.

1.3 Control Structures

The C language contains all the program control statements required to write well-structured programs. C supports the following program control statements:

- Loop constructs (including both *case* statements similar to the PASCAL CASE statement, and statements that perform loop tests at the top or the bottom of the loop).
- Statements used to terminate loop iteration prematurely or to exit from a loop.
- Conditional statements (such as the *if-else* statement) used for "short-circuit evaluation" of a condition.

"Short-circuit evaluation" means that you always evaluate the expressions in the conditional part of a statement from left to right. Branching occurs as soon as you can establish the validity of the conditional statements. Short-circuit evaluation streamlines the evaluation of conditional statements containing multiple parts and reduces the need for deeply nested *if* statements.

1.4 Logical and Arithmetic Operators

C contains a rich set of basic arithmetic and logical operators. Bitwise and simple Boolean operators are provided, as are left-shift, right-shift, and ternary conditional operators. The ternary conditional operator selects one of two user-supplied values on the basis of a third value. Ternary operations in effect provide an *if-then-else* capability that you can use within expressions.

1.5 Symbolic Constants and Conditional Compilation

The C compiler software package contains a sophisticated preprocessor called *cpp* that is used for the processing of symbolic constants and for the expansion of macros. The preprocessor also makes possible the in-line substitution of constant expressions referenced in C source files and text-forming macros. Macro arguments are substituted one-for-one in the position specified in the macro definition.

Chapter 2
Using the Compiler

This chapter tells you how to compile and execute C programs. The use of the loader, runtime libraries, and debuggers is also described.

2.1 Compilation Process

The portable C compiler compiler uses a four-step process to translate C language source programs into code that you can execute directly on a CONVEX computer. This process is as follows:

1. The compiler calls the C preprocessor, which processes the symbolic constants, conditional compilation statements, include file references, and textual macro substitutions found in the source file. The preprocessor performs these translations and expansions and generates a temporary file.

2. The compiler translates the expanded C source text into symbolic assembly-language code and creates a second temporary file.

3. The compiler invokes the assembler, which transforms the symbolic assembly-language source text into a compiled binary object module.

4. Finally, the compiler invokes the loader if an executable program image is to be created. The loader combines the compiled object modules of the program with compiled modules from a library (or libraries) of UNIX system routines. The final executable output from the loader is written to a file called *a.out*.

2.2 Invoking the Compiler

The C compiler is usually invoked from the UNIX shell. Once invoked, the compiler produces an executable image file from any number of C source files, assembly-language source files, and previously compiled object files. Use the following command to invoke the compiler:

cc [*options*] *file(s)*

where:

options refers to the compiler options shown in the following table, while *file(s)* refers to any number of C source files, assembly-language source files, and previously compiled object files.

Compiler Options

-B*string* Uses the compiler programs, *cpp*, *ccom*, and *cpo* in the files named *string* to which the suffixes *cpp*, *ccom*, and *cpo*, respectively, are appended. If no *string* is specified, the standard backup files (*/lib/old.cpp*, */lib/old.ccom*, and */lib/old.cpo*) are used.

-c Suppresses the loading phase of the compilation and forces production of an object file even if only one program is compiled.

-fd Causes generated code to operate on 32-bit (single-precision) floating-point numbers. This option may enhance program execution speed, but the results may have less precision than if 64-bit floating-point numbers were used.

-fi Specifies that real constants are to be translated into IEEE format and processed in IEEE mode. If you specify this option, your machine must be equipped with the IEEE support hardware or an error message occurs and compilation terminates. If you do not specify a floating-point format, your site default is used.

NOTE

The CONVEX hardware and software only support the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifiations for arithmetic.

-fn Specifies that real constants are to be translated into native format and processed in native mode. If you do not specify a floating-point format, your site default is used.

-g Causes the compiler to produce additional symbol table information for *csd* and *pmd*. Also instructs the compiler to pass the *-lg* option to the loader.

-O Invokes an object code improver.

-OL Generates an optional improvement summary onto standard error after performing object code improvement.

-p Produces code that counts the number of times each routine is called. If loading takes place, replaces the standard startup routine by one that automatically calls *monitor*(3) at the start and arranges to write a *mon.out* file at normal termination of the object program. An execution profile can then be generated using *prof*(1).

-pg Causes the compiler to produce counting code in the manner of *-p*, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. The compiler searches a profiling library in lieu of the standard C library. An execution profile can then be generated using *gprof*(1).

-pb Causes the compiler to produce source-level counting code that produces an execution profile named *bmon.out* at normal termination. Listings of source-level execution counts can then be obtained using *bprof*(1).

-E Runs only the preprocessor on the named C programs and writes the result onto standard output.

-C Prevents the preprocessor from removing comments from the processed source text.

-D*name*[=*def*] Causes the preprocessor to define the symbolic constant *name* as if a *#define* directive had been read in the source file. If no definition is given with "=", the name is defined with the value 1. The names "convex" and "unix" are predefined.

-I*dir* Creates a search path for *#include* files whose names do not begin with /. This option may be used more than once.

-o *name* Causes the compiler to produce an executable file called *name*. If this option is not specified, the default name assigned to the executable file is *a.out*.

-S Causes the compiler to place the generated assembly-language source text into the corresponding *.s* files.

-U*name* Causes the preprocessor to remove any built-in definition of *name*. The only built-in names defined by the compiler are "convex" and "unix".

-V Displays the compiler version number in the form *a.b.c.d*, where *a* is the major version number, *b* is the minor version number, *c* is the internal release number, and *d* is the internal generation number.

-w Suppresses the output of diagnostic warning messages.

-t[p][0][2] Instructs the compiler to use only the designated substitute compiler program on the files named with the *-B* option. (If you do not use the *-B* option, the *-Bstring* is taken to be */lib/exp.*) If you specify *-tp*, the preprocessor phase (*cpp*) is selected; if you specify *-t0*, the compiler phase (*ccom*) is selected; if you specify *-t2*, the object code improvement phase (*cpo*) is selected. If you specify *-t* without any other arguments, all phases are selected.

