CONVEX Vector C Compiler
User's Guide

Document No. 720-000630-200

Version 2.0
January 12, 1988

Sharon Lammers

CONVEX Computer Corporation

Revision/Update Information for CONVEX Vector C Compiler User's Guide

Ver. No.	Date	Description
1.0	August 1986	First release.
2.0	February 1988	Document updated for software version 2.0, reformatted to 8-1/2 x 11, and converted to numbered headings. New command line options added and obsolete options deleted. Description of IEEE formats and new optimization features added. Complete editorial revision performed, including rewrite of Chapter 1. Technical and typographical errors in manual corrected.

Table of Contents

Appendices

List of Tables

List of Figures

Preface

Introduction

This manual describes the CONVEX Vector C compiler and the C programming language as implemented for the family of supercomputers produced by CONVEX Computer Corporation.

Although no official ANSI, NBS, or ISO standards for C exist at this time, the generally accepted definition of the language appears in Appendix A of B.W. Kernighan and D.M. Ritchie's *The C Programming Language* (Prentice-Hall, 1978). This book is considered to be the standard for C on most systems for which UNIX is available.

Intended Audience

It is assumed that the reader is familiar with the C language. Programmers familiar with other programming languages, but not with C, should read Appendix A of *The C Programming Language*.

Organization

- Chapter 1 is an overview of CONVEX C. It covers the procedures for compiling, linking, and executing a program.
- Chapter 2 describes the CONVEX extensions to the C language.
- Chapter 3 describes machine-independent and machine-dependent optimizations. Both vectorization and optimization techniques are described. Sections on using argument lists and vector intrinsic functions conclude the chapter.
- Chapter 4 describes the CONVEX C call conventions that enable you to call routines written in languages other than C. It details the CONVEX calling standards and shows machine code examples of calling conventions.
- Chapter 5 describes the services available in the C runtime library.
- Chapter 6 describes how to debug C programs using the *adb*, *csd*, and *pmd* debugging tools.
- Appendix A describes the C data representations supported by CONVEX C and shows how they are stored in memory.
- Appendix B describes the various sources of diagnostic messages: compiler, loader, and runtime library.
- Appendix C lists and describes the *vcpp* preprocessor statements.
- Appendix D lists the runtime libraries with pathnames.
- Appendix E describes the compiler directives.

Notational Conventions

The following conventions have been used in this document:

1. Mnemonics enclosed in "less than" and "greater than" signs designate ASCII nonprintable characters. For example, <CR> stands for carriage return.

2. Brackets ([ ]) designate optional entries.

3. Horizontal ellipsis (. . .) shows repetition of the preceding item(s).

4. Vertical ellipsis shows continuation of a sequence where not all the statements in an example are shown.

5. 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.

6. *Italics* within text indicate commands, filenames, or programs.

7. 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.

Associated Documents

CONVEX provides the following related documents:

- *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 C Compiler User's Guide* - Describes how to use *cc*, the scalar CONVEX C compiler.
- *CONVEX Consultant User's Guide* - Documents an optional program package that includes the CONVEX *csd* debugger.
- *CONVEX Guide to Software Development* - Describes some of the available tools and procedures for software development.
- *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.

Reply Form

Information on how to request additional documentation and a reply form for questions and comments are available at the back of this manual.

1

Overview

1.1 Introduction

C is a general-purpose programming language that provides many of the features of other modern high-level languages. Subroutines written in C can be linked with both FORTRAN and assembly language routines to run as a package. The data types and control structures of the C language are efficiently supported by the architecture of the CONVEX computers.

The C language is closely associated with the UNIX operating system. In fact, the kernel of the CONVEX UNIX operating system and its associated utilities are written in C.

The CONVEX Vector C Compiler (*vc*) provides a variety of features to improve programming efficiency and to enhance the performance of programs written in C. These features include:

- Source code optimizations to eliminate unnecessary computations during program execution.
- Vectorization to improve performance by eliminating loop overhead.
- Machine-dependent optimization to enhance the object code produced by the compiler.

The Vector C compiler generates reentrant object modules that can be shared and can include symbol tables used by the CONVEX symbolic debugging tools. The runtime libraries support calls for all UNIX system services.

The data types in Vector C exploit the internal data representations found on CONVEX computers. All the vector and scalar data types defined by the CONVEX architecture are supported in Vector C. In addition:

- C supports structured aggregate data (called *struct*) in multidimensional arrays.
- Structured variables may include bit-field definitions.
- Like FORTRAN, 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 supports an enumerated scalar data type, *enum*, that permits you to assign the ordinal value of a set of scalar values to a variable.
- The values returned by C functions may be of any data type, except arrays, including *struct* and *union*.

The C language passes arguments to functions using the "call-by-value" approach; that is, it sends functions the value of arguments, rather than the addresses of the arguments. You can also pass variable addresses to achieve the effects of the FORTRAN "call-by-reference" operation.

1.2 Compiling C Programs

To invoke the Vector C compiler, enter the following command line:

vc [*options*] *files* [*loader-options*]

In the command line, *options* is one or more of the options specified in Table 1-1. The parameter *files* can specify C source files, assembly language source files, object modules, or libraries. The *loader-options* are certain of the options recognized by the loader (*ld*).

Files specified on the compiler command line have standard suffixes as shown in the following table:

Filenames for...	End With the Extension...
C source files	.c
Compiled object module files	.o
Symbolic assembly-language files	.s
Libraries	.a

The following loader options can be specified on the command line: A, D, E, l, M, m, o, r, s, T, t, u, X, x, and y. These options are explained in the *CONVEX Loader User's Guide.*

Examples:

The following command compiles the source file *span.c* performing local and global scalar optimization and vectorization. The executable file is then loaded into *a.out.*

```
vc -O2 span.c
```

The next example compiles the C source file *mflops.c,* assembles the assembly language file *timer.s,* and links them together, generating the symbol table information necessary to run the *csd* debugger. No optimization is performed and the executable file is written to *a.out.*

```
vc -no -db mflops.c timer.s
```

The following command line compiles the C program *therm.c,* assembles the file *assem.s,* and links the resulting object files with *myobj.o* and functions from library *mylib.a* to form the executable object file *a.out.*

```
vc therm.c assem.s myobj.o mylib.a
```

1.3 Linking C Programs

When compilation is complete, the compiler automatically calls the loader (*ld*) and passes to it any loader options that were specified on the compiler command line.

The preferred method of invoking the loader for C programs is to use the *vc* command. This approach ensures that the proper C libraries are loaded in the proper order. Any libraries specified on the *vc* line with the *-l* loader option take precedence over the standard libraries. Appendix D identifies the pathnames and contents of the C runtime libraries.

You can also invoke the loader using the *ld* command. For information on alternative methods of invoking the loader, see the *CONVEX Loader User's Guide.*

Examples:

In the following example, the loader is automatically invoked by the compile command *vc*. The resulting executable file is redirected to the file *myprog*.

```
vc parsec.c -o myprog
```

In the following example, the loader is invoked with the *ld* command to combine three modules, which have already been compiled, with the standard startup routine and libraries. The output is written to *a.out*.

```
ld /lib/crt0.o moda.o modb.o modc.o -lvc-Bi -lc
```

1.4 Executing C Programs

To execute a program after it has been generated by the loader, simply type the name of the executable file. The default name of the executable file is *a.out*. If the *-o name* option was specified on the compiler or loader command line, the name of the executable file is *name*.

Example:

The following line executes *a.out*, reading input from the file *input* and writing output on the file *output*.

```
a.out < input > output
```

Table 1-1: Command Line Options

Option	Description
-B*string*	Finds substitute compiler (*cocc* and *vcpp*) in the directory named *string*. If *string* is empty, use a standard backup version in the directory */usr/convex/oldvc*. For example, the command *-B/usr/new* invokes */usr/new/cocc* and */usr/new/vcpp* instead of the default */usr/convex/cocc* and */usr/convex/vcpp*.
-C	Tells the macro preprocessor not to delete comments.
-c	Suppresses the loading phase of the compilation. The compiled object module that is generated from the files *file.c* or *file.s* is written to *file.o*.
-D*name*[=*def*]	Defines a name to the preprocessor as if the *#define* preprocessor statement had been used. If no definition is given, the name is defined as 1.
-db	Produces additional information for use by the symbolic debugger, *csd*, and the *pmd* utility. Also passes the *-lg* option to the loader. This option can be used with all levels of optimization. If the *-O* option is used to specify optimization at some level, there may be source statements for which no debugging information is generated for *csd*.
-E	Runs only the C preprocessor on the named C programs and sends the result to standard output.
-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.
-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.

Table 1-1: Command Line Options (continued)

Option	Description
-fx	Specifies either that the routine contains no real constants or that the routine is dual mode. If you use this option and the routine does contain real constants, they are translated into native format and processed in native mode. For further information on dual-mode programming, please obtain the dual-mode application note from the CONVEX Technical Assistance Center (TAC).
–I*dir*	Names an alternate directory to search for *#include* files. Several *-I* options can be used to create a search path for *#include* files. The directories are searched in the order specified on the command line. The current working directory is searched before any directories specified by the *-I* option.
–na	Suppresses all advisory diagnostic messages.
–no	No optimization is performed.
–nv	Suppresses all vectorization summary messages.
–nw	Suppresses all warning diagnostic messages.
–O*n*	Performs optimization at the level specified by *n*. The levels are 0 (local scalar optimization), 1 (local and global scalar optimization), and 2 (local and global scalar optimization plus vectorization). If this option is not specified, the compiler optimizes at the -O0 level.
–o *name*	Specifies that the executable program produced by the loader (*ld*) is to be called *name*. The default name is *a.out*.
–p	Causes the compiler to produce 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 out a *mon.out* file at normal end of execution of the object program. Also, a profiled library is searched instead of the standard C library. An execution profile can then be generated by use of *prof*(1) (optional product).
–pb	Causes the compiler to produce statement-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) (optional product).

Table 1-1: Command Line Options (continued)

Option	Description
-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. An execution profile can then be generated by use of *gprof*(1) (optional product).
-S	Generates symbolic assembly code for each program unit in a source file. Assembler output for the source file *x.c* is put on file *x.s*; the assembly file is not assembled.
-sc	Instructs the compiler to check the program for compilation errors. No optimization or code generation is performed with this option.
-ss *n*	Reduces the size of the windows that the compiler uses in optimizing programs to *n* executable statements. This option is useful for compiling large programs that might otherwise compile very slowly.
-tl *time*	Sets the maximum CPU time limit for compilation to *time* minutes. If the CPU time exceeds the specified time, compilation is aborted.
-U*name*	Removes any initial *vcpp* definition of *name*. The only built-in names defined by *vc* are "__LINE__", "__FILE__", and "convexvc".
-uo	Performs potentially unsafe optimizations, i.e., moves the evaluation of common subexpressions and/or invariant code from within conditionally executed code.
-va	Tells the compiler that formal array parameters of basic types are to be treated as arrays rather than as pointers and implies that actual array arguments do not overlap each other or any external variable that is modified in the function. If these rules are followed, the need for *no_recurrence* directives in functions is reduced.
-vn	Identifies compiler version. Outputs the version number of *vc*, *vcpp*, and *cocc* (the compiler). The version numbers of *vc*, *vcpp*, and *cocc* go to *stderr*.

1.5 Diagnostic Messages

This section describes the diagnostic messages that you can receive from the compiler or during runtime.

1.5.1 Compiler Diagnostic Messages

The compiler generates four kinds of user diagnostic messages: user errors, warnings, advisories, and vectorization summaries. The compiler diagnostic messages are directed to standard error. Appendix B lists the compiler diagnostic messages. Each messages consists of:

- The compiler name (vc).
- The line number on which the error occurred.
- The character position within the line.
- The pathname of the source file containing the line in error.
- A brief description of the error.

Examples:

```
vc: Error on line 7.22 of test.c: label referenced but not defined.

vc: Warning on line 3.6 of zodiac.c: divide by zero is possible at runtime.

vc: Loop on line 1.7 of matrix.c (1 loop) fully vectorized.
```

Use a text editor to locate the line containing the error using the line number from the diagnostic message. Alternatively, you can generate a listing with line numbers using the UNIX command *cat*.

Example:

```
cat -n statfile
```

You may also use the *error* utility to insert diagnostic messages into your source file as comments. This method is a convenient way to find errors while you are editing a source file.

The compiler places output files in the current directory. It directs a compilation summary to standard output (*stdout*), and the vectorization summary and diagnostic messages to standard error (*stderr*). You can redirect these messages to a file using standard UNIX redirection commands. For further information, see *csh*(1) and *sh*(1).

Example:

```
vc stat.c |& error
```

This command compiles *stat.c* and sends the standard output and standard error output to the *error* utility, which then inserts the diagnostic messages into the source file *stat.c*.

If the compiler has an internal error, it generates a message in the following format:

```
vc: >>>>>       COMPILER ERROR            <<<<<
>>>>> See your system manager for help <<<<<
```

The preceding lines are followed by a line that describes the nature of the error. Report such errors to your system manager or to the CONVEX Technical Assistance Center (TAC).

1.5.2 Runtime Error Messages

Runtime error messages are directed to standard error. A math routine error message has the form:

```
routine_name: [error_number] description
```

Example:

```
mth$r_sqrt: [300] square root undefined for negative values
```

1.6 Program Interfaces

The following sections describe the interfaces between *vc* and the other programs it invokes to produce an executable program. During the processing, error messages, warning messages, and vector summaries are written to *stderr*. The individual steps by which a source file is converted to an executable file are as follows:

1. The Vector C preprocessor (*vcpp*) processes the source files and *include* files. The resulting output file is passed to the compiler.

2. The Vector C compiler (*cocc*) receives the output from the preprocessor and generates object code. Assembly language code is not produced unless you have specified the *-S* option on the *vc* command line.

3. The loader (*ld*) produces an executable file from the compiled object code along with any libraries that are required.

1.6.1 Preprocessor

The C compiler software package contains a preprocessor (*vcpp*) that is invoked automatically by the compiler. The preprocessor processes symbolic constants and performs the expansion of macros. The preprocessor also makes possible the inline substitution of constant expressions referenced in C source files, and text-forming macro bodies. Macro arguments are substituted one-for-one in the position specified in the macro definition.

The preprocessor supports positional argument substitution and includes a conditional compilation facility. See Kernighan and Ritchie's *The C Programming Language* or *vcpp*(1) for a detailed explanation of this facility.

The preprocessor statements begin with the # symbol in column 1 and are syntactically independent of the rest of C. The preprocessor statements are described in Appendix C.

1.6.2 Runtime Support System

The runtime system provides both scalar and vector versions of the math functions. The compiler determines which version of the routine to call. The standard calling sequences for the runtime system are described in Chapter 4.

1.6.3 Debuggers

There are three debugging tools that can be used with C programs. These tools are as follows:

- The source-level (*csd*) debugger provides statement-level execution control and access to program variables through symbolic names. In order to use *csd*, you must compile your program with the *-db* option on the *vc* command line.
- The object-level debugger (*adb*) requires no special support (such as recompilation of programs) from the compiler. Unlike *csd*, you need not specify any options when compiling programs for use with *adb*.
- The optional post-mortem dump (*pmd*) utility displays information about failed programs that can be used for debugging. This information includes the signal that caused the program failure, an approximate source line at which the failure occurred, the contents of machine registers and global variables, and a summary of the resources used by the program.

1.6.4 *lint* Utility

The *lint* utility examines C programs for potential portability problems and detects such errors as uninitialized variables or mismatched argument types between separately-compiled programs.

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. The *lint* utility accepts multiple input files and library specifications and checks them for consistency. It is suggested that you run *lint* after any program changes. For more information, see *lint*(1).

2

CONVEX Extensions to C

2.1 Introduction

The CONVEX Vector C compiler (*vc*) supports the use of four data types not specified in the portable C compiler. Three of these data types (*structure*, *void*, and *enumeration*) are now supported in most versions of C. The fourth data type, *long long int*, is a 64-bit integer data type developed expressly for use on CONVEX computers.

In addition to the data type extensions, the following differences exist between the portable C compiler and CONVEX Vector C:

- The portable C compiler treats the result of the *sizeof* operator as an *int*; the Vector C compiler treats it as *unsigned*. This treatment causes differences in the generated code if the result of the *sizeof* is subtracted from a smaller *int* value.
- The portable C compiler supports arrays with up to 14 dimensions. The Vector C compiler supports arrays with up to 7 dimensions.

2.2 Structure Data Type

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

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

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

```
old_emp = new_emp;
```

rather than the three statements:

```
strncpy (&old_emp.name, &new_emp.name, 40);
old_emp.age = new_emp.age;
old_emp.sex = new_emp.sex;
```

Vector C also extends the use of structures in function calls. Structures can now be passed to functions by value. The entire structure is pushed onto the stack. If *new_emp* declared in the previous example is passed to a function *update_emp*, 48 bytes are pushed on the stack. The extra three bytes maintain stack alignment for performance reasons.

