CONVEX FORTRAN User's Guide

Document No. 720-000030-203

Eighth Edition
October 1988

Sharon Lammers

CONVEX Computer Corporation
Richardson, Texas

1988: First release of the manual.

Table of Contents

Appendices

List of Tables

List of Figures

Preface

This guide tells you how to use the CONVEX FORTRAN compiler. Subjects discussed include compiling, loading, and executing programs. Other pertinent information includes input/output operations, error processing, program optimization, utility libraries, and debugging.

It is assumed throughout that you are an experienced FORTRAN programmer. For further discussion of the CONVEX FORTRAN language and other CONVEX software, please consult the bibliography at the end of this Preface.

If you are unfamiliar with the CONVEX UNIX operating system, also consult the bibliography. Although a detailed knowledge of the operating system is not necessary to an understanding of this document, some familiarity with the system is beneficial.

Organization

This manual is organized into the following chapters and appendixes:

- Chapter 1 contains an overview of the CONVEX FORTRAN compiler and describes how to compile, load, and execute a program.
- Chapter 2 describes how to use the CONVEX input/output facilities. It describes the I/O statements and their parameters, file specifications, record structures, and record access.
- Chapter 3 discusses how to use character data: building character substrings, using character constants, declaring character data, initializing character variables, and using character library functions.
- Chapter 4 describes vectorization, parallelization, global optimization, local optimization, inline substitution, and machine dependent optimization in CONVEX FORTRAN.
- Chapter 5 describes the CONVEX FORTRAN calling conventions and describes how to call routines written in languages other than FORTRAN.
- Chapter 6 describes the use of CONVEX UNIX system services.
- Chapter 7 presents an overview of the debugging tools available for use with CONVEX FORTRAN.
- Chapter 8 discusses runtime error processing and describes how the runtime library processes errors, what the defaults are, and how to override the defaults.
- Appendix A describes the data types supported by CONVEX FORTRAN and shows their internal representations.

- Appendix B lists the error and advisory messages that can occur during compilation or runtime.
- Appendix C lists and describes the runtime library and routines.
- Appendix D tells you how to report software and documentation problems.

An index and reader reply forms are included at the back of the guide.

Notational Conventions

The following conventions are used in this document:

- Brackets ([ ]) designate optional entries.
- A caret (^) is used to represent the space character.
- Horizontal ellipsis (. . .) shows repetition of the preceding item(s). In an example, horizontal ellipsis indicates that statements are omitted.
- Vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- References to the *CONVEX UNIX Programmer's Manual* appear in the form *fc*(1F), where the name of the manual page is followed by its section number enclosed in parentheses.
- *Italics* within text denote commands, options, filenames, or programs.
- Within command sequences set apart from text, **boldface** type indicates literals. Words appearing in boldface should be typed as they appear. *Italics* within command sequences indicate generic information, such as options or filenames. Substitute actual information for the italicized words. For example, the command sequence

  ld [*options*] [*object files*] [*libraries*]

  instructs you to type the command *ld*, followed by your choice of options, object files, and/or libraries.

Associated Documents

The following documents, available from CONVEX Computer Corporation, are recommended to the CONVEX FORTRAN programmer:

- *CONVEX FORTRAN Language Reference Manual* describes the FORTRAN language and the CONVEX extensions to the language.
- *CONVEX UNIX Primer* contains basic self-instruction for learning and using the CONVEX UNIX operating system.
- *CONVEX UNIX Programmer's Manual,* Parts I and II, contains complete reference material on the UNIX operating system for the CONVEX family of supercomputers.
- *CONVEX adb Debugger User's Guide,* a tutorial and reference manual, describes the functions and operations of the CONVEX *adb* debugger.
- *CONVEX Consultant User's Guide* (optional product) describes the functions and

operations of the CONVEX *csd* debugger, post-mortem dump (*pmd*) utility, and the *gprof* and *prof* profilers.

- *CONVEX Loader User's Guide*, a tutorial and reference manual, describes the CONVEX loader.
- *CONVEX COVUEshell Reference Manual* describes COVUEshell. COVUEshell is an optional CONVEX product that provides a VMS-type interface, giving the user access to a subset of Digital Command Language (DCL) commands.
- *American National Standard Programming Language, FORTRAN* manual (ANSI X3.9-1978) defines the standard language.

Section 3F of the *CONVEX UNIX Programmer's Manual* contains the FORTRAN runtime library functions.

Chapter 1
Compiling Programs

This chapter presents an overview of the CONVEX FORTRAN compiler and explains how to compile, load, and execute programs written in CONVEX FORTRAN.

CONVEX FORTRAN is a high-level programming language that contains standard FORTRAN as defined by the American National Standard FORTRAN-77 (ANSI X3.9-1978), VAX-11 functions, and unique CONVEX extensions. A complete description of the CONVEX FORTRAN language is contained in the *CONVEX FORTRAN Language Reference Manual.*

1.1 Overview

The CONVEX FORTRAN compiler translates a source file containing one or more FORTRAN program units into an object module. The object module can then be linked with library routines or other object modules for execution on a CONVEX computer. Previously compiled programs written in CONVEX assembly language, C, or Ada can interface with CONVEX FORTRAN object code to produce an executable program.

The CONVEX FORTRAN compiler automatically generates code that takes full advantage of the architecture of the CONVEX family of supercomputers. In addition, you can specify that the compiler perform optimization, vectorization, and parallelization of your source code for maximum efficiency of execution.

Compiler options let you request Cray FORTRAN, VAX FORTRAN, or Sun FORTRAN compatibility, IEEE-standard representation of floating-point values, and inline substitution of subroutines. IEEE functionality requires that the target machine be equipped with the IEEE support hardware.

To enhance compilation speed, the CONVEX FORTRAN compiler generates object code directly. At your option, however, the compiler can produce assembly-language code, which may be inspected, modified, or assembled directly.

To assist in program checkout, the compiler interfaces with several debuggers and utility routines, including a post-mortem dump utility, a source level debugger, and an assembly-level debugger.

You may also compile CONVEX FORTRAN programs under COVUEshell. COVUEshell is a CONVEX product that provides a VMS-like interface and supports many of the Digital Command Language (DCL) commands. For further information, please refer to the *CONVEX COVUEshell Reference Manual.*

1.2 File-Naming Conventions

The compiler distinguishes a file type by the extension added to the end of the filename. A FORTRAN source file is identified either by the extension .f or by the extension .FOR. The compiled object file has the same name as the source file except that it ends with .o.

When you compile and load a single source file during one invocation of the compiler, the object file is normally deleted. Unless you specify otherwise on the compiler command line, the executable module produced by the loader is placed into the file *a.out*.

Although the compiler normally produces object code directly, you may request that symbolic assembly code be generated. In this case, the name of the file that is produced is the same as that of the source file except that it ends with .s.

The following table summarizes the file naming conventions used by CONVEX.

Filenames for...	End with the extension...
FORTRAN source files	**.f** or **.FOR**
Object files	**.o**
Symbolic assembly-language files	**.s**
Libraries	**.a**
Inline intermediate files	**.fil**

1.3 Compiling Programs

To invoke the CONVEX FORTRAN compiler, use the following command line:

fc [*options*] *files* [*loader-options*]

In the command line, *options* is one or more of the compiler options described in the following sections. Any options contained in an OPTIONS statement within a program override those specified on the command line.

The parameter *files* represents one or more FORTRAN source files to be compiled, object files to be loaded, or symbolic assembly-language files to be assembled.

The parameter *loader-options* is one or more loader options as described in the *CONVEX Loader User's Guide*. If specified, these options are passed to the UNIX loader when compilation is complete.

Language-Compatibility Options

-cfc Causes the compiler to use the Cray FORTRAN language definition instead of the standard CONVEX FORTRAN definition. This option cannot be used with the *-i* or *-r* options.

-F66 Selects FORTRAN-66 language interpretation rules in cases of incompatibility.

-sa Prevents FORTRAN from generating pre-compiled argument packets in the text segment. All arguments are placed on the stack. This option should only be used when an application contains user-supplied C programs called from FORTRAN. Using it with applications coded only in FORTRAN slows down the application.

-sfc Causes the compiler to use the available subset of Sun f77 language features instead of the corresponding CONVEX FORTRAN features. The subset is described in Appendix I of the *FORTRAN Language Reference Manual*.

-vfc Causes the compiler to accept certain language extensions implemented in VAX FORTRAN instead of the corresponding CONVEX FORTRAN features.

Optimization Options

-ep *n* Specifies the expected number of processors (*n*) on which the program is going to run. If the value of *n* is not an integer from 1 to 4, the behavior of the compiler is indeterminate.

The compiler parallelizes a loop whenever doing so appears to decrease the turnaround time, assuming the given number of processors. Use this option with caution since it may lead to inefficient use of processors.

-il Instructs the compiler to prepare an intermediate language (*.fil*) file for a subprogram that is to be used for inline substitution. The *-il* option cannot be used with the *-c*, *-cs*, or *-S* options. Optimization levels are ignored.

-is *directory* Instructs the compiler to perform inline substitution of each subprogram for which there exists a *.fil* file in the specified *directory*. This option must be repeated for each directory containing *.fil* files to be used for inline substitution.

-no Specifies that the compiler is to perform no optimization. This option is the default if the -O option is not specified.

-O*n* Performs machine-independent optimizations at the specified level. You can specify the following optimization levels:

Level	Description
-O0	Local scalar optimization
-O1	Local scalar optimization and global scalar optimization
-O2	Local scalar optimization, global scalar optimization, and vectorization
-O3	Local scalar optimization, global scalar optimization, vectorization, and parallelization

If this option is not used, the compiler performs no machine-independent optimization.

-rl Causes the compiler to perform loop-replication optimizations on loops selected by the compiler on the basis of profitability. The loop-replication options include loop unrolling and dynamic loop selection.

This option may not be used unless the -O2 option is also specified. When the -O2 option is specified, the compiler generates scalar and vector versions of eligible loops and selects the best version at runtime.

-uo Performs potentially unsafe optimizations, e.g., moves the evaluation of common subexpressions and/or invariant code from within conditionally executed code. Such moved code may be executed unconditionally.

Code-Generation Options

-c Suppresses the loading phase of the compilation. For example, output from the file *file.f* or *file.s* is written to *file.o*.

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

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

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

-i*n* Controls how the compiler interprets INTEGER and LOGICAL declarations with unspecified lengths. The default interpretation is INTEGER*4 and LOGICAL*4. The option changes the interpretation to INTEGER*2 and LOGICAL*2, or INTEGER*8 and LOGICAL*8. *n* may be 2, 4, or 8.

-r*n* Controls how the compiler interprets REAL declarations having unspecified lengths. The default interpretation is REAL*4. *n* may be 4 or 8.

-re Causes the compiler to generate reentrant code for parallel or recursive invocation of subprograms. This option makes it possible to call subroutines from inside parallel loops.

Each invocation of a subroutine has its own copy of local variables. Arguments are passed on the stack instead of by means of argument packets. Common variables and saved or initialized variables are still shared among invocations.

If you compile a program using the *-re* option, you must initialize all local variables.

-S Generates symbolic assembly code for each program unit in a source file. Assembler output for source *myfile.f* is written to *myfile.s*. The assembly file is not assembled to produce object code.

-tm *target* Specifies the target machine architecture for which compilation is to be performed. The value for *target* can either be c1 or c2 (C1 or C2 may also be used). If you specify a target machine, the instruction set for that machine is used regardless of the machine on which the compiler is running. If you do not specify a target machine, the compiler generates instructions for the class of machine on which it is running.

Debugging and Profiling Options

-a1 Causes noncharacter arrays declared with a last dimension of 1 to be treated as if they were declared assumed-size (last dimension of *). Subscript checking can then be performed if the *-cs* option is also specified. The *-a1* option can be used in the OPTIONS statement.

-cs Compiles code to check that each subscript is within its array bounds. Does not check the bounds for arrays that are dummy arguments for which the last dimension bound is specified as * or 1. The *-cs* option can be used in the OPTIONS statement.

-db Produces additional information for use by the symbolic debugger, *csd*, and passes the *-lg* option to the loader. This option can be used with all levels of optimization. If the -O option is specified, there may be source statements for which no debugging information is generated for *csd*.

-dc Specifies that a line with a D in column 1 is to be compiled and not treated as a comment line. Statements with a D in column 1 can be conditionally compiled, making this feature a useful debugging tool.

-p Produce code that counts the number of times each routine is called. If loading takes place, the standard startup routine is replaced with one that automatically calls *monitor* at the start and arranges to write out a *mon.out* file at normal termination of the object program.

Also, a profiling library is searched instead of the standard FORTRAN library. An execution profile can then be generated by use of *prof* (optional product).

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

-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* (optional product).

-sc Provides a syntax check. Stops compilation of each program unit in a source file after the program has been determined to be a valid FORTRAN program. Using this option during program development reduces compilation times.

Message and Listing Options

-na Suppresses all advisory diagnostic messages.

-nv This option is no longer available. Use *-or* instead.

-nw Suppresses all warning diagnostic messages.

-or *table* Specifies the contents of the optimization report to be produced; either the loop table, the array table, or both, can be displayed. The value for *table* can be *all, none, loop,* or *array.* If this option is not specified, only the loop table is displayed. Section 1.6.2 of this manual describes the optimization report.

-xr Calls the *fxref* cross-reference generator. The following options are related to this option:

Option	Description
-iw *n*	Specify the column width for identifiers. *n* can range from 8 to 32. The default is 16.
-pw *n*	Specify the logical page width used by the output formatter. The default is 132.
-sl	Produce a source listing with line numbers that precedes the cross-reference table.

-xr1 Calls the *fxref* cross-reference generator and puts all objects (such as variables and arrays) into one table, rather than printing a separate table for each class of objects.

Miscellaneous Options

-B*string* Finds the substitute compiler (*fskel* and *fpp*) in the directory named *string*. The default directory is */usr/convex/oldfc*, which contains the previous version of the compiler for use as a backup.

-o *name* Assigns *name* as the name of the executable file produced by the loader. The default name is *a.out*. If the loader is not invoked because the *-c* option is specified and if there is only one file to compile or assemble, then *name* becomes the name of the object module.

-tl *n* Sets the maximum CPU time limit for compilation to *n* minutes. If the time limit is exceeded, compilation terminates with the message *System error in /usr/convex/fskel.*

-vn Display information concerning the version of the compiler that is being used. Output goes to *stderr*.

-72 Causes the compiler to process only the first 72 characters of each program line. (The compiler normally processes all characters.) Continued Hollerith and character constants are not padded. A line with fewer than 72 characters ending with a Hollerith constant is padded with blanks until the constant is completed, or until 72 characters are processed for that line. A line with fewer than 72 characters ending with the first characters of a character constant is padded with blanks until 72 characters have been processed. A tab counts as one character.

1.4 Loading Programs

Two types of files, object files and libraries, are used as input to the UNIX loader (*ld*). Object files contain the binary output produced by the compiler. Library files contain frequently used object modules that are inserted into programs by the loader as necessary.

The preferred method of invoking the loader for FORTRAN programs is to use the *fc* command. This approach ensures that the proper FORTRAN libraries are loaded in the proper order. The compiler, in turn, passes any loader options on the *fc* command line to the loader. You can suppress the loading phase of the compilation with the *-c* option. For information on invoking the loader directly, see the *CONVEX Loader User's Guide*.

Any libraries specified on the *fc* line with the *-l* loader option are searched before the standard libraries. Table 1-1 lists the names and contents of the standard FORTRAN runtime libraries.

Table 1-1: FORTRAN Runtime Libraries

Name	Contents
/usr/lib/libF77.a	Intrinsic function library
/usr/lib/libF77_p.a	Profiled intrinsic function library
/usr/lib/libI77.a	FORTRAN I/O library
/usr/lib/libI77_p.a	Profiled FORTRAN I/O library
/usr/lib/libU77.a	UNIX interface library
/usr/lib/libu77_p.a	Profiled UNIX interface library
/usr/lib/libI66.a	FORTRAN-66 I/O initialization
/usr/lib/libV77.a	Constants for VAX FORTRAN compatibility
/usr/lib/libvfn.a	VMS-to UNIX filename translation routines
/usr/lib/libD77.a	Dummy VMS-to-UNIX filename translation routines
/lib/libc.a	C library (system utilities)
/usr/lib/libc_p.a	Profiled C library
/usr/lib/libm.a	Math library
/usr/lib/libm_p.a	Profiled math library
/usr/lib/libmathC1.a	Math library optimized for C1 architecture
/usr/lib/libmathC1_p.a	Profiled math library optimized for C1 architecture
/usr/lib/libmathC2.a	Math library optimized for C200 architecture
/usr/lib/libmathC2_p.a	Profiled math library optimized for C200 architecture.

The */usr/lib/libI66.a* library is required only for FORTRAN-66 compatibility. This library is included automatically if the -F66 option is specified on the compiler command line. Appendix C contains detailed information on the FORTRAN intrinsic library.

1.5 Executing Programs

To execute your program after it has been processed by the loader, type the name of the executable file. The default name for the executable file is *a.out*. If you have included the *-o name* option on the *fc* command line, the name of the executable file is *name*.

1.6 Messages

This section presents an overview of the messages that can result during compilation and at runtime. The messages are grouped into the following categories:

- Compiler messages
- Optimization report
- Runtime messages

1.6.1 Compiler Messages

The compiler produces various messages during compilation—error messages, warnings, advisories, vectorization, and parallelization messages. Messages produced by the compiler are directed to the standard error file (*stderr*). A compiler message consists of the line number of the text in which the error occurs, the pathname of the source file containing the line of text in error, and a brief description of the error.

The following examples show typical compiler messages.

```
fc: Error on line 7.1 of testprog.f:  Label defined but never referenced.
fc: Warning on line 3.4 of myprog.f:  Divide by zero may occur at runtime.
```

If an internal compiler occurs, the compiler outputs a message that begins with the words "COMPILER ERROR." Such a message should be reported to the CONVEX Technical Assistance Center (TAC).

1.6.2 Optimization Report

If a program is compiled with the -O2 or -O3 option, the compiler generates an optimization report for each program unit. This report consists of a loop table, an array table, or both. You can specify which tables are to be included in the optimization report by means of the *-or* compiler option.