2.3 File Naming Conventions

The compiler distinguishes file types by their extensions. C source programs, for example, are identified by the *.c* extension. The CONVEX file naming conventions are as follows.

File Names for...	End With the Extension...
C source files	*.c*
Assembly-language source files	*.s*
Compiled object module files	*.o*
Libraries	*.a*

The object modules produced during the compilation of source file *myfile.c* is given the name *myfile.o*. Compiling and loading a single C program all at once normally causes the *.o* file to be deleted.

2.4 Compilation Examples

The following command causes the C source text contained in the file *myfile.c* to be compiled, assembled, and loaded. The executable program file created during this process is named *a.out*.

cc myfile.c

The following command executes the program:

a.out

Assume *myfile.c* is a module that supplements a large program contained in a file named *main.c*. To compile both of these files, enter the following command. The resulting executable module is written to *a.out*.

cc main.c myfile.c

To produce an executable *a.out* file from the object modules *main.o* and *myfile.o*, enter the command sequence:

cc main.o myfile.o

Since no *.c* files are listed on the command line, no actual compilation takes place.

When you do not use the *-c* option, the compiler invokes the loader after any successful compilation. The executable file created is called, by default, *a.out*. The *-o* option can be used to specify a different name for the executable file. For example, if you enter the command sequence:

cc -o final main.c myfile.c

The command causes the C source files *main.c* and *myfile.c* to be compiled and loaded to create the executable program file *final*.

2.5 Generating Assembly-Language Source Files

Occasionally, the assembly-language source program produced during the compilation of a C source program may have to be examined. The *-S* compiler option causes the assembly-language source text generated by the compiler to be written to a file but does not invoke the assembler. The assembly-language source is written to a file with the same name as the file that contains the C source text except that the *.c* file name extension is replaced with *.s*.

If necessary, you can assemble the generated assembly-language source with the *as* command. For example, the command sequence:

cc -S sample.c
as sample.s

compiles the file *sample.c* into an assembly-language source file named *sample.s* and then assembles that file to produce an object module file named *sample.o*.

For a description of the CONVEX Assembler, refer to the *CONVEX Assembly-Language User's Guide*.

2.6 Loading Programs

Normally, the compiler invokes the loader and supplies any necessary command line arguments. If required, the loader can be invoked directly to create an executable file. For information regarding the manual loading of C programs, refer to the *CONVEX Loader User's Guide.*

2.7 Using Runtime Libraries

The C runtime library */lib/libc.a* is only one of many system libraries supplied with UNIX. These system libraries contain precompiled routines that can be combined with user programs. Runtime libraries of precompiled functions can be used to supply graphics packages and other applications software.

Most of the runtime libraries reside in the directories */lib* and */usr/lib*. To scan these libraries, use the *-l* loader option. This option has the following format:

-l*string*

where *string* refers to the root name of a library. If you use this option with the *cc* command, the option is passed to the loader for processing.

For example, the C library */lib/libc.a* is scanned whenever you specify the *-lc* option on the loader command line. Similarly, the mathematical library */usr/lib/libm.a* is scanned whenever a *-lm* option appears in an *ld* command. Because the libraries are scanned in the order entered on the command line, references within a library to a library not yet scanned are not understood.

Runtime libraries are created by the *ar* and *ranlib* utilities. See the *CONVEX UNIX Programmer's Manual* and the *CONVEX Loader User's Guide* for descriptions of these utilities.

2.8 Debugging Programs

Several CONVEX debuggers are available to execute, test, and debug C programs. The assembly-level debugger, *adb*, is described in the *CONVEX adb Debugger User's Guide.* The *csd* debugger is part of an optional package and is used for the source-level debugging of high-level languages. The *csd* debugger is described in the *CONVEX Consultant User's Guide.* The optional post-mortem dump (*pmd*) utility displays information about failed programs that can be used for debugging. To use *pmd*, you must compile your program using the *-g* compiler option.

Executable program files created by the compiler are ready for use with *adb*. For example, the following command sequence invokes the compiler, loader, and *adb* in turn:

```
cc diag.c
adb
```

To debug the program with *csd*, use the following commands:

```
cc -g diag.c
csd
```

In addition to the *CONVEX adb Debugger User's Guide* and the *Convex Consultant User's Guide*, the *CONVEX Guide to Software Development* contains information on debugging techniques.

2.9 Using *lint*

Many C programs consist of separately compiled modules. Because the C compiler operates on single source files, it does not check the number and data types of function arguments in function calls to ensure that they agree with the arguments declared in the function itself. A separate utility, *lint*, performs this function and other semantic checks that the compiler does not make. You should always run *lint* after any program changes. (For more information, see the *CONVEX UNIX Programmer's Manual.*)

Chapter 3
CONVEX Extensions

This chapter describes data types supported by the portable C compiler that are not described in *The C Programming Language*. Two of these data types, *structure* and *enumeration*, are now supported in most versions of C. The third data type, *long long int*, is a 64-bit integer data type developed expressly for use on CONVEX computers.

3.1 Structure Data Type

The portable C compiler allows the assignment of one structure to another via the "=" operator. Structures may also be passed as arguments to functions. Like simple variables, the runtime system copies all of the structures elements to the stack when a structure is passed by value.

Functions may return structures as results in a static memory location. This fact does not impact the recursive nature of such functions, but prevents them from being reentrant in the presence of asynchronous interrupt signals.

3.2 Enumeration Data Type

The *enum* data type described in Chapter 4 is not described in *The C Programming Language*. In the portable C compiler, enumerated variables are distinct from integers. Do not use enumerated values in any context where an integer value may be used unless you use an explicit data type conversion, or *cast*, to convert the enumerated variable into an integer.