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

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

2.3 Enumeration Data Type

In Vector C, enumerated variables are accepted as integers. The definition of the *enum* data type permits the use of enumerated variables wherever integers can be used.

The *vc* compiler does not provide *csd* support for enumeration data types. References to these types are treated as integers.

2.4 Void Data Type

The *void* data type has no values and performs no operations. It is used to specify the return type of a function that returns no value. You can also use *void* to cast when you want to discard a return value.

2.5 64-Bit Integer Data Type

The *vc* compiler supports the native 64-bit long 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/output.

2.6 64-Bit Integer Variables

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

You may declare *long long int* 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 32-bit unsigned integer variables 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. To avoid argument misalignment, declare the corresponding parameter declaration in the function as a *long long int*.

In Vector C, 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 promoted to a 64-bit format before the operation is performed. Signed values are converted to 64-bit format via sign extension. Unsigned values are converted with zero extension. Quantities assigned to a 64-bit integer quantity are converted to 64-bit format. Signed 64-bit integers should not be used to contain pointer data, since their portability is not assured.

2.7 64-Bit Integer Constants

The Vector C 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 uppercase *LL*, lowercase *ll*, or mixed *Ll*, *lL*; the uppercase designation is preferred because it is easier to read.

As with smaller integer constants, you may use both octal (leading 0) and hexadecimal (leading 0x). The default radix for all numeric constants is decimal. Decimal constants with values that exceed the largest signed 32-bit integer are taken to be type *long long*. Likewise, octal or hexadecimal constants with values that exceed the largest unsigned 32-bit integer are taken to be type *long long*.

2.8 64-Bit Integer Descriptors

Format strings that 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*, *%lu*, or *%lo* specify *long* (32-bit) conversions, *%lld*, *%llx* *%llu*, or *%llo*, specify *long long* integers.

You can use the library routines *printf* and *scanf* to manipulate 64-bit values. These routines are documented in Chapter 5 of this guide, and in Section 3S of the *CONVEX UNIX Programmer's Manual*.

3

Optimization

3.1 Overview

The CONVEX Vector C compiler optimizations produce code that results in enhanced performance. Optimization involves carefully manipulating operations in the source programs being compiled; the result is an object program that can run more efficiently. When you compile programs using one of the optimization options (*-O0, -O1, -O2*), compilation time increases as more optimizations are performed. Also, use of the optimization option can affect accuracy of the *csd* symbolic debugger.

The Vector C compiler performs the following types of optimization:

- Local optimization
- Global optimization
- Vectorization
- Machine-dependent optimization

3.2 Local Optimization

Local optimization is machine-independent scalar optimization performed on a sequence of consecutive statements with one entrance and one exit. Scalar optimization uses information within the source code to eliminate unnecessary computations during program execution. The *-O0* option on the command line causes the compiler to perform local optimization only.

3.2.1 Assignment Substitution

Assignment substitution is the process by which the compiler removes redundant loads and stores. The process involves substituting a preassigned value of a variable for all succeeding uses of the variable. For example:

```
x = y + c;
  ...
x1 = x;
  ...
x2 = x;
x = y + z;
```

Effectively, the $y+c$ replaces all uses of x up to the next assignment to x. As a result, the compiler can eliminate the loads on x if x can be retained in a register. Not only does this optimization save space and time, but it also allows other optimizations such as constant folding and redundant subexpression elimination.

3.2.2 Redundant-Assignment Elimination

Redundant-assignment elimination removes redundant assignments to the same variable. An assignment to a variable can be followed by another assignment to the same variable, overriding the result of the first assignment. Since there is no need for the program to perform the first assignment, the compiler eliminates it.

3.2.3 Redundant-Use Elimination

This optimization collapses all uses of a variable between two assigns into one use; it is a simplistic form of redundant-subexpression elimination. As a result, the compiler can eliminate loads provided it can retain the variable in a register.

3.2.4 Redundant-Subexpression Elimination

Redundant-subexpression elimination involves removing repeated evaluations of equivalent arithmetic, logical, and relational operations that are recognized as common subexpressions. When the compiler detects a common subexpression, it removes all but one common subexpression from the program and replaces it with the one retained in a register. In the following example, the first $y+c$ replaces the second occurrence but not the third:

```
z1 = y + c;
 ...
z2 = y + c;
 ...
y = x + 1;
 ...
z2 = y + c;
```

3.2.5 Constant Propagation and Folding

Constant propagation in a program means that when you assign a constant to a variable, everywhere the variable occurs later, the compiler replaces it with the constant. For example, if you assign $x = 5$, wherever x occurs later, constant propagation replaces it with the constant 5.

In constant folding, when the compiler comes across an operation on constants, like $y = 5 + 7$, it replaces the operation with its value (here, 12). The compiler may assign the new value to y, so that y can now be propagated.

Example:

Original Program	Transformed Program

`i = 5;`
`j = 0;`
`...`
`j = j + 2;`
`...`
`k = k + i * j;`
	`i = 5;`
	
	`...`
	`j = 2;`
	`...`
	`k = k + 10;`

Compile-time type conversions cause the compiler to perform type conversions on mixed-mode expressions. The conversion of constants and folded constants is performed as part of the

constant propagation and folding optimization. For example, if the program contains the assignment $x = 1$, where x is the double data type, the compiler converts the 1 to double. The effect is the same as if $x = 1.0$ had been written.

3.2.6 Algebraic Simplification

The compiler performs algebraic and trigonometric simplifications as shown in the following table.

The expression...	Is converted to...
x+0	x
x*1	x
x*0	0
x-0	x
x&-1	x
x&0	0
x|-1	-1
x|0	x
-1*x	-x
x-x	0
x/-1	-x
x/x	1
0-x	-x
0/x	0
sin(x)*cos(x)	.5*sin(2x)
sin(x)/cos(x)	tan(x)
cos(x)/sin(x)	1/tan(x)

The obvious variants of these operations are performed for the commutative operators; for example, $x+0+y \Rightarrow x+y$.

3.2.7 Simple Strength Reduction

The compiler attempts to replace time-consuming operations with those that execute faster, for example, replacing a multiply operation with a shift.

Examples:

```
x/c => (1/c)*x
5*x => (x<<2)+x
```

3.2.8 Common Subexpression Elimination

The compiler recognizes common subexpressions and retains the value in a register to avoid repetitious load operations. For example, the compiler recognizes $b+c$ as a common subexpression of $a+b+c+d$ and $c+d+b$.

3.3 Global Optimization

Global optimization is machine-independent scalar optimization that is performed over an entire program unit, including conditional statements and loops. The *-O1* option on the *vc* command line causes the compiler to perform both global and local optimization.

3.3.1 Constant Propagation and Folding

Global constant propagation and folding is similar to local constant propagation and folding, except the folded constant is propagated across the function.

Example:

Original Program	Transformed Program

`main()`
`{`
`  int a, b, c, i;`
`  a = 5;`
`  b = 15;`
`  scanf("%d", &i);`
`  if (i <= 0) {`
`  a = 6;`
`  c = a;`
`  b = a + c;`
`  } else {`
`  c = a + b;`
`  b = a + 8 + c;`
`  }`
`  printf("%d %d %d\n",`
`  a, b, c);`
`}`
	`main()`
	`{`
	`  int a, b, c, i;`
	`  a = 5;`
	`  b = 15;`
	`  scanf("%d", &i);`
	`  if (i <= 0) {`
	`  a = 6;`
	`  c = 6;`
	`  b = 12;`
	`  } else {`
	`  c = 20;`
	`  b = 33;`
	`  }`
	`  printf("%d %d %d\n",`
	`  a, b, c);`
	`}`

3.3.2 Dead-Code Elimination

As a result of constant propagation and folding, the control expression of an *if* statement may be folded. The alternative (*if, else*) that is now unreachable is eliminated.

An example of the use of dead-code elimination is conditional compilation. Code that is to be conditionally compiled is enclosed by an *if* statement that tests a variable whose value is set to 1 (compile enclosed code) or 0 (do not compile enclosed code) by an assignment or a preprocessor *# define* statement, or data initialization.

Example:

Original Program	Transformed Program

c = 0;
if (c) {
  a = a + 1;
  b = 10;
}
	c = 0;

3.3.3 Copy Propagation

Copy propagation occurs when the compiler replaces a variable with another variable to which it has been equated. For example, if you assign $x = y$, the compiler may replace later occurrences of x with y.

Example:

```
x = y;                    x = y;
 ...          becomes      ...
t = z - x;                t = z - y;
```

3.3.4 Redundant-Assignment Elimination

Redundant-assignment elimination involves removing assignment statements (definitions) that are not used. The final assignments to a dummy parameter, extern variable, or static local variable of a function are never eliminated. Also, if the right side of an assignment statement contains a function call, the function call is not eliminated.

Example:

Original Program	Transformed Program

main()
{
  int a, b, c, x, y, z;
 	
  x = y * z;
  if (a > 0)
  a = x * y;
  else
  x = a - b * c;
}
	main()
	{
	  int a, b, c, x, y, z;
	}

Note that all the executable code in this function is removed because the values assigned to x and a are never used.

3.3.5 Redundant-Subexpression Elimination

Like local redundant-subexpression elimination, global redundant-subexpression elimination involves the removal of common subexpressions. Instead of retaining the value of one common subexpression in a register, however, the compiler assigns the value to a compiler-generated temporary; all other occurrences are replaced by this temporary.

Example 1:

Original Program

```
main()
{
   double a, b, c, e, f, k, j, l;
      ...
   scanf("%lf", &c);
   if (k < l)
      a = b + (c * 4) / (-(j * b) + sqrt(c));
   else
      e = e - (b + (c * 4) / (-(j * b) + sqrt(c)));
   f = b + (c * 4) / (-(j * b) + sqrt(c));
      ...
}
```

Transformed Program

```
main()
{
   double a, b, c, e, f, k, j, l, t1;
      ...
   scanf("%lf", &c);
   t1 = b + (c * 4) / (-(j * b) + sqrt(c));
   if (k < l)
      a = t1;
   else
      e = e - t1;
   f = t1;
      ...
}
```

The common subexpression is moved only if it is safe; that means the subexpression is always evaluated.

Example 2:

Original Program

```
main()
{
   double a, b, c, e, f, j, k, l;

   scanf("%ld", &c);
   a = b + (c * 4) / (-(j * b) + sqrt(c));
   if (k < l)
      l = 5;
   else
      l = 6;
   f = e - (b + (c * 4) / (-(j * b) + sqrt(c)));
      ...
}
```

Transformed Program

```
main()
{
   double a, b, c, e, f, j, k, l, t1;
      ...
   scanf("%ld", &c);
   t1 = b + (c * 4) / (-(j * b) + sqrt (c));
   a = t1;
   if (k < l)
      l = 5;
   else
      l = 6;
   f = e - t1;
      ...
}
```

In certain cases, it may be advantageous to move redundant subexpressions out of *if* blocks or loops, even if it is unsafe (for example, if the alternative that did not evaluate the expression is seldom executed). The compiler can be directed to perform unsafe optimizations with the *-uo* option. Use this option only if you know that no exceptions will be generated by moving redundant subexpressions.

3.3.6 Code Motion

Code motion involves taking invariant computations in a loop and moving them before the loop. An invariant computation is one that yields the same result independent of the number of times the loop is executed. The computation can be a subexpression or assignment. For safety reasons, no code motion is performed on an invariant expression whose evaluation point does not lie on a path to all loop exits.

Example 1:

Original Program

```
f(a, b, c)
double a, b, c;
{
   double ar[10], d;
   int i;
      ...
   scanf("%ld", &d);
   for (i = 0; i < 10; i++) {
      a = b + (c * 4) / (-(e * b) + sqrt(c));
      ar[i] = 2 + b * c;
   }
      ...
}
```

Transformed Program

```
f(a, b, c)
double a, b, c;
{
   double ar[10], e, t1;
   int i;
      ...
   scanf("%ld", &e);
   a = b + (c * 4) / (-(e * b) + sqrt (c));
   t1 = 2 + b * c;
   for (i = 0; i < 10; i++)
      ar[i] = t1;
      ...
}
```

Example 2:

Original Program

```
f(a, b, c)
int a, b, c[10];
{
   int i;
   for (i = 0; i < 10; i++) {
      a = b;
      c[i] = 0;
   }
      ...
}
```

Transformed Program

```
f(a, b, c)
int a, b, c[10];
{
   int i;
   a = b;
   for (i = 0; i < 10; i++)
      c[i] = 0;
      ...
}
```

3.3.7 Strength Reduction

Strength reduction involves replacing an operator whose operands are either a loop induction variable or a loop constant, with an operator that executes faster. A loop induction variable is one whose value is changed within the loop linearly, that is, incremented by a constant amount. A loop constant is a constant or variable that is loop invariant, i.e., whose value is not changed within the loop. Typical operators subject to strength reduction are multiplications involved in the address calculation of subscripted variables.

Strength reduction of mixed-mode multiplies is not performed. The reduced operations are not numerically equivalent because of the imprecision of floating point for large numbers. For safety reasons, no strength reduction is performed on an expression whose evaluation point does not lie on a path to all loop exits.

Example:

Original Program

```
main()
{
   int c, i, x;
   i = 1;  /* i is a loop induction variable */
   do {
      x = i * c;  /* c is a loop invariant */
       ...
      i = i + 2;
   } while (i <= 100);
}
```

Transformed Program

```
main()
{
   int i, c, x, t1, t2;

   i = 1;
   t1 = i * c;
   t2 = 2 * c;
   do {
      x = t1;
      ...
      t1 = t1 + t2;
      i = i + 2;
   } while (i <= 100);
}
```

If *i* is dead (not used before being assigned) on exit from the loop and there are no other uses of *i* in the loop except in the incrementation and test, the incrementation can be eliminated. The test can be replaced by a test on the induced induction variable—the induced temporary. This optimization is known as linear-function test replacement. After linear-function test replacement, the equivalent transformed program is shown in the next example.

Example:

```
main()
{
   int i, c, x, t1, t2, t3;

   i = 1;
   t1 = i * c;
   t2 = 2 * c;
   t3 = 100 * c;
   do {
      x = t1;
      ...
      t1 = t1 + t2;
   } while (t1 <= t3);
      ...
}
```

3.4 Vectorization

Vectorization converts scalar operations on data arrays into equivalent vector operations. Vector operations use the vector registers in the CONVEX processors to perform operations on arrays of data simultaneously. Vector operations can manipulate up to 128 operands with a single instruction.

The *-O2* option on the *vc* command line causes the compiler to perform vectorization, global optimization, and local optimization.

The vectorizer vectorizes innermost *for* loops directly. For example, vector code is generated for the following loop:

```
for (i = 0; i < 100; i++)
   a[i] = b[i] + c[i];
```

Instead of generating a loop to load elements of *b* and *c*, add them, store into *a*, and advance *i*, vector code is generated to load 100 elements of *b* into a vector register, load 100 elements of *c* into another vector register, add them, and store the 100 resulting elements from the result vector register into *a*.

The algorithms that the CONVEX vectorizer uses are general. Loops containing nested *if* statements and nonlinear subscripts (subscripts whose values on succeeding iterations of a loop do not form arithmetic progressions) can be vectorized. For example, the following loop is fully vectorized:

```
for (i = 0; i < 100; i++) {
   a[i] = b[kk[i] + c[i*i]];
   if (a[i] < 0){
      if (a[i] < -100)
         a[i] = 0;
   } else
      a[i] = sqrt(a[i]);
}
```

The vectorizer does have some limitations, however, which are described later in this chapter.

Figure 3-1 shows examples of vectorizable loops in C.

Figure 3–1: Vectorizable Loops in C

```
main()
{
   double a[10], b[10], c[10];
   int i;

   /*
    * for loop to do multiple array initialization
    */
   for (i = 0; i < 10; i++) {
      a[i] = b[i] = c[i] = i;
   }

   /*
    *  for loop, similar to a FORTRAN DO loop
    */
   for (i = 0; i < 10; i++) {
      a[i] = b[i] + c[i];
   }

   /*
    *  while loop,
    *     the compiler must be able to
    *     find an induction variable
    */
   i = 0;
   while (i < 10)
   {
      a[i] = b[i] + c[i];
      i += 1;
   }

   /*
    *  do while loop,
    *     the compiler must be able to
```

```
     *    find an induction variable
     */
    i = 0;
    do {
       a[i] = b[i] + c[i];
    } while (++i < 10);
}
```

3.4.1 Strip Mining

The vector registers of the CONVEX processor holds up to 128 elements. When the number of iterations of a vectorizable loop exceeds (or could exceed) 128 elements, the vectorizer "strip mines" the loop before vectorizing it. Strip mining replaces the loop with two loops, the innermost of which has an iteration count that never exceeds 128. For example, strip mining makes the conversion shown in the following example:

Example:

Original Loop	Converted Loop

`for (i = 0; i < n; i++)`
`  a[i] = b[i] + c[i];`
	`i = 0;`
	`for (lv = n; lv > 0; lv -= 128) {`
	`  j = i + (lv < 128 ? lv : 128);`
	`  for (iv = i; iv < j; ++iv)`
	`  a[iv] = b[iv] + c[iv];`
	`  i = i + 128;`
	`}`

Here, *lv* is a variable introduced by the compiler to count the number of elements remaining to be processed, and the loop on *iv* represents a vector operation.