1.6.2.1 Loop Table, Part 1

The loop table lists the optimizations that were performed on each loop and, if appropriate, the reasons while a possible optimization was not performed. The loop table can consist of one or two parts. Part 1 of the loop table is always printed and contains the following information:

- **Line Num.**

  Specifies the source line of the beginning of the loop. If the line number has two parts, separated by a hyphen, the second number is the distributend number (due to loop distribution).

- **Iter. Var.**

  Specifies the name of the iteration variable controlling the loop or *NONE*. If the iteration variable has two parts, separated by a colon, the second part is the inline substitution instantation of that variable.

- **Reordering Transformation**

  Indicates which reordering transformations were performed. A reordering transformation does not eliminate operations from a program or replace them by simpler operations, but rearranges them so they can be more efficiently executed. This column contains one of the following values:

Value	Explanation
Scalar	No transformation of this type was performed.
nn% VECTOR	The loop was partially vectorized, with the percentage (*nn*) specified being executed in vector mode.
FULL VECTOR	The loop was fully vectorized, with all operations being executed in vector mode.
PARALLEL	The loop runs in parallel mode.
PARA/VECTOR	The loop was vectorized, and the strip mine loop runs in parallel mode.
Dist	Loop distribution was performed.
Inter	Loop interchange was performed.

- **Optimizing/Special Transformation**

  Indicates which optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with simpler operations. A special transformation allows the compiler to vectorize or parallelize code under special circumstances. This column contains one of the following values:

Value	Explanation
Unroll	The loop was completely or partially unrolled.
Reduction	The compiler recognized a reduction and vectorized the loop.
Pattern	The compiler recognized a special pattern and vectorized the loop.
Synch	The compiler inserted synchronization code to ensure correct execution of a parallel loop.

- **Mode**

  If used, this column refers to multiple execution modes controlled by dynamic selection. This column can contain the following values:

Value	Explanation
S	Specifies scalar execution.
V	Specifies vector execution.
P	Specifies parallel execution.
Z	Specifies parallel-outer execution, vector-inner execution.

1.6.2.2 Loop Table, Part 2

Part 2 of the loop table is only printed if there is relevant information to be shown. Part 2 of the loop table contains the following information:

- **Line Num.**

  Source line of the beginning of the loop

- **Iter. Var.**

  Name of the iteration variable controlling the loop or *NONE*.

- **Analysis**

  Why a transformation or optimization was not performed, or additional information on what was done.

1.6.2.3 Array Table

The array table lists array references that prevented optimization or on which special optimizations were performed. The array table contains the following information:

- **Line Num.**

  Source line on which the reference occurs.

- **Var. Name**

  Name of the array being referenced

- **Optimization**

  Contains one of the following values:

Value	Explanation
Hoist	The vector load was found to be loop invariant and was moved outside the loop.
Sink	The vector store was found to be loop invariant and was moved outside the loop.

- **Dependencies**

  Shows the names of other variables in a recurrence, in the form *name@ linenumber*. If the reference could be to any memory location, it is in the form **MEM*@ linenumber*.

1.6.3 Runtime Error Messages

Runtime error messages are directed to the standard error file (*stderr*). Error messages can be generated by math routines, I/O operations, or trap errors. I/O error messages can contain up to four lines of information depending on the type of I/O operation involved.

The following examples show typical runtime error messages. The numbers in brackets indicate an error number associated with a particular error condition.

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

mth$r_sqrt: [300] square root undefined for negative values
sqrt( -1.7014117E+38)= 1.3043818E+19

write sfe: [100] error in format
logical unit 6, named 'stdout'
lately: writing sequential formatted external IO
part of last runtime format: (f6.2,x,v5|,x,i

dofio: [115] read unexpected character
logical unit 7, named 'fort.7'
lately: reading sequential formatted external IO
part of last pre-compiled format: (14,F7.2,E10.4|,E10
```

Some runtime errors also produce a stack trace.

1.7 Program Interfaces

The calling sequences for the runtime system include the use of the short-form call instruction (*callq*) and additional conventions related to the use of registers and the values in registers after calls. The compiler provides versions of the intrinsics that use the standard calling sequence. This feature allows you to pass intrinsic names as arguments to subprograms.

NOTE

There is a performance penalty for invoking intrinsics passed as arguments.

The runtime system provides scalar and vector versions of the math intrinsics. The compiler determines which version of the routine to call.

If you code part of your program in a language other than FORTRAN, such as C or assembly, you must use the same names that are output by FORTRAN or the program cannot link properly. The naming conventions are as follows:

main program	_MAIN__	(1 preceding underscore and 2 following)
blank common	___blnk_	(3 preceding underscores and 1 following)
named common	__name_	(2 preceding underscores and 1 following)
subprogram	_name_	(1 preceding underscore and 1 following)

where *blnk* and *name* represent the symbolic name.

To get names generated by FORTRAN, you must append and prefix the appropriate number of underscores. For example, since C prefixes a single underscore on function names and external variables, a call to a FORTRAN routine FFT must use FFT_.

Chapter 2
Input/Output Operations

This chapter describes the particulars of FORTRAN input/output operations as they are performed under CONVEX FORTRAN and presents specific information pertaining to UNIX files.

2.1 Units

The unit number in a CONVEX FORTRAN input/output statement can range from 0 through 255. A unit number can be specified either explicitly or implicitly. The following WRITE statement, for example, specifies an explicit unit of 9.

```
WRITE (9, 100) I, X, Y
```

In certain forms of the READ and WRITE statements and in statements such as ACCEPT, PRINT, or TYPE, the unit number is implicit. Table 2-1 shows the general forms of CONVEX FORTRAN I/O statements in which the unit is specified implicitly. In the table, *f* indicates the FORMAT statement number.

Table 2-1: Implicit Units

FORTRAN Statement	Implicit Unit
READ (*, *f*) *list*	5
READ *f*, *list*	5
ACCEPT *f*, *list*	5
WRITE (*, *f*) *list*	6
PRINT *f*, *list*	6
TYPE *f*, *list*	6

UNIX defines the terms "standard input" (*stdin*), "standard output" (*stdout*), and "standard error" (*stderr*). By default, a program looks at *stdin* for input, writes output to *stdout*, and sends error messages to *stderr*. All three of these designators are normally assigned to your terminal but can be redirected by UNIX commands or by the OPEN statement.

By default, CONVEX FORTRAN assigns unit 5 to *stdin*, unit 6 to *stdout*, and unit 0 to *stderr*. If you use an asterisk (*) in a READ or WRITE statement, unit 5 or 6 is always assigned, regardless of whether or not these units have ever been specified in a CLOSE or OPEN statement.

2.2 Logical Names

Every unit in CONVEX FORTRAN, except unit 0, is associated, by default, with a logical name of the form FOR*nnn*, where *nnn* is the unit number. This logical name is used to create a default UNIX filename in the form *fort.nnn* to which the unit is automatically "preconnected." Preconnected means that the file is connected to the unit when the program begins executing and can be referenced by input/output statements without prior execution of an OPEN statement.

The following statement opens and writes to file *fort.35*.

```
WRITE (35,10) data
```

Table 5-2 shows examples of units, the logical names associated with the units by default, and the UNIX filenames to which they are preconnected by default.

Table 2-2: Default Logical Names

Unit	Default Logical Name	Default UNIX File Name
8	FOR008	fort.8
52	FOR052	fort.52
230	FOR230	fort.230

Units 5 and 6 (*stdin* and *stdout*) are not automatically preconnected to files *fort.5* and *fort.6*. You must perform explicit CLOSE and OPEN statements to get this connection as shown in the following example. In the example, since no filename is specified in the OPEN statement, the default filename (*fort.5*) is assigned.

Example:

```
CLOSE (5)
OPEN (5)      ! Unit 5 (stdin) is now connected to fort.5
```

You can usually override the preconnection of a unit by means of an explicit OPEN statement or with environment variables as described later in this section. If, however, the logical name specified in an OPEN statement is of the form FOR*nnn*, the UNIX file actually generated has the corresponding *fort.nn* name.

Example:

The following statement generates a file named *fort.4*.

```
OPEN (UNIT=1, FILE='FOR004')
```

The files *stderr*, *stdin*, and *stdout* can also be redirected from the command line as described in the description of the C shell (*csh*) in the *CONVEX UNIX Programmer's Manual*.

2.3 The OPEN Statement

The OPEN statement connects a unit to a UNIX file so that any subsequent input/output operations to that unit access the specified file. An OPEN statement may contain a FILE= clause whose value can either be a logical name or the name of a UNIX file. The value specified in the FILE= clause overrides the default logical name or filename associated with the unit.

An OPEN statement without a FILE= clause opens the default file for that unit unless STATUS='SCRATCH' is specified. If STATUS='SCRATCH' is specified, a temporary file is generated with the name *tmp.Fcppppnnn*, where *c* is a special character, *ppppp* is the process ID and *nnn* is the unit number. By default, a temporary file is deleted when the program terminates.

Examples:

The following example opens a file named *fort.94* in the current home directory.

```
OPEN (94)
```

The following example opens a temporary file, *tmp.Fcppppnnn*, in the current working directory and deletes it when program execution is complete.

```
OPEN (24, STATUS= 'SCRATCH')
```

The following example opens the specified UNIX file.

```
OPEN (10, FILE='/usr/tmp/myfile')
```

2.4 Assigning Logical Names

You can customize your CONVEX UNIX working environment to assign FORTRAN logical names to UNIX files. You can also change the default name assigned to scratch files by changing the environment variable FORTEMP. For a discussion of the UNIX working environment, please refer to the *CONVEX UNIX Primer* or to *environ*(7).

In the most commonly-used UNIX command interpreter, the C shell (*csh*), the commands *setenv* and *unsetenv* control the setting and resetting of variables in your working environment. The format of these commands is

```
setenv    name   value
unsetenv  name
```

To examine the environment variables that are currently set, use the *csh* command *printenv*.

When a unit is opened, the logical name associated with the unit is compared to the environment variables. The logical name can be specified in the FILE= clause of an OPEN statement or can be the default logical name associated with the unit. If an environment variable matches the logical name, the value assigned to that variable is substituted for the logical name and the comparison process is repeated.

Examples:

The following UNIX command causes the compiler to generate scratch filenames in the current directory of the form MYFILE*cppppppnnn* instead of tmp.F*cppppppnnn*:

```
setenv FORTEMP MYFILE
```

The following UNIX command causes the compiler to generate scratch filenames in the directory */tmp* of the form TEMP*cppppppnnn* instead of tmp.F*cppppppnnn*:

```
setenv FORTEMP /tmp/TEMP
```

The following example causes data to be written to the file *output.dat* in the current working directory.

```
setenv  FOR021  ~/output.dat
        . . .
WRITE (21,*) A, B, C
```

The following example causes data to be written to the file */acct/smith/data*.

```
setenv  FOR055  OUTPUT
setenv  OUTPUT  /acct/smith/data
        . . .
WRITE (55,*)  BASELINE
```

The following example writes data onto *stdout*. You must use the backslash (\) as an escape character.

```
setenv  FOR021  SYS\$OUTPUT
          . . .
WRITE (21,10)  IOLIST
```

The following example causes data to be written to the file */acct/smith/printfile*.

```
setenv  PRINT  /acct/smith/printfile
          . . .
OPEN (21,FILE='PRINT')
WRITE (21,50) I, J, A
```

The following example causes data to be written to ./PRINT.

```
unsetenv  PRINT
            . . .
OPEN (21,FILE='PRINT')
WRITE (21,50) I, J, A
```

2.5 Forms of Input/Output

CONVEX FORTRAN supports formatted, list-directed, namelist-directed, and unformatted input/output (I/O). Formatted I/O statements have explicit format specifiers that control data translation from internal binary form within a program to external, readable-character form in the records, or vice versa.

Although similar to formatted statements in function, list-directed and namelist-directed I/O statements use data types rather than explicit format specifiers to control data translation from one form to another.

Unformatted (or binary) I/O statements do not translate the data being transferred and can be used when output data is later to be used as input. Unformatted I/O saves execution time; it eliminates the translation process, maintains greater precision in the external data, and conserves file storage space.

I/O statements transfer all data as records. How much data a record can hold depends on whether unformatted or formatted I/O is used for data transfer. With unformatted I/O, the I/O statement determines how much data is to be transferred. With formatted I/O, the I/O statement and its associated format specifier determine how much data is to be transferred.

Usually, data transferred by an I/O statement is read from or written to a single record. A formatted, list-directed, or namelist-directed I/O statement, however, can transfer more than one record.

2.6 File Type

There are two types of files: external and internal. An external file is associated with a disk file, terminal, or some other device. An internal file is associated with internal storage space and consists of a character variable, array element, array, or substring.

An internal file that contains a single character variable, array element, or substring consists of one record whose length is the same as that of the character variable, array element, or substring. An internal file that contains a character array consists of a sequence of records, each of which consists of a single array element. The order of subscript progression determines the record sequence in an internal file.

Before data is transferred, an internal file is always positioned at the beginning of the first record.

2.7 Access Modes

The method for retrieving and storing records in a file is the access mode, which is specified by each I/O statement. CONVEX FORTRAN supports sequential and direct access modes.

2.7.1 Sequential Access

Sequentially accessed records are written to or read from the file starting at the beginning and continuing through the file, one record after another. You can access a particular record only after all the records preceding it have been read and can write new records only at the end of the file.

Example:

```
READ (10,*) A, B
```

In this example, the READ statement causes the next two real values to be read into *A* and *B*.

2.7.2 Direct Access

This mode allows you to choose the order in which records are read and written. Each READ or WRITE statement must include the record number.

Examples:

```
WRITE (10,REC=28) I
WRITE (10,REC=15) J
```

The first statement writes the value I to record 28, and the second statement writes the value J to record 15.

2.8 Logical Records

The definition of a logical record depends on the combination of the I/O form and the mode specified by the FORTRAN I/O statement. Each execution of a FORTRAN unformatted I/O statement causes a single logical record to be read or written.

Each execution of a FORTRAN formatted I/O statement causes one or more logical records to be read or written.

2.8.1 Direct-Access External File

A logical record in a direct-access external file is a string of bytes, the length of which you specify when you open the file. READ and WRITE statements must not attempt to access more data than fits into one record. Shorter logical records are allowed. Unformatted direct WRITE statements leave the unfilled part of the record undefined. Formatted direct WRITE statements pad the unfilled record with blanks.

2.8.2 Sequential-Access External File

A logical record in a sequentially accessed external file may be of any length. The size of the items in the list of I/O values (the I/O list) determines the logical record length for unformatted sequential files. For formatted WRITE statements, the format statement interacting with the I/O list at execution time determines the logical record length. Formatted sequential access causes one or more logical records ending with the "newline" (hexadecimal 0A) character to be read or written.

2.8.3 Namelist-Directed Input/Output

Namelist-directed I/O statements are similar to list-directed statements in function. For variable-length files the namelist-directed statement uses data types instead of explicit format specifiers to control data translation and formatting. For fixed-record-length files, the namelist-directed statement reads and writes records of a fixed length.

2.8.4 Internal Files

The logical record length for an internal READ or WRITE is the length of the character variable or array element. Thus, a simple character variable is a single logical record.

2.9 Input/Output Statement Summary

Table 2-3 summarizes the input/output statements available in CONVEX FORTRAN. The *CONVEX FORTRAN Language Reference Manual* describes these statements in detail.

Table 2-3: Input/Output Statements

Type	Statement	Use
Input	READ	Transfers data from an external file into internal storage or between internal storage locations.
	ACCEPT	Sequentially reads data from the implicit input unit.
	DECODE	Transfers data between arrays or variables in internal storage and translates the data from character to internal form.
Output	WRITE	Transfers data from internal storage to an external device or between internal storage locations.
	PRINT	Transfers formatted records to the implicit output device.
	TYPE	Same as PRINT.
	ENCODE	Transfers data between arrays or variables in internal storage and translates the data from internal to character form.
Auxiliary	OPEN	Connects an existing external file to the specified unit, changes the attributes of a connected file, or creates a new file and connects it to the specified unit.
	CLOSE	Disconnects a file from a unit.
	REWIND	Positions a file at its initial point.
	INQUIRE	Determines the specified properties of a file or of a unit on which a file can be opened.
	BACKSPACE	Positions a file to the preceding record.
	ENDFILE	Writes an endfile record on the file connected to the specified unit.
	FIND	Positions a direct-access file to a particular record.

The ACCEPT, DECODE, TYPE, ENCODE, and FIND statements are CONVEX extensions to the FORTRAN-77 standard.

Chapter 3
Character Data

This chapter describes the use of character data and shows you how to build character strings, substrings, and constants, and how to declare character data, initialize character variables, manipulate data for longer arguments, and build character library functions.

3.1 Character Constants

A character constant is a string of characters enclosed in apostrophes. A space is considered as a valid character. To include an apostrophe as part of a character constant, use two consecutive apostrophes with no intervening blanks. The following examples show how to assign a character value to a character variable:

Examples:

```
STRING = ' ab c'
ABC = 'BAR'
CANNOT = 'CAN''T'
```

If the size of the variable is smaller than the number of characters being assigned to the variable, the string is truncated on the right; if the size of the variable is larger than the number of characters being assigned, the string is padded on the right with blanks up to the designated length of the variable.

Examples:

```
CHARACTER*2 ABC
ABC = 'BAR'         ! The value BA is stored in ABC

CHARACTER*6 ABC
ABC = 'BAR'         ! The value BAR^^^ (^ = blank) is stored in ABC
```

You can use the PARAMETER statement to give character constants symbolic names. For example:

```
CHARACTER*(*) POEM
PARAMETER (POEM = 'BEOWULF')
```

You can now use the symbolic name POEM anywhere a character constant is allowed.

CONVEX FORTRAN does not allow the variable on the left hand side of a character assignment statement to appear on the right hand side.

Example:

```
CHARACTER*30 A

A(1:10) = A(5:15)      ! This statement is invalid
A(1:10) = A(20:30)     ! This statement is valid
```

3.2 Declaring Character Variables

The CHARACTER statement is used to declare character variables or arrays as shown in the following examples.

Examples:

```
CHARACTER*8 GAME(10), TENNIS     ! Declare the array GAME with ten 8-character
                                 ! elements and the variable TENNIS, which is
                                 ! 8 characters long