3.3 64-Bit Integer Data Type

The portable C compiler supports the native 64-bit integer data type that exists in the CONVEX computer architecture. The compiler supports signed and unsigned 64-bit integer variables, 64-bit integer literal constants, and 64-bit integer input and output.

3.3.1 64-Bit Integer Variables

Use the *long long int* declaration specifier to declare 64-bit integer variables. The *long int* type is 32-bits for backward compatibility with other UNIX implementations.

As with other integers, you may declare *long long* integer variables to be either signed or unsigned. Signed 64-bit integers may be declared wherever signed 32-bit integer variables may be declared, and are designated either *long long int* or *long long*.

Unsigned 64-bit integer variables may be declared wherever a 32-bit unsigned integer variable may be declared, and are designated either *unsigned long long int* or *unsigned long long*.

Function arguments declared as *long long* integers are passed in 64-bit format. The corresponding parameter declaration in the function must be declared as *long long int* or argument misalignment occurs.

In the portable C compiler, 64-bit integers have the same arithmetic properties as 32-bit integers. The 64-bit integers also participate in the data-type conversions routinely performed for arithmetic, logical, and assignment operations.

If either operand of an arithmetic or logical operation is a 64-bit integer, the other operand is converted to 64-bit format before the operation is performed. Signed values are converted to 64-bit format by sign extension; unsigned values are converted through the use of zero extension. Quantities assigned to a 64-bit integer quantity are converted to 64-bit format. Stack misalignment occurs when 64-bit quantities are passed as arguments to routines that have declared the corresponding parameter to be a 32-bit integer. Bit fields may not be larger than 32 bits in size. Also avoid using signed 64-bit integers to contain pointer data, since their portability is not assured.

3.3.2 64-Bit Integer Constants

The portable C compiler compiler supports the use of *long long* integer constants. The format of these constants is *nnnLL*, where *nnn* is a digit string. Constants may be specified with either the uppercase *LL* or the lowercase *ll*; the uppercase designation is easier to read and is preferred.

As with smaller integer constants, you may use both octal (leading 0) and hexadecimal (leading 0x) radices. The default radix for all numeric constants is decimal. Integer constants with binary equivalents longer than 32 bits are automatically treated as *long long* constants.

3.3.3 64-Bit Integer Descriptors

Format strings which include specifications of the form *%ll* can be used for the input or output of 64-bit integer variables. Format modifiers, such as *u*, *x*, *d*, and *o* used with 32-bit integer values, can also be used with 64-bit integer values. The number of significant digits maintained is 16 for hexadecimal, 22 for octal, and 20 for decimal values. Larger fields are padded with blank values.

In *printf* and *scanf*, just as *%ld*, *%lx*, or *%lo* specify *long* (32-bit) conversions, *%lld*, *%llx*, or *%llo*, specify *long long* integers.

NOTE

The library routines *printf* and *scanf* can be used to manipulate 64-bit values. These routines are documented in Section 3S, *CONVEX UNIX Programmer's Manual*.

Chapter 4
Data Types

This chapter describes the basic data types supported by the portable C compiler and specifies the requirements for the proper alignment of code and data in memory.

All the scalar data types defined by the CONVEX architecture are supported in the portable C compiler; vector data types are not directly supported.

The portable C compiler supports the following data types:

- **Integer data types:** *short int, int, long int, long long int.* To declare 16- , 32- , and 64-bit integer variables, use the *short int, long int,* and *long long int* keywords, respectively. The 32-bit *int* data type is the same as *long int,* and is the default integer data type. Twos-complement arithmetic is used for all signed arithmetic on CONVEX computers. Integer types may also be unsigned. Each data type is described in detail later in this chapter.

- **Character data type:** *char.* Character data may be considered to be a fourth integer data type. Character data is stored as bytes. The *char* representation contains a single ASCII character. Integer arithmetic operations can be performed on characters.

- **Floating-point data types:** *float, double.* Declare 32-bit single-precision floating-point data with the keyword *float,* and 64-bit double-precision floating-point data with the keyword *double.* Floating-point numbers may be represented in either native format or IEEE format.

- **Enumerated (scalar-type):** *enum.* Each *enum* datum holds the ordinal value of a member of a set of enumerated values.

- **Pointer (address).** Each pointer datum holds a virtual memory address.

- **Record structure:** *struct.* Record structure data is an aggregate of fields of data types. Structure data may be of any size.

- **Array (matrix).** Array data is also an aggregate, although each item of an array must be of the same data type. Array data is not documented in this guide since the CONVEX implementation of array data is identical to the specification found in *The C Programming Language.*

You may create user-defined data types out of combinations of the data types described above. Logical (Boolean) operations may be performed on signed or unsigned data of the following types: *char, short int, int, long int,* and *long long int.*

4.1 CONVEX Data Representations

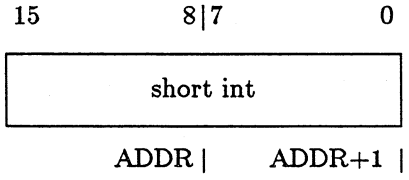
The following data representations are internal representations of the data types supported by the portable C compiler.

4.1.1 Short Integer Data

To declare 16-bit integer variables use the designations *short int* or *short*; *short int* variables may be either signed or unsigned. Unsigned 16-bit integers range in value from 0 to 65,535 (2^{16}-1). Signed 16-bit integers may range in value from -32,768 to +32,767 (-2^{15} to $+2^{15}$-1).

Figure 4-1 shows the internal representation of the *short int* data type. The least-significant bit in the data representation is numbered 0. The numbers shown at the top of the figure represent bit numbers. Byte offsets are shown at the bottom of the figure. A similar layout is used in all the figures in this chapter.