3.4.2 Loop Distribution

Nests of loops are vectorized by first distributing the outermost loop, then vectorizing each of the resulting loops or loop nests; e.g., consider the following nest of loops:

```
for (i = 0; i < n; i++) {
   b[i][0] = 0;
   for (j = i; j < m; j++)
      a[i] = a[i] + b[j][i] * c[j][i];
   d[i] = e[i] + a[i];
}
```

Distribution of the outer loop yields intermediate code equivalent to the following three loops:

```
for (i = 0; i < n; i++)
   b[i][0] = 0;

for (i = 0; i < n; i++)
   for (j = i; j < m; j++)
      a[i] = a[i] + b[j][i] * c[j][i];

for (i = 0; i < n; i++)
   d[i] = e[i] + a[i];
```

3.4.3 Loop Interchange

The first and last loops on *i* and the loop on *j* of the previous example are all innermost loops and can be vectorized directly. However, to yield additional performance improvement, the vectorizer performs loop interchange optimization on the middle nest replacing it with the following nest:

```
for (j = 0; j < m; j++)
   for (i = j; i < n; i++)
      a[i] = a[i] + b[j][i] * c[j][i];
```

When the vector code is generated for the *i* loop, elements of *b* and *c* are accessed contiguously as they are loaded into vector registers.

3.4.4 Vectorization Summary

When a program is compiled with the *-O2* option, the compiler produces summary information that tells you what vectorization was performed and indicates any conditions that prevented a particular loop from being vectorized.

Two types of summary information are produced—vectorization messages and the vectorization summary report. Vectorization messages, Figure 3-2, are produced for the source file as a whole. The vectorization summary report, Figure 3-3, contains more detailed information and is produced for each subroutine.

Figure 3–2: Vectorization Messages

```
vc: Loop on line 8.5 of sort.c (i-loop) fully vectorized
vc: Loop on line 15.5 of sort.c (i-loop) fully vectorized
vc: Loop on line 25.5 of sort.c (i-loop) fully vectorized
vc: Loop on line 36.5 of sort.c (i-loop) fully vectorized
vc: Loop on line 47.1 of sort.c (i-loop) fully vectorized
```

Figure 3–3: Vectorization Summary Report

```
          Vectorization Summary for Routine main

Source Iter.                                  Vector-
Line   Var.      Start     Stop     Step     ization     Reason
-------------------------------------------------------------
     7 i         *EXPR*      10        1    FULL  Vector
    13 i         *EXPR*      10        1    FULL  Vector
    20 i         *EXPR*      10        1    FULL  Vector
    28 i         *EXPR*      10        1    FULL  Vector
```

3.4.5 Limitations on the Vectorizer

The vectorizer has the following limitations:

1. Loops containing function calls or that have more than one exit or entrance cannot be vectorized.

2. If an outer loop contains a nested loop with an induction variable whose start value, stop value, or step value varies with iterations, the outer loop is not vectorized. (An induction variable is a variable that is incremented or decremented by the same amount on each iteration of the loop.)

3. A vectorized loop may give incorrect results if one of its induction variables has zero-stride. For example:

```
zero = 0;
for (i = 0; i < n; i++) {
   j = j + zero;
   b[i] = a[j];
   a[j] = c[i];
}
```

   If vectorized, all resulting values of *b* are identical. In a scalar loop, *b[2]* would equal *c[1]*, *b[3]* would equal *c[2]*, and so on.

4. A vectorized loop may fail if the indexes for a conditionally referenced array fall outside the array bounds. For example:

```
int a[100];
for (i = 0; i < 10000; i++) {
    if (i < 100)
       x += a[i];
}
```

   Here the vectorized version of the loop tries to load *a[10000]*, yet *a* has only 100 elements. A scalar version of the loop executes properly, since *i* never exceeds 100 when the conditionally executed statement *x += a[i]* is executed.

5. Optimization is disabled if you use assembly-language statements in your C code.

3.4.6 Recurrence

In addition to these limitations, a loop may not be vectorized or may be only partially vectorized if a recurrence (real or apparent) is present. A recurrence is present when an assignment stores a value that is used during a later iteration to compute the value on the right side of the same assignment. For example:

```
for (i = 1; i < n; i++)
   a[i] = a[i-1] + 1;
```

Here, on the first iteration *a[1] = a[0]+1*, and on the second iteration *a[2] = a[1] + 1*, using the value of *a[1]* computed on the first iteration. Such a computation is inherently serial and cannot be vectorized.

More generally, vectorization is inhibited if two array references are so related that neither could validly be placed first in vectorized code, or the compiler cannot determine which to place first. Situations like these are also referred to as recurrences.

For example, the following loop cannot be vectorized if the sign of *n* is unknown:

```
for (i = 0; i < 100; i++){
   a[i+n] = 1;
   a[i] = 0;
}
```

If *n* were +1, the value of *a[2]* on termination of the loop would be 0, implying that the assignment to *a[i]* would have to follow the assignment to *a[i+n]*. However, if *n* were -1, the value of *a[2]* on termination would be 1, implying that the assignment to *a[i+n]* would have to follow the assignment to *a[i]*.

The previous example illustrates the most common reason for the compiler failing to vectorize a vectorizable loop—the addition of a loop constant quantity of unknown sign to a subscript. Another frequent cause of apparent recurrences is the use of array references in subscripts. For example:

```
for (i = 0; i < 100; i++)
   a[j[i]] = a[j[i]] + 1;
```

This loop is vectorizable but the compiler cannot ignore the possibility that elements of the *j* array may be repeated. Therefore, the assignment to *a[j[i]]* could produce a value that would be used in computation of its right side on a later iteration, and the compiler must assume that the references to *a[j[i]]* are in a recurrence.

You can use the ***no_recurrences*** directive to vectorize loops where vectorization would otherwise be prevented by apparent recurrences. (See Appendix E.)

The compiler vectorizes a special class of recurrence called a reduction. In general, a reduction has the form:

```
x = x op y
```

where *x* is a scalar variable (or scalar relative to the loop in question); *y* is any expression not involving *x* (*x* is not assigned or used elsewhere in the loop); and *op* is one of the operators +, -, *, &, |, or ^.

For example, the following loop computes the sum of the first 100 elements of the array *a* with a sum reduction:

```
for (sum = i = 0; i < 100; i++)
   sum += a[i];
```

The vectorizer sometimes inserts vector temporaries to enable a loop with a recurrence to be partially vectorized. For example, the following loop cannot be vectorized as is:

```
for (i = 0; i < n; i++)
   a[i] = a[i-1] + b[i] * c[i];
```

The vectorizer recognizes, however, that the multiplication *b[i]*c[i]* can be vectorized. To do so, it introduces a temporary array (here represented by *t[i]*) and splits the loop into two loops:

```
for (i = 0; i < n; i++) {
  t[i] = b[i] * c[i];
}

for (i = 0; i < n; i++)
   a[i] = a[i-1] + t[i];
```

The first loop is then vectorized and a sequential loop is generated for the second loop.

3.4.7 Examples

1. Use of a scalar temporary does not inhibit vectorization. The following loop is fully vectorized.

```
for (i = 0; i < n; i++){
   x = a[i] + b[i];
   a[i] = c[i];
   z[i] = y[i] + x;
}
```

2. A loop containing only a scalar reduction is fully vectorized. For example,

```
k = 1;
for (i = 0; i < 10; i++)
   k += 1;
```

   is fully vectorized. The loop is replaced by the equivalent code:

```
k = 11;
```

3.5 Machine-Dependent Optimization

Machine-dependent optimization enhances the object code produced by the compiler to take advantage of the machine architecture. Machine-dependent optimization is always performed regardless of the optimization level.

3.5.1 Instruction Scheduling

Instruction scheduling determines an order of instructions that effectively uses the function units on the computer. You have no control over this scheduling. The compiler rearranges the instructions in the program to achieve a high level of concurrent operation. In debug mode, instruction scheduling is done only within (not between) statements, so that *csd* can correlate instructions with the lines in the original program.

The compiler schedules instructions across numbers of statements instead of in one statement only, thereby achieving substantial performance improvements. For example:

```
a = b + c;
d = e - f;
```

Regular Code		Optimized Code	
ld.w	b,s0	ld.w	b,s0
ld.w	c,s1	ld.w	c,s1
add.s	s0,s1	ld.w	e,s2
st.w	s1,a	ld.w	f,s3
ld.w	e,s0	add.s	s0,s1
ld.w	f,s1	sub.s	s3,s2
sub.s	s1,s0	st.w	s1,a
st.w	s0,d	st.w	s2,d

In the left example, the subtraction cannot execute until the addition is completed. In the right example, these two operations proceed almost concurrently.

3.5.2 Span-Dependent Instructions

The compiler attempts to generate a 2-byte branch or a 4-byte jump instruction for conditional and unconditional transfers of control within a program. These short form instructions, which conserve memory and improve execution speed, can be generated when the span (that is, the distance from the branch or jump instruction to the target location) is within the limits defined for these instructions.

3.5.3 Branch Optimization

Many compilers generate branch instructions that branch to the next sequential instruction. The CONVEX Vector C compiler generates such branches internally, then removes them by branch optimization before the object code is produced.

3.5.4 Register Allocation

Register allocation is an optimization that is performed automatically. The register allocation scheme in the CONVEX Vector C compiler is different from other machines because of the machine's unique architecture. Most machines try to minimize the number of registers allocated for a given expression; the CONVEX compiler attempts to maximize the number to achieve more parallelism. Register declarations do not affect register allocation except in the presence of *asm* statements.

The only time you should be aware of this optimization is when you are invoking an assembly-language routine. The compiler assumes on any call that all the registers are destroyed; as a result, it saves and restores any that are active. Therefore, you need not be concerned about the contents of the registers when coding an assembly-language routine.

When an *asm* statement is encountered, you can assume that the value of the first register variable in the scope is loaded into S7, the second into S6, the third into S5, and the fourth into S4. Any other register variables are treated as automatics.

3.5.5 Hoisting Scalar and Array References

The compiler "hoists" scalar and array references out of innermost loops if the value referred to does not change during the execution of the loop. Array references may be hoisted out of vectorized loops if:

- The array is indexed only by constants, some combination of variables that do not change within the vectorized loop, and the iteration variable of the vectorized loop.
- There are no intrinsic calls within the loop.

3.5.6 Matching Paired Vector References

Under certain circumstances, a vector register can be treated as a set of accumulator registers, making it possible to move loads and stores of that vector register outside of a vectorized loop. The simplest situation under which this can occur is a matrix multiply.

Array references may be matched in vectorized loops if:

- The vectorized loop is part of a nest of loops.
- There is only one use and one assign to the array within the vectorized loop.
- The array use and array assign have identical subscripts.
- The array use can be hoisted, either as is or after the interchange of two of the scalar loops in the nest.

3.5.7 Strength Reduction and the Code Generator

The code generator performs certain strength-reduction operations on instruction-level operations. For example, instead of multiplying by a power of 2, the code generator transforms the operation into a shift.

3.5.8 Tree-Height Reduction

Tree-height reduction is best explained by example. Consider the following expression:

```
a+b+c+d+e+f+g+h
```

Two ways of evaluating this expression are as follows:

Method 1	Method 2
`(((((((a+b)+c)+d)+e)+f)+g)+h)`	`(((a+b)+(c+d))+((e+f)+(g+h)))`

Method 1 requires that *a+b* be evaluated first. The result of that calculation is then used to compute *(a+b) + c*, and so on. None of the additions can proceed simultaneously, because each must wait for the result of the addition to its right.

Method 2 allows four additions to execute in parallel: *(a+b)*, *(c+d)*, *(e+f)*, and *(g+h)* can be computed simultaneously, because none of these additions requires the results from any other addition. Furthermore, when the results from these additions become available, the additions *(a+b)+(c+d)* and *(e+f)+(g+h)* can also execute in parallel.

In general, the time required to evaluate a particular parenthesization is roughly proportional to the depth of the expression, i.e., the deepest nesting level of parentheses. The nesting level is 6 for the first method, but only 3 for the second.

When the compiler has a choice, as in the previous example, of what order in which to evaluate expressions, it chooses the order that yields the least depth and therefore the highest degree of parallelism. (Internally, the compiler represents expressions as "trees," the height of which corresponds to the depth of the expression.) This may result in a slightly different numerical result for floating-point operators because of rounding off in the least-significant bits.

4

Calling Conventions

4.1 Introduction

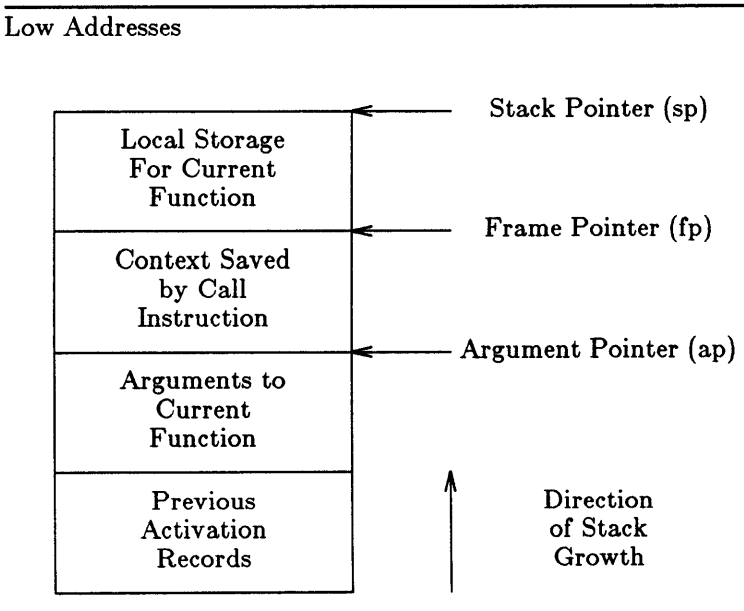
When C function calls are executed, the current state of several hardware registers used by the calling function must be preserved. The contents of these registers are pushed onto the runtime stack as a part of an activation record. The called function may then alter the machine registers as it runs.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

The CONVEX Vector C compiler does not preserve the value of registers across function calls. Called functions that restore the frame pointer (fp) register to its original state are allowed to modify any register passed to them.

4.2 Function Stack Layout

Figure 4-1 shows the top of the runtime stack. The stack pointer (sp) register contains the address of the topmost location on the runtime stack. The fp register contains the address of the last frame pushed on the runtime stack by a *call* or *calls* instruction. The ap register contains the address of the arguments to the current function.

Figure 4–1: Top of the Runtime Stack



4.3 Standard 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 ap register. The updated register should point to the first argument in the argument list. (The first argument in the list is the last one pushed.)

3. Push an additional word. This word should contain the count of the number of arguments passed.

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

Executing a *calls* instruction places a stack frame on the runtime stack. The stack frame contains the current values of the program counter (pc) (return address), the program status word (psw), the fp, and the ap. The fp is set equal to the sp, then the sp is updated to point to the new top of stack.

The conventions that apply to function calls are as follows:

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

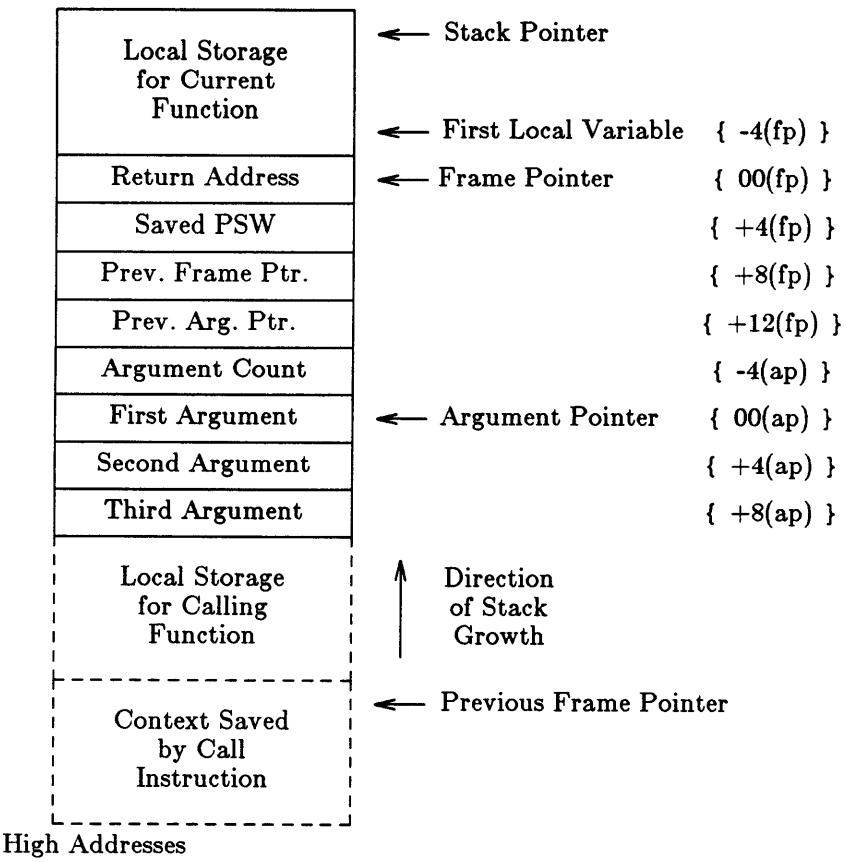
2. The called function need not preserve the contents of any register except the fp. The called function uses the current value of the ap to access the arguments passed to the function by its parent.