CHARACTER*8 TEAM, POLO*2, GAME   ! Declare TEAM and GAME as 8-character variables
                                 ! and POLO as a 2-character variable
```

For more information on the CHARACTER statement, please refer to the *CONVEX FORTRAN Language Reference Manual.*

3.3 Initializing Character Variables

You can use the DATA statement to initialize a character variable.

Examples:

```
CHARACTER*8  GAME(10), TENNIS
DATA GAME(1), TENNIS /'HOCKEY', 'CONNORS'/

CHARACTER*10 TEAM
DATA TEAM /'COWBOYS'/
```

You can also initialize character variables in the CHARACTER declaration statement.

Example:

```
CHARACTER*10 TEAM /'COWBOYS'/
```

If necessary, the value used to initialize the variable is extended with blanks or truncated in the same manner as for the assignment statement (see Section 3.1).

3.4 Character Substrings

A character substring is a portion of a character string and is consists of the name of a character variable or array element followed by delimiters that define the leftmost and rightmost characters in the substring. Substrings have the form:

v (*e1*:*e2*) ! Substring of a character variable

or

a (*s*,*s* ...) (*e1*:*e2*) ! Substring of an array element

where

v	is a character variable name.
a(*s*,*s*...)	is a character array element name.

e1, e2 are integer expressions and are called substring expressions.

The value *e1* specifies the leftmost character position of the substring, and the value *e2* the rightmost character position. These values allow you to select certain segments (substrings) from a character variable or array element. For example, assume the following character string:

```
POE = 'ONCE UPON A MIDNIGHT DREARY'
```

The following examples show how to extract various substrings.

Examples:

```
POE(13:20)      ! Extracts the substring MIDNIGHT
POE(:11)        ! Extracts the substring ONCE UPON A
POE(13:)        ! Extracts the substring MIDNIGHT DREARY
```

3.5 Concatenating Character Strings

Use double slashes (//) to concatenate strings from two or more separate strings. Thus, to create a variable called BUDGET from the following strings:

```
'FOUR'
'BILLION'
'DOLLARS'
```

define each as a character variable with a specified length:

```
CHARACTER*20 BUDGET
CHARACTER*5  S1/'FOUR'/
CHARACTER*8 S2/'BILLION'/
CHARACTER*7  S3/'DOLLARS'/
```

Next, use the double slashes to create your new string:

```
BUDGET = S1//S2//S3
```

This string contains all the values assigned to each of the substrings. You can select a given substring using a character substring reference. The following character substring references access those portions of BUDGET containing the concatenated values:

```
BUDGET(:5)
BUDGET(6:13)
BUDGET(14:20)
```

3.5.1 Character Input/Output

The CHARACTER data type enables you to read and write character strings of any length.

Example:

```
      CHARACTER*20 HEADER
          .
          .
          .
      READ (6,100) HEADER
100   FORMAT (A)
```

The preceding code causes 20 characters to be read from unit 6 and stored in the variable HEADER.

3.6 Character Library Functions

There are eight character intrinsic functions and two character utility functions:

Character Intrinsic Functions	Utility Functions
ICHAR CHAR LEN INDEX LGE, LGT, LLE, LLT	RINDEX LNBLNK

3.6.1 ICHAR Function

The ICHAR function returns the decimal value of a specified ASCII character. This function has the form:

ICHAR (*c*)

where *c* is an ASCII character expression. If *c* is longer than one character, only the value of the first character is returned; the rest of the expression is ignored.

Example:

```
CHARACTER*5 STATE/'TEXAS'/
I=ICHAR(STATE(1:1))        !sets I to 84, the ASCII value of 'T'
```

3.6.2 CHAR Function

The CHAR function returns the ASCII character corresponding to a specified decimal value. CHAR returns an ASCII value from 0 through 255. This function has the form:

CHAR (*i*)

where *i* is an integer expression equivalent to an ASCII code. Unlike FORTRAN, C strings are usually null terminated. You can get this effect in FORTRAN by concatenating a CHAR(0) at the end of the string.

Example:

```
CHARACTER*80 CSTR
PRINT *, CHAR(84)              !prints the letter T
CSTR = 'ABC' //CHAR(0)         !places a null at the end of a string for C
```

3.6.3 LEN and LNBLNK Functions

The LEN function returns an integer length to show the length of a character expression. This function is useful for finding the true length of an object declared CHARACTER *(*). The LEN function has the form:

LEN (*c*)

where c is a character expression.

Example:

```
I=len('A'//'B'//'C')  !sets I to 3
```

The LNBLNK function finds the position of the rightmost nonblank character:

```
I=LNBLNK('ABC^^^^')     !sets I to 3
```

3.6.4 INDEX and RINDEX Functions

The INDEX function searches a specified string for the occurrence of a substring; the RINDEX function finds the last occurrence of a string.

If the substring exists, INDEX returns an integer value corresponding to the character position at which the substring begins. If no substring exists, INDEX returns a value of zero. If the substring appears more than once, INDEX returns the starting position of the leftmost substring. This function can be used to search for specific characters, words, or sentences located in a given text. The INDEX function has the form:

INDEX (*c1,c2*)

where

c1 is a character expression that specifies the string to be searched.

c2 is a character expression representing the substring for which a match is desired.

Example:

```
CHARACTER*20 STRING /'NOW IS THE TIME'/
I=INDEX(STRING,'THE')        ! sets I to 8
J=INDEX(STRING,'TIME')       ! sets J to 12
R=INDEX(STRING,'I')          ! sets R to 5

INTEGER RINDEX,R
R=RINDEX(STRING,'I')         ! sets R to 13
```

3.6.5 Lexical Comparison Functions

Four intrinsic functions (LGE, LGT, LLE, and LLT) are used for comparing the standard lexical relationships of two character strings and returning a logical value of .TRUE. or .FALSE.

You can get the same results by using the arithmetic relational operators (such as .GE.) instead of the lexical functions because CONVEX machines store strings in ASCII.

Table 3-1 describes the lexical comparison functions.

Table 3-1: Lexical Intrinsic Functions

Function	Value is true if...
LGE(c1,c2)	The string c1 follows or equals the string c2 in the ASCII collating sequence.
LGT(c1,c2)	The string c1 follows the string c2 in the ASCII collating sequence.
LLE(c1,c2)	The string c1 precedes or equals the string c2 in the ASCII collating sequence.
LLT(c1,c2)	The string c1 precedes the string c2 in the ASCII collating sequence.

Chapter 4
Optimization

The CONVEX FORTRAN compiler offers several types of optimization that you can use to produce more efficient code and to enhance the speed of execution of your program. You can specify the optimization to be performed on your program by means of compiler directives, command line options, and the OPTIONS statement.

4.1 Types of Optimization

The types of optimization that can be performed are

- Vectorization
- Parallelization
- Scalar optimization
- Inline substitution
- Loop replication

Regardless of the optimization level you specify, the compiler always performs machine-dependent optimization as described later in this chapter.

Higher levels of optimization usually require more compilation time. The use of optimization also affects the use of the *csd* debugger. Many of the optimizations described in this chapter are not performed if your program uses equivalenced variables and arrays.

NOTE

Be sure to read the descriptive text before attempting to use either inline substitution or loop replication.

4.2 Vectorization

Vectorization converts scalar operations on data arrays into their equivalent vector operations. Vector operations use the vector registers in the CONVEX processors to perform simultaneous operations on multiple elements of a data array. For example, vector operations can add up to 128 elements of an array with a single instruction.

The -O2 option on the *fc* command line causes the compiler to perform vectorization as well as global optimization and local optimization on the program being compiled.

4.2.1 Basic Operation

An innermost DO loop is vectorized directly. For example, vector code is generated for the following loop:

```
      DO 10 I = 1,100
      A(I) = B(I)+C(I)
10    CONTINUE
```

Instead of generating a loop to load elements of B and C, add them, store them 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 resulting vector register into A.

DO 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:

```
      DO 10 I = 1, 10
      A(I) = B(KK(I))+C(I*I)
      IF (A(I).LT.0)THEN
            IF (A(I).GT.-100) A(I) = 0
      ELSE
            A(I) = SQRT(A(I))
      ENDIF
10    CONTINUE
```

4.2.2 Strip Mining

The vector registers of the CONVEX processor hold 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, the following code:

```
      DO 10 I = 1,N
        A(I) = B(I)+C(I)
10    CONTINUE
```

becomes

```
      I = 1
      DO 10a LV = N, 0, -128
      DO 10b IV = I, I + MIN(128,LV)-1
10b     A(IV) = B(IV) + C(IV)
        I = I + 128
10a   CONTINUE
```

where LV is a variable introduced by the compiler to count the number of elements remaining to be processed, and the 10b loop on IV represents a vector operation.

If you request parallelization, the compiler may select a strip-mine length other than 128. The compiler determines the strip-mine length to achieve a balance between parallelization and vector length for fastest execution. Among the factors examined are the iteration count of the loop and the amount of code contained in the loop body.

4.2.3 Loop Distribution

Nests of DO loops are vectorized by first distributing the outermost loop, then vectorizing each of the resulting loops or loop nests. For example, consider the following nest of DO loops:

```
      DO 20 I = 1,N
        B(I,1) = 0
      DO 10 J = 1,M
        A(I) = A(I)+B(I,J)*C(I,J)
```

```
10    CONTINUE
        D(I) = E(I)+A(I)
20    CONTINUE
```

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

```
      DO 20a I = 1,N
        B(I,1)=0
20a   CONTINUE
      DO 20b I = 1,N
      DO 10  J = 1,M
        A(I) = A(I) + B(I,J) * C(I,J)
10    CONTINUE
20b   CONTINUE
      DO 20c I = 1,M
        D(I) = E(I) + A(I)
20c   CONTINUE
```

where 20a, 20b, and 20c represent labels created by the compiler.

4.2.4 Loop Interchange

The 20a and 20c loops and the 10 loop of the preceding example are all innermost loops and can be vectorized directly. To yield additional performance improvement, however, the vectorizer performs the loop interchange optimization on the middle nest of loop 20b and loop 10, replacing it with the following nest:

```
      DO 10 J = 1,M
      DO 20b I = 1,N
      A(I) = A(I) + B(I,J) * C(I,J)
20b   CONTINUE
10    CONTINUE
```

When the vector code is then generated for the 20b loop, elements of B and C are accessed contiguously as they are loaded into vector registers. This procedure provides a substantial performance improvement over the noncontiguous access that results if the interchange is not performed.

4.2.5 Semantic Differences With Vectorization

When using a vectorized loop, you may get incorrect results if one of its induction variables has zero-stride. For example:

```
      do 10 i = 1,n
      j = j + zero
      b(i) = a(j)
      a(j) = c(i)
10    continue
```

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

```
      dimension a(10000), b(10000), c(10)
      data a/10*-5, 9990*0/
      do 10 i = 1,10000
      if (a(i).lt.0)b(i) = a(i) + c(i)
10    continue
```

4.2.6 Vectorizer Limitations

The vectorizer has the following limitations:

- Loops containing computed or assigned GOTO statements or more than one exit cannot be vectorized or partially vectorized.
- Loops containing function or subroutine calls or I/O statements can be partially vectorized; the calls or I/O statements cannot themselves be vectorized.
- If an outer loop contains a nested loop with an induction variable whose start value or step value varies with iterations, the outer loop is not processed. An induction variable is a variable that is incremented or decremented by the same amount on each iteration of the loop and is usually the DO loop control variable.
- Although loops containing multiple entries cannot be vectorized, the compiler attempts to vectorize any loops contained within the multiple entry loops.

4.2.7 Recurrence

In addition to the previously noted 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 subsequent iteration to compute the value on the right side of the same assignment. For example:

```
      DO 10 I = 2,100
      A(I) = A(I-1)+1
10    CONTINUE
```

On the first iteration A(2) = A(1)+1; on the second iteration A(3) = A(2)+1, using the value of A(2) computed on the first iteration. Since such a computation is inherently serial, it 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. Such situations are also referred to as recurrences.

For example, the following loop cannot be vectorized because the sign of N is unknown:

```
      DO 10 I = 1,100
      A(I+N) = 1
10    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). If N were -1, however, 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:

```
      DO 10 I = 1,100
      A(J(I)) = A(J(I)) + 1
10    CONTINUE
```

This loop can probably be vectorized 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 to be used in computation of its right side on a subsequent iteration, and the compiler must assume that the references to A(J(I)) are in a recurrence.

The NO_RECURRENCE directive can be used to vectorize loops in which vectorization is otherwise prevented by apparent recurrences.

4.2.8 Reductions

The compiler vectorizes one important special class of recurrences known as reductions. In general, a reduction has the form

$x = x \; oper \; y$

where

x is a scalar variable (or scalar relative to the loop in question).

y is any expression not involving *x*, and *x* is not assigned or used elsewhere in the loop

oper is one of the operators +, -, *, .AND., .OR., .EQV., .NEQV., or one of the intrinsic functions for maximum or minimum.

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

```
      SUM = 0
      DO 10 I = 1,100
      SUM = SUM + A(I)
10    CONTINUE
```

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:

```
      DO 10 I = 1,N
      A(I) = A(I-1) + B(I)*C(I)
10    CONTINUE
```

The vectorizer recognizes, however, that the multiplication B(I)*C(I) can be vectorized. To do so, it introduces a temporary array, in this case T(I), and splits the loop into two loops:

```
      DO 10a I = 1,N
      T(I) = B(I)* C(I)
10a   CONTINUE
      DO 10b I = 1,N
      A(I) = A(I-1)+T(I)
10b   CONTINUE
```

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

4.2.9 Conditional Induction Variables

Conditional induction variables are variables that are incremented by a constant amount within a loop but not on each iteration of the loop. The compiler can frequently recognize conditional induction variables and generate vector code for expressions involving them. For example:

```
      k=0
      do 10 i=1,10
      if (cond(i)) then
          k=k+1
          a(i)=b(k)
          c(k)=d(i)
      endif
10    continue
```

In the preceding example, *k* is a conditional induction variable. A vector of length less than 10 is fetched from *b*, expanded with the *merg* instruction, and stored into *a*. A vector of length 10 is fetched from *d*, compressed with the *cprs.t* instruction, and stored in *c*.

This feature is useful in dealing with calculations on sparse vectors. You can create compressed copies of the vectors to be worked on, perform the necessary computations with the compressed vectors, then merge the results of the compressed computations with the original result vectors. If the compressed vectors are much shorter than the original vectors and extensive computations are to be performed, a substantial performance improvement can be achieved.

4.3 Parallelization

If you specify the -O3 option, the compiler attempts to generate parallelized code for all eligible loops in the program. This code allows the processor to execute independent, nonsynchronized iterations of a loop in parallel on separate processors. A loop can be parallelized without synchronization if there are no dependencies between iterations; that is, if the results computed by one iteration do not depend on the results of earlier iterations. A loop cannot be parallelized if it includes

- Loop-carried dependencies
- Exits
- Calls to subroutines

When you specify the -O3 option on the *fc* command line, the compiler performs parallelization as well as vectorization, global optimization, and local optimization on the program being compiled.

Parallelization is generally performed on the outer loop of a nest of loops or to the strip-mine loop generated by vectorization. Consider the following example:

```
      DO 10 J = 1,M
      DO 10 I = 1,N
      A(I,J) = B(I,J) + C(J,I)
10    CONTINUE
```

In the preceding example, the I loop is vectorized but the J loop is parallelized. Up to N processors can be simultaneously executing vector operations. The compiler may further perform parallel strip mining on the outer loop to minimize the synchronization overhead required for scheduling iterations.

Parallel strip mining breaks the parallel loop into two loops, then parallelizes only the outer loop. Each processor executes a contiguous strip of iterations of the original parallel loop.

When no loop is outside the vector loop, the strip-mine loop is usually parallelized. The following example:

```
      DO 10 I = 1,N
10    A(I) = B(I) + C(I)
```

becomes

```
      DO 10 I = 1,N,128
      DO 10 II = I,MIN (N,I+128)
10    A(II) = B(II) + C(II)
```

In the preceding example, a nest of two loops is created, which can be vectorized and parallelized. If N is a relatively small constant, but larger than 64, the compiler may generate a strip-mine length smaller than 128 to ensure that enough iterations are in the outer loop to make it worth parallelizing. If N is a variable, the compiler performs dynamic vector strip mining to select the best strip-mine length at runtime.

The compiler can also handle most scalar assignments and reductions within parallel loops. For example, the following loop will be both vectorized and parallelized:

```
      DO 10 I = 1,N
      IF (A(I) .GT. 0) GOTO 10
      S = S + B(I)*C(I)
      X = B(I)
10    CONTINUE
```

4.4 Global Optimization

Global optimization is machine-independent scalar optimization that is performed over an entire program unit—in particular, conditional statements and loops. The -O1 option on the *fc* command line causes the compiler to perform global optimization as well as local optimization on the program being compiled.

The following sections describe the various types of global optimization that are performed.

4.4.1 Constant Propagation and Folding

Global constant propagation and folding is similar to local constant propagation and folding, except that the folded constant is propagated across the program unit, provided the constant appears later in the program and is actually used.

Example:

Original Program	Optimized Program
`INTEGER A,B,C`	`PROGRAM GFOLD1`
`A=5`	`INTEGER A,B,C`
`B=15`	`A=5`
`READ *,I`	`B=15`
`IF (I) 10,10,15`	`READ *,I`
`10 A=6`	`IF (I) 10,10,15`
`C=A`	`10 A=6`
`GOTO 20`	`C=6`
`15 C=A+B`	`GOTO 20`
`GOTO 25`	`15 C=20`
`20 B=A+C`	`GOTO 25`
`GOTO 30`	`20 B=12`
`25 B=A+8+C`	`GOTO 30`
`30 PRINT *,A,B,C`	`25 B=33`
`END`	`30 PRINT *,A,B,C`
	`END`

4.4.2 Dead-Code Elimination

As a result of constant propagation and folding, the arithmetic or logical expression of an IF statement may be folded to .TRUE. or .FALSE.. The alternative that is unreachable is eliminated.

Conditional compilation is a good example of the use of dead-code elimination. Code that is to be conditionally compiled is enclosed by an IF statement that tests a variable whose value is set to .TRUE. (compile enclosed code) or statement, or data initialization.