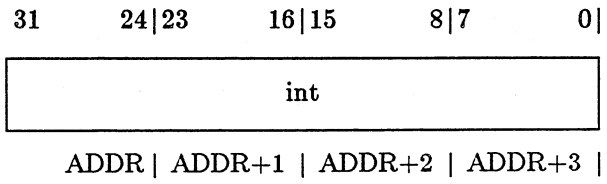
Figure 4–1: Short Integer Data Representation



4.1.2 Integer Data

In the portable C compiler, 32-bit integer variables are declared as *int* variables. *Int* variables may be declared as either signed or unsigned. Unsigned *int* variables range in value from 0 to +4,294,967,295 (0 to $+2^{32}$-1). Signed *int* variables range in value from -2,147,483,648 to +2,147,483,647 (-2^{31} to $+2^{31}$-1). Figure 4-2 depicts the representation of this data type.

Figure 4–2: Integer Data Representation

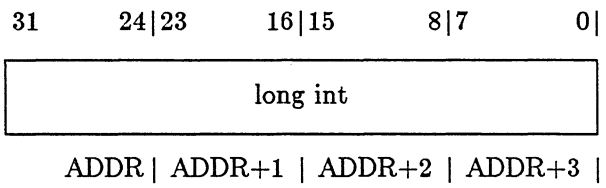


4.1.3 Long Integer Data

In the portable C compiler, both the *long int* and *int* data types have 32-bit representations. *Long* integers are defined as 32-bit integers to provide maximum compatibility with C compilers designed for 32-bit minicomputers.

Long integers may be declared with the designations *long int* or *long*. The long integer data representation is shown in Figure 4-3.

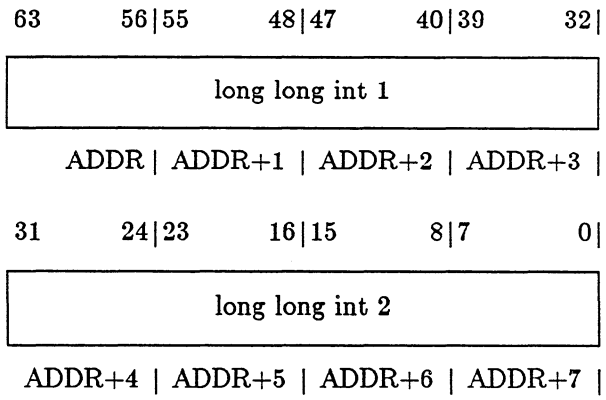
Figure 4–3: Long Integer Data Representation



4.1.4 Long Long Integer Data

In the portable C compiler, a 64-bit integer is declared with one of two designations: *long long int* or *long long*. You may declare *long long* variables to be either signed or unsigned. Unsigned 64-bit integers range in value from 0 to +18,446,744,073,709,551,615 (0 to $+2^{64}$-1). Signed *long long* variables range in value from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $+2^{63}$-1). Figure 4-4 shows the *long long* data representation.

Figure 4–4: Long Long Integer Data Representation



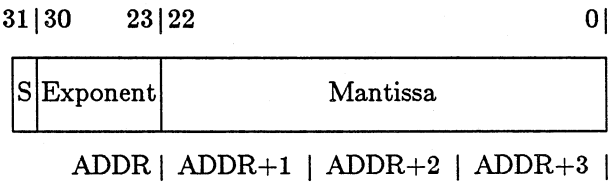
Although a *long long* integer may normally be used wherever a *long* integer may be used, you cannot pass *long long* integers as arguments to functions that do not expect to receive them. In general, *char* and *short* values can be passed to routines which expect *int* or *long* values. The compiler converts 8- and 16-bit quantities to a 32-bit format before pushing them onto the run-time stack. Since 64-bit values are not truncated, however, improper stack alignment occurs when *long long int* values are passed to routines that expect *int* arguments.

4.1.5 Single-Precision Floating-Point Data

Single-precision (32-bit) floating-point variables are declared with the *float* keyword and can be represented in either native format or in IEEE format. If you want to process your floating-point data in IEEE mode, your machine must be equipped with the IEEE support hardware.

Figure 4-5 shows the internal representation of single-precision floating-point data. The positioning of the sign, exponent, and mantissa apply to both native and IEEE formats; the particulars of each format are described following the figure.

Figure 4–5: Single-Precision Floating-Point Data Representation



4.1.5.1 Single-Precision Native

In single-precision native floating point, the range of numbers that can be represented is:

$2.9387359 \times 10^{-39}$ through $1.7014117 \times 10^{+38}$

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.

4.1.5.2 Single-Precision IEEE

In single-precision IEEE floating point, the range of numbers that can be represented is:

$1.1754944 \times 10^{-38}$ through $3.4028235 \times 10^{+38}$

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.

NOTE

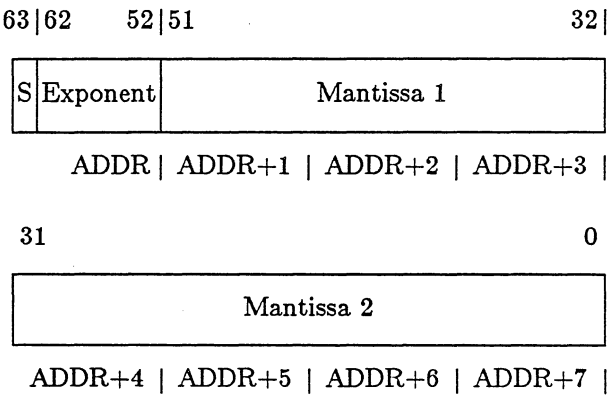
The CONVEX hardware and runtimes only support the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

4.1.6 Double-Precision Floating-Point Data

Double-precision (64-bit) floating-point variables are declared with the *long float* or with the *double* keyword and can be represented in either native format or in IEEE format. If you want to process your floating-point data in IEEE mode, your machine must be equipped with the IEEE support hardware.

Figure 4-6 shows the internal representation of double-precision floating-point data. The positioning of the sign, exponent, and mantissa apply to both native and IEEE formats; the particulars of each format are described following the figure.