3. The fp points to the context block pushed by the caller. The called function references the local storage it has allocated on the runtime stack by referencing negative offsets from the fp.

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

Figure 4-2 shows the layout of the stack as seen by a function after it has been called, and after it has allocated some storage for local variables on the top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 4–2: Stack Layout



Called functions return to their parents by placing a return value in register s0, then executing the *rtn* instruction.

When the *rtn* instruction is executed, automatic storage allocated by the called function is automatically deallocated. This instruction also restores the 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 function.

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

4.4 Code Generated for Standard Calls

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

Example:

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

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

```
        .globl  _child           ; called function
_child:
        sub.w   #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
```

4.5 Standard Function Names

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

4.6 Standard Function Arguments and Return Values

C arguments are passed using the "call-by-value" method; that is, the value (rather than the address) of an argument is passed. C programs may pass addresses as arguments to functions by using pointer variables or the & address operator. Note that arrays are passed by address.

Structures in C are passed by value. All the elements of the structure are pushed onto the runtime stack before the call. The calling process is faster 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 an area on the stack and return a pointer to that area in S0.

5

Runtime Library

5.1 Introduction

This chapter describes the standard C library functions provided with the CONVEX Vector C compiler. The C library contains interfaces to the system calls and various specialized functions (e.g., cursor manipulation functions) that are not described in this chapter. See Section 3 of the *CONVEX UNIX Programmer's Manual* for descriptions of the library functions and Section 2 for a list of system calls.

The functions in this chapter are listed in alphabetical order within the following categories:

- Character handling functions
- *libm* math functions
- Other math functions
- Nonlocal jump functions
- Signal handling functions
- Standard buffered I/O functions
- Low-level I/O functions
- General utility functions
- String handling functions
- Date and time functions
- Error handling functions

Each section contains a description of each function, examples of function use, and a reference section. Appendix D contains a list of libraries with their locations and contents.

5.1.1 Calling Format

The easiest way to call runtime library functions is to use the associated header. The header is a file that declares a group of functions along with any types and *#define* macros needed to use the functions. To include a header, use the *#include* preprocessor control line in the following format:

#include < *file.h* >

where <*file.h*> is the name of the header. The header always ends with the *.h* suffix and must be surrounded by angle brackets < > on the command line as shown.

To use a library function with its header, include the header file before referencing the function. For example, you can use the *getc* and *feof* functions as follows:

```
#include <stdio.h>
main()
{
    FILE*F;
    char*p;

    ...
```

```
        while (!feof(F)){
            *p++= getc(F);
        }
        ...
    }
```

Some functions do not use header files and this fact is so noted in the text. (See the *CONVEX UNIX Programmer's Manual.*)

5.2 Character Handling Functions <*ctype.h*>

The character handling functions are used for classifying ASCII-coded integer values by table lookup. Each function returns nonzero for true or zero for false. The character-handling functions are as follows:

isalnum true for any letter or digit.

isalpha true for any letter.

isascii true for any ASCII character.

iscntrl true for any control character. Control characters are nonprinting and are implementation-defined.

isdigit true for any decimal digit.

islower true for any lower-case letter.

isprint true for any printing character (including space).

ispunct true for any punctuation character. These characters are defined as all printing characters except spaces, digits, controls, or letters.

isspace true for any space, tab, carriage return, newline, or form feed.

isupper true for any uppercase letter.

isxdigit true for any hexadecimal digit.

In the following list, *tolower* and *toupper* are character-mapping functions; *_tolower*, *_toupper*, and *toascii* are macros.

tolower returns the corresponding lowercase letter when the argument is an uppercase letter. If the argument is not an uppercase letter, it returns the argument unchanged.

toupper returns the corresponding uppercase letter when the argument is a lowercase letter. If the argument is not a lowercase letter, it returns the argument unchanged.

_tolower returns the same data as *tolower*, except it has a restricted domain and runs faster. If the argument to *_tolower* is not an uppercase letter, its result is undefined.

_toupper returns the same data as *toupper*, except it has a restricted domain and runs faster. If the argument to *_toupper* is not a lowercase letter, its result is undefined.

toascii returns the argument with all bits turned off that are not part of the standard ASCII character.

5.2.1 Examples

The following function converts all uppercase letters in *str* to lowercase.

```
#include <ctype.h>
lowercase(str)
char *str;
{
    while(*str)
        *str = tolower(*str++);
}
```

Another way to accomplish the conversion is to use the macro *_tolower* as shown below. The check for an uppercase letter must occur before you use *_tolower* in order to avoid conversion of non-letter characters. This example is more efficient than using the *tolower* function because it does not include function call overhead.

```
#include <ctype.h>
lowercase(str)
char *str;
{
    while(*str) {
        if(isupper(*str))
            *str = _tolower(*str);
        str++;
    }
}
```

5.2.2 References

For further information about these functions, see *ctype*(3) and *ascii*(7).

5.3 *libm* Math Functions *<fastmath.h>* and *<math.h>*

The include file *<fastmath.h>* enables the fast calling sequence for scalar math functions and provides access to the vector math functions. In the fast calling sequence, the *callq/rtnq* mechanism is used and arguments are passed in registers rather than on the stack. *<fastmath.h>* contains a *#define* statement for each CONVEX math function accessible from C. For example, the double- and single-precision tangent functions are defined as:

```
#define tan(x)  _mth$d_tan((double)(x))
#define stan(x) _mth$r_tan((float)(x))
```

The compiler recognizes these symbols as keywords for functions with the fast calling sequences. The C vectorizer uses this feature to generate calls to vector functions. For example, if

```
a(i) = tan(b(i));
```

appears in a vectorizable loop, and the file *<fastmath.h>* is included, the compiler recognizes *_mth$d_tan* as a function with the fast calling sequence and generates a call to the corresponding vector function, *_mth$vd_tan*.

The *_mth$* functions are located in the C library that is automatically loaded by *vc*.

If you want to use the standard C calling convention (Chapter 3), the library *libm.a* must be loaded using the *-lm* option on the *vc* command line. The header *<math.h>* contains declarations for this library. The following functions always use the standard C calling convention and cannot be vectorized: *ceil, floor, gamma, j0, j1, jn, y0, y1,* and *yn*. If these functions are used, you must include *-lm* on the *vc* command line.

A portable way to include *<fastmath.h>* when compiling with Vector C, and *<math.h>* when using other C compilers, is to use the predefined preprocessor name *convexvc*. For example:

```
#ifdef convexvc
#include <fastmath.h>
#else
#include <math.h>
#endif
```

5.3.1 Math Errors

Math errors are grouped into two categories: domain errors (EDOM) and range errors (ERANGE). Domain errors occur when a function argument is out of the domain of that function. Range errors result when the computed value cannot be represented within the machine precision or when the size of the argument would lead to significant inaccuracy of function performance. The math errors are listed in the include file *<errno.h>*, and are generally more specific than EDOM and ERANGE. When an error occurs, the global integer *errno* is set to the appropriate error code and a message is printed to *stderr*.

5.3.2 Special Constants

The include files *<fastmath.h>* and *<math.h>* define the following special constants:

- HUGE—the largest double-precision floating-point number for native format.
- HUGEI—the largest double-precision floating-point number for IEEE format.

If used, these constants must be in the proper context. That is, HUGE must be used if you are compiling for native mode and HUGEI must be used if you are compiling for IEEE mode so that internal representations are correct.

5.3.3 Function List

The *libm* math functions are as follows (trigonometric functions assume arguments in radians unless otherwise noted):

abs	integer absolute value.
acos	double-precision arc cosine.
asin	double-precision arc sine.
atan	double-precision arc tangent.

atan2	double-precision arc tangent of two arguments.
cabs	double-precision Euclidean distance function (complex absolute value).
ceil	ceiling function returning the smallest integer not less than the argument.
cos	double-precision cosine.
cosh	double-precision hyperbolic cosine.
exp	double-precision exponential.
fabs	double-precision absolute value.
floor	floor function, returns the largest integer not greater than the argument.
gamma	log gamma function.
hypot	double-precision Euclidean distance function (complex absolute value).
ipow	integer power function of two arguments.
j0	Bessel functions of the first kind (j1, order 0).
j1	Bessel functions of the first kind (j1, order 1).
jn	Bessel functions of the first kind (j1, order n).
log	double-precision natural logarithm.
log10	double-precision base 10 logarithm.
lpow	long long integer power function of two arguments.
pow	double-precision power function of two arguments.
sacos	single-precision arc cosine.
sasin	single-precision arc sine.
satan	single-precision arc tangent.
satan2	single-precision arc tangent of two arguments.
scabs	single-precision complex absolute value.
scos	single-precision cosine.
scosh	single-precision hyperbolic cosine.
sexp	single-precision exponential.

sfabs	single-precision absolute value.
shypot	single-precision Euclidean distance function.
sin	double-precision sine.
sinh	double-precision hyperbolic sine.
slog	single-precision natural logarithm.
slog10	single-precision base 10 logarithm.
spow	single-precision power function of two arguments.
sqrt	double-precision square root.
ssin	single-precision sine.
ssinh	single-precision hyperbolic sine.
ssqrt	single-precision square root.
stan	single-precision tangent function.
stanh	single-precision hyperbolic tangent
tan	double-precision tangent function.
tanh	double-precision hyperbolic tangent.
y0	Bessel functions of the second kind (y1, order 0).
y1	Bessel functions of the second kind (y1, order 1).
yn	Bessel functions of the second kind (y1, order n).

5.3.4 Examples

The following is an example of a single-precision complex sine function. It uses single-precision math functions. The inclusion of *<fastmath.h>* enables the fast calling sequence.

```
#include <fastmath.h>
typedef struct {
    float real;
    float imag;
} complex;

csin(y,x)        /* single-precision complex sine */
complex *x, *y;
{
    /* real part of result */
    y->real = ssin(x->real) * scosh(x->imag);
    /* imaginary part of result */
    y->imag = scos(x->real) * ssinh(x->imag);
}
```

The next example computes the square root of a vector of double precision values. If compiled with level 2 optimization (*-O2*), the *for* loop is vectorized, resulting in calls to the vector square root function.

```
#include <fastmath.h>
vsqrt(a,b,n)    /* double precision vector square root */
int n;          /* length of vector */
double *a, *b; /* input vector in a, result in b */
{
    int i;
    for(i=0;i<n;i++) {
        b[i] = sqrt(a[i]);
    }
}
```

5.3.5 References

For further information about these functions, see *abs*(3), *exp*(3M), *floor*(3M), *gamma*(3M), *hypot*(3M), *j0*(3M), *sin*(3M), and *sinh*(3M).

5.4 Other Math Functions

The other math functions do not use a header file. All these functions, except *random*, are used in the formatted I/O runtimes. For example, *atof* performs ASCII to floating-point conversion for *scanf*, *sscanf*, and *fscanf*. The math functions are as follows:

atof performs ASCII to floating-point conversions.

atoi performs ASCII to integer conversions.

atol performs ASCII to long integer conversions.

atoll performs ASCII to long long integer conversions.

ecvt performs floating-point to ASCII conversions.

fcvt performs floating-point to ASCII conversion using FORTRAN F format.

frexp extracts mantissa and exponent from a double precision float.

gcvt performs floating-point to ASCII conversion using FORTRAN G format.

ldexp loads a mantissa and exponent into a double float.

modf extracts the integer and fractional parts of a double float.

random generates a pseudo-random number.

5.4.1 Examples

The following example shows how the runtimes *srandom* and *random* can be used to generate random floating-point values in the range 0 to 1.

```
#define   RANDMAX   2147483647.0          /* 2**31-1 */
main()
{
    int random(), srandom(), seed;
    float x;
    ...

    /* initialize random number generator */
    seed = 1;
    srandom(seed);
    ...

    /* get next random number and scale
       to the range [0,1] */
    x = ( (float) random() / RANDMAX);
    ...
}
```

5.4.2 References

For further information about these functions, see *atof*(3), *ecvt*(3), *frexp*(3), and *random*(3).

5.5 Nonlocal Jump Functions <*setjmp.h*>

The nonlocal jump functions enable nonlocal *goto* statements and are useful for handling errors and interrupts that occur in low-level subroutines within a program. The nonlocal jump functions are as follows:

longjmp restores the environment saved by the last call of the *setjmp* function. It then returns so that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*, which must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called.

setjmp saves its stack environment in *env* for later use by *longjmp*. It returns the value 0.

5.5.1 Example

This example shows how *setjmp* and *longjmp* can be used to process error conditions. First, *setjmp* is called to save the current environment in *env*. Next, p0 is called and, in turn, calls p1. If either function fails, a *longjmp* is executed, restoring the environment saved by *setjmp*. The call to *longjmp* causes control to return to the beginning of the switch statement as if the call to *setjmp* had just returned the second argument to *longjmp*.

```
#include <stdio.h>
#include <setjmp.h>
#define PASSED    0
#define P0_FAILED 1
#define P1_FAILED 2

jmp_buf env;
```

```
main()
{
    switch(setjmp(env)) {
    case PASSED:
        p0();
        printf("p0 and p1 passed\n");
        break;
    case P0_FAILED:
        printf("p0 failed\n");
        break;
    case P1_FAILED:
        printf("p1 failed\n");
        break;
    }
}

p0()
{
    int error;

    ...

    /* if error condition, return to beginning
       of switch stmt */
    if(error)
        longjmp(env,P0_FAILED);
    else
        return(p1());
}

p1()
{
    int error;

    ...

    /* if error condition, return to beginning
       of switch stmt */
    if(error)
        longjmp(env,P1_FAILED);
    else
        return(PASSED);
}
```

5.6 Signal Handling Functions *<signal.h>*

The signal handling functions allow signal manipulation within a program. Signals are generated by the user at the terminal, by a program error, by request of another program, or by a process in background needing access to its control terminal.

psignal produces a short message on the standard error file describing the indicated signal. *psignal* prints the argument string *s*, then a colon, then the name of the signal and a newline.

signal establishes a condition handler that allows signals either to be ignored or to cause an interrupt to a specified location. When a process receives a signal, the default

action is to cleanup and abort. You may write an alternative signal-handling routine by making a call to *signal* and specifying the signal number, the condition handler, and a flag. You may not, however, override the default action for the SIGKILL and SIGSTOP signals. Table 5-1 contains a list of the signals and their meanings.

Table 5-1: Signal Names

Signal Name	No.	Meaning
SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating-point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal from kill
SIGURG	16**	Urgent condition present on I/O channel
SIGSTOP	17***	Stop (cannot be caught or ignored)
SIGTSTP	18***	Stop signal from keyboard
SIGCONT	19**	Continue a stopped process
SIGCHLD	20**	Child status has changed
SIGTTIN	21***	Background read attempted from control terminal
SIGTTOU	22***	Background write attempted to control terminal
SIGIO	23**	I/O is possible on a descriptor
SIGXCPU	24	CPU time limit exceeded
SIGXFSZ	25	File size limit exceeded
SIGVTALRM	26	Virtual time alarm
SIGPROF	27	Profiling time alarm
SIGWINCH	28	Window changed
SIGLOST	29	Resource lost
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2

(*) the default action for the signal is to terminate the program with a core dump; (**) the default action is to ignore the signal; (***) the default action is to stop the program. The default action for all other signals is to terminate the program.

Examples:

Define condition handler for keyboard-generated interrupts.

```
sigc = signal (SIGTSTP, sigdie);
```

Restore default action for keyboard-generated interrupts.

```
sigc = signal (SIGTSTP, SIG_DFL);
```

Ignore keyboard-generated interrupts.

```
sigc = (SIGTSTP, SIG_IGN);
```

5.6.1 Exceptions

An exception is an event that disrupts the running of a program. Exceptions occur because of problems in the currently executing program (e.g., arithmetic inconsistencies or address translation faults) or as a result of some asynchronous event (e.g., an interrupt or hardware failure). Exceptions result in the transfer of control to a predetermined address known as an exception or signal handler. Table 5-2 defines the mapping of exceptions to signals and codes.