4.4.3 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.

In the following example, a substitution is performed if, by doing so, the assignment to x becomes redundant and can be eliminated.

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

becomes

```
t=z-y
```

4.4.4 Redundant-Assignment Elimination

Redundant-assignment elimination removes assignment statements (definitions) that are never used. Label assignments (i.e., ASSIGN statements) and assignments to a dummy parameter, name of a function, common variable, or local variable of a subprogram are never eliminated. Also, if the right side of an assignment statement contains a function call, the assignment is not eliminated because the function could have side effects.

Example:

Original Program	Optimized Program

program grae1
...
x=y*z
if (a.gt.0) then
...
* x not used
a=x*y+ ...
else
...
* x not used
x=...
end if
...
* a,x not used
end
	program grae1
	...
	if(a.gt.0) then
	...
	else
	...
	end if
	...
	end

4.4.5 Redundant-Subexpression Elimination

Global redundant-subexpression elimination removes common subexpressions. Instead of retaining the value of a common subexpression in a register, the value is assigned to a compiler-generated temporary; all other occurrences are replaced by use of this temporary.

Example 1:

Original Program	Optimized Program

program gcse1
...
read *,c
if (k .lt. 1) then
a=b+(c*4)/(-(j*b)+sqrt(c))
else
e=e-(b+(c*4)/(-(j*b)+sqrt(c)))
end if
f=b+(c*4)/(-(j*b)+sqrt(c))
...
end
	program gcse1
	...
	read *,c
	t1=b+(c*4)/(-(j*b)+sqrt(c))
	if (k .lt. 1) then
	a=t1
	else
	e=e-(t1)
	end if
	f=t1
	...
	end

Example 2:

Original Program	Optimized Program

program gcse2
...
read *,c
a=b+(c*4)/(-(j*b)+sqrt(c))
if (k .lt. 1) then
l=5
else
l=6
end if
f=e-(b+(c*4)/(-(j*b)+sqrt(c)))
...
end
	program gcse2
	...
	read *,c
	t1=b+(c*4)/(-(j*b)+sqrt(c))
	a=t1
	if (k .lt. 1) then
	l=5
	else
	l=6
	end if
	f=e-t1
	...
	end

4.4.6 Code Motion

Code motion moves invariant computations in a loop to a position in front of the loop. An invariant computation is one that yields the same result independent of the number of times through the loop. 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 unless the *-uo* option is specified.

Example 1:

Original Program	Optimized Program
program cm1 common a,b,e dimension ar(10) ... read *,c do i=1,10 a=b+(c*4)/(-(e*b)+sqrt(c)) ar(i)=a+b*c enddo ... end	program cm1 common a,b,e dimension ar(10) ... read *,c a=b+(c*4)/(-(e*b)+sqrt(c)) t1=a+b*c do i=1,10 ar(i)=t1 enddo ... end

Example 2:

Original Program	Optimized Program

subroutine cm2
common a,b,c(10)
do i=1,10
  a=b
  c(i)=0
enddo
  ...
end
	subroutine cm2
	common a,b,c(10)
	a=b
	do i=1,10
	  c(i)=0
	enddo
	  ...
	end

4.4.7 Strength Reduction

Strength reduction replaces 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, i.e., 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. Thus, for a loop on i containing $j=j+i$, j is not an induction variable. A typical operator subject to strength reduction is multiplication, such as multiplications used to calculate the address of subscripted variables.

Strength reduction of $i*r$ is not performed. The reduced operations are not numerically equivalent due to 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	Optimized Program

program sr1
  i=1 ! i is a loop induction variable
10 x=i*c ! c is loop invariant
  ...
  i=i+2
  if (i .le. 100) goto 10
  ...
  end
	program sr1
	  i=1
	  t1=i*c
	  t2=2*c
	10 x=t1
	  ...
	  t1=t1+t2
	  i=i+2
	  if(i .le. 100) goto 10
	  ...
	  end

If i is dead on exit from the loop, i.e., is not used before being assigned, and there are no other uses of i in the loop except in the incrementation and test, the incrementation can be eliminated and the test 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:

```
      program sr1
      i=1
      t1=i*c
      t2=2*c
      t3=100*c
10    x=t1
```

```
...
t1=t1+t2
if (t1 .le. t3) goto 10
...
end
```

4.5 Local Optimization

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

The following pages describe the various types of local optimization that can be performed.

4.5.1 Assignment Substitution

Assignment substitution eliminates redundant loads and stores. The compiler substitutes a preassigned value of a variable for all succeeding uses of the variable. For example:

```
x=y+c
  ...
   x
  ...
   x
x=f(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 opens up opportunities for other optimizations, such as constant folding and redundant subexpression elimination.

4.5.2 Redundant-Assignment Elimination

This optimization removes redundant assignments to the same variable. An assignment to a variable can be followed by another assignment to the same variable, wiping out the result of the first assignment. Thus, there is no need for the program to perform the first assignment and the compiler eliminates it.

4.5.3 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.

4.5.4 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*.

4.5.5 Constant Propagation and Folding

Propagating constants in a program means that when a constant is assigned 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 it is replaced with the constant 5.

In constant folding, when the compiler encounters an operation on constants, such as 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.

The most frequently-used intrinsics are folded at compile time if applied to constant arguments; for example, *sin(0.0) => 0.0*. Constant folding is also applied to ** at compile time.

Example:

Original Program	Optimized Program

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

Compile-time type conversions operate 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 expression x = 1, the compiler converts the 1 to REAL data type as if x = 1.0 had been written.

If during constant folding an integer overflow occurs, you get a user-error message ("Integer constant truncation"). If floating-point underflow occurs, the folded result is zero. If floating-point overflow occurs, a user-error message displays ("Real constant either too large or too small"). The user cannot override these compiler actions. It is necessary to modify the source, replacing the operator or constants with the correct constant.

4.5.6 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 .and. -1	x
x .and. 0	0
x .or. -1	-1
x .or. 0	x
-1*x	-x
x-x	0
x/-1	-x
-1**x	1-(x .and. 1)*2
x**.5	sqrt(x)
x**0	1
1**x	1
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 is converted to x+y.

4.5.7 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:

In the first example, *c* is constant so that *1/c* can be folded. The first optimization is not performed unless the *-uo* option is specified.

```
x/c  =>  (1/c)*x
5*i  =>  ISHFT(i,2) + i
```

4.6 Inline Substitution

Inline substitution (inlining) replaces a subroutine or function call with the actual body of the subprogram. During the substitution, actual arguments are mapped to dummy arguments and local identifiers are assigned unique names.

Inlining can improve the performance of a program because inlined code can be fully optimized by the compiler. The inlined code can be customized for a particular call based on the actual arguments passed in the call. Portions of inlined code can become "dead" code, the vectorization of loops can be enhanced, and call overhead is removed.

If a subprogram is to be inlined, a specially-compiled version of the subprogram must reside in an intermediate language (*.fil*) file from which it can be accessed by the compiler.

Inlining can be performed to any depth, that is, a subprogram that is inlined can itself call a subprogram that is to be inlined, and so on. Recursive inlining is not permitted; that is, a subprogram cannot call itself, either directly or indirectly.

4.6.1 When to Use Inlining

It may not be advantageous to inline every call in your program. Before compilation you should determine which, if any, subprograms or functions should be inlined. To make this determination, profile your program using the *prof* utility and inline those subprograms that are called most often. Repeat the profiling once more to determine if additional subprograms should be inlined.

You should not attempt to inline very large subprograms, especially if these programs are called several times. The speed of compilation could be significantly reduced and the size of the program could be significantly increased.

4.6.2 How to Use Inlining

In order to perform inlining, you must do the following:

- Make certain that a *.fil* file exists for each subprogram to be inlined.
- Specify the *-is* option on your *fc* command line.

4.6.3 Creating *.fil* Files

To create *.fil* files, specify the *-il* option on the *fc* command line. The compiler generates a separate *.fil* file for each subprogram in the source file. The *-il* option cannot be used with the *-c*, *-cs*, or *-S* options. Optimization levels are ignored.

The name assigned to a *.fil* file is the name of the subprogram with the extension *.fil*. For example, if a file contains subprograms a, b, and c, the files created are *a.fil*, *b.fil*, and *c.fil*. After compilation, if you do not want a particular subprogram to be inlined, you must delete its

NOTE

Whenever you upgrade to a newer release of the compiler, you must regenerate all your *.fil* files before they can be used for inlining.

If *.fil* output cannot be produced, the compiler generates an explanatory message. No *.fil* file is generated if the subprogram:

- Contains alternate entry points.
- Contains a SAVE statement.
- Contains a NAMELIST statement.
- Is compiled with the *-cs* (check subscript) option.
- Uses an adjustable array.
- Contains a character dummy argument
- Function returns character type
- Contains DATA statements or type statements with initial values

4.6.4 Using the *-is* Option

The *-is* option on the *fc* command line instructs the compiler to perform the actual inline substitution of subprograms for which there exists a *.fil* file. The format of this option is:

-is *directory*

where *directory* is the name of a user or library directory to be searched for the *.fil* files. If you want to specify more than one directory, repeat the " *-is directory* " pair as many times as necessary on the command line. Directories are searched in the order specified.

If you want to inline only specific subprograms, copy their *.fil* file(s) into a special directory and specify that directory name with the *-is* option. Once you have specified the *-is* option, inlining is performed on every subprogram call for which a corresponding *.fil* file exists. It is not possible to exclude specific calls.

If inlining cannot be performed, the call is retained and the compiler generates an explanatory message. Inline substitution is not performed on subprograms:

- When a name in a common block conflicts with a name in the main program.
- When data types and sizes in common do not match.
- When the actual arguments to the called routine do not agree in number and type with the corresponding dummy arguments.
- When the actual argument is an array name whose number of dimensions is not the same as the number of dimensions of the corresponding dummy argument.
- When the actual argument is an array name whose lower and upper dimension bounds are not the same as the corresponding lower and upper bounds of the corresponding dummy argument.
- When the dummy argument is referenced as a subroutine and the actual argument is not a subroutine name.
- When the dummy argument is referenced as a function and the actual argument is not a function name.
- When the actual argument is a function whose type does not agree with the type of the dummy argument.

- When the actual argument is an intrinsic function and the arguments of the dummy function reference do not agree in number and type with that defined for the intrinsic.

4.6.5 Restrictions on Inlining

Incorrect results may occur if the same subprogram is inlined in different, separately-compiled program units but relies on the retention, between calls, of a local static variable. Local static variables are not global between separately-compiled program units but can be made global by putting them into a common block.

The use of a SAVE statement suppresses inlining. To allow a subprogram that contains SAVE statements to be inlined, variables that occur in SAVE statements should be put into a common block and the SAVE statement should be removed.

Inline substitution affects the use of the *csd* debugger because the user breakpoints at the start of the inlined code instead of at the start of the subprogram. It is not possible to breakpoint within the inlined code; breakpoints are specified in terms of line numbers of the source file. In addition, the symbols of the inlined routine are compiler-generated names and cannot be referenced.

A cross-reference listing is produced in terms of the original source program and is not based on the results of inline substitution.

4.7 Loop Replication

The CONVEX FORTRAN compiler offers a set of optimizations that replicate the body of a loop. To request these optimizations, you must specify both the -O2 and the -rl compiler options. The loop replication optimizations are

- Loop unrolling
- Dynamic loop selection

NOTE

When the compiler performs loop replication, your program requires more memory due to the additional code being generated.

The compiler automatically selects loops for which replication is appropriate. Individual loops may be selected manually by means of the UNROLL and SELECT directives described in the *CONVEX FORTRAN Language Reference Manual.*

4.7.1 Loop Unrolling

Loop unrolling reduces loop overhead by replicating the body of the loop. Loop unrolling is performed on both scalar loops and vector loops. An individual loop can be flagged for unrolling by means of the UNROLL directive.

Unrolling may either be complete or partial. Complete unrolling unrolls a loop before vectorization, from the innermost loop outward, to facilitate outer loop vectorization.

Example:

```
DO I = 1,3
   C(I) = D(I)
ENDDO
```

becomes

```
C(1) = D(1)
C(2) = D(2)
C(3) = D(3)
```

Partial unrolling is attempted after vectorization on loops that have not been vectorized.

Example:

```
DO I = 1,N
   A(I) = B(I)
ENDDO
```

becomes

```
DO I = 1,IAND (N,3)
   A(I) = B(I)
ENDDO

DO J = I,N,4
   A(J)   = B(J)
   A(J+1) = B(J+1)
   A(J+2) = B(J+2)
   A(J+3) = B(J+3)
ENDDO
```

4.7.2 Dynamic Loop Selection

Dynamic loop selection causes the compiler to create multiple versions of a loop and to select at runtime which version to execute. As required, the compiler can generate scalar and vector versions of a loop.

You can control dynamic loop selection in one of two ways: by means of the *-rl* compiler option and by means of the SELECT directive. The *-rl* option operates on all loops in the program unit according to internal compiler algorithms. The SELECT directive operates on the loop immediately following the directive according to the parameters supplied in the directive.

The SELECT directive is described in the *CONVEX FORTRAN Language Reference Manual.*

4.8 Machine-Dependent Optimization

Machine-dependent optimization, as described in the sections that follow, 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.

4.8.1 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 performed only within (not between) statements, so that *csd* can correlate instructions with the lines in the original program.

Instruction scheduling on CONVEX processors schedules instructions across numbers of statements instead of in one statement only, often 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 are nearly concurrent.

4.8.2 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.

4.8.3 Branch Optimization

Many compilers generate branch instructions that branch to the next sequential instruction. Such branches are generated internally by the CONVEX FORTRAN compiler but are removed by "branch optimization" before the object code is produced.

4.8.4 Register Allocation

Register allocation is performed automatically. The register allocation technique used by the CONVEX compiler takes advantage of the unique architecture of CONVEX supercomputers. Most machines try to minimize the number of registers allocated for a given expression; the CONVEX compiler attempts to maximize the number in order to achieve more parallelism.

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

4.8.5 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, 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.

4.8.6 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. The innermost loop of such an operation contains code similar to the following:

```
C(I,K) = C(I,K) + A(I,J) * B(J,K)
```

If the DO I or DO K loops are vectorized, the C(I,K) references form a matching pair. The matching pair perform a reduction in parallel in each of the cells of the vector register.

Array references may be matched in vectorized loops if:

- 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.

4.8.7 Strength Reduction and the Code Generator

The code generator performs certain strength-reduction operations on instruction-level operations. For example, instead of performing integer multiplies by a power of 2, the code generator transforms them to shifts.

4.8.8 Tree-Height Reduction

Tree-height reduction is best explained by example; consider the following expression:

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

Two methods of evaluating this expression are

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

Method 1 requires g+h to be evaluated first. The result of that calculation is then used to compute f+(g+h) 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. That is, (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 an expression is about proportional to its depth, i.e., the deepest nesting level of parentheses in the expression. This is 6 for the first method, but only 3 for the second.

When the compiler can choose the 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, and hence the name of the optimization.) This may result in a different numerical result for floating-point operators due to rounding off in the least-significant bits. The compiler cannot, however, override an order of evaluation made explicit by the use of parentheses. For example, if you write a+(b+(c+(d+(e+(f+(g+h)))))), the compiler generates code to evaluate first g+h, then f+(g+h), and so forth. To get faster code, omit the parentheses.

Chapter 5
Calling Conventions

Subprogram calls in CONVEX FORTRAN use the same calling convention as that used by other CONVEX language processors. This CONVEX standard permits routines written in CONVEX assembly language, C, or Ada to be called from a FORTRAN program and vice versa.

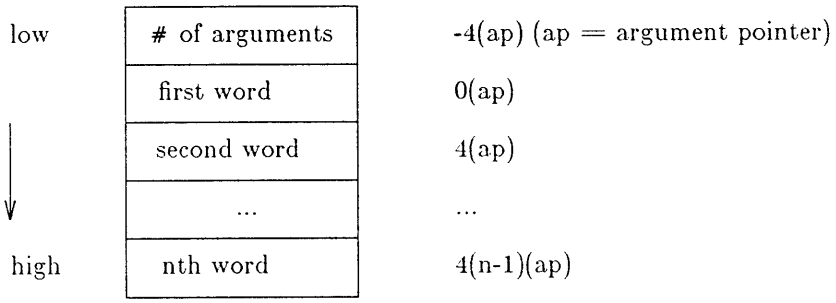
5.1 FORTRAN Subprogram Calling Convention

The basis of the calling standard is the passing of actual arguments or parameters. The standard supports two argument-passing mechanisms: call by value and call by reference. Although FORTRAN arguments are passed by reference, CONVEX FORTRAN provides built-in functions (%VAL and %REF) to support both mechanisms. A routine call involves passing a pointer to the list of arguments, called the "argument packet".

5.1.1 FORTRAN Argument Packets

An argument packet is a sequence of word (4-byte) entries. Only precompiled argument lists are allocated space in memory statically. When possible, the compiler creates argument list entries at compile time. The compiler cannot precompile an argument list if any argument is a dummy argument or array element with nonconstant subscripts. Figure 5-1 shows the layout of an argument packet in memory.

Figure 5–1: Argument Packet: Example 1



The first word is normally the address of the first argument; the second word is the address of the second argument; and so on. For character arguments, an extra by-value word is added to the end of the list containing the length of the character entity. There is one extra word for each character argument, occurring in the same order as the addresses of the character arguments.

For functions that return character and complex values, an extra argument is added before the first user-specified argument to receive the function result. For a character-valued function, this extra argument consists of two words: the first is the address of the character string to receive the value of the function; the second is its maximum length.

Figure 5-2 shows an example of both the FORTRAN code and the resulting layout of the argument packet in memory.

Figure 5–2: Argument Packet: Example 2