Figure 4–6: Double-Precision Floating-Point Data Representation



4.1.6.1 Double-Precision Native

In double-precision native floating point, the range of numbers that can be represented is:

$5.562684646268003 \times 10^{-309}$ through $8.988465674311584 \times 10^{+307}$

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.

4.1.6.2 Double-Precision IEEE

In double-precision IEEE floating point, the range of numbers that can be represented is:

$2.225073858507201 \times 10^{-308}$ through $1.797693134862317 \times 10^{+308}$

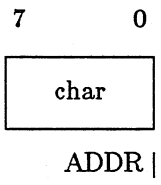
In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and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

4.1.7 Character Data

Character (*char*) data is stored in 8-bit bytes. Each byte can contain one of the ASCII character codes. In the portable C compiler, *char* data items may be treated either as 8-bit integers or as ASCII characters.

You may declare *char* variables as either signed or unsigned. Unsigned *char* variables may range in value from 0 to +255 (0 to $+2^8$-1). Signed *char* variables may range in value from -128 to +127 (-2^7 to $+2^7$-1). Figure 4-7 shows the representation of this data type.

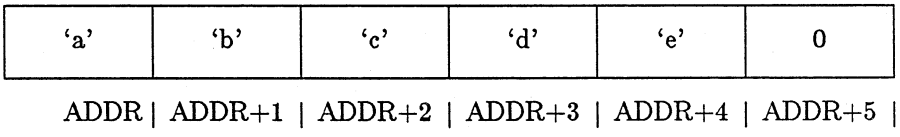
Figure 4–7: Character Data Representation



Single-character constants in C are surrounded by apostrophes. ASCII codes are specified as *char* variables when you place the one- to three-digit octal number representing the desired character code between apostrophes with a preceding backslash (\ ) character.

Arrays of character data are stored in ascending memory addresses, regardless of 32-bit word boundaries. By convention, C character strings are terminated with a null (0) byte. Thus, the character string "abcde" appears in memory with the configuration shown in Figure 4-8.

Figure 4–8: Character String Representation



4.1.8 Enumerated Data

In the portable C compiler, an enumerated data type is a user-defined data type that has a finite number of possible values. 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 take on 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 above example) is stored with the ordinal value of zero. Subsequent enumerated values are represented by sequential integer values. In the example shown above, blue = 1, and green = 2. The default ordinal values are overridden when they are followed by an equal sign and a new ordinal (e.g., *enum color {red=10, blue=20, green=30}*).

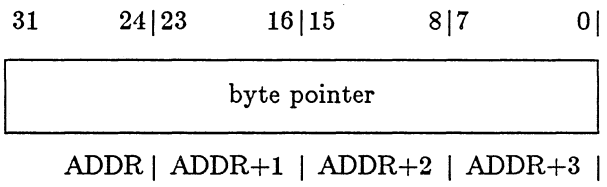
4.1.9 Pointer Data

A pointer is a variable that contains the 32-bit address of another variable. For example, the declaration:

```
char *cp
```

designates a pointer named *cp*, which may be assigned the address of a *char* variable. All pointers defined in the portable C compiler refer to the location of a byte in memory. Pointers have the same range of possible values as unsigned integers. All possible unsigned integer values may not be used as valid pointers, however. A pointer may contain the address of a memory location that has been specially protected by UNIX. 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 the pointer refers. In the portable C compiler, it is an error for a program to access the contents of a null (0) pointer. Figure 4-9 is an example of pointer data representation.

Figure 4–9: Pointer Data Representation



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, since data with aligned addresses can be encached in high-speed memory by the CONVEX hardware. The compiler attempts to keep addresses properly aligned.

4.1.10 Structure Data

Structures are collections of data items that are related in some way and may contain bit fields less than 32 bits long. The alignment of fields within a structure depends on the data types of the fields. Each field in the structure is aligned on a boundary appropriate for its data type. The alignment boundaries for fields within structures are the same as the alignment values for variables on the runtime stack.

4.2 Storage Alignment Requirements

This section describes the requirements for alignment of instructions and data in memory. In general, if you use only high-level languages, you need not be concerned about the alignment of either code or data. This information is useful, however, if you are interested in avoiding performance penalties that result from misaligned data in assembly-language code.

4.2.1 Data Alignment

Data is properly aligned when it resides in memory at an address that is a multiple of the size of the datum in bytes. The alignment is as follows:

- 8-bit data items are always aligned, regardless of their addresses.
- 16-bit data items are aligned when their addresses are multiples of 2 bytes; that is, when they are aligned on even address boundaries.
- Both 32-bit and 64-bit data items are aligned when their addresses are multiples of 4 bytes.

In most circumstances, the portable C compiler and the UNIX runtime system can keep program variables aligned to take maximum advantage of the hardware.

Data on runtime stacks should be aligned using these same guidelines. To maximize performance for push and pop operations, align the top of the runtime stack on a 32-bit boundary. Data stored within C structures should also be properly aligned. The alignment of data within structures may cause "holes" in the structures.

4.2.2 Code Alignment

Code is properly aligned when

- The starting addresses of the instructions lie on even address boundaries, and
- The least-significant bit of the program counter is zero.

Since each instruction in the CONVEX instruction set is a multiple of 16 bits, the program counter becomes misaligned only after it has been altered. When control is transferred through the use of a branch instruction, the least-significant bit in the program counter is usually ignored. Executing instructions that were loaded on odd address boundaries produces unpredictable results. To avoid this occurrence in assembly-language code, use the assembler directive *.align* to force the alignment of instructions if odd-length data is stored in the text segment of the program.

4.2.3 Bit Field Alignment

Bit fields within structures in C programs are allocated from the most-significant bit in a 32-bit word (bit 31) toward the least-significant bit in the word (bit 0). The most-significant bit in a word is the leftmost bit; the least-significant one is the rightmost bit. Bit fields should be no longer than 32 bits. Bit fields spanning a 32-bit boundary are realigned to start at the next available 32-bit boundary. This realignment results in a hole between the bit fields. A hole is always in the least-significant bit position.