Condition handlers receive three arguments that describe the signal and the state of the program when the signal occurred. The first argument is the signal number; the second argument is the exception code; the third argument describes the state of the program when the trap was taken and is defined in the include file *<signal.h>*.

Example:

In the following example, the function ***newhandler*** is established as the condition handler for keyboard-generated interrupts.

```
#include <signal.h>
main()
{
    ...
    /* use newhandler for keyboard-generated exceptions */

    oldhandler = signal(SIGTSTP, newhandler);
    ...
}

newhandler(sig,code,scp)
int sig, code;
struct sigcontext *scp;
{
    ...
}
```

Table 5-2: Mapping Exceptions

Hardware	Signal	Code
Arithmetic Traps:	SIGFPE(8)	
Integer overflow	SIGFPE	FPE_INTOVF_TRAP (1)
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP (2)
Floating overflow	SIGFPE	FPE_FLTOVF_TRAP (3)
Floating division by zero	SIGFPE	FPE_FLTDIV_TRAP (4)
Floating underflow	SIGFPE	FPE_FLTUND_TRAP (5)
Reserved Operand	SIGFPE	FPE_RESOP_TRAP (6)
Segmentation Violations:	SIGSEGV (11)	
Read access violation	SIGSEGV	SEG_READ_TRAP (1)
Write access violation	SIGSEGV	SEG_WRITE_TRAP (2)
Execute access violation	SIGSEGV	SEG_EXEC_TRAP (3)
Invalid segment descriptor	SIGSEGV	SEG_INVSDR_TRAP (4)
Invalid page table reference	SIGSEGV	SEG_INVPTP_TRAP (5)
Invalid data reference	SIGSEGV	SEG_INVDATA_TRAP (6)
I/O access violation	SIGSEGV	SEG-IOACC_TRAP (7)
Ring Violations:	SIGBUS (10)	
Inward ring address reference	SIGBUS	BUS_INWADDR_TRAP (1)
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP (2)
Inward ring return	SIGBUS	BUS_INWRTN_TRAP (3)
Invalid syscall gate entry	SIGBUS	BUS_INVGATE_TRAP (4)
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP (5)
Illegal Instruction:	SIGILL (4)	
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP (0)
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP (1)
Undefined op code	SIGILL	ILL_UNDFOP_TRAP (4)
Trace pending	SIGTRAP(5)	
Bpt instruction	SIGTRAP (5)	

5.7 Standard Buffered I/O Functions *<stdio.h>*

The standard buffered I/O functions constitute a user-level buffering scheme. A file associated with buffering is called a stream, and is declared to be a pointer to a defined type file. There are three normally-open streams with constant pointers declared in the include file and associated with the standard open files: *stdin* (standard input file), *stdout* (standard output file), and *stderr* (standard error file). The constant pointer NULL means no stream at all. The integer constant EOF is returned at end of file or error by integer functions that deal with streams. Any routine that uses the standard input/output functions must include the header file *<stdio.h>*.

Data can be transferred a character at a time (*getc, putc, fgetc, fputc*), a string at a time (*gets, puts, fgets, fputs*), a word at a time (*getw, putw, fgetw, fputw*), or in large blocks (*fread, fwrite*). Formatted I/O is supported by *printf, fprintf, scanf,* and *fscanf.* In-memory format conversions are done with *sprintf* and *sscanf.*

The standard buffered I/O functions are as follows:

clearerr resets the error indication on the named stream.

fclose empties any buffers for the named stream and closes the file.

fdopen matches a stream with a file descriptor obtained from *open*, *dup*, *creat*, or *pipe*(2).

ferror returns non-zero when an error has occurred reading or writing the named stream.

fflush writes any buffered data for the named output stream to that file. The stream remains open.

fgetc returns the next character from the named input stream.

fgets reads *n*-1 characters, or up to a newline character, from the stream into a string.

fileno returns the integer file descriptor associated with the stream.

fopen opens a file specified by *filename* and matches a stream with it.

fprintf performs formatted output on the named output stream.

fputc outputs a character to the named output stream. *fputc* performs the same function as *putc*, except that *fputc* is a function, whereas *putc* is a macro.

fputs copies the null-terminated string *s* to the specified output stream.

fread reads a block of data from the specified input stream. *fread* returns the number of items actually read.

fscanf reads formatted input from stream.

fseek sets the position of the next input or output operation on the stream.

ftell returns the current value of the offset relative to the beginning of the file associated with the specified stream. The value is measured in bytes.

fwrite writes a block of data to the named output stream. *fwrite* returns the number of items actually written.

getc returns the next character from the named input stream.

getchar returns the next character from *stdin*.

gets reads a string from the standard input stream *stdin*.

getw returns the next 32-bit word from the specified input stream.

mktemp replaces a template with a unique file name and returns a pointer to the template. The template should look like a filename with six trailing Xs that are replaced with the current process ID and a unique letter.

popen creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used to write to the standard input of the command or read from its standard output.

printf	performs formatted output on the standard output stream, *stdout*.
putc	outputs a character to the specified output stream.
putchar	outputs a character to the standard output stream, *stdout*.
puts	writes the null-terminated string to the standard output stream, *stdout*, and appends a newline character.
putw	outputs a word to the output stream.
rewind	rewinds the named input stream.
scanf	reads formatted input from the standard input stream, *stdin*.
setbuf	assigns a new buffer of a predetermined size to a stream after it has been opened but before it is read or written.
setbuffer	assigns a new buffer of a specified size to a stream after it has been opened, but before it is read or written.
setlinebuf	changes *stdout* or *stderr* from block buffered or unbuffered to line buffered.
sprintf	performs formatted output and places the output in a null-terminated string.
sscanf	reads formatted input from a character string.
ungetc	pushes a character back on an input stream.

5.7.1 Examples

The following function copies the contents of file *f* to file *g*. Both files have been opened before the call to copy.

```
#include <stdio.h>
copy (f,g)
FILE *f, *g;
{
  int c;

  while((c = getc(f)) != EOF)
       putc(c,g);
}
```

The function *concat* concatenates two files horizontally. That is, it reads a line from each file, and stores the concatenation of the two lines in a third file. In all three files, each line is terminated with a newline character. The files have been opened before the call to *concat*. Although not shown in this example, the return values from the I/O functions are used to indicate error conditions. For example, *fwrite* returns the number of items actually transferred if the operation was successful; otherwise, 0 is returned.

```
#define n ...   /* maximum record size */

concat(f,g,h)   /* concat lines from f and g to h */
FILE *f, *g, *h;
{
    char s[n], t[n];
```

```
    while((fgets(s,n,f)) != EOF && (fgets(t,n,g)) != EOF) {
                /* write line from f */
        fwrite(s,sizeof(char),strlen(s),h);
                /* write line from g */
        fwrite(t,sizeof(char),strlen(t),h);
                /*terminate w/ newline*/
        putc('\n',h);
    }

    if(feof(f)) {
                /* copy remaining lines from g */
        while(fgets(t,n,g) != EOF) {
            fwrite(t,sizeof(char),strlen(t),h);
            putc('\n',h);
        }
    } else if(feof(g)) {
                /* copy remaining lines from f */
        fwrite(s,sizeof(char),strlen(s),h);
        while(fgets(s,n,f) != EOF) {
            fwrite(s,sizeof(char),strlen(s),h);
            putc('\n',h);
        }
    }
}
```

5.7.2 References

For further information, see the description of the individual functions in Section 3S of the *CONVEX UNIX Programmer's Manual.*

5.8 Low-Level I/O Functions

The low-level I/O functions are system calls accessible from the C library. The buffered I/O routines are optimized for the UNIX file system and their use is generally more optimal than the low-level functions. The low-level I/O functions do not use buffering. These functions identify files with integers called file descriptors.

The low-level I/O functions are as follows:

close closes the connection between an open file and a file descriptor, freeing the file descriptor for use with other files.

creat creates a new file or prepares to rewrite an existing file.

lseek moves the read/write pointer. *lseek* returns the pointer location as measured in bytes from the beginning of the file.

open opens a file for reading or writing, or creates a new file. *open* returns a file descriptor for the requested file.

read reads data from the object referenced by a descriptor. *read* returns the number of bytes actually read.

unlink removes an entry for a file from its directory.

write writes data to the object referenced by a descriptor. *write* returns the number of bytes actually written.

Example:

In this example, *lseek* and *read* are used for random access input. Here, the offset specified for *lseek* is from the file beginning. It may also be from the current position or end of file, depending on the value of the last argument.

```
#define RECLENGTH ...     /* record length */

get_rec(fd,buf,n)         /* read the n'th record from fd */
int fd, n;
char *buf;
{
    int cnt;
        /* convert record number to byte count */
    cnt = (n-1)*RECLENGTH;

        /* position to count bytes from beginning */
    lseek(fd,cnt,0);

        /* read n-th record */
    return(read(fd,buf,RECLENGTH));
}
```

5.8.1 References

For detailed information about these functions, see *close*(2), *creat*(2), *lseek*(2), *open*(2), *read*(2), *seek*(2), *unlink*(2), and *write*(2).

5.9 General Utility Functions

The general utility functions use no header file. *alloca*, *calloc*, *malloc*, *realloc*, and *free* are used for dynamic memory allocation. *abort* and *exit* are used for nonstandard program termination, where *abort* produces a core dump and *exit* cleans-up any open files. The remaining general utilities are *getenv*, *qsort*, *sleep*, and *system*.

abort executes an instruction that is illegal in user mode. This creates a signal that terminates the process with a core dump useful for debugging.

alloca temporarily allocates memory. This space is freed in future calls to *alloca*.

calloc allocates memory. The space is initialized to zeros.

exit terminates a process after calling the standard I/O library function *_cleanup* to flush any buffered output. *exit* never returns.

free deallocates a block of memory allocated by *malloc* and friends.

getenv searches the environment list for a string of the form *name=value*. Returns a pointer to the string *value* if such a string is present. If such a string is not present, *getenv* returns the value 0 (NULL).

malloc allocates a block of memory and returns a pointer to the block. *malloc*, along with *free*, is a simple general-purpose memory allocation package.

qsort is a quick-sort utility. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments that are pointers to the elements being compared.

realloc changes the size of the block allocated by *malloc* and returns a pointer to the new block. The contents are unchanged up to the lesser of the new and old sizes.

sleep suspends the current process from executing for the number of seconds specified by the argument.

system issues a shell command. The current process waits until the shell has completed the *string*, then returns the exit status of the shell.

5.9.1 Examples

The following function inserts a new element into the linked list, following element p. If *malloc* cannot allocate the requested amount of memory, it returns 0.

```
typedef struct list {
    struct list *next;
    double x,y;
} llist;

insert(p,x,y)   /* insert new list item after p */
llist *p;
double x,y;
{
    llist *q;

                /* allocate list item */
    q = (llist *)malloc(sizeof(llist));

    if(q) {
                /* set x & y fields */
        q->x = x;
        q->y = y;
                /* insert list item */
        q->next = p->next;
        p->next = q;
    }
    return(q);
}
```

The next example uses the function *realloc* to increase the size of an array.

```
main()
{
    long long *arr, x;
```

```
        int cnt, size, incr;
        ...
            /* check for array full */
        if(cnt >= size) {
            size = size+incr;
            arr = (long long *) realloc((char *)arr,
                    size, sizeof(long long));
        }

            /* add new element */
        arr[cnt++] = x;
        ...
}
```

5.9.2 References

For detailed information about these functions, see *abort*(3), *exit*(3), *getenv*(3), *malloc*(3), *qsort*(3), *sleep*(3), and *system*(3).

5.10 String Handling Functions <*strings.h*>

The string handling functions are used to manipulate character arrays and blocks of memory. *bcmp*, *bcopy*, and *bzero* are block memory functions optimized for execution on the CONVEX hardware. The remaining functions perform operations on null-terminated character strings.

The string handling functions are as follows:

bcmp compares two byte strings, returning zero if they are identical, nonzero if they are not.

bcopy copy a block of data.

bzero places "0" bytes into the specified string.

ffs finds the first bit set in the argument received and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

index returns a pointer to the first occurrence of a character in a string or zero if the character does not occur in the string.

rindex returns a pointer to the last occurrence of a character in a string or zero if the character does not occur in the string.

strcat appends a copy of a string to the end of another string.

strcmp compares two strings and returns an integer greater than, equal to, or less than 0, depending on whether the first string is lexicographically greater than, equal to, or less than the second string.

strcpy copies a null-terminated string.

strlen returns the number of non-null characters in a string.

strncat appends a copy of a string to the end of another string. At most *n* characters are copied.

strncmp compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether the value of the first string is greater than, equal to, or less than the value of the second string. Unlike *strcmp*, this function compares no more than *n* characters.

strncpy copies exactly *n* characters from the second string to the first string, truncating or null-padding if necessary.

5.10.1 Examples

```
#include <strings.h>
main()
{
    char *s, *t, *v;
    ...

    /* concatenates s and t, with the
    lexicographically smaller one first */
    if(strcmp(s,t) <= 0)
        v = strcat(s,t);
    else
        v = strcat(t,s);
    ...
}
```

5.10.2 References

For detailed information about these functions, see *bstring*(3) and *string*(3).

5.11 Date and Time Functions <*time.h*>

The date and time functions return CPU and real time and perform conversions on these times.

asctime converts a broken-down time returned by *localtime* or *gmtime* into ASCII.

ctime converts into ASCII a time pointed to by *clock* as returned by *time*(3C).

ftime fills a *timeb* structure with the current time.

getdate converts most common time specifications as returned by *ftime* to standard UNIX format.

getrusage returns information about process resource use, including process CPU time.

gettimeofday returns current system time and time zone.

gmtime converts time returned by *time*(3) to a structure containing the broken-down time using GMT, the standard UNIX time.

localtime converts time returned by *time*(3) to a structure containing the broken-down time, correcting for time zone and possible daylight savings time.

time returns the date and time of day measured in seconds.

timezone returns the name of the time zone associated with its first argument, measured in minutes westward from the Greenwich meridian; *ftime*(3C) supplies the zone and *ds = arguments*.

5.11.1 Example

This example computes elapsed real time and CPU time. For real time, *gettimeofday* returns time since January 1, 1970. Therefore, two calls are required to get the program elapsed real time. For CPU time, *getrusage* returns the CPU usage in seconds for the executing process.

```
#include <sys/time.h>
#include <sys/resource.h>
main()
{
    struct timeval realtime0, realtimen;
    struct timezone zone;
    struct rusage *ru;
    unsigned long t;
    float cputime;

        /* starting real time */
    gettimeofday(&realtime0,&zone);

    ...

        /* ending real time */
    gettimeofday(&realtimen, &zone);

        /* print elapsed real time in seconds */
    t = realtimen.tv_sec - realtime0.tv_sec;
    printf("elapsed real time: %d secs\n", t);

        /* print cpu time in seconds */
    getrusage(RUSAGE_SELF, &ru);
    cputime = (float) ru.ru_exutime.tv_sec
            + ru.ru_exutime.tv_usec/1000000.0
            + (float) ru.ru_stime.tv_sec
            + ru.ru_stime.tv_usec/1000000.0;
    printf("elapsed cpu time: %f secs\n", cputime);
}
```

5.11.2 References

For further details about these functions, see *ctime*(3), *getdate*(3), *time*(3C), *times*(3C), *getrusage*(2), and *gettimeofday*(2) .

5.12 Error Handling Functions *<errno.h>*

The error handling function, *perror*, retrieves system error message numbers. When an error occurs, the global integer *errno* is set to the appropriate error number, and an error message is written to *stderr*. The header file *<errno.h>* contains a complete list of all errors and their names. These errors are also listed in Appendix B, and *intro*(2).

perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. *perror* uses the following variables:

- *sys_errlist*—a table that simplifies variant formatting of messages. *errno* can be used as an index in this table to get the message string without the newline.
- *sys_nerr*—the number of messages provided in the *sys_errlist table*.

Example:

```
#include <stdio.h>
main()
{
    FILE *f;
    ...
    f = fopen("data", "r");
    if(f==NULL)
        perror("failed open");
    ...
}
```

If the file data does not exist, the following error message is output.

```
failed open: No such file or directory
```

5.12.1 References

For further information about these functions, see *perror*(3).

6

Debugging C Programs

6.1 Overview

This chapter presents an overview of the methods available for debugging C programs. For complete information on this subject, please see the *CONVEX adb Debugger User's Guide* and the *CONVEX Consultant User's Guide* (optional product).