```
character*(*) function f (x, ch1, a, b, i, ch2, j)
character*10 ch1
character*5  ch2
real a
real*8 b
complex x
integer*4 j
integer*2 i
end
```

low	# of arguments (11)	-4(ap) (ap is the argument pointer)
	Address of result	0(ap)
	Max. length of result	4(ap)
	Address of x	8(ap)
	Address of ch1	12(ap)
	Address of a	16(ap)
	Address of b	20(ap)
	Address of i	24(ap)
	Address of ch2	28(ap)
	Address of j	32(ap)
	Length of ch1	36(ap)
high	Length of ch2	40(ap)

5.1.2 Argument-Passing Mechanisms

The CONVEX calling standard supports two methods of passing arguments to subprograms:

- Passing arguments by immediate value, where the argument-packet entry is the value.
- Passing arguments by reference, where the argument-packet entry is the address of the value.

Use the second of these mechanisms to pass all numeric arguments. For character arguments, two words are actually passed—the string itself (by reference) and its length (by value).

Table 5-1 illustrates the different numeric actual arguments and the subprogram calling built-in functions that pass these arguments. Numeric actual arguments can be logical, integer, or real. The size of the argument is *4 for *1, *2, and *4 arguments; *8 arguments cannot be passed. Each of these assignments is right justified. Complex is not allowed. The lengths are appended to the argument packet in the order of the character arguments.

Table 5-1: Built-in Functions and Argument Types

Argument Type	%VAL	%REF
Numeric	Value	Address
Character	Not allowed	Address
Array	Not allowed	Address of first element
Subprogram	Not allowed	Address of entry point

5.1.3 Argument Packet Built-in Functions

It is not always possible to pass arguments to routines not written in FORTRAN using the default FORTRAN calling convention. In this instance, CONVEX FORTRAN provides the built-in functions %VAL and %REF. These functions are not used to call a subprogram written in FORTRAN and can only appear in actual argument lists.

5.1.3.1 %VAL

This built-in function ensures that the argument packet entry uses the immediate value mechanism. It is represented as:

%VAL(*arg*)

The argument packet entry is the value of the actual argument, *arg*. See Table 5-1 for details concerning values and the size of the argument.

Example:

```
CALL SUB(3,%VAL(10))
```

In this example, the first constant is passed by reference, the second by immediate value.

5.1.3.2 %REF

This built-in function ensures that the argument packet entry uses the reference mechanism. It is represented as:

%REF(*arg*)

The argument packet entry is the address of the actual argument, *arg*. The argument value can be a numeric or character expression, array, or procedure name. See Table 5-1 for details concerning values and the size of the argument.

5.1.3.3 Function Return Values

The method of returning a value of a function depends on the data type of the value as summarized in Table 5-2.

Table 5-2: Function Return Values

Data Type	Return Method
LOGICAL INTEGER REAL	Scalar register S0
COMPLEX	If the function returns a complex, the first word of the argument list is the address of the result.
CHARACTER	If the function returns a character, the first word of the argument list is the address of the result and the second word is the length of the result.

5.1.3.4 %LOC

This function is used to assign the address of a storage element as an INTEGER*4 value, which can be used in an arithmetic expression.

Example:

```
I = %LOC(J)-4
```

In the example, I now holds in storage the address of the word preceding J. %LOC is useful when passing argument data structures containing the address of storage elements to non-FORTRAN routines.

5.2 Non-FORTRAN-to-FORTRAN Calling Sequence

If you are writing in assembly language or C and want to call a FORTRAN routine, follow the procedures outlined below. Figure 5-3 is an example of code you might write in assembler or C to call a FORTRAN subroutine. See the *CONVEX Architecture Handbook* for information on the instruction set and calling procedures. The sequence of steps in the calling procedure is:

1. Push the arguments onto the runtime stack in reverse order.
2. Update the argument pointer to point to the top of the runtime stack.
3. Push an additional argument, the number of arguments, onto the stack.
4. Call the subroutine (using the calls instruction).

Where possible, the arguments are precompiled and the calling sequence is reduced to:

1. Load the address of the argument packet into the argument pointer.
2. Call the subroutine (using the calls instruction).

Figure 5–3: Calling a FORTRAN Subroutine

```
call sub        (a,b,c)         !FORTRAN call with three real arguments
                                ! equivalent assembler calling sequence
pshea           c               ! push the third argument's address
pshea           b               ! push the second argument's address
pshea           a               ! push the first argument's address
mov             sp,ap           ! set up the ap
pshea           3               ! push number of words in argument list
calls           _sub_           ! call the subroutine
ld.w            12(fp)a         ! restore ap
add.w           #16,sp          ! restore sp

sub_ (&a,&b,&c);                /* equivalent C call */
```

The arguments are pushed in reverse, because the stack grows toward low memory addresses. Thus, the last argument pushed ends up at the top of the list.

5.2.1 Procedure Names

The compiler adds a leading and trailing underscore to the name of a common block or a FORTRAN subprogram to distinguish it from a C procedure or external variable with the same user-assigned name. FORTRAN library procedure names have embedded underscores to avoid conflict with user-assigned subroutine names.

5.2.2 Data Representations

Table 5-3 shows the corresponding FORTRAN and C declarations. In FORTRAN, all integer, logical, and real data occupy the same amount of memory.

Table 5-3: FORTRAN and C Declarations

FORTRAN	C
INTEGER*1 x	char x;
INTEGER*2 x	short int x;
INTEGER x	int x;
INTEGER*8 x	long long int x;
LOGICAL*1 x	char x;
LOGICAL*2 x	short int x;
LOGICAL x	long int x;
LOGICAL*8 x	long long int x;
REAL x	float x;
DOUBLE PRECISION x	double x;
COMPLEX x	struct {float r, i;} x;
DOUBLE COMPLEX x	struct {double dr, di;} x;
CHARACTER*6 x	char x[6];

5.2.3 Return Values

A function of type INTEGER, LOGICAL, REAL, or DOUBLE PRECISION declared as a C function returns the corresponding type. A COMPLEX or DOUBLE-COMPLEX function is equivalent to a C routine with an additional initial argument pointing to where the return value is to be stored. Thus,

Complex function...	Is equivalent to...
COMPLEX function f(...)	void f_(temp,...) struct {float r, i;} *temp; ...

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

Character function...	Is equivalent to...	Can be invoked in C by...
CHARACTER*15 function g(...)	void g_(result,length,...) char result[ ]; long int length; ...	char chars[15] ... g_(chars,15L,...);

Subroutines are invoked as if they were integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
call nret(*1, *2, *3)
```

is treated as if it were the computed *goto*

```
goto (1, 2, 3), nret( )
```

5.2.4 Argument Packets

All FORTRAN arguments are passed by address. For every argument that is of type CHARACTER or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are *int* quantities passed by value.) The order of arguments is:

1. Extra arguments for complex and character functions
2. Address for each datum or function
3. A *long int* for each character argument

Thus,

The call in...	Is equivalent to the call in...

external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
	int f( );
	char s[7];
	long int b[3];
	...
	sam_(f, &b[1], s, 7L);

The first element of a C array always has subscript 0, while FORTRAN arrays begin at 1 by default. FORTRAN arrays are stored in column-major order; C arrays are stored in row-major order.

5.3 Examples

The following examples illustrate how to interface to FORTRAN from other languages and various argument-passing techniques.

Example 1:

This example shows a simple FORTRAN procedure and how it is written in C and assembly language for interfacing to FORTRAN.

FORTRAN Source Code:

```
subroutine sub (i,r,d)  ! in fortran
integer i
real r
double precision d
d = i + r
end
```

C Source Code:

```
sub(i,r,d) int *i;
float *r;
double *d;
{
*d = *i + *r;
}
```

Object Code:

```
sub: ld.w      @4(ap),s1     ; s0 = i
     ld.w      @0(ap),s0     ; s1 = r
     cvtw.s    s0,s0         ; convert i to real
     add.s     s1,s0         ; add it to r
     cvts.d    s0,s0         ; convert the result to real*8
     st.l      s0,@8(ap)     ; store the result in d
```

Example 2:

This example shows a call involving two character arguments separated by other arguments and the compiler generated the object code:

FORTRAN Source Code:

```
subroutine sub1
character*5 a,b
real x,y
call charargs (a,x,y,b)
end
```

Object Code:

```
ds.w   6            ; 6 arguments
ds.w   LU           ; address of A
ds.w   LU+12        ; address of X
ds.w   LU+16        ; address of Y
ds.w   LU+5         ; address of B
ds.w   5            ; length of A
```

```
ds.w    5              ; length of B

ldea    LC+4,ap        ; load packet pointer
calls   _charargs_     ; restore cp
```

Example 3:

This example illustrates a call to a function that returns a character value:

FORTRAN Source Code:

```
subroutine sub2
character*10 a,f
a = f(1.7)
end
```

Object Code:

```
...
ld.w    #0x000000a,s0    ; #3, 10
pshea   LC               ; #3, ?LC
psh.w   s0               ; #3
pshea   -10(fp)          ; #3, ?ch1
mov     sp,ap            ; #3
pshea   3                ; #3
calls   _f_              ; #3, F
add.w   #0x0000010,sp    ; #3, 16
ldea    LU,a5            ; #3, A
ldea    -10(fp),a1       ; #3, ?ch1
ld.l    0x0(a1),s1
st.l    s1,0x0(a5)
ld.h    0x8(a1),s1
st.h    s1,0x8(a5)
```

Example 4:

This example illustrates a call to a function that returns a complex value:

FORTRAN Source Code:

```
subroutine sub3
complex x,f
x = f (10)
end
```

Object Code:

```
pshea   LC+32        ; address of argument 10
pshea   -8(fp)       ; address of function result
mov     sp,ap        ; packet address
pshea   2            ; number of arguments
calls   _f;
ld.w    12(fp),ap    ; restore ap
add.w   #12,sp       ; restore sp
```

Example 5:

This example shows how subprogram arguments are passed:

FORTRAN Source Code:

```
subroutine sub4
external f
call useit (f,x)
end
```

Object Code:

```
LC:  ds.w   2        ; 2 arguments
     ds.w   _f_      ; address of user external function f
     ds.w   LU+40    ; address of X

     ldea   LC+4,ap
     calls  _useit_
```

Example 6:

This example shows how an array argument is passed:

FORTRAN Source Code:

```
subroutine sub5
real a(20)
call usearray (a,x,y)
end
```

Object Code:

```
LC:ds.w  3             ; 3 arguments
   ds.w  LU+44         ; address of A
   ds.w  LU+124        ; address of X
   ds.w  LU+128        ; address of Y

   ldea  LC+4,ap
   calls _usearray_
   ld.w  12(fp),ap
```

Chapter 6
System Utilities

The FORTRAN system utility routines provide a runtime interface between CONVEX FORTRAN programs and the UNIX operating system. The library in which the utility routines reside (*/usr/lib/libU77.a*) also contains useful character and math functions. Any referenced utility is automatically loaded during compilation.

6.1 How to Call Utility Routines

A utility routine is called in the same manner as a user-written subroutine. Take, for example, the *chdir* function listed below and described in full in Section 3F of the *CONVEX UNIX Programmer's Manual*. (The manual page gives the name, synopsis, description, and file location of each of the utilities.) The synopsis gives the information required for referencing the utility:

integer function chdir (*dirname*) **character*(*)** *dirname*

This synopsis tells you that *chdir* is a function that returns an integer value, and you must pass it one parameter (a directory name) in a character variable of arbitrary length. The following program uses *chdir* to change to the */tmp* directory:

```
INTEGER*4 FUNCTION CHDIR
CHDIR ("/tmp")
END
```

The routines described in this chapter that accept INTEGER*4 arguments must always be passed INTEGER*4 arguments regardless of the *-i1*, *-i2*, or *-i8* options. Similarly, routines that accept REAL*4 arguments must be passed REAL*4 arguments regardless of which *-r* option is set.

6.2 UNIX Utilities

The calling sequences for the routines shown in Table 6-1 are also described in Section 3F of the *CONVEX UNIX Programmer's Manual* under the heading indicated in the Reference column of the table.

Table 6-1: Calling Sequences for CONVEX UNIX Utilities

Name	Reference	Description
abort	abort	terminate with memory image
access	access	determine accessibility of file
alarm	alarm	execute a subroutine after a specified time
bessel	bessel	calculate bessel functions of two kinds for integer orders
chdir	chdir	change default directory
chmod	chmod	change mode of file
ctime	stime	return system time
dffrac	flmin	return fractional accuracy of double-precision float
dflmax	flmin	return maximum positive double-precision float
dflmin	flmin	return minimum positive double-precision float
drand	rand	return random values
dtime	etime	return elapsed execution time since last call to dtime
errtrap	errtrap	enable or disable certain signal traps
etime	etime	return elapsed execution time
exit	exit	terminate process with status **
fdate	fdate	return date and time in ASCII string
ffrac	flmin	return fractional accuracy of single-precision float
fgetc	getc	get a character from a logical unit
flmax	flmin	return maximum positive single-precision float
flmin	f1min	return minimum positive single-precision float
flush	flush	flush output to a logical unit
fork	fork	create a copy of this process
fputc	putc	write a character to a FORTRAN logical unit
fseek	fseek	reposition file on logical unit
fstat	stat	get file status
ftell	fseek	reposition file on logical unit
gerror	perror	get system error message
getarg	getarg	return command line arguments
getc	getc	get a character from a logical unit
getcwd	getcwd	get current working directory
getenv	getenv	get value of environment variables
getgid	getuid	get group ID of caller
getlog	getlog	get user login name
getpid	getpid	get process id
getuid	getuid	get user ID of the caller
gmtime	stime	return system time
hostnm	hostnm	return name of current host
iargc	getarg	return command line arguments
ierrno	perror	get system error messages
inmax	flmin	return the maximum positive integer value
ioinit	ioinit	change default settings of I/O attributes
irand	rand	return random values
isatty	ttynam	find name of terminal port
itime	idate	return date or time in numerical form
kill	kill	send a signal to a process
link	link	make a link to an existing file
lnblnk	rindex	return index of last nonblank character in a string
loc	loc	return the address of an object
longjmp	longjmp	restore stack environment
lstat	stat	get file status

Table 6-1 Calling Sequences for CONVEX UNIX Utilities (cont.)

Name	Reference	Description
ltime	stime	return system time
perror	perror	get system error messages
putc	putc	write a character to a FORTRAN logical unit
qsort	qsort	perform quick sort
rand	rand3.f	return random values
rename	rename	rename a file
rindex	index	return index of last occurrence of a substring
setjmp	setjmp	save stack environment
signal	signal	change the action for a signal
sleep	sleep	suspend execution for an interval
stat	stat	get file status
stime	stime	return system time
system	system	execute a UNIX command
symlnk	link	make a symbolic link to an existing file
time	time	return time in an ASCII string
topen	topen	provide low-level interface for magnetic tape devices
traceback	traceback	print names of routines in call stack
traper	traper	trap floating-point underflow and integer overflow
ttynam	ttynam	find name of a terminal port
unlink	unlink	remove a directory entry
wait	wait	wait for a process to terminate

6.3 Using the *system* Utility

To call UNIX utilities not included in Table 6-1, use the *system* utility. *system* executes a UNIX command and is used as follows:

integer function system *(string)*
character*(*) *string*

The system call causes *string* to be given to the shell (*/bin/sh*) as input, where it is executed as if it were a command. Consult the manual page for *system* in Section 3F of the *CONVEX UNIX Programmer's Manual.*

Example:

```
i = system ('mv file1 file2')
```

6.4 VAX-11 FORTRAN System Utilities

These utilities are provided for VAX compatibility. If your program currently calls a VAX/VMS system service, you must change it so that it can call one of the UNIX utilities listed in Table 6-1, or one of the functions listed in the following subsections.

6.4.1 *date*

The *date* utility returns the current date as dd-mmm-yy.

SUBROUTINE **date***(buf)*
CHAR*9 *buf*

6.4.2 *idate*

The *idate* utility returns the current month (i), day (j), and year (k).

SUBROUTINE idate*(i,j,k)*
INTEGER*4 *i,j,k*

6.4.3 *errsns*

The *errsns* utility is similar to the UNIX function *ierrno* and returns information about the last runtime error.

SUBROUTINE errsns(fnum,rmssts,rmsstv,iunit,condval)
INTEGER*4 fnum, rmssts, rmsstv, iunit, condval

fnum is the most recent FORTRAN runtime error number. The remaining arguments are not used.

6.4.4 *exit*

The *exit* utility terminates a process and makes the argument status available to the parent process. It is equivalent to the UNIX utility function of the same name.

SUBROUTINE exit(status)
INTEGER*4 status

6.4.5 *secnds*

The *secnds* utility returns the system time in seconds, less the value of its argument.

FUNCTION secnds(x)
REAL*4 x

6.4.6 *time*

The *time* utility returns the current system time in an ASCII string as hh:mm:ss.

SUBROUTINE time*(buf)*
CHAR*8 *buf*

6.4.7 *ran*

The *ran* utility returns random values and is similar to the UNIX utility *rand.*

FUNCTION ran(i)
INTEGER*4 i

6.4.8 *mvbits*

The *mvbits* utility transfers *len* bits from positions *i* through *i+len-1* of the source location, *m*, to positions *j* through *j+len-1* of the destination location, *n*. The values of *i+len* and *j+len* must be less than 32.

```
SUBROUTINE mvbits(m,i,len,n,j)
INTEGER*4 m,i,len,n,j
```

Chapter 7
Debugging Programs

This chapter contains a general overview of the tools and techniques that you can use to debug FORTRAN programs. These tools include *fxref*, a cross reference generator *pmd*, a post-mortem dump analyzer *csd*, a source-level debugger *adb*, an assembly-level debugger.

7.1 General Considerations

Before using any of the debugging tools, consider the following strategies:

- If runtime errors involve segmentation violations, reserved operands, or inner ring references, try checking that array references do not exceed their array bounds. Try recompiling the program with the *-cs* option. The *-a1* option may also be helpful.
- If you are compiling at the -O3 or -O2 optimization levels, try recompiling at a lower level of optimization.
- Check that the problem is not being caused by the use of floating-point numbers. Some of the more common problems are truncation error, round-off error, and ill conditioning.

7.2 Cross-Reference Generator

The cross-reference generator (*fxref*) produces identifier cross-reference tables for FORTRAN source programs. Arguments whose names end with .f are assumed to be FORTRAN source programs. The command to invoke *fxref* is:

fxref [*options*] *filename*

The *options* are:

-iw *n* — Specifies the column width (*n*) for identifiers. The width can range from 8 to 32; the default value is 16.