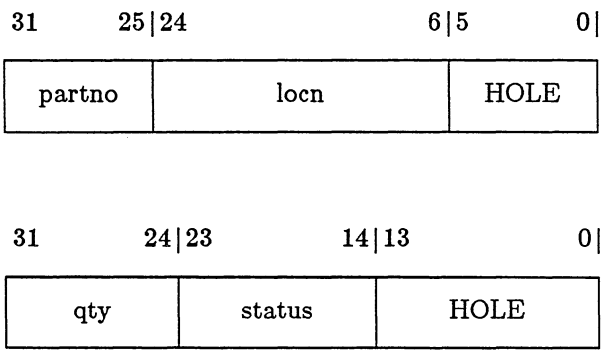
For example, consider this bit field structure:

```
struct {
        int     partno:7;
        int     locn:19;
        int     qty:8;
        int     status:10;
} bitz;
```

Two 32-bit words are needed to contain an instance of the structure *bitz*. If the field named *qty* appears in memory immediately after *locn*, it spans a 32-bit boundary. A 6-bit hole is inserted by the compiler to align the *qty* bit field on a 32-bit boundary. Fourteen bits remain unallocated in the second word of the structure and form a second hole.

Figure 4-10 shows how the compiler allocates bit fields beginning with the most-significant bit in a word and extending to the least-significant bit. The compiler assigns fields to words beginning with the first bit field that appears in the structure declaration.

Figure 4–10: Bit Field Alignment Example



Chapter 5
Calling Conventions

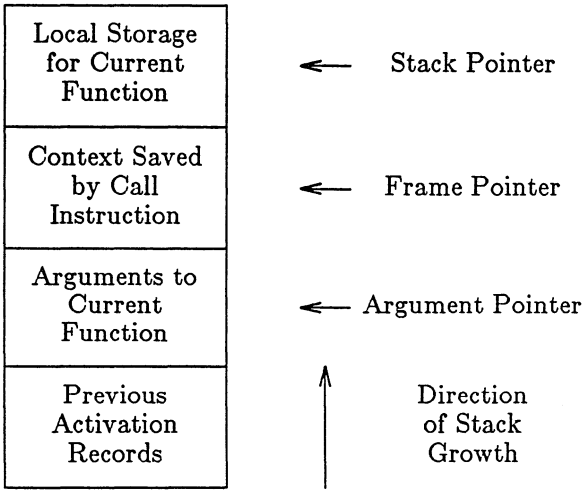
This chapter describes the general layout of the activation records used with C functions. When C function calls are executed, the current state of several hardware registers used by the calling routine must be preserved. The contents of these registers are pushed onto the runtime stack as part of an activation record. The called function may then alter the machine registers as it runs. The hardware, as part of the return to the original calling routine, restores the old values of the saved registers.

The portable C compiler does not preserve the value of every register across a function call. Only registers required to maintain the state of the runtime stack are preserved. Called routines that can restore the frame pointer register to its original state are allowed to modify any register passed to them.

5.1 Function Stack Layout

Figure 5-1 shows the top of the runtime stack. Although the stack grows downward in the address space, this diagram shows the stack as growing upward on the page. The stack pointer register contains the address of the topmost location on the stack. The frame pointer register contains the address of the last frame pushed onto the stack by a *call* or *calls* instruction. The argument pointer register contains the address of the arguments to the current routine.

Figure 5–1: Top of the Runtime Stack



5.2 Function Calling Sequence

The general steps that the compiler generated code performs for a function call are:

1. Push the values of the arguments to the function onto the runtime stack in reverse order.

2. Update the argument pointer 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 should contain a count of the number of arguments passed.

4. Call the routine with a *calls* instruction.

Executing *calls* instruction places a stack frame on the runtime stack. The stack frame contains the current values of the program counter (return address), the current value of the program status word, the old value of the frame pointer, and the current value of the argument pointer.

The conventions that apply to function calls are as follows:

- The called routine can allocate storage for local variables on top of the runtime stack. No stack references in portable C compiler code are made relative to the top of the stack. Storage allocated on the stack by called routines is automatically deallocated when the function returns.

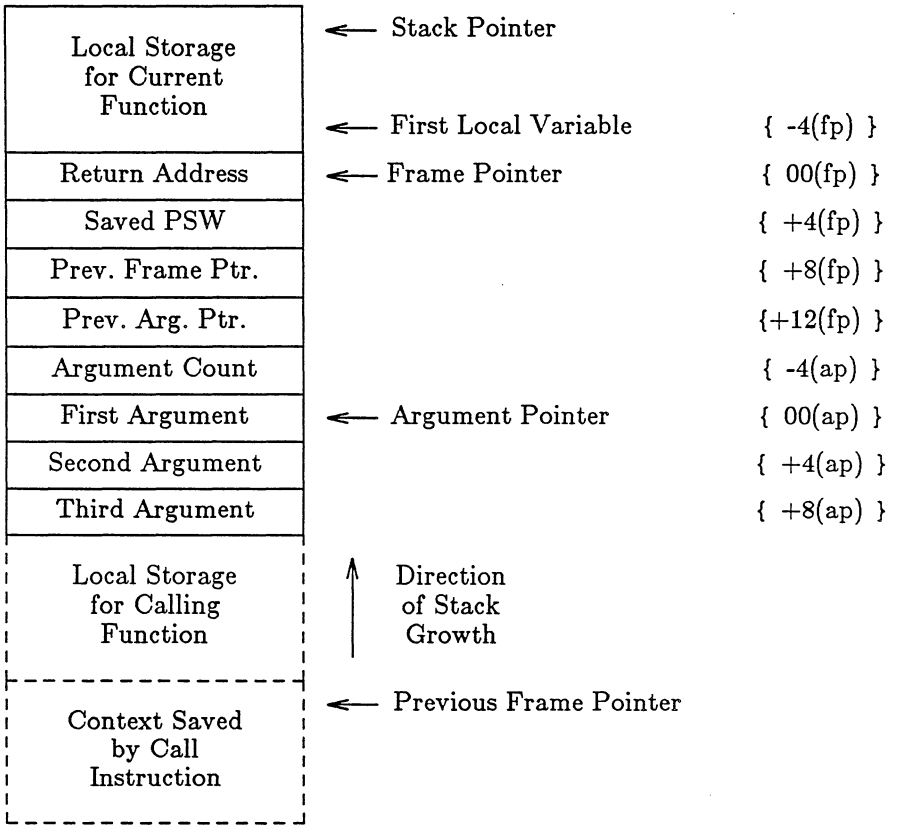
- The called routine need not preserve the contents of any register except the frame pointer. The called routine uses the current value of the argument pointer register to access the arguments passed to the routine by its parent.