There are three tools for debugging C programs: *pmd*, a post-mortem dump analyzer, *csd*, a source-level debugger, and *adb*, an assembly-level debugger. Both *csd* and *pmd* require special support from the compiler and loader and are probably the best tools for debugging. On the other hand, *adb* requires no support from the loader or compiler and is most useful for examining core dumps from failed programs.

Before beginning debugging, make certain that the problem is not being caused by the use of floating point numbers. Some of the more common problems are:

- Truncation error—Although it is not feasible to run an algorithm indefinitely, the more iterations of the series you can run, the greater the reliability of the answer. Conversely, the fewer iterations you run, the larger the possible error is likely to be.

- Round-off error—For exact representation, floating point numbers sometime require more space than the fixed word length provided. Rounding the least significant portion of the number can introduce errors. Although using single-precision numbers is considerably faster, using double-precision greatly reduces the significance of the introduced error.

- Propagated error—An error in the original data can be aggravated by either of the previously mentioned errors.

If you think that your errors may be related to floating point arithmetic, you may want to consult a numerical analysis textbook for more information on floating point errors and possible solutions.

6.2 Post-Mortem Dump Analyzer (*pmd*)

The post-mortem dump analyzer (*pmd*) displays formatted listings when your program terminates abnormally. You run *pmd* at the same time you run your program. If the program aborts, *pmd* displays information that can help you locate the problem.

The *pmd* output is written to *stderr*, so that it appears on your screen. If you need to re-examine the post-mortem information, you can do so using the *dump* command in *csd* (see "The Dump Command" later in this Chapter). The information displayed includes:

1. The signal that caused the program to abort.

2. A runtime stack backtrace and the approximate source line location of where program took the exception.

3. The contents of the machine registers.
4. A dump of active local variables in each routine on the runtime stack.
5. A dump of global, or common, variables.
6. The region of disassembled object code where the exception took place.
7. A summary of resources used by the program (execution time, elapsed time, percent of time in CPU, size of shared memory and unshared memory, page faults, and swaps).

6.2.1 Invoking *pmd*

To use *pmd*, compile your program using the *-db* option on the compiler command line. Then invoke *pmd* as follows:

pmd [**-a**] [**-d** *n:n...*] [**-l**] [**-S**] [**-s**] [**-t** *limit*] [**-v**] *command* [*args*]

where

-a Print the contents of the address registers in hex, decimal, and floating point formats.

-d *n:n* Prints up to *n* elements of arrays in the post-mortem dump listing. Up to seven dimensions for arrays can be specified.

-l Displays a post-mortem dump in long format. The long format includes items 1 through 7 described in the previous section.

-S Excludes from the post-mortem dump the approximate source code location where *command* took the exception.

-s Displays a post-mortem dump in short format. The short format, which is the default format, includes:

1. The signal that caused the program to abort.
2. A runtime stack backtrace.
3. The approximate source code location of the exception.

-t *n* Limits the cpu time for *command* to *n* seconds.

-v Includes the contents of the machine vector registers in the post-mortem dump listing.

6.2.2 Restrictions on *pmd*

You must compile your program with the *-db* option in order for *pmd* to obtain information about the runtime stack backtrace, source line locations, and active local variables.

The higher the optimization level specified when compiling the program, the greater the uncertainty of variables and source line location accuracy. The following assumptions can be made:

1. Regardless of the optimization level, memory locations of common block variables are accurate at the call to a subprogram.

2. The validity of memory location contents of active local variables cannot be guaranteed. In most code, however, the memory locations for local variables are likely to be accurate.

3. The mapping of object code to source code is granular to the basic block. A basic block is defined as a sequence of statements with no branches.

4. The values of subprogram arguments at entry points to subprograms are accurate.

Since the core file must be created for *pmd* to produce post-mortem dump information, you must have write permission in your current working directory. In addition, you cannot exceed the maximum core file size. Set the maximum core file size in your shell with the *limit* parameter (see *csh*(1)).

6.3 *csd* Debugger

csd was designed specifically for debugging C and FORTRAN programs executing on the CONVEX UNIX operating system. Its features include:

- Statement-level execution control—*csd* debugs programs at the statement level, rather than at machine level.

- Enhanced capabilities for examining core files and program files in a variety of formats—you can examine core dumps with *csd* to find out the line on which the program failed.

- Ability to debug multiple source-module applications—*csd* automatically updates the debugging environment as execution branches from module to module.

- Symbolic access to program variables—*csd* can access program variables by name rather than by absolute address.

The information that follows summarizes some of the features of *csd*. For a complete description, please refer to the *CONVEX Consultant User's Guide*.

6.3.1 Invoking *csd*

To use *csd*, you must first compile your program with the *-db* compiler option. The command to invoke *csd* has the format:

csd [**-r**] [**-I** *dir*] [...] [*objfile*] [*corefile*]

where

-r instructs *csd* to execute *objfile* immediately (without waiting for *csd* commands). If the program terminates successfully, *csd* exits. Otherwise, *csd* reports the reason for termination, and you can enter *csd* commands. *Csd* reads from */dev/tty* when you specify *-r* and the standard input is not a terminal. If you do not specify this option, *csd* displays a prompt and waits for a command.

-I *dir* directs *csd* to add the specified directory to the list of directories searched when *csd* looks for a source file. Note that a space is required between *-I* and the directory name.

objfile is an object file that has been directed with the appropriate option to produce symbol table information. If you do not specify *objfile*, *csd* produces a query.

corefile refers to the pathname for a file containing a core dump generated as the result of an abnormal program termination. The *corefile* is generally named "core".

The *corefile* contains an image of the state of the program at its termination. Once you access the corefile using *csd*, you can determine which routines were active, their arguments, and the current value of all the active program variables. After *csd* loads the core image, you can determine the final program state by examining stack traces and variable contents.

6.3.2 Running *csd*

Once you have invoked *csd*, use the *run* command to start executing the program to be debugged. The program that you are debugging is known as the "child".

The *run* command has the following format:

run [*args*] [*<filename*] [*>filename*]

The *run* command arguments control the execution of the programs to be debugged. These arguments are the same arguments used when the program is run as a shell command. You can redirect standard input and output to the child using the last two arguments shown above. Entering *<filename* reads data from the filename specified; entering *>filename* writes data to the file specified. Note that there are no blanks between the symbols < and > and *filename*. If you invoke the *run* command more than once, the child variables are re-initialized with each invocation before execution begins.

NOTE

If you compile your program using some level of optimization, the optimized object code may not always match the original source code. Portions of your original program may have been optimized out or rearranged. To avoid this problem for debugging purposes, you can compile your program using the *-no* option on the *vc* command line.

6.3.3 Stopping *csd*

csd stops executing the child and asks for user input when the program being debugged exits, when an interrupt signal occurs, when breakpoints are encountered, or when a fatal program error occurs. Fatal errors return you to the *csd* command interpreter rather than to the UNIX shell. *csd* lists the number of the line containing the error, enabling you to locate the position in the source file at which the error occurred.

You may also use the *quit* command to exit *csd*. When you exit *csd*, program control passes to the program that invoked *csd* (often the UNIX shell).

6.3.4 *help* Command

You can display a short menu of commonly used *csd* commands with the *help* command. The menu includes the syntax and function of the basic *csd* command groups. To use the *help* command, enter

help

at the *csd* command level.

6.3.5 Other Basic Operations

The following commonly used *csd* commands perform basic operations. Invoke each of these commands by entering the name of the command at the *csd* command level.

Use the *cont* command to resume execution of the program after a stop (such as encountering a breakpoint). You cannot continue execution if a standard program exit or abnormal termination has occurred. You can use the *run* command, however, to restart the program.

Use the *step* command to execute one or more high-order language source lines. The format of the *step* command is

step [*count*]

where *count* is an optional step count; the default is one source line. Note that you must start the child with the *run* command before single-stepping can be performed. The *next* command also enables you to execute source lines. However, *next* skips over subroutine calls and stops at the next line in the calling routine, while *step* stops at the first line of the subroutine block. The format for the *next* command is

next [*count*]

where *count* is an optional step count (default is one source line).

6.3.6 Current Address

csd maintains a current address that is similar in function to the current pointer in the UNIX text editor, *ed*(1). The *csd* current address pointer is a line number in a given source file. To move from this address, use the *step* command, which executes one source line. Similarly, the *next* command executes the next source line, but stays in the current file. The *cont* command also modifies the current address. If execution stops in another source file, *csd* automatically updates the current address within that file.

6.3.7 Environment

The environment, or scope, of a symbol refers to the subroutines and modules in which it is declared. In C, the scope of a variable may be further defined in terms of nesting levels, or blocks, inside the subroutine. *csd* automatically assigns a name to nested blocks. This name, which begins with "$b" and ends with a nesting level number, qualifies references to these variables.

For program variables, the environment describes how to access a particular variable on the runtime subroutine stack. For data declarations, the environment defines how references to that declaration are resolved.

Specify the scope or environment of a symbol in the following formats:

[*module_name.*][*function_name.*][*block_name.*]*symbol_name*

The default environment is the function currently executing.

To reference nonunique variables in functions other than the one currently executing, you must use the full environmental pathname. Modify the default environmental pathname with the *file* and *func* commands.

The *file* command, has the following format:

file [*filename.c*]

This command changes the current source name to the name specified in the filename argument. If you do not specify an argument, *csd* displays the current source file pathname. Use the *list* command described in the next section to display source lines from this file. The *func* command changes the current subroutine environment in which *csd* is working, thereby changing the default pathname. The command has the following format:

func [*function_name*]

If you do not specify a function name, *csd* displays the current function name.

In cases where subroutines call one another recursively, both subroutines are on the runtime stack at the same time. To move the current subroutine up or down the stack in order to resolve names, use the *up* or *down* commands. The formats for these commands are:

up [*count*]
down [*count*]

where *count* is the argument for the number of levels to move on the stack. If no *count* argument is specified, the default is 1.

The *vregs* command dumps the contents of the vector registers. If the vector registers are empty, *vregs* dumps all zeros. *csd* displays the output of these commands in hex on the terminal screen.

6.3.8 *print* Command

The *print* command displays data values, symbol addresses, and expressions. Its format is:

print *expression* [, *expression*][...]

Expression is a list of constants or variables and operators. You can perform basic arithmetic operations on the constants and variables within the expression.

The *print* command displays the current value of the selected expression. Variables having the same name as the one in the current block may be fully qualified. You can use the field reference operator "." with pointers and records, making the C operator "->" unnecessary (although it is supported).

6.3.9 *whatis* Command

The *whatis* command prints the data type of a variable. The format of this command is as follows:

whatis *name*

on the command line, where *name* is any variable name.

6.3.10 *which* Command

The *which* command prints the default qualified pathname for a symbol. The format of this command is as follows:

which *name*

where *name* is the name of a variable or symbol.

6.3.11 *whereis* Command

The *whereis* command displays the environmental pathname of each symbol name matching the given identifier. The format of this command is as follows:

whereis *name*

6.3.12 *where* Command

The *where* command displays a stack trace of the subroutine calls that led to the current program state. The output consists of subroutine arguments and return addresses. You can redirect output to a specific file by using the > symbol. The format of this command is as follows:

where [>*filename*]

6.3.13 *list* Command

The *list* command lists lines from the current source file. The *list* command can be entered in several formats, including those listed below.

Entering the command in the following format lists lines in the current source file from the first specified line number to the second specified line number, inclusive. If you do not specify line numbers, the next 10 lines are listed.

list [*source_program_line-number* [[,]
source_program_line-number]]

Entering the command with the name of the function displays a small region of text around the first line of the function named.

6.3.14 *dump* Command

There are several different listings that the *dump* command can generate. If you are using *csd* without having run *pmd*, the *dump* command generates a listing consisting of a stack backtrace, the signal that caused the abort, and the approximate source code location of the exception. Output goes to *stdout* if you do not specify a file.

If you have run your program with *pmd* before using *csd*, the *dump* command generates a listing according to the *pmd* options you specify.

The *dump* command can be used on active programs as well as programs that ended abnormally. For instance, running *csd* on a program, you can use *dump* to examine the variable values up to a set breakpoint. If your program has aborted, you can use *dump* to examine the state of the program at that time.

This command has the following format:

dump [> *filename*]

where *filename* is the name of the file to which output is directed.

6.3.15 Setting Breakpoints

The *stop* command is used to set breakpoints. When *csd* encounters a breakpoint, it stops execution of the program and returns to command mode. *csd* commands are then used to instruct *csd* to display information for monitoring the progress of a program.

You can instruct *csd* to stop execution either at a specific line number or when a particular subroutine is called. In both cases, *csd* stops program execution at the beginning of the line. You can also instruct *csd* to stop at a particular location only if a certain user-defined condition is encountered. *Condition* here is a Boolean expression. Table 6-1 shows alternative forms of the command with brief explanations.

Table 6-1: Forms of the *stop* Command

Command	Result
stop at *source-line* [**if** *condition*]	Program execution stops at the specified source line number (if the condition is true).
stop in *function* [**if** *condition*]	Program execution stops at the first line of the function named (if the condition is true).
stop *variable* [**if** *condition*]	Program execution stops when the value of the named variable changes (if the condition is true).
stop if *condition*	Program execution stops when the condition is true.

6.3.16 Setting Tracepoints

Tracepoints, like breakpoints, instruct *csd* to print information that can be used to monitor the progress of the program. Unlike breakpoints, however, tracepoints do not return control to you when this information is displayed. Program execution continues until the program terminates normally or until it encounters a breakpoint or fatal error or is stopped. *Condition* is a Boolean expression evaluated before printing the tracing information; if the expression is false, *csd* does not display the tracing information. The condition can involve any program variables and can be as complex as necessary.

To set tracepoints, use the *trace* command. The first argument used with the *trace* command describes the program element to be traced. If you do not specify an argument, each source line is displayed before processing, resulting in slow program execution. Table 6-2 shows alternative forms of the command.

Table 6-2: Forms of the *trace* Command

Command	Result
trace [**in** *function_name*] [**if** *condition*]	Prints trace information only when the given function is being processed (if condition is true).
trace *source_line_number* [**if** *condition*]	Prints the line identified immediately before being processed (if condition is true).
trace *function_name* [**in** *function_name*] [**if** *condition*]	Prints lines of information each time the function is called, including the name of the calling routine, parameters passed and value returned if argument is a function (if the condition is true).
trace *expression* **at** *source-program line number* [**if** *condition*]	Prints the value of the expression each time the identified source line is reached (if the condition is true).
trace *variable* [**in** *function_name*] [**if** *condition*]	Prints the name and value of the variable each time it changes (if the condition is true).

6.3.17 Deleting Breakpoints and Tracepoints

The *delete* command deletes breakpoints and tracepoints. This command has the following format:

delete *command-number*

where *command-number* is the unique number identifying the breakpoint or tracepoint.

The *status* command is used to determine the command number of currently active *trace* or *stop* commands. This command has the following format:

status [*>filename*]

csd routes output from the command to any filename used as an argument. If you do not specify a filename, output is routed to *stdout*. Output includes a list of breakpoints and tracepoints and their command numbers.

6.4 *adb* Debugger

adb is an object-code debugger that requires no special support (such as the recompilation of programs) from the loader or compilers. The two major uses of the *adb* debugger are:

- To examine core dumps resulting from failed programs
- To enable interactive debugging of programs at the assembly-language level

adb can also be used to run programs with embedded breakpoints, to patch files, and to print output in a variety of formats. Since *adb* does not require special support from the compiler, its only effect on optimization is to make your program run more slowly.

The information that follows summarizes some of the features of *adb*. For a complete description, please refer to the *CONVEX adb Debugger User's Guide*.

6.4.1 Command Formats

The command to invoke *adb* is as follows:

adb [**-w**][*objfil* [*corfil*]]

where

-w creates both *objfil* and *corfil*, if necessary, and opens them for reading and writing so that *adb* can modify files. If you specify either *objfil* or *corfil* as "-", that file is ignored.

objfil is an executable UNIX file (default is *a.out*).

corfil is a core image file produced by the operating system when a job terminates abnormally (default is *core*).

Once you have invoked *adb* and receive the prompt (adb), you can enter requests for various functions. The basic format for these requests is:

[*address*] [,*count*] [*command*] [;]

If you specify *address*, *adb* sets the current address (referred to as dot) to the specified address. *Address* is explained later in this chapter.

Count specifies the number of times that *adb* executes the command. Use a semicolon to separate multiple command statements. Table 6-3 lists some of the most commonly-used requests.

Table 6-3: General *adb* Requests

Command	Meaning
?	Print contents from program file.
/	Print contents from core file.
=	Print value of dot.
:	Breakpoint control.
$	Miscellaneous requests.
;	Request separator.
!	Escape to shell.

6.4.2 Displaying Memory Locations

Usually, the object file contains the text portion of the program and the core file contains data. The ? command display texts (instructions) and the / command displays the data as found when the core file was created.