-pw *n* — Specifies the logical page width used by the output formatter; the default value is 132.

-sl — Produces a source listing, with line numbers, preceding the cross-reference table.

-xr1 — Puts all objects (such as variables and arrays) into one table, rather than printing a separate table for each class of objects.

-72 — Truncate the source at column 72.

The *filename* specifies the file for which a cross-reference table is to be produced. Names longer than 32 characters are truncated. If two names differ only after the first 32 characters, they are treated as the same identifier.

The cross-reference generator produces a 5-column table. The contents of the table are as follows:

Column 1 — Name of the identifier.

Column 2 — Name of the program, function, subroutine, or block data program in which the identifier was found.

Column 3 — Data type of the identifier

R - real
I - integer
L - logical
C - character
Z - complex
blank - no type

Followed by an asterisk (*), then the width in bytes.

Column 4 — The object class (applicable if the *-xrl* option is used)

ARY - array
BLK - block data
COM - common block
DO - DO loop head
ENT - entry point
EXT - external
FUN - function
ITR - intrinsic
LAB - statement label
NML - namelist
PAR - parameter
PRG - program
STF - statement function
SUB - subroutine
VAR - variable

Column 5 — Line number and usage class of each reference

d - defined (DIMENSION, EQUIVALENCE, COMMON)
i - initialized (DATA, PARAMETER)
a - assigned
p - passed as argument to function or subroutine
blank - referenced

For further information, please refer to the *CONVEX UNIX Programmer's Manual*, Section *fxref*(1F).

7.3 Post-Mortem Dump (*pmd*)

The post-mortem dump (*pmd*) generates information to assist your debugging efforts if the program running under it aborts and dumps memory. To run a program under *pmd*, you must first compile the program using the *-db* option on the compiler command line.

The command to invoke *pmd* is:

pmd [*options*] *program* [*arguments*]

The *options* are:

-a	Dereference the address registers and print their contents in hexadecimal, decimal, and floating-point format.
-c	Display data within the specified common block only.
-d *n:n...*	Print up to *n* elements of arrays up to the given subscripts for each dimension. Up to seven dimensions can be specified; the default is 100:10:1.
-l	Display a post-mortem dump in long format.
-s	Generate a post-mortem dump in short format.
-S	Exclude the approximate source code location.
-t *time*	Limit the execution time to *time* seconds.
-v	Include the contents of the vector registers in the dump.

The parameter *program* is the name of an executable module, compiled with the *-db* option, containing the program to be run and *arguments* are the arguments to be used by the program during execution.

Depending on the option specified, *pmd* produces either a short-form dump or a long-form dump containing the information shown below. If no option is specified, the short-form dump is the default.

Short Form	The signal that caused the program to abort.
	A runtime stack backtrace.
	The approximate source line location at which the exception occurred.
Long Form	The signal that caused the program to abort.
	A runtime stack backtrace.
	The approximate source line location at which the exception occurred.
	The contents of the machine registers.
	A dump of active local variables in each routine on the runtime stack.
	A dump of global, or common, variables.
	The region of disassembled object code where the exception took place.
	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).

For further information, please refer to the *CONVEX Consultant User's Guide*

7.4 CONVEX Symbolic Debugger (*csd*)

The CONVEX symbolic debugger (*csd*) provides statement-level control of program execution and access to program variables through symbolic names. To run a program under *csd*, you must first compile the program with the *-db* option on the compiler command line.

The *csd* program provides the ability to

- Debug the program at the statement level rather than at the machine level.
- Examine core dumps to find tne exact line at which the program failed.
- Debug multiple source program modules.
- Access program variables by name rather than by absolute address.
- Debug optimized cods.

The command to invoke *csd* is

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

where

-r	Instructs *csd* to execute *objfile* immediately (without waiting for *csd* commands).
-I *dir*	Directs *csd* to add the specified directory to the list of directories searched when *csd* looks for a source file.
objfile	Is an executable file produced by the compiler with the *-db* option.
corefile	Is the pathname for a file containing a core dump generated as the result of an abnormal program termination. The *corefile* is usually 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 use it to 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.

Once you have invoked *csd*, use the *run* command to start executing the program to be debugged.

The *run* command has the following format:

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

The *run* command arguments drive the program 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 program using the last two arguments shown above. Entering <*filename* reads data from the filename specified, while entering >*filename* writes data to the file specified. There are no blanks between the symbols < and > and *filename*.

If you invoke the *run* command more than once, the variables of the program are reinitialized with each invocation before execution begins.

Table 7-1 lists some of the more commonly-used *csd* commands. For the syntax of these commands and for a complete description of all the *csd* commands, please refer to the *CONVEX Consultant User's Guide* or to the *csd*(1) man page.

Table 7-1: Commonly Used csd Commands

Command	Use
cont	Continue execution.
down	Move routine environment down the call stack.
dump	Print active variables and a stacktrace.
file	Print or change source file environment.
func	Print or change routine environment.
help	Display a list of *csd* commands.
list	List source lines in current source file.
print	Print variables, expression values, etc.
quit	Exit *csd* and return to shell.
return *s*	Stop execution when routine *s* is on top of the stack.
status	Display current trace and stop commands.
step *n*	Execute *n* source lines.
stop	Stop execution at specified point.
trace	Print trace information while program is executing.
up	Move routine environment up the stack.
use	Set list of directories to search.
whatis *s*	Print declaration of *s*.
where	Print active routines on the call stack.
whereis *s*	Print all environments where *s* exists.
which *s*	Print full environment for *s*.

7.5 Assembly-Language Debugger

The assembly-language debugger (*adb*) is an object-code debugger that requires no recompilation or special compiler options. The *adb* debugger allows you to examine core dumps from failed programs and to perform interactive debugging of programs at the assembly-language level.

Since the *adb* debugger runs programs under its control, it is always aware of the state of the program and the values of all variables. Using *adb*, you can

- Display the assembly-language instructions of the program
- Stop program execution at any point
- Examine the values of program variables
- Modify the value of any program variable
- Execute a program one line at a time
- Display the values of machine registers
- Modify the values of machine registers

The *adb* debugger can be used to debug programs at all optimization levels, including vector code and programs running on multiple processors. For a detailed description of the *adb* debugger and complete instructions on its use, please refer to the *CONVEX adb (Assembly-Language Debugger) User's Guide.*

Chapter 8
Runtime Errors and Exceptions

This chapter discusses runtime error processing and describes how the runtime library processes errors, what the defaults are, and how to override the defaults.

The runtime system contains software modules required to support features of FORTRAN that are not handled by the compiler itself. The modules that make up the runtime system are packaged in precompiled files called libraries, which are accessible by the CONVEX loader, *ld*.

During runtime, error or exception conditions may occur during I/O operations, from system-detected errors, invalid input data, arithmetic errors, or argument errors in calls to the mathematical library. The runtime library provides default processing for errors, sends the appropriate messages, and takes steps to recover from errors, if possible. To override default actions, use the following:

- ERR (error) and END (end-of-file) specifiers in I/O statements to transfer control to error-handling code within the program.
- IOSTAT (I/O status) specifier in I/O statements to identify FORTRAN-specific errors based on the values of IOSTAT.
- CONVEX signal-handling facility to modify error processing to your needs.

8.1 I/O Error Processing

When an I/O error occurs during program execution, the runtime default action is to print an error message and terminate the program with a core dump. Error numbers lower than 100 are generated by UNIX.

8.1.1 ERR and END Specifiers

To override program termination upon detection of an I/O error, use the ERR or END specifier in I/O statements to transfer control to a specified point in the program. Execution continues at the specified statement and no error message prints. For example, in a program containing the WRITE statement:

```
WRITE (8,50,ERR=400)
```

if an error occurs during its execution, the runtime library transfers control to the statement at label 400. You can also use the END specifier to treat an end-of-file condition that otherwise might be treated as an error. For example:

```
READ (12,70,END=550)
```

ERR can also be specified as a keyword in an OPEN, CLOSE, or INQUIRE statement, as in the following example:

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

Detection of an error while this statement is executing transfers control to statement 999.

8.1.2 IOSTAT Specifier

To continue program execution after an I/O error and return data on I/O operations, use the IOSTAT specifier. This specifier can augment or replace the END and ERR transfers. Executing an I/O statement containing the IOSTAT specifier suppresses printing of an error message and defines the specified integer variable or integer array element as one of the following:

- A value of -1 when an end-of-file condition occurs
- A value of 0 when no error condition or end-of-file condition occurs
- A positive integer value when an error condition occurs. This value is one of the system errors or FORTRAN I/O errors.

After the I/O statement executes and an IOSTAT is assigned a value, control transfers to the END or ERR statement label, if one exists. When no control transfer occurs, normal execution continues.

Example:

```
      READ (5,*,IOSTAT=IERR,ERR=10,END=20)I,J,K
      ...
      (process input record)
      ...
10    PRINT *,'ERROR DURING READ:',IERR
      STOP
20    PRINT *,'END OF FILE'
      STOP
```

8.2 Signals and Exceptions

This section describes the signals and exceptions that can occur at runtime.

8.2.1 Signals

A signal is generated by an abnormal event, a user at a terminal (*quit, interrupt, stop*), a program error (e.g., bus error), the request of another program (*kill*), or when a process is stopped to access its control terminal while in background mode. Signals can also be generated when a process resumes after being stopped, when the status of child process changes, or when input is ready at the control terminal.

Table 8-1 lists the name, number, and meaning of each of the runtime signals. Each signal has a default action associated with it. Except for the SIGKILL and SIGSTOP signals, the *signal* utility allows this default action to be overridden.

Table 8-1: Signal Names and Numbers

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
SIGURG	16**	Urgent condition present on socket
SIGSTOP	17***	Stop (cannot be caught or ignored)
SIGTSTP	18***	Stop signal generated from keyboard
SIGCONT	19**	Continue after stop
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 timer alarm
SIGWINCH	28	Window changed
SIGLOST	29	Resource lost
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2

In the table, * indicates that the default action for the signal is to terminate the program and produce a core dump; ** indicates that the default action is to ignore the signal; *** indicates that the default action is to stop the program. The default action for all other signals is to terminate the program.

8.2.2 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 8-2 shows the mapping of exceptions to signals and codes.

Table 8-2: Mapping Exceptions to Signals and Codes

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 trap	SIGFPE	FPE_FLTOVF_TRAP (3)
Floating division by zero	SIGFPE	FPE_FLTDIV_TRAP (4)
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP (5)
Reserved Operand trap	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	SIGSEGV	SEG_INVSDR_TRAP (4)
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP (5)
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP (6)
I/O access violation	SIGSEGV	SEG_IOACC_TRAP (7)
Ring Violations	SIGBUS (10)	
Inward 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	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 (1)
Privileged instruction	SIGILL	ILL_PRIVIN_TRAP (2)
Undefined op code	SIGILL	ILL_UNDFOP_TRAP (4)
Trace pending	SIGTRAP(5)	
Bpt instruction	SIGTRAP (5)	

8.3 Error-Processing Utilities

This section describes the general error-processing utilities you can use to attach your own signal handler, enable or disable certain arithmetic traps, and retrieve error numbers. The error-processing utilities are located in */usr/lib/libU77.a* and are described in the *CONVEX UNIX Programmer's Manual*, Section 3F.

8.3.1 *setjmp* and *longjmp* Utilities

The *setjmp* and *longjmp* utilities save and restore the stack environment and the signal mask (*sigmask*), respectively, and can be used in processing errors and interrupts encountered in a low-level subroutine. These utilities provide a mechanism for performing statement-level recovery from errors.

Example

```
integer*4 env(10), val
i = setjmp(env)
...
call longjmp(env,val)
```

The first time that *setjmp* is called, it returns a value of 0; otherwise, the value returned is the second argument to *longjmp*..

The *_setjmp* and*_longjmp* utilities save and restore the stack and registers but not the signal mask (*sigmask*). You cannot use *longjmp* to restore the environment saved by *_setjmp* and you cannot use *_longjmp* to restore the environment saved by *setjmp*.

8.3.2 *errtrap* Utility

The *errtrap* utility allows you to enable or disable signal trapping. The following signals can be trapped:

- integer overflow
- floating-point underflow
- intrinsic errors
- integer divide by zero
- floating-point divide by zero
- floating-point overflow
- reserved operand fault

The *errtrap* utility enables or disables trapping for the designated errors by setting or resetting the appropriate bits in the process status word. If trapping is enabled for a particular error and that error occurs, signal SIGFPE is sent to the process. Upon completion of *errtrap*, the previous value of the flags is restored.

The argument to *errtrap* is produced by summing the appropriate flags from the following table:

Flag	Meaning	Default
'01'X	Trap integer overflow	off
'02'X	Trap floating underflow	off
'04'X	Trap intrinsic errors	off
'08'X	Trap integer divide by zero	on
'10'X	Trap floating point (divide by zero, overflow, reserved operand fault)	on

By default (if you do not use *errtrap*), the following error traps are enabled during execution: integer divide by zero, floating-point overflow, floating-point divide by zero, and reserved operand fault. By default, the following traps are disabled during execution: floating-point underflow, integer overflow, and intrinsic errors.

The *errtrap* utility supersedes *traper*, which is maintained for upward compatibility purposes.

Example

The following code enables integer overflow and floating-point underflow and disables intrinsic errors, integer divide by zero, and floating-point trap:

```
i = errtrap('3'X)
```

8.3.3 *signal* Utility

The *signal* utility allows you to designate a signal-handling routine. The *signal* utility has three parameters: the signal number, the condition handler, and a flag.

Example 1:

```
sigf = signal(18, sigdie,-1)
```

This statement establishes the condition handler, *sigdie*, for stop signals generated from the keyboard. The old handler is returned in *sigf*.

Example 2:

```
sigf = signal(18,0,0)
```

This statement restores the default action for stop signals generated from the keyboard.

Example 3:

```
sigf = signal(18,0,1)
```

This statement causes stop signals generated from the keyboard to be ignored.

8.3.4 *traceback* Utility

The *traceback* utility prints out the names of the routines currently in the call stack. Control then returns to the calling program.

Example 1:

This example shows *traceback* as the result of an exception.

```
*** Floating Point Exception: Floating divide by zero: at 800010d8.

signal(8,4,8002bf84,80001262) from ffffd084 [ap = 8002bf74]
curbrk+6d0(80018000,80018004) from 800010d8 [ap = 800010fc]
_MAIN__() from 80001230 [ap = ffffce54]
_main(1,ffffce98,ffffcea0) from 80001074 [ap = ffffce8c]
```

Example 2:

This example shows *traceback* called by the user.

```
_sub2_(80001148,80001154) from 800010de [ap = ffffcde0]
_sub1_(80001148) from 800010ba [ap = 80001150]
_MAIN__() from 800012f4 [ap = ffffce1c]
_main(1,ffffce60,ffffce68) from 80001074 [ap = ffffce54]
```

8.3.5 *traper* Utility

The *traper* utility enables traps for floating-point underflow and integer overflow by setting status bits in the process status word. (Also see the description of the *errtrap* utility.)

If you enable trapping for integer overflow and integer overflow condition occurs, signal SIGFPE is sent. Likewise, if you enable trapping for floating-point overflow and floating-point overflow occurs, signal SIGFPE is sent. By default, an intrinsic error does not terminate execution of the program. If the intrinsic error trap is set, however, the program terminates on the first intrinsic error encountered. The intrinsic error trap value applies to the entire program.

NOTE

Integer overflow is not detected for multiplies and divides of the constants 2, 4, and 8 at optimization levels -O0, -O1, and -O2.

The argument to *traper* is produced by summing the appropriate flags from the following table:

Flag	Meaning
'1'X	Trap integer overflow
'2'X	Trap floating underflow
'4'X	Trap intrinsic errors

If you enable trapping of math errors, when a math error occurs an error message is printed to *stderr* and the program terminates with a core dump. Using the default setting (trap disabled) causes an error message to print and the program to continue with an appropriate default value. For example, using the default setting as shown in the following example produces an error message but no core dump.

```
x = -1
y = sqrt(x)
print *, 'x: ', 'y: ', y
```

The following message is produced:

```
mth$r_sqrt:  [300] square root undefined for negative values
x:    -1.0000000      y:    1.000000
```

Enabling the trap as shown below, however, produces an error message and terminates the program with a core dump.

```
x = -1
i = traper (4)
y = sqrt(x)
print *, 'x: ', 'y: ', y
end
```

The following message is produced:

```
mth$r_sqrt:  [300] square root undefined for negative values
Illegal instruction (core dumped)
```

8.3.6 *perror*, *gerror*, and *ierrno* Utilities

The *perror*, *gerror*, and *ierrno* utilities retrieve the system error message numbers. *perror* writes a message to FORTRAN unit 0 appropriate to the last detected system error. *gerror* returns the system error message in a character variable and may be called either as a subroutine or as a function. *ierrno* returns the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call that indicates what caused the error condition.

8.4 Examples of Signal Handling

The following examples illustrate the use of the error processing utilities.

Example 1:

```
c This example establishes a signal handler for interrupts
c
       integer signal      ! integer function
       integer oldhandler  ! save old signal value
       integer newhandler  ! new handler address
       external newhandler
       oldhandler = signal (2, newhandler, -1)
       print *,'Hit control-C (^C) to generate a SIGINT signal...'
       read *,i            ! wait here until user enters ^c
       end

c Subroutine to intercept signals
c
       subroutine newhandler (sig, code, scp)
       integer sig         ! signal number
       integer code        ! signal subcode
       integer scp(5)      ! signal context
                           ! (1) /* sigstack state to restore */
                           ! (2) /* signal mask to restore */
                           ! (3) /* sp to restore */
                           ! (4) /* pc to restore */
                           ! (5) /* psw to restore */
       write (*,00002) sig,code,scp(4)
00002  format(/,
     $         ' Signal number [SIGINT].....:',I10,/,
     $         ' Signal lsubcode [0].......:',I10,/,
     $         ' Program counter [pc].....:',Z10,/,
     $         ' ---------------------------------')
       end
```

Example 2:

```
       program pr3527f
       external sighandler
       integer*4 env(10),i,code,setjmp,longjmp
       common env

c Establish a signal handler for arithmetic exceptions
       oldhan = signal(8,sighandler,-1)

c Establish an environment for recovery in case an error occurs
c within the routine work.  In case of error, routine fixup is
c called to repair the program state so execution may continue.
       i = setjmp(env)

c Initially, setjmp returns 0.  If a subsequent longjump is performed,
c the value returned is the second argument to longjump.
c Returning a value of zero is not recommended.
       if (i .eq. 0) then
         call work ()
       else
```

```
        call fixup ()
      endif
     ...
      end

      subroutine sighandler (sig,code,scp)
c Intercept (SIGFPE) floating point exceptions
      integer*4 sig,code,scp(5),env(10)
      common env

c Return the error subcode and execute global goto
      call longjump(env,code)
      end

      subroutine work
      integer a,b,c
      print *,'Doing meaningful work.'
      read *,a,b,c
      a = b/c
      end

      subroutine fixup
      print *,'Fixing results.'
      end
```

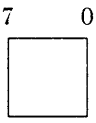
A

FORTRAN Data Representations

This appendix describes the data types supported by CONVEX FORTRAN and shows how each is stored in memory. The numbers on top of the figures are the bit ordering; the numbers on the bottom are the byte ordering.