- The frame pointer register points to the context block pushed by the caller when calling the child routine. The called child routine references the local storage it has allocated on the runtime stack by referencing positive offsets from the frame pointer.

- The called routine references arguments passed to it by its parent by referencing positive offsets from the argument pointer register. The word with an address of -4 relative to the argument pointer contains a count of the number of arguments passed to the routine.

Figure 5-2 shows the layout of the stack as seen by a routine 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 5–2: Stack Layout



Called routines return to their parents by:

1. Placing a return value in register s0, then
2. Executing the *rtn* instruction.

When the *rtn* instruction is executed, automatic storage allocated by the called routine is automatically deallocated. This instruction also restores the program 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 routine. After control returns to the parent, the stack pointer register points to the location that contains the pushed argument count. The parent routine adds a fixed 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 prior to 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 offset 12 bytes from the frame pointer.

5.3 Code Generated for Function Calls

The following example shows a section of CONVEX assembly-language code used for a function call:

```
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 following code is used in the called child function:

```
        .globl  _child          ; called function
_child:
        sub.w   #localsize,sp   ; allocate bytes for local variables
        ld.w    (0)ap,s0        ; load the value of the first arg
                                ; 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
```

5.4 Function Names

Function names and global variables produced by the compiler in object code are unrestricted in length. An underscore character is prefixed to each global variable. Be sure to include this character when using the assembly-language debugger or when writing assembly-language functions that are called by C routines.

5.5 Function Arguments and Return Values

C arguments are passed by the "call-by-value" method (that is, the values of arguments, rather than their addresses, are passed). C programs may pass addresses as arguments to functions by using pointer variables. Structures in C are also passed by value. All elements of the structure are pushed onto the runtime stack before the call. The calling process accelerates when structure pointers, rather than the structures themselves, are passed. Results returned from functions are returned in scalar register S0. Functions that return structures as results place the function result in a static area in the data segment and return a pointer to that area in S0.

A

Compiler Error Messages

This appendix lists the error messages that the portable C compiler may produce. These messages fall into the following categories:

- **User Error Messages**: errors in the program detected by the compiler. If the compiler generates any error messages, the compiled output file may not be correct.
- **User Warning Messages**: messages about possible errors in the program detected by the compiler. These messages are informative in nature and may or may not indicate an actual error in the program. The compiler still produces a compiled output file when warning messages are produced. You may suppress these messages with the *-w* option on the command line.
- **Compiler Error Messages**: messages that indicate faults detected in the compiler itself. These messages may indicate a bug in the compiler or an excessive number of user error messages.

The remainder of this Appendix lists messages in each of the first two categories only.

User Error Messages

***name* undefined**
The identifier *name* is not declared prior to the current statement.

=<*c* illegal
Some character other than "<" followed the characters "=<".

>*c* illegal
Some character other than ">" followed the characters "=>".

Null dimension
Some array dimension is explicitly zero.

array of functions is illegal
An array is declared with functions (instead of function pointers) as elements.

assignment of different structures
Two different shaped structures are being assigned.

bad asm construction
The argument to the *asm* function is not a character string.

bad scalar initialization
An element of a data initialization list should be a scalar quantity but is not.

cannot initialize extern or union
An attempt is being made to initialize a variable declared as *extern* or *union*.

case not in switch
A *case* statement is outside the scope of any *switch* statement.

constant expected
A variable expression appears in a context where only a constant is legal.

declared argument *name* is missing
An argument declaration exists for a name that does not appear in the argument list of a function.

default not inside switch
A *default* case label is outside the scope of any *switch* statement.

division by 0
An expression contains a divisor that evaluates to zero.

duplicate case in switch, *number*
Two or more cases numbered *number* appear in a switch statement.

duplicate default in switch
Two or more *default* cases appear in a switch statement.

empty character constant
A character constant enclosed in apostrophes contains no characters.

expression depth exceeded
An expression in the statement is too complex for the compiler to handle.

-f: invalid floating point format
An invalid floating point option was specified.

-f option obsolete; you must use -fd instead
Message self-explanatory.

field outside of structure
A bit field appeared in a declaration that is not part of a structure declaration.

field too big
A bit field in a structure is larger than 32 bits.

function declaration in bad context
A function declaration appears inside another function or declaration.

function has illegal storage class
A function is declared with an illegal storage class.

function illegal in structure or union
A structure or union element may not be a function.

function returns illegal type
A function returns an illegal data type.

illegal break
A *break* statement appears outside any looping construct.

illegal character: *number* (octal)
The character constant is not a valid 8-bit octal number.

illegal class
A storage class is specified for a variable declaration that may contain no class.

illegal comparison of enums
Two incompatible *enum* expressions are being compared.

illegal continue
A *continue* statment appears outside any looping construct.

illegal field size
A bit field is specified to contain more bits that the declared data type contains.

illegal field type
A bit field is specified of an invalid type.

illegal function
The expression for a function name does not resolve to an identifier that is a function name.

illegal hex constant
A hexadecimal constant contains invalid characters.

illegal indirection
An indirection operator appears with an expression that does not resolve to a pointer type.

illegal initialization
A data initialization clause appears for an invalid declaration such as a bit field.

illegal lhs of assignment operator
The left-hand side of an assignment operator does not resolve to an address.

illegal member use: *name*
An identifier is used as a structure field name but is not declared as such.

illegal pointer subtraction
Pointers to different size objects cannot be subtracted.

illegal register declaration
A *register* class is specified for a type that cannot be loaded into a register.

illegal type combination
A declaration contains a set of type keywords that do not form a legal data type.

illegal types in :
The two operands of the colon operator must be convertible to the same type.

illegal use of field
A structure field has been given an invalid storage class.

illegal {