These two requests generally have the following format:

address **?** *format*

or

address / *format*

where *format* is the format of the output.

adb can print output data in several formats. Specify the format using a collection of letters and characters. If you do not specify a format, output will appear in the previously requested format. Table 6-4 shows the most commonly used format letters; for a complete list, consult *adb*(1).

Table 6-4: Format Letters

Format Letters	Definition
bx	One byte in hex
c	One byte as a character
wo	One word in octal (4 bytes = 1 word)
wt	One word in signed decimal
f	One word in floating point
i	CONVEX instruction
s	A null terminated character string
a	The value of dot
u	One word as unsigned decimal
^	Backup dot

6.4.3 Current Address and Expressions

adb maintains a current address, called dot, that is similar in function to the current pointer in the UNIX editor, *ed*(1). Specifying an address causes dot to be set to that location. A set of expressions in the program being debugged is used to represent addresses. These expressions are composed of decimal, octal, and hexadecimal integers and may be combined with operators to alter the location of dot. Note that all arithmetic in *adb* is performed with 64 bits of precision. Table 6-5 lists the expression operators. The expressions are described following the table.

Table 6-5: Expression Operators

Expression	Description
.	The value of dot
+	The value of dot incremented by the current increment
^	The value of dot decremented by the current increment
"	The last address typed
%	Integer division
*	The contents of the location addressed by *exp* in *corfil*
&	Bitwise AND
|	Bitwise OR
#	Round up to the next multiple

Some of the major expressions within *adb* are as follows. For a complete list of expressions, consult *adb*(1).

integer — is a number. *adb* interprets integers according to the prefix. Integer prefixes are: 0o or 0O (zero oh) for octal radix, 0t or 0T for decimal radix, and 0x or 0X for hexadecimal radix. Thus, 0o20 = 0t16 = 0x10 = 16. If you do not specify a prefix, the default radix is used. The default radix is initially hexadecimal, but you can change it with the $*x* command.

integer.fraction — is a 32-bit floating-point number.

<name — is either a variable name or a register name. If *name* is a register name, then *adb* gets the value from the system header in *corfil*. Display the register names using the $*r* command.

symbol — is a sequence of uppercase or lowercase letters, underscores, or digits. The symbol cannot start with a digit. You can use the backslash (\) character to escape other characters. *adb* takes the value of *symbol* from the symbol table in *objfil*, and prefixes an initial underscore character if needed.

_symbol — is the true name of an external symbol in C. You may have to specify the underscore before the symbol to distinguish it from internal or hidden variables in a program.

(exp) — is the value of the expression *exp*.

6.4.4 Examining Core Dumps

The most common use of *adb* is to examine core dumps from failed programs. The first step to examine a core dump is to invoke *adb* with the command:

adb a.out core

Once *adb* is running, you can enter requests to determine why your program failed. The command

$c

will print a C or FORTRAN backtrace through the functions that were called. (This command may not work if the failure is such that it scrambles the state of the runtime stack.) You can then examine the values of the arguments displayed to determine where the error occurred.

6.4.5 Finding Variables

When examining a core dump, it can be helpful to look for certain variables. You can use the command

$e

to display a list of all known global variables and their current values. These are the only variables that you can examine by name using *adb*.

6.4.6 Miscellaneous Operations

Requests of the form

?f

print locations starting at the *address* in *objfil* in the format specified by *f*. *adb* increments dot by the sum of the increments for each format letter given.

Requests of the form

/f

print locations starting at the *address* in *corfil* in the format specified by *f*. *adb* increments dot by the sum of the increments for each format letter given. Requests of the form

=f

print the value of *address* itself in the format specified by *f*. Since *adb* uses current address to remember its current location, you can use this command to reference locations relative to the current address. For example

```
.-0T10,0T10/wt
```

prints 10 decimal numbers starting at dot -10 in the *corfil*.

To enter UNIX requests from *adb*, type

!*command*

where *command* is a UNIX command.

If you invoked *adb* from *csh*(1), you can suspend *adb* return to *csh* by typing:

^z

You may return to *adb* later by typing:

fg

6.4.7 Running and Debugging With *adb*

Breakpoints are used to instruct *adb* to print information that can be used to monitor the progress of a program. When *adb* encounters a breakpoint, it stops execution of the program and returns to the *adb* command mode. To set breakpoints, you must first invoke *adb*. Once you have done this, the command to set breakpoints is

address [,*c*]**:b** [*request*]

where *address* is the address of a function. (You should only set breakpoints at function entry points unless you are familiar with the code generated by the compiler.) If you specify a value as the *c* (count) modifier, *adb* bypasses the specified breakpoint *c-1* times before stopping.

If you specify *request*, *adb* executes the specified requests each time it encounters the breakpoint. If you do not specify *request*, or if the request sets dot to zero, the breakpoint stops the program. Separate multiple *requests* on a single line with semicolons. For example, the request:

```
_main+10,10:b
```

stops the program the 10th time it reaches the address *_main+10*. The request

```
_fopen:b $c
```

stops the program at *_fopen* and prints a stack backtrace.

To delete a breakpoint at *address*, use the command:

address:**d**

To display all the breakpoints in a program, use the command:

$b

Output from this command is similar to Figure 6-1.

Figure 6–1: Display Breakpoint Output

```
$b
breakpoints
count          bkpt          command
1              _settab       settab,5?i; input?s
3              _fgetc
1              _fopen
```

The *:r* request runs the *objfil* as a subprocess. If you specify an *address* with this request, the program begins execution at that point; otherwise, the program starts executing at the standard entry point. The format for this request is:

[*address*]**:r** [*arguments*]

If you specify *arguments*, they are passed to the program as command-line arguments. Arguments may include < and > to specify input and output redirection. Use the *k* request to terminate the process you are currently debugging. The format of this request is:

:k

A

Vector C Data Types

A.1 Introduction

This appendix describes the basic CONVEX data types and specifies the requirements for the proper alignment of code and data in memory.

The data types supported by Vector C exploit the internal data representations found on the CONVEX family of supercomputers. All the scalar data types defined by the CONVEX architecture are supported in Vector C.

Vector C 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. Two's-complement arithmetic is used for all signed arithmetic on CONVEX computers. Integer types may also be unsigned.

- **Character data type:** *char. char* 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. Characters can also be signed and unsigned.

- **Floating-point data types:** *float, double.* Declare 32-bit single-precision data with the keyword *float,* and 64-bit double-precision 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 integer value of a member of a set of enumerated values.

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

- **Void.** The *void* data type is used to declare functions that do not return a useful value. Using this data type ensures that undefined values returned by void functions are not used.

- **Record structure:** *struct.* Record structure data differs from the basic data types in that it is an aggregate of data type fields. Structure data may be of any size.

- **Union.** This data type uses the syntax of structures to reserve space in memory for variables whose data types may vary.

- **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.*

A.2 Vector C Data Representations

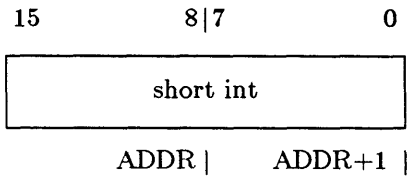
The following data representations are internal representations of the data types supported by Vector C.

A.2.1 Short Integer Data Representation

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

Figure A-1 illustrates the internal representation of the *short int* data type. Note that 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.

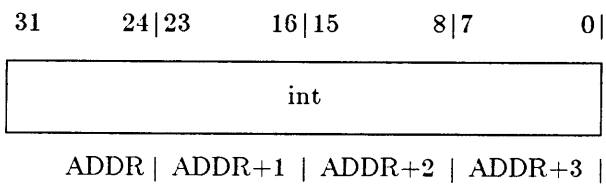
Figure A–1: Short Integer



A.2.2 Integer Data Representation

A 32-bit integer variable is declared as *int* (or *long int*) and may be 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 A-2 shows the *int* data representation.

Figure A–2: Integer

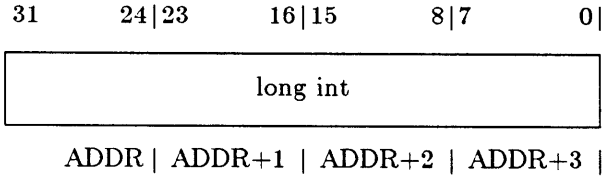


A.2.3 Long Integer Data Representation

In Vector C, 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 A-3.

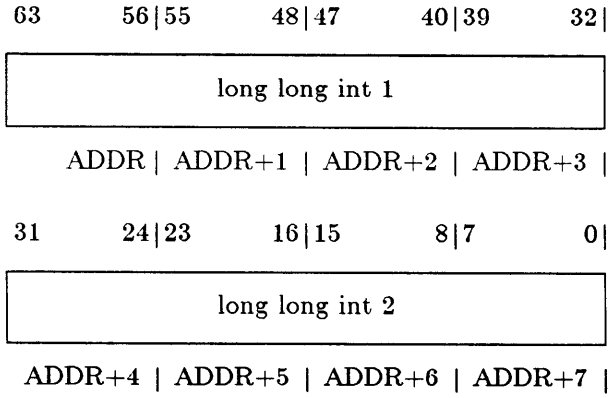
Figure A–3: Long Integer



A.2.4 Long Long Integer Data Representation

In Vector C, 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 A-4 shows the *long long* data representation.

Figure A–4: Long Long Integer



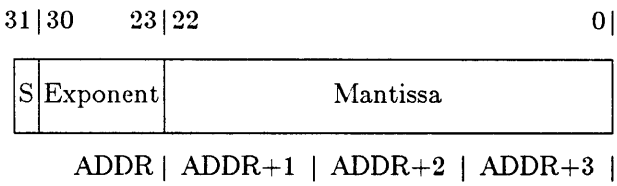
Although a *long long* integer may normally be used wherever a *long* integer may be used, you cannot pass *long long* integers as arguments to functions that do not expect to receive them. In general, *char* and *short* values can be passed to routines that expect *int* or *long* values. *vc* converts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

A.2.5 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 A-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 A–5: Single-Precision Floating



A.2.5.1 Single-Precision Native

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

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

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.

A.2.5.2 Single-Precision IEEE

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

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

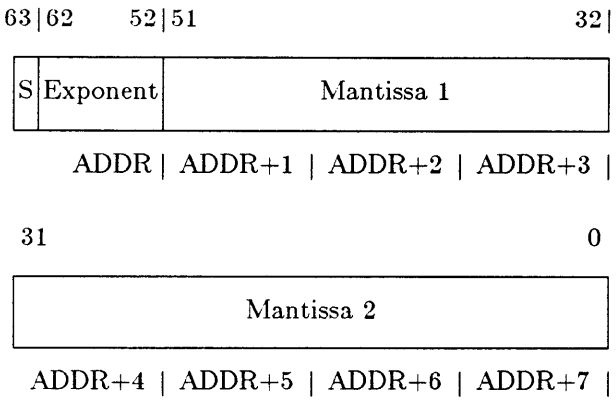
In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias 0f 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and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

A.2.6 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 A-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 A–6: Double-Precision Floating



A.2.6.1 Double-Precision Native

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

$5.562684646268003x10^{-309}$ to $8.988465674311584x10^{+307}$

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.

A.2.6.2 Double-Precision IEEE

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

$2.225073858507201x10^{-308}$ to $1.797693134862317x10^{+308}$

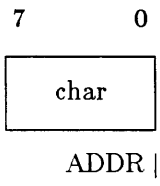
In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and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

A.2.7 Character Data Representation

Character data is stored in 8-bit bytes. Each byte can contain one of the ASCII character codes. In Vector C, *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 character variables may range in value from 0 to +255 (0 to $+2^8$-1). Signed character variables may range in value from -128 to +127 (-2^7 to $+2^7$-1). Figure A-7 shows the representation of this data type.

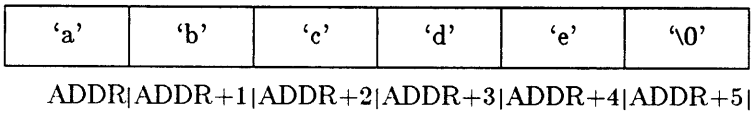
Figure A–7: Character



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 preceded by a 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 by a null (0) byte. Thus, the character string "abcde" would appear in memory with the configuration shown in Figure A-8.

Figure A–8: Character String



A.2.8 Enumerated Data Representation

In Vector C, an enumerated data type is a user-defined data type that has a finite number of possible values. You specify each of the 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 instant.

Internally, *enum* values are stored as integer representations. By default, the first enumerated value (red in the above example) is represented 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, for example:

```
enum color {red=10, blue=20, green=30};
```

Overriding default ordinal values affects the values of the following members. In the following example, green assumes a value of 21:

```
enum color {red, blue, yellow=20, green};
```

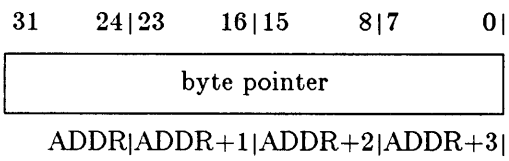
A.2.9 Pointer Data Representation

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

```
char *cp
```

designates a pointer named *cp* that may be assigned the address of a *char* variable. All pointers defined in Vector C are byte-granular virtual addresses (that is, they 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. It is an error for a program to access the contents of a null (0) pointer. Figure A-9 is an example of pointer data representation.

Figure A–9: Pointer



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. *vc* attempts to keep addresses properly aligned.

A.2.10 Structure Data Representation

Structures are collections of data items that are related in some way. Structures are analogous to records in PASCAL, or to Level 1 data items in COBOL. Structures may contain bit fields whose length is less than or equal to 64 bits.

The alignment of fields within a structure depends on the data types of the fields. No field in a structure lies on an alignment boundary of that field type. That is, an *int* field does not cross a 32-bit aligned boundary. This does not imply that all *int* fields are aligned on 32-bit boundaries. For example, two 16-bit *int* fields fit in the same 32-bit package.

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

A.2.11 Union Data Representation

The union data type uses the syntax of structures to reserve space in memory for variables whose data types may vary. A union is aligned with its largest member. The most significant bit of all the union members are aligned.

A.2.12 Void Data Representation

Since the void data type allocates no storage, there are no alignment considerations related to this data type.

A.3 Storage Alignment Requirements

This section describes the requirements for alignment of instructions and data in memory. In general, if you use only high-level programming 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 the performance penalties that result from misaligned data in assembly language code.

A.3.1 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.

- 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.
- 32-bit data items are aligned when their addresses are multiples of 4 bytes.
- 64-bit data items are aligned when their addresses are multiples of 8 bytes.

Generally, the Vector C compiler and the UNIX runtime system can keep program variables aligned to take maximum advantage of the hardware.

Align data on runtime stacks 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. Note that the alignment of data within structures may cause "holes" in the structures.

A.3.2 Code Alignment

Code is properly aligned when:

- The starting addresses of the instructions lie on an even address boundaries, and
- The least significant bit of the program counter is 0.

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 by a branch instruction, the least-significant bit in the program counter is usually ignored. Executing instructions loaded on odd address boundaries produce unpredictable results. To avoid this occurrence in assembly language code, use the *.align* assembler directive to force the alignment of instructions if odd-length data is stored in the text segment of the program.

A.3.3 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.

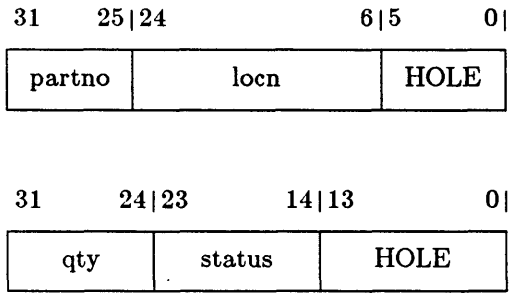
Bit fields should be no longer than permitted by the type declaration. Bit fields spanning a natural address alignment boundary are realigned to start at the next available alignment boundary. This realignment results in a "hole" between the bit fields in the least-significant bit positions.

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* appeared in memory immediately after *locn*, it will span a 32-bit boundary. Therefore 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 A-10 demonstrates that 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 A–10: Bit Field Alignment Example



B

Compiler and Runtime Messages

B.1 Introduction

This appendix describes the types of error messages that can occur when you compile a C program. First, there are messages generated by the Vector C preprocessor, *vcpp*. Next, the Vector C compiler, *cocc*, issues four kinds of diagnostic messages: error, warning, advisory, and vector summarization. All messages are output to *stderr*.

When the compiler has completed the syntactic and semantic analyses of a program, it aborts the compilation if user errors remain. This can also occur during optimization, e.g., integer truncation during constant folding.

The message may contain a parameter indicated by %s or %c. For example, in the *vcpp* error message *%s redefined*, the %s is filled in with the macro name.