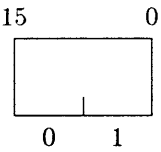
A.1 Logical Representation

LOGICAL*1

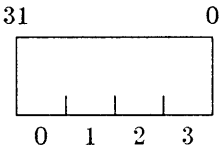


.FALSE. = all 0's
.TRUE. = all 1's

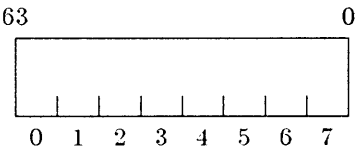
LOGICAL*2



LOGICAL*4



LOGICAL*8

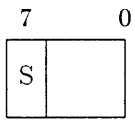


The leftmost byte (8 bits) is always stored in memory at the lowest byte address.

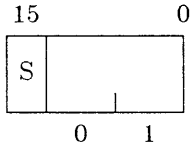
A.2 Integer Representation

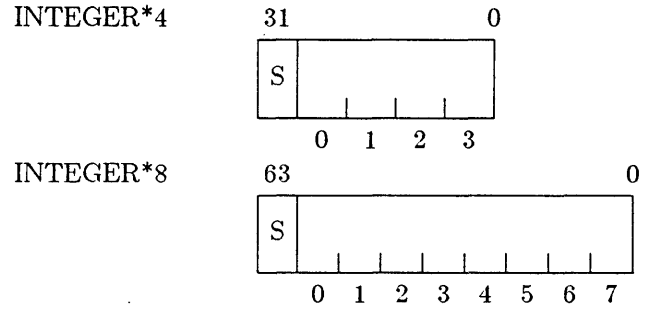
Integer data is declared with the INTEGER*1 (BYTE), INTEGER*2, INTEGER*4, and INTEGER*8 keywords. In the internal representations, the sign bit (S) is 0 for positive integers and 1 for negative integers. INTEGER data types use the two's complement format.

INTEGER*1
BYTE



INTEGER*2





INTEGER*1 values are in the range -128 to +127. INTEGER*2 values are in the range -32,768 to +32,767. INTEGER*4 values are in the range -2,147,483,648 to +2,147,483,647. INTEGER*8 values are in the range -2^{63} to $+2^{63}-1$.

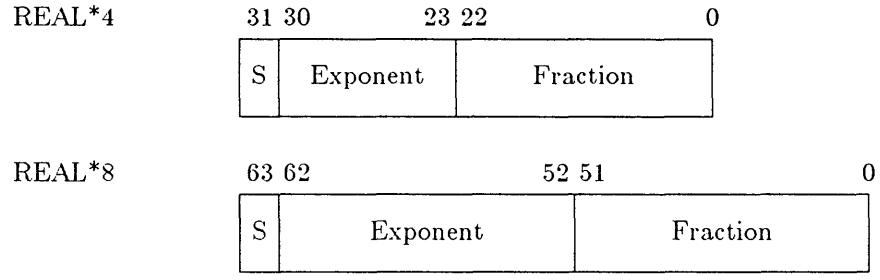
A.3 Real Data Representation

Single-precision (32-bit) floating-point variables are declared with the REAL*4 keyword; double-precision (64-bit) floating-point variables are declared with the REAL*8 keyword. Both types of floating-point data can be represented in either native format or in IEEE format. If you want to process floating-point data in IEEE mode, your machine must be equipped with the IEEE support hardware.

NOTE

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

The following figure shows the internal representations of single-precision and 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.



A.3.1 Native Floating-Point

The CONVEX native floating-point representation defines the following types of operands:

Operand	Explanation
Normalized	The exponent is not all zeros or all ones.
Reserved operand(Rop)	The exponent is 0, the sign is 1, and the fraction can have any value.
Zero	The exponent is 0, the sign is 0, and the fraction can have any value. True zero has a sign of 0, an exponent of 0, and a fraction of 0.

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

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

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

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

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

In the internal representation the sign bit (S) 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.3.2 IEEE Floating-Point

The CONVEX IEEE floating-point representation defines the following types of operands:

Operand	Explanation
Normalized	The exponent is not all zeros or all ones.
Denormalized	The exponent is 0, the fraction is nonzero, and the sign is 0 or 1. This number is always treated as true zero.
Not a number (NaN)	The exponent is all ones, the fraction is nonzero, and the sign is 0 or 1.
Infinity (Inf)	The exponent is all ones, the fraction is 0, and the sign is 0 or 1.

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

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

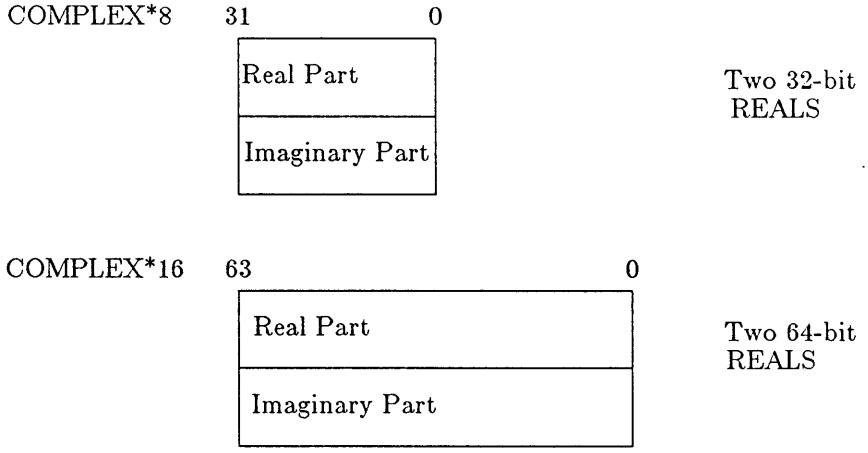
In the internal representation the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has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

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

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

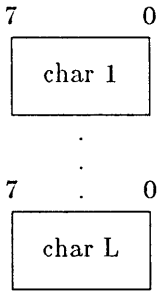
In the internal representation the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and 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.4 Complex Representation



A.5 Character Representation

A character string is stored internally as a sequence of bytes.



A character constant is limited to 4000 characters. Character strings formed at runtime may be of arbitrary length.

A.6 Hollerith Representation

A Hollerith constant is stored internally as a sequence of bytes and is limited to 2000 characters.

B

Compiler and Runtime Messages

The CONVEX FORTRAN compiler 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. An abort can also occur during optimization, e.g., integer truncation during constant folding.

B.1 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: >& file name. If you do not redirect the messages, they appear on your screen.

Example:

```
fc file.f
```

sends the messages to the screen, while

```
fc file.f >&out
```

sends the messages to the file *out*. You may also use the *error* utility to insert diagnostic messages into your source file, where they appear as comments. This is a particularly convenient way to find the bugs while editing your source file.

Example:

```
fc foo.f |& error
```

This command compiles *foo.f* and pipes the standard output and standard error output to the error utility, which then inserts the diagnostic messages back in the source file *foo.f*. 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. Refer to the UNIX documentation supplied with the system for details on writing *csh* scripts.

B.2 Runtime Error Messages

The runtime library reports errors encountered during execution. Runtime errors can be system-detected, arithmetic, or I/O errors. The runtime library provides default error processing and generates the necessary error messages to the user. All error messages are written to unit 0, *stderr*.

B.2.1 System Errors

System errors can be returned either by the FORTRAN I/O library or by the FORTRAN utility library. In the former case, system errors are in the form of an I/O error message; in the latter case, the error number is returned as the value of the utility function (see Section 3F of the *CONVEX UNIX Programmer's Manual*). The system errors generated by the UNIX operating system are described in the *CONVEX UNIX Programmer's Manual, Part II* under the section *INTRO(2)*.

B.2.2 I/O Errors Generated by Runtime Library

The following system error messages are generated by the FORTRAN I/O runtime library:

100 error in format

See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested (), or an extremely long format statement.

101 illegal unit number

You cannot close logical unit 0. Valid unit numbers are in the range 0 to 255.

102 formatted io not allowed

The logical unit was opened for unformatted I/O.

103 unformatted io not allowed

The logical unit was opened for formatted I/O.

104 direct io not allowed

The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 sequential io not allowed

The logical unit was opened for direct-access I/O.

106 can't backspace file

The file associated with the logical unit cannot seek. May be a device or a pipe.

107 off beginning of record

The format specified a left tab off the beginning of the record.

108 can't stat file

The system cannot return status information about the file. Perhaps the directory is unreadable.

109 no * after repeat count

Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

110 off end of record

A formatted write tried to go beyond the logical end-of-record. An unformatted read or write also causes this.

112 incomprehensible list input

Bad input data for list-directed read.

113 out of free space

The library dynamically creates buffers for internal use. Not enough memory was available at the time of the request.

114 unit not connected

The logical unit was not open.

115 read unexpected character

Certain format conversions cannot tolerate nonnumeric data. Logical data must be T or F.

116 blank logical input field

117 'new' file exists

You tried to open an existing file with status='new'.

118 can't find 'old' file

You tried to open a nonexistent file with status='old'.

119 unknown system error

Contact the Technical Assistance Center (TAC).

120 requires seek ability

Direct-access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 illegal argument

Certain arguments to OPEN, etc., are checked for legitimacy. Often only nondefault forms are looked for.

122 negative repeat count

The repeat count for list-directed input must be a positive integer.

123 illegal operation for unit

124 new record not allowed

Encode and decode can only write and read single records.

125 numeric keyword variable overflowed

A keyword variable such as ASSOCIATEVARIABLE overflowed.

126 record number is out of range

A direct-access was attempted to a record number less than one or greater than MAXREC specified in the OPEN statement.

127 file is read-only

Writing is not permitted to files opened with the READONLY keyword.

128 variable record format not allowed

Direct-access files may not have variable-length records.

129 record length exceeded

A read was attempted past the end of a record in a sequentially accessed file with RECL set on OPEN.

130 exceeds maximum number of open files

A maximum of 255 files may be open at one time. This is a system-dependent limit.

131 data type size too small for REAL

Format code of variables less than 4 bytes cannot be read or written with the E, O, F, or G format code.

132 infinite loop in format

133 fixed record type not allowed for print files

134 attempt to read nonexistent record

Returned for direct-access reads when an attempt is made to read a record that does not exist.

135 reopening file with different unit not allowed

136 io list item type is incompatible with format code

137 unknown record length

A record length must be specified for the file.

138 asynchronous io not allowed on this file

139 synchronous io not allowed on this file

140 incompatible format structure - recompile

The internal representation of parsed format strings has changed. The routine must be recompiled.

141 namelist error

An error has been detected in the use of namelist-directed I/O.

142 apparent recursive logical name definition

143 recursive input/output operations

144 out of free space, possibly from performing unformatted I/O

145 Error in conversion of string to numeric

C

Runtime Libraries

This appendix describes the FORTRAN intrinsic library, the CONVEX math library, and the FORTRAN input/output (I/O) library.

C.1 FORTRAN Intrinsic Library and CONVEX Math Library

This section summarizes the runtime entry points in the FORTRAN Intrinsic Library and the CONVEX Math Library. These libraries include runtimes for FORTRAN intrinsics, mathematical programmed operators, and character-string programmed operators. They are loaded automatically by the FORTRAN compiler.

C.1.1 Calling Conventions

The CONVEX Math Library runtimes are accessible to all language processors. The functions must be called via the *callq/rtnq* mechanism (in assembly language) with arguments passed by value in the scalar and/or vector registers and function results returned in the appropriate type register(s). Runtimes, which accept multiple vector arguments, e.g., complex division, restrict all arguments to the same length, and the length of the resultant vector is the same as the argument vector length.

Complex values are represented as pairs of registers, with the real part in the low-order register and the imaginary part in the high-order one. Since vector arguments and results are passed in registers, vector lengths are restricted to a maximum of 128. This requires strip-mining by the compiler prior to a vector runtime call.

The functions in *libF77* use the standard FORTRAN calling conventions. For many intrinsics, there are entry points in both the intrinsic library and the math library. The *libF77* entry points are provided for compatibility with the standard FORTRAN calling convention and must be used when intrinsics are referenced as dummy arguments. In most cases, *libF77* routines do not perform the intrinsic operation but call the corresponding CONVEX math runtime. Not all intrinsics require runtime routines. For example, scalar truncation can be accomplished with a single convert instruction, and, as such, is implemented as inline code.

C.1.2 Function-Naming Convention

The runtime names are constructed as follows:

<prefix><argument-type(s)>_<function-type>_<result-type>

The prefix is either:

mth$ (CONVEX Math Library) or for$ (FORTRAN Intrinsic Library)

The function-type is typically the generic intrinsic name. For example, *mth$d_sqrt* is the REAL*8 scalar square root entry point in the CONVEX Math Library.

Scalar intrinsics have entry points in both the FORTRAN Intrinsic Library ("for$" prefix) and the CONVEX Math Library ("mth$" prefix); vector intrinsics have entry points only in the math library.

If the argument and result types are the same, the result type is omitted from the function name. If the function has multiple arguments all of the same type, then a single argument code is used rather than a sequence of codes.

The codes in Table C-2 are used to indicate the result and argument types.

Table C-1: Function Naming Convention

Code	Type of Result/Arguments
h	INTEGER*1
i	INTEGER*2
j	INTEGER*4
k	INTEGER*8
r	REAL*4
d	REAL*8
c	COMPLEX*8
z	COMPLEX*16
s	CHARACTER*N
l	LOGICAL*4
vh	vector of INTEGER*1
vi	vector of INTEGER*2
vj	vector of INTEGER*4
vk	vector of INTEGER*8
vr	vector of REAL*4
vd	vector of REAL*8
vc	vector of COMPLEX*8
vz	vector of COMPLEX*16
vm	vector mask register

When multiple arguments of different types are used, the order of the arguments conforms to the intrinsic definition. For example, the scalar/vector KISHIFT intrinsic is implemented with the following runtime:

mth$jvj_shft	performs a logical shift an INTEGER*4 scalar by an INTEGER*4 vector and returns the result as an INTEGER*4 vector.

This runtime is used in the following situation:

```
DO I=1,N
   K(I) = KISHFT(L,M(I))
ENDDO
```

C.1.3 Intrinsic Runtimes

Table C-3 summarizes calling sequences for the intrinsic runtimes. Braces { } indicate the prefixes that are available for each runtime. The specific intrinsic names are provided for cross-reference purposes. There is not always a one-to-one correspondence between intrinsic references and runtimes. In some cases, two different intrinsics generate a call to the same runtime (these are separated by commas) or the application of multiple intrinsics generates a call to a single runtime (this is denoted by the use of parentheses). Some of the runtimes listed in this table are used as programmed operators. For example, the assignment of an INTEGER*4 vector to a REAL*4 vector generates a call to a *mth$vj_cvt_vr*.