A left brace appears in a data initialization list at a point that does not agree with the shape of the item to be initialized.

member of structure or union required
The name that follows a period or "->" operator is not the name of a field in a *struct* or *union.*

newline in string or char constant
An ASCII newline character cannot appear in a string or character constant.

no automatic aggregate initialization
An array or structure on the stack (*auto*) cannot be data initialized.

non-constant case expression
The expression in a *case* statement does not resolve to a constant value.

nonunique name demands struct/union or struct/union pointer
A nonunique field name appears.

operands of *operation* **have incompatible types**
The operands of an operator must be convertible to a common type.

pointer required
Some context in the current statement requires an object of type pointer.

redeclaration of *name*
A variable has been redeclared in a manner that is incompatible with a previous declaration.

redeclaration of formal parameter, *name*
A formal parameter has been declared more than once.

redeclaration of: *name*
A field name has been declared more than once.

structure reference must be addressable
The expression to the left of a period does not resolve to the address of a structure.

this machine does not have hardware support for IEEE
The -fi option was specified but the machine does not contain the IEEE support hardware.

too many characters in character constant
Excess characters appear between apostrophes.

too many initializers
To many data initialization items appear for the number of elements to be initialized.

type clash in conditional
The operands of a conditional operator cannot be converted to a common type.

unacceptable operand of &
The operand of the address operator resolves to a value that has no address, like a constant.

undefined structure or union
The name to the left of a period has not been previously defined.

unexpected EOF
The compiler reached the end of the source file without reaching the end of a program unit or declaration.

unknown size
The size of the argument to a *sizeof* function cannot be determined by the compiler.

void function *name* **cannot return value**
A function declared as *void* returns a value.

void type for *name*
A non-function variable is declared with type *void.*

void type illegal in expression
An identifier of type *void* appears within an expression.

zero size field
A bit field within a structure has a width of zero.

zero sized structure
A structure is defined that contains no fields.

} expected
The number of data initializers for an aggregate is greater than the number of elements to be initialized.

User Warning Messages

name **redefinition hides earlier one**
The identifier *name* has been redefined in such a way as to prevent access to the an earlier definition.

& before array or function: ignored
An address operator appears before an expression that already yields an address.

a function is declared as an argument
The declaration of a function argument is itself a function.

ambiguous assignment: assignment op taken
An old-style assignment (like =+) has been seen.

bad arg temp
An address expression in the current statement contains the address of a function argument.

constant argument to NOT
A constant expression follows an exclamation point.

constant in conditional context
The condition expression of a conditional statement resolves to a constant.

empty array declaration
An implicitly-sized array contains an initialization list with no initializers.

enumeration type clash, operator *op*
The operands of operator *op* are not both of enumerated type.

illegal array size combination
An operation in the current statement involves pointers to different size arrays.

illegal combination of pointer and integer, op *op*
If one operand of operator *op* is a pointer, the other must be either a pointer or zero.

illegal member use: *name*
Field name *name* appears together with a structure name of which it is not a member.

illegal member use: perhaps *union.member*?
Union member name *member* appears together with a union name of which it is not a member.

illegal pointer combination
A combination of pointer types results in illegal type.

illegal structure pointer combination
A combination of pointer types results in structure of a different size.

illegal zero sized structure member: *name*
The declaration of structure *name* results in a zero-sized structure.

loop not entered at top
A looping construct, like *for*, is entered via a *goto*.

non-null byte ignored in string initializer
More bytes appear in a string initializer than are declared for the array of characters.

old-fashioned assignment operator
An obsolete "=*op*" assignment operator appears in the current statement.

old-fashioned initialization: use =
An obsolete initialization expression appears without an equal size.

sizeof returns 0
The *sizeof* function used for a zero-sized value.

statement not reached
There is no control path that executes the current statement.

struct/union or struct/union pointer required
The expression appearing before a period or "->" operator is not of an appropriate type.

structure typed union member must be named
An element of a union is a structure without a tag name.

undeclared initializer name *name*
One of the expressions in a list of initializers is an unknown identifier.

zero or negative subscript
A subscript expression resolves to a non-positive constant.

B

Reporting Problems

This appendix introduces the CONVEX Technical Assistance Center (TAC) and the *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.

B.1 Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address the 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.

B.2 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 them.

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.

B.3 Prerequisites

To use *contact* requires

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

B.3.1 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 system to another. The *uucp* (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

B.3.2 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*, only faster.

B.3.3 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.

B.4 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

B.4.1 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.

B.4.2 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.

B.4.3 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.

B.4.4 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, press **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*).

B.4.5 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.

B.4.6 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

B.5 Using the *contact* Utility

The *contact* utility prompts for the following information:

- your name, title, phone number, and corporate name
- the name and version of the product
- a one-line summary of the problem
- a detailed description of the problem
- the priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation supporting 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 the 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 the documentation supporting the product. Indicate if the documentation could be revised to address the question.

Step 9 The *contact* utility asks for the 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. Instead, *contact* treats a tilde in this section to mean your home directory. This convention is based on use of the tilde for expanding file names in *csh*.

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

To specify a directory, combine the 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 environment.

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 environment.

Index

V

W

CONVEX Portable C Compiler User's Guide
Document No. 720-000530-202, Fourth 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

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)