B.2 vcpp Messages

The Vector C preprocessor, *vcpp*, generates the C-specific error messages and warning messages. All *vcpp* error and warning messages have the form:

file: line_number: error message

B.3 Compiler Messages

The compiler messages are error, warning, advisory, and vector summarization. You can redirect these messages to any specified file using the UNIX output redirection characters: >& *filename*. If you do not redirect the messages, they appear on your screen.

Example:

vc *options file.c*

sends the messages to the screen, while

vc *options file.c* **>&out**

sends the messages to the file *out*.

Another option available to you is the *error* utility. You can use *error* to insert diagnostic messages into your source file, where they appear as comments. This is a convenient way to find the bugs while editing your source file.

Example:

```
vc titan.c | & error
```

This command compiles *titan.c* and pipes the standard output and standard error output to the error utility, which then inserts the diagnostic messages back in the source file *titan.c*. You can write a simple *csh* script using the error utility to produce listings with embedded error messages that do not modify the source file itself.

An error message typically generates as:

Error on line 53 of filexxx: "Illegal pointer subtraction"

B.4 Runtime Error Messages

The runtime library reports errors encountered during execution. Runtime errors can be system-detected or mathematical. The runtime library provides default error processing and generates the necessary error messages.

All error messages are written *stderr*. Please consult the *CONVEX UNIX Programmer's Manual*, introduction to Section 2 for detailed information on UNIX-generated system calls and error numbers.

B.4.1 System Errors

System errors are returned by either the C I/O library or the C utility library. If it is the C I/O library, the system errors appear as I/O error messages. If the error is returned by the C utility library, the error number is returned as the value of the utility function (see Section 3 of the *CONVEX UNIX Programmer's Manual.*)

Table B-1 lists system errors generated by the UNIX operating system. Table B-2 lists the math errors.

Table B-1: System Errors Generated by UNIX

---

1 Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super user. It is also returned for attempts by ordinary users to do things allowed only to the super user.

2 No such file or directory

This error occurs when a file name is specified and the file should exist but does not, or when one of the directories in a path name does not exist.

3 No such process

The process whose number was given to *kill* and *ptrace* does not exist, or is already dead.

4 Interrupted system call

An asynchronous signal (such as interrupt or quit), that the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it appears as if the interrupted system call returned this error condition.

5 I/O error

Some physical I/O error occurred during a *read* or *write*. This error may occur on a call following the one to which it actually applies.

6 No such device or address

I/O on a special file refers to a subdevice that does not exist, or is beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected or a disk pack is not loaded on a drive.

7 Arg list too long

An argument list longer than 10240 bytes is presented to *execve*.

8 Exec format error

A request is made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number, see *a.out*(5).

9 Bad file number

Either a file descriptor refers to no open file or a read (or write) request is made to a file that is open only for writing (or reading).

10 No children

Wait and the process has no living or unwaited-for children.

11 No more processes

In a *fork*, the system's process table is full or you are not allowed to create any more processes.

12 Insufficient free swap space

During an *execve* or *break*, a program asks for more core or swap space than the system is able to supply. A lack of swap space is normally a temporary condition but a lack of core is not. The maximum size of the text, data, and stack segments is a system parameter.

13 Permission denied

An attempt was made to access a file in a way forbidden by the protection system.

14 Bad address

The system encountered a hardware fault in attempting to access the arguments of a system call.

15 Block device required

A plain file was mentioned where a block device was required.

16 Mount device busy

An attempt was made to mount a device that was already mounted or to dismount a device on which there is an active file directory (open file, current directory, mounted-on file, active text segment).

17 File exists

An existing file was mentioned in an inappropriate context.

18 Cross-device link

A hard link to a file on another device was attempted.

19 No such device

An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 Not a directory

A nondirectory was specified where a directory is required, for example in a path name or as an argument to *chdir*.

21 Is a directory

An attempt was made to write on a directory.

22 Invalid argument

Some invalid argument: dismounting a nonmounted device, mentioning an unknown signal in *signal*, reading or writing a file for which *seek* has generated a negative pointer. Also set by math functions.

23 File table overflow

The system table of open files is full, and temporarily no more *opens* can be accepted.

24 Too many open files

Customary configuration limit is 20 per process.

25 Not a typewriter

The file mentioned in an *ioctl* is not a terminal or another device to which these calls apply.

26 Text file busy

An attempt was made to execute a pure-procedure program that is currently open for writing (or reading). Also an attempt was made to open for writing a pure-procedure

program that is being executed.

27 File too large

The size of a file exceeded the maximum allowed (about 10**9 bytes).

28 No space left on device

During a *write* to an ordinary file, there is no free space left on the device.

29 Illegal seek

An *lseek* was issued to a pipe. This error may also be issued for other nonseekable devices.

30 Read-only file system

An attempt to modify a file or directory was made on a device-mounted read-only.

31 Too many links

An attempt was made to make more than 32767 hard links to a file.

32 Broken pipe

A write was made on a pipe or socket for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 Argument too large

The argument of a function in the math package (3M) is out of the domain of the function.

34 Result too large

The value of a function in the math package (3M) cannot be represented within machine precision.

35 Operation would block

An operation that would cause a process to block was attempted on an object in nonblocking mode. See *ioctl* (2).

36 Operation now in progress

An operation that takes a long time to complete (such as a *connect*) was attempted on a nonblocking object. For more information, see *ioctl*(2).

37 Operation already in progress

An operation was attempted on a nonblocking object that already had an operation in progress.

38 Socket operation on non-socket

Self-explanatory.

39 Destination address required

A required address was omitted from an operation on a socket.

40 Message too long

A message sent on a socket was larger than the internal message buffer.

41 Protocol wrong type for socket

A protocol was specified thats does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 Protocol not available

A bad option was specified in a *getsockopt* (2) or *setsockopt* (2) call.

43 Protocol not supported

The protocol has not been configured into the system or no implementation for it exists.

44 Socket type not supported

The support for the socket type has not been configured into the system or no implementation for it exists.

45 Operation not supported on socket

For example, trying to accept a connection on a datagram socket.

46 Protocol family not supported

The protocol family has not been configured into the system or no implementation for it exists.

47 Address family not supported by protocol family

An address incompatible with the requested protocol was used. For example, you would not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

48 Address already in use

Only one usage of each address is normally permitted.

49 Can't assign requested address

Normally results from an attempt to create a socket with an address not on this machine.

50 Network is down

A socket operation encountered a dead network.

51 Network is unreachable

A socket operation was attempted to an unreachable network.

52 Network dropped connection on reset

The host you were connected to crashed and rebooted.

53 Software caused connection abort

A connection abort was caused internal to your host machine.

54 Connection reset by peer

A connection was forcibly closed by a peer. This normally results from the peer executing a *shutdown*(2) call.

55 No buffer space available

An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.

56 Socket is already connected

A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

57 Socket is not connected

A request to send or receive data was disallowed because the socket is not connected.

58 Can't send after socket shutdown

A request to send data was disallowed because the socket had already been shut down with a previous *shutdown* call.

59 Too many references: can't splice

Currently unused.

60 Connection timed out

A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period depends on the communication protocol.)

61 Connection refused

No connection was made because the target machine actively refused it. This usually results from trying to connect to an inactive service on the foreign host.

62 Too many levels of symbolic links

A path name lookup involved more than 8 symbolic links.

63 File name too long

A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

64 Host is down

Self-explanatory.

65 Host is unreachable

Self-explanatory.

66 Directory not empty

A directory with entries other than . and .. was supplied to a remove directory or rename call.

Table B-2: Math Error Messages

300	square root undefined for negative values
301	exponential overflowed
302	logarithm undefined for nonpositive values
303	power undefined with negative base
304	power undefined with zero base and nonpositive exponent
305	power overflowed
306	argument for sin is too large
307	argument for cosine is too large
308	argument for tangent is too large
309	argument for cotangent is too large
310	tangent overflowed
311	cotangent overflowed
312	argument is out of range for arc sin
313	argument is out of range for arc cosine
314	arc tangent of 0.0/0.0 is undefined
315	hyperbolic sin overflowed
316	hyperbolic cosine overflowed
317	complex log undefined for 0.0
318	complex divide undefined for divisor = 0.0

C

Preprocessor Statements

The compiler contains a preprocessor that allows macro substitution, inclusion of file names, and conditional compilation. Preprocessor statements begin with the # symbol and are syntax-independent of the compiler. You can extend long statements over more than one line by entering a backslash (\ ) at the end of the line to be continued.

The preprocessor statements are described in the following paragraphs.

C.1 #define Statement

The #define statement causes the preprocessor to replace subsequent instances of an identifier with a given string of tokens. The format of this statement is as follows:

#define *identifier token-string*

or

#define *identifier(identifier,..., identifier) token-string*

The token string in the definition replaces the identifier. The arguments in the call in the second form are token strings separated by commas. Note that commas within quoted strings or protected by parentheses do not separate arguments. The corresponding token string from the call replaces every identifier mentioned in the formal parameter list of the definition, and the number of formal and actual parameters must be the same. Text inside a string or a character is not replaced.

Blanks are significant in *#define* statements. For example, *#define x(y,z) y+z* is not the same as *#define x (y,z) y+z*. The argument list must immediately follow the macro name.

C.2 #undef Statement

The #undef statement causes the identifier preprocessor definition to be deleted. The format of this statement is as follows:

#undef *identifier*

C.3 #include Statement

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

#include "*filename*"

or

#include <*filename*>

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

C.4 #if Statement

The #if statement checks whether a C-style constant expression evaluates to nonzero (#if), whether the identifier is currently defined in the preprocessor (#ifdef), or whether the identifier is currently undefined in the preprocessor (#ifndef). The format of this statement is as follows:

#if *constant-expression*

or

#ifdef *identifier*

or

#ifndef *identifier*

These forms are followed by an arbitrary number of lines, which may contain a control line *#else*. The last line must be *#endif*. If the specified condition is true, the lines between the *#else* and the *#endif* are ignored. If the checked condition is false, the lines between the *#if* and an *#else* (or the *#endif*, if no *#else* exists) are ignored. These constructs may be nested.

C.5 #line Statement

The #line statement tells the compiler that the number of the next source line is given by the constant and the current input file is named by the identifier. If no identifier is specified, the current file name does not change. The information resulting from the *#line* command provides more informative error messages for use in diagnostics. The format of this statement is as follows:

#line *constant identifier*

D

Runtime Libraries

The runtime libraries and initialization routines are shown in Table D-1. The table includes the purpose of each library or routine, and the directory in which each resides.

Table D-1: Runtime Libraries

Library/ Init. Routine	Location	Purpose
libc.a	/lib	Contains standard C library functions, *stdio* functions, network functions, interfaces to system calls, and programmed operators (e.g., udiv64).
libm.a	/usr/lib	Contains C interface to math functions.
crt0.o	/lib	C program initialization routine (no profiling).
mcrt0.o	/usr/lib	C program initialization routine (used with *prof*).
gcrt0.o	/usr/lib	C program initialization routine (used with *gprof*).
bcrt0.o	/usr/lib	C program initialization routine (used with *bprof*).

libc.a is loaded automatically by *vc*. *libm.a* is loaded when you include the *-lm* option on the *vc* command line.

The initialization routines are loaded automatically by *vc* unless you choose to load your program manually. (The *CONVEX Loader User's Guide* describes the manual loading process.) If you specify profiling with *prof* using the *-p* option on the *vc* command line, mcrt0.o is used. If you specify profiling with *gprof* (*-pg* option), gcrt0.o is used. Programs profiled with *bprof* (*-pb* option) use the bcrt0.o routine. crt0.o is used if you do not specify profiling.

E

Compiler Directives

E.1 Introduction

A compiler directive provides information that the compiler cannot deduce and instructs the compiler to override conditions that inhibit optimization or vectorization. A compiler directive has the form

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

where

/*$dir Indicates that the comment is a compiler directive. The characters $dir must be the first characters in the comment notation.

directive Is a compiler directive. Do not include other comments in the same comment delimiters with a directive. Use a separate set of comment delimiters (/* */) if you want to put comments near the directive.

The scope of a compiler directive is the program unit in which it appears. For directives having to do with vectorization, the scope is the loop that immediately follows the directive in the program text; it does not apply to loops nested within that loop. Since the compiler ignores comments, you may surround directive lines by any number of comment lines.

E.2 no_side_effects Directive

This directive has the following format:

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

where *func* is a user-defined function.

This directive can be used to increase the amount of optimization in a function. This directive is only legal within a function, and applies from the point it appears in the text to the end of the function. Use the directive if the compiler gives the advisory message "More optimization is possible if this function call has no side effects."

The "no_side_effects" directive instructs the compiler that the named functions have no side effects, i.e., they do not modify the value of a parameter or external variable, or perform a read or write. This permits scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that does not reach a use. The function call can be removed since the function has no side effects—it does not matter whether the call is made. Such optimization opportunities usually arise after other optimizations are performed and rarely occur in the original source text.

Example 1:

```
/*$dir no_side_effects (F1,F2)*/
 ...
            x = y * F1(5,z) - w;
 ...

/*if the x= does not reach a use of x, the assignment*/
/*statement may be removed*/
```

A function call with no side effects is invariant with respect to a loop if its arguments are loop invariant. The call may then be moved out of the loop.

Example 2:

A function call may inhibit code motion. Here, the directive is not applicable, and you must perform the optimization at the source level. (The source would have to be modified anyway to add the directive.)

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

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

   z = f(3);
}
```

Equivalent code is:

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

Code motion moves the z=.

E.3 scalar Directive

This directive has the following format:

scalar

When placed just before a loop, this directive prevents that loop from being vectorized. The body of the loop may still be vectorized if an outer loop interchanges with the scalar loop.

The scalar directive is useful in preventing vectorization when the iteration count for the loop is too low to compensate for the vectorization overhead or when the numerical results must be exactly the same as for a scalar loop.

The results of a vectorized loop can differ from its scalar equivalent. For example, floating-point sum and product reduction operators may give different answers because of rounding off in the least-significant bits. That is, the result depends on the order in which the operands are processed.

The scalar directive can also prevent loop interchange, which may make incorrect decisions when it cannot deduce the iteration counts of the loops involved.

Example 1:

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

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

Example 2:

Here, neither iteration count is enough to warrant vectorizing the loops:

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

E.4 no_recurrence Directive

This directive has the following format:

no_recurrence

Place this directive just before a loop if the loop was not vectorized because of an apparent recurrence but each statement could have been vectorized in the order specified in the original source program.

Use caution when using this directive. You may get incorrect results if you mistake a real recurrence for an apparent one. Test vector results versus scalar results if you are unsure whether a recurrence is real or apparent.

This directive applies only to the loop immediately following it. It does not affect recurrences caused by a contained loop. You could use the directive on each loop of a nest to give the vectorizer maximum opportunity for improving the performance of the nest.

Example:

Here, if j is positive, there is no recurrence:

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

F

Reporting Problems

F.1 Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

You must know the name and version number of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command. The syntax for the command is

vers *filename*

where *filename* is the the full pathname of the program. If you don't know the full pathname of the program, type

which *program*

For more information on these commands, see *vers*(1) and *which*(1) in the *CONVEX UNIX Programmer's Manual*, Part I.

F.2 Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.
2. The name and version of the product involved. Use the *vers* command if you don't know the version number of the program or utility.
3. A short (1 line) summary of the problem.
4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.
5. The priority of the problem. You are shown a list of six levels from which to select.
6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else you attempted to make your program run.

7. Any other comments about the problem or files you wish to submit.

You will have a chance to review your report before you submit it. You can edit the report if you find an error in what you have typed. If you change your mind and don't want to submit the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report*.

The following figure is a sample *contact* session. User input is in bold lettering, and the system response is in constant-width lettering.

Figure F–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University r
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

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.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

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.
> 3 (RETURN)

Problem report submitted.
%
```

Index

Software Documentation

Index Enhancements

So that we can continue to provide better indexing in CONVEX documentation, please keep track of the words or phrases you look up in an index, but don't find. Then, list under which index entry you ultimately found the information you were seeking. You can mail one of these postage-paid forms to the CONVEX Software Documentation Department monthly, or you can submit the information to the Technical Assistance Center in the form of a bug report. You can get more forms by writing to CONVEX at the address below, or by calling us. You can also photocopy this form and mail it back in an envelope. Thank you for helping us to serve you better.

Name:_____ **Company:**_____

Phone:_____ **Date:**_____

Manual Title/Rev. No.	Looked Up This Word	Found Information Under This Word

CONVEX Computer Corporation — P.O. Box 833851, Richardson, TX 75083-3851 (214) 952-0200

(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)

CONVEX Vector C Compiler User's Guide
Document No. 720-000630-200, V2.0

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and will help us continue in our efforts to generate quality customer documentation. (Please list the page number for questions and comments):

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Location	Phone Number
Continental USA	(214) 952-0200
Europe	011-44-483-69000

Direct Mail Orders to:

CONVEX Computer Corporation
Customer Service
Educational Department
P.O. Box 833851
Richardson, Texas 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)