Table C-2: Intrinsic Functions

Intrinsic	Runtime Name	Arguments	Result
Square root			
SQRT	{for,mth}$r_sqrt	r	r
	mth$vr_sqrt	vr	vr
DSQRT	{for,mth}$d_sqrt	d	d
	mth$vd_sqrt	vd	vd
CSQRT	{for,mth}$c_sqrt	c	c
	mth$vc_sqrt	vc	vc
CDSQRT	{for,mth}$z_sqrt	z	z
	mth$vz_sqrt	vz	vz
Natural logarithm			
LOG	{for,mth}$r_log	r	r
	mth$vr_log	vr	vr
DLOG	{for,mth}$d_log	d	d
	mth$vd_log	vd	vd
CLOG	{for,mth}$c_log	c	c
	mth$vc_log	vc	vc
CDLOG	{for,mth}$z_log	z	z
	mth$vz_log	vz	vz
Common logarithm			
LOG10	{for,mth}$r_log10	r	r
	mth$vr_log10	vr	vr
DLOG10	{for,mth}$d_log10	d	d
	mth$vd_log10	vd	vd

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Exponential			
EXP	{for,mth}$r_exp	r	r
	mth$vr_exp	vr	vr
DEXP	{for,mth}$d_exp	d	d
	mth$vd_exp	vd	vd
CEXP	{for,mth}$c_exp	c	c
	mth$vc_exp	vc	vc
CDEXP	{for,mth}$z_exp	z	z
	mth$vz_exp	vz	vz
Sine			
SIN	{for,mth}$r_sin	r	r
	mth$vr_sin	vr	vr
DSIN	{for,mth}$d_sin	d	d
	mth$vd_sin	vd	vd
CSIN	{for,mth}$c_sin	c	c
	mth$vc_sin	vc	vc
CDSIN	{for,mth}$z_sin	z	z
	mth$vz_sin	vz	vz
Sine (degree)			
SIND	{for,mth}$r_sind	r	r
	mth$vr_sind	vr	vr
DSIND	{for,mth}$d_sind	d	d
	mth$vd_sind	vd	vd
Cosine			
COS	{for,mth}$r_cos	r	r
	mth$vr_cos	vr	vr
DCOS	{for,mth}$d_cos	d	d
	mth$vd_cos	vd	vd
CCOS	{for,mth}$c_cos	c	c
	mth$vc_cos	vc	vc
CDCOS	{for,mth}$z_cos	z	z
	mth$vz_cos	vz	vz
Cosine (degree)			
COSD	{for,mth}$r_cosd	r	r
	mth$vr_cosd	vr	vr
DCOSD	{for,mth}$d_cosd	d	d
	mth$vd_cosd	vd	vd
Tangent			
TAN	{for,mth}$r_tan	r	r
	mth$vr_tan	vr	vr
DTAN	{for,mth}$d_tan	d	d
	mth$vd_tan	vd	vd
CTAN	{for,mth}$c_tan	c	c
	mth$vc_tan	vc	vc
CDTAN	{for,mth}$z_tan	z	z
	mth$vz_tan	vz	vz

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Tangent (degree)			
TAND	{for,mth}$r_tand	r	r
	mth$vr_tand	vr	vr
DTAND	{for,mth}$d_tand	d	d
	mth$vd_tand	vd	vd
Arc sine			
ASIN	{for,mth}$r_asin	r	r
	mth$vr_asin	vr	vr
DASIN	{for,mth}$d_asin	d	d
	mth$vd_asin	vd	vd
Arc sine (degree)			
ASIND	{for,mth}$r_asind	r	r
	mth$vr_asind	vr	vr
DASIND	{for,mth}$d_asind	d	d
	mth$vd_asind	vd	vd
Arc cosine			
ACOS	{for,mth}$r_acos	r	r
	mth$vr_acos	vr	vr
DACOS	{for,mth}$d_acos	d	d
	mth$vd_acos	vd	vd
Arc cosine (degree)			
ACOSD	{for,mth}$r_acosd	r	r
	mth$vr_acosd	vr	vr
DACOSD	{for,mth}$d_acosd	d	d
	mth$vd_acosd	vd	vd
Arc tangent			
ATAN	{for,mth}$r_atan	r	r
	mth$vr_atan	vr	vr
DATAN	{for,mth}$d_atan	d	d
	mth$vd_atan	vd	vd
Arc tangent (degree)			
ATAND	{for,mth}$r_atand	r	r
	mth$vr_atand	vr	vr
DATAND	{for,mth}$d_atand	d	d
	mth$vd_atand	vd	vd
Arc tangent with two arguments			
ATAN2	{for,mth}$r_atan2	r,r	r
	mth$vr_atan2	vr,vr	vr
	mth$vrr_atan2	vr,r	vr
	mth$rvr_atan2	r,vr	vr
DATAN2	{for,mth}$d_atan2	d,d	d
	mth$vd_ata2n	vd,vd	vd
	mth$vdd_atan2	vd,d	vd
	mth$dvd_atan2	d,vd	vd

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Arc tangent with two arguments (degree)			
ATAN2D	{for,mth}$r_atan2d	r,r	r
	mth$vr_atan2d	vr,vr	vr
	mth$vrr_atan2d	vr,r	vr
	mth$rvr_atan2d	r,vr	vr
DATAN2D	{for,mth}$d_atan2d	d,d	d
	mth$vd_atan2d	vd,vd	vd
	mth$vdd_atand2	vd,d	vd
	mth$dvd_atand2	d,vd	vd
Hyperbolic sine			
SINH	{for,mth}$r_sinh	r	r
	mth$vr_sinh	vr	vr
DSINH	{for,mth}$d_sinh	d	d
	mth$vd_sinh	vd	vd
Hyperbolic cosine			
COSH	{for,mth}$r_cosh	r	r
	mth$vr_cosh	vr	vr
DCOSH	{for,mth}$d_cosh	d	d
	mth$vd_cosh	vd	vd
Hyperbolic tangent			
TANH	{for,mth}$r_tanh	r	r
	mth$vr_tanh	vr	vr
DTANH	{for,mth}$d_tanh	d	d
	mth$vd_tanh	vd	vd
Absolute value			
IIABS	{for,mth}$i_abs	i	i
	mth$vi_abs	vi	vi
JIABS	{for,mth}$j_abs	j	j
	mth$vj_abs	vj	vj
KIABS	{for,mth}$k_abs	k	k
	mth$vk_abs	vk	vk
ABS	{for,mth}$r_abs	r	r
	mth$vr_abs	vr	vr
DABS	{for,mth}$d_abs	d	d
	mth$vd_abs	vd	vd
CABS	{for,mth}$c_abs_r	c	r
	mth$vc_abs_vr	vc	vr
CDABS	{for,mth}$z_abs_d	z	d
	mth$vz_abs_vd	vz	vd
Float-to-fix conversion			
INT1(IINT)	mth$vr_cvt_vh	vr	vh
IINT,IIFIX	{for,mth}$r_cvt_i	r	i
	mth$vr_cvt_vi	vr	vi
JINT,JIFIX	{for,mth}$r_cvt_j	r	j
	mth$vr_cvt_vj	vr	vj
KINT,KIFIX	{for,mth}$r_cvt_k	r	k
	mth$vr_cvt_vk	vr	vk
INT1(IIDINT)	mth$vd_cvt_vh	vd	vh
IIDINT,IIDINT	{for,mth}$d_cvt_i	d	i
	mth$vd_cvt_vi	vd	vi

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
JIDINT,JIDINT	{for,mth}$d_cvt_j	d	j
	mth$vd_cvt_vj	vd	vj
KIDINT,KIDINT	{for,mth}$d_cvt_k	d	k
	mth$vd_cvt_vk	vd	vk
Integer conversion			
INT1	mth$vi_cvt_vh	vi	vh
	mth$vj_cvt_vh	vj	vh
	mth$vk_cvt_vh	vk	vh
INT2	mth$vh_cvt_vi	vh	vi
	mth$vj_cvt_vi	vj	vi
	mth$vk_cvt_vi	vk	vi
INT4	mth$vh_cvt_vj	vh	vj
	mth$vi_cvt_vj	vi	vj
	mth$vk_cvt_vj	vk	vj
INT8	mth$vh_cvt_vk	vh	vk
	mth$vi_cvt_vk	vi	vk
	mth$vj_cvt_vk	vj	vk
Nearest integer			
ININT	{for,mth}$r_nint_i	r	i
	mth$vr_nint_vi	vr	vi
JNINT	{for,mth}$r_nint_j	r	j
	mth$vr_nint_vj	vr	vj
KNINT	{for,mth}$r_nint_k	r	k
	mth$vr_nint_vk	vr	vk
IIDNNT	{for,mth}$d_nint_i	d	i
	mth$vd_nint_vi	vd	vi
JIDNNT	{for,mth}$d_nint_j	d	j
	mth$vd_nint_vj	vd	vj
KIDNNT	{for,mth}$d_nint_k	d	k
	mth$vd_nint_vk	vd	vk
ANINT	{for,mth}$r_nint	r	r
	mth$vr_nint	vr	vr
DNINT	{for,mth}$d_nint	d	d
	mth$vd_nint	vd	vd
Fix-to-float conversion			
FLOATI (INT2)	mth$vh_cvt_vr	vh	vr
DFLOTI (INT2)	mth$vh_cvt_vd	vh	vd
FLOATI	{for,mth}$i_cvt_r	i	r
	mth$vi_cvt_vr	vi	vr
DFLOTI	{for,mth$i_cvt_d	i	d
	mth$vi_cvt_vd	vi	vd
FLOATJ	{for,mth}$j_cvt_r	j	r
	mth$vj_cvt_vr	vj	vr
DFLOTJ	{for,mth}$j_cvt_d	j	d
	mth$vj_cvt_vd	vj	vd
FLOATK	{for,mth}$k_cvt_r	k	r
	mth$vk_cvt_vr	vk	vr
DFLOTK	{for,mth}$k_cvt_d	k	d
	mth$vk_cvt_vd	vk	vd

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Integer part of real			
AINT	{for,mth}$r_int	r	r
	mth$vr_int	vr	vr
DINT	{for,mth}$d_int	d	d
	mth$vd_int	vd	vd
REAL*4 to REAL*8 conversion			
DBLE	{for,mth}$r_cvt_d	r	d
	mth$vr_cvt_vd	vr	vd
REAL*8 to REAL*4 conversion			
SNGL	{for,mth}$d_cvt_r	d	r
	mth$vd_cvt_vr	vd	vr
Real part of complex			
REAL,SNGL	{for}$c_real	c	c
DREAL,DBLE	{for}$z_real	z	z
Imaginary part of complex			
AIMAG	{for}$c_imag	c	c
DIMAG	{for}$z_imag	z	z
Complex conjugate			
CONJG	{for}$c_conjg	c	c
DCONJG	{for}$z_conjg	z	z
Maximum (pairwise operation—not a reduction)			
IMAX0	mth$vi_max	vi,vi	vi
	mth$vii_max	vi,i	vi
JMAX0	mth$vj_max	vj,vj	vj
	mth$vjj_max	vj,j	vj
KMAX0	mth$vk_max	vk,vk	vk
	mth$vkk_max	vk,k	vk
AMAX1	mth$vr_max	vr,vr	vr
	mth$vrr_max	vr,r	vr
DMAX1	mth$vd_max	vd,vd	vd
	mth$vdd_max	vd,d	vd
Minimum (pairwise operation— not a reduction)			
IMIN0	mth$vi_min	vi,vi	vi
	mth$vii_min	vi,i	vi
JMIN0	mth$vj_min	vj,vj	vj
	mth$vjj_min	vj,j	vj
KMIN0	mth$vk_min	vk,vk	vk
	mth$vkk_min	vk,k	vk
AMIN1	mth$vr_min	vr,vr	vr
	mth$vrr_min	vr,r	vr
DMIN1	mth$vd_min	vd,vd	vd
	mth$vdd_min	vd,d	vd

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
REAL*8 product of REAL*4's			
DPROD	{for,mth}$r_prod_d	r	d
	mth$vr_prod_vd	vr	vd
	mth$vrr_prod	vr,r	vd
Positive difference			
IIDIM	{for,mth}$i_dim	i,i	i
	mth$vi_dim	vi,vi	vi
	mth$vii_dim	vi,i	vi
	mth$ivi_dim	i,vi	vi
JIDIM	{for,mth}$j_dim	j,j	j
	mth$vj_dim	vj,vj	vj
	mth$vjj_dim	vj,j	vj
	mth$jvj_dim	j,vj	vj
KIDIM	{for,mth}$k_dim	k,k	k
	mth$vk_dim	vk,vk	vk
	mth$vkk_dim	vk,k	vk
	mth$kvk_dim	k,vk	vk
DIM	{for,mth}$r_dim	r,r	r
	mth$vr_dim	vr,vr	vr
	mth$vrr_dim	vr,r	vr
	mth$rvr_dim	r,vr	vr
DDIM	{for,mth}$d_dim	d,d	d
	mth$vd_dim	vd,vd	vd
	mth$vdd_dim	vd,d	vd
	mth$dvd_dim	d,vd	vd
Remainder			
IMOD	{for,mth}$i_mod	i,i	i
	mth$vi_mod	vi,vi	vi
	mth$vii_mod	vi,i	vi
	mth$ivi_mod	i,vi	vi
JMOD	{for,mth}$j_mod	j,j	j
	mth$vj_mod	vj,vj	vj
	mth$vjj_mod	vj,j	vj
	mth$jvj_mod	j,vj	vj
KMOD	{for,mth}$k_mod	k,k	k
	mth$vk_mod	vk,vk	vk
	mth$vkk_mod	vk,k	vk
	mth$kvk_mod	k,vk	dvk
AMOD	{for,mth}$r_mod	r,r	r
	mth$vr_mod	vr,vr	vr
	mth$vrr_mod	vr,r	vr
	mth$rvr_mod	r,vr	vr
DMOD	{for,mth}$d_mod	d,d	d
	mth$vd_mod	vd,vd	vd
	mth$vdd_mod	vd,d	vd
	mth$dvd_mod	d,vd	vd

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Transfer of sign			
IISIGN	{for,mth}$i_sign	i,i	i
	mth$vi_sign	vi,vi	vi
	mth$vii_sign	vi,i	vi
	mth$ivi_sign	i,vi	vi
JISIGN	{for,mth}$j_sign	j,j	j
	mth$vj_sign	vj,vj	vj
	mth$vjj_sign	vj,j	vj
	mth$jvj_sign	j,vj	vj
KISIGN	{for,mth}$k_sign	k,k	k
	mth$vk_sign	vk,vk	vk
	mth$vkk_sign	vk,k	vk
	mth$kvk_sign	k,vk	vk
SIGN	{for,mth}$r_sign	r,r	r
	mth$vr_sign	vr,vr	vr
	mth$vrr_sign	vr,r	vr
	mth$rvr_sign	r,vr	vr
DSIGN	{for,mth}$d_sign	d,d	d
	mth$vd_sign	vd,vd	vd
	mth$vdd_sign	vd,d	vd
	mth$dvd_sign	d,vd	vd
Bitwise AND			
IIAND	{for,mth}$i_and	i,i	i
JIAND	{for,mth}$j_and	j,i	j
KIAND	{for,mth}$k_and	k,k	k
Bitwise OR			
IIOR	{for,mth}$i_or	i,i	i
JIOR	{for,mth}$j_or	j,j	j
KIOR	{for,mth}$k_or	k,k	k
Bitwise XOR			
IIEOR	{for,mth}$i_xor	i,i	i
JIEOR	{for,mth}$j_xor	j,j	j
KIEOR	{for,mth}$k_xor	k,k	k
Bitwise complement			
INOT	{for,mth}$i_not	i,i	i
JNOT	{for,mth}$j_not	j,j	j
KNOT	{for,mth}$k_not	k,k	k
Bitwise shift			
IISHFT	{for,mth}$i_shft	i,i	i
	mth$vi_shft	vi,vi	vi
	mth$ivi_shft	i,vi	vi
JISHFT	{for,mth}$j_shft	j,j	j
	mth$vj_shft	vj,vj	vj
	mth$jvj_shft	j,vj	vj
KISHFT	{for,mth}$k_shft	k,k	k
	mth$vk_shft	vk,vk	vk
	mth$kvk_shft	k,vk	vk

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Bitwise extract			
IIBITS	{for,mth}$i_bits	i,i,i	i
	mth$vi_bits	vi,vi,vi	vi
	mth$vivii_bits	vi,vi,i	vi
	mth$viivi_bits	vi,i,vi	vi
	mth$viii_bits	vi,i,i	vi
JIBITS	{for,mth}$j_bits	j,j,j	j
	mth$vj_bits	vj,vj,vj	vj
	mth$vjvjj_bits	vj,vj,j	vj
	mth$vjjvj_bits	vj,j,vj	vj
	mth$vjjj_bits	vj,j,j	vj
KIBITS	{for,mth}$k_bits	k,k,k	k
	mth$vk_bits	vk,vk,vk	vk
	mth$vkvkk_bits	vk,vk,k	vk
	mth$vkkvk_bits	vk,k,vk	vk
	mth$vkkk_bits	vk,k,k	vk
Bitwise set			
IIBSET	{for,mth}$i_set	i,i	i
	mth$vi_set	vi,vi	vi
	mth$vii_set	vi,i	vi
	mth$ivi_set	i,vi	vi
JIBSET	{for,mth}$j_set	j,j	j
	mth$vj_set	vj,vj	vj
	mth$vjj_set	vj,j	vj
	mth$jvj_set	j,vj	vj
KIBSET	{for,mth}$k_set	k,k	k
	mth$vk_set	vk,vk	vk
	mth$vkk_set	vk,k	vk
	mth$kvk_set	vk,vk	vk
Bitwise test			
BITEST	{for,mth}$i_test	i,i	i
	mth$vi_test	vi,vi	vi
	mth$vii_test	vi,i	vi
	mth$ivi_test	i,vi	vi
BJTEST	{for,mth}$j_test	j,j	j
	mth$vj_test	vj,vj	vj
	mth$vjj_test	vj,j	vj
	mth$jvj_test	j,vj	vj
BKTEST	{for,mth}$k_test	k,k	k
	mth$vk_test	vk,vk	vk
	mth$vkk_test	vk,k	vk
	mth$kvk_test	k,vk	vk

Table C-3: Intrinsic Functions (continued)

Intrinsic	Runtime Name	Arguments	Result
Bitwise clear			
IIBCLR	{for,mth}$i_clr	i,i	i
	mth$vi_clr	vi,vi	vi
	mth$vii_clr	vi,i	vi
	mth$ivi_clr	i,vi	vi
JIBCLR	{for,mth}$j_clr	j,j	j
	mth$vj_clr	vj,vj	vj
	mth$vjj_clr	vj,j	vj
	mth$jvj_clr	j,vj	vj
KIBCLR	{for,mth}$k_clr	k,k	k
	mth$vk_clr	vk,vk	vk
	mth$vkk_clr	vk,k	vk
	mth$kvk_clr	k,vk	vk
Bitwise circular shift			
IISHFTC	{for,mth}$i_shftc	i,i,i	i
	mth$vi_shftc	vi,vi,vi	vi
	mth$vivii_shftc	vi,vi,i	vi
	mth$viivi_shftc	vi,i,vi	vi
	mth$viii_shftc	vi,i,i	vi
JISHFTC	{for,mth}$j_shftc	j,j,j	j
	mth$vj_shftc	vj,vj,vj	vj
	mth$vjvjj_shftc	vj,vj,j	vj
	mth$vjjvj_shftc	vj,j,vj	vj
	mth$vjjj_shftc	vj,j,j	vj
KISHFTC	{for,mth}$k_shftc	k,k,k	k
	mth$vk_shftc	vk,vk,vk	vk
	mth$vkvkk_shftc	vk,vk,k	vk
	mth$vkkvk_shftc	vk,k,vk	vk
	mth$vkkk_shftc	vk,k,k	vk
String length			
LEN	for$s_len_j	s	j
String index			
INDEX	for$s_index_j	s	j
Character relationals			
LLT	for$s_llt_l	s	l
LLE	for$s_lle_l	s	l
LGT	for$s_lgt_l	s	l
LGE	for$s_lge_l	s	l
IEEE/Native conversions			
RCVTIR	{for,mth}$r_cvti	r	r
	mth$vr_cvti	vr	vr
DCVTID	{for,mth}$d_cvti	d	d
	mth$vd_cvti	vd	vd
IRCVTR	{for,mth}$r_icvt	r	r
	mth$vr_icvt	vr	vr
IDCVTD	{for,mth}$d_icvt	d	d
	mth$vd_icvt	vd	vd

C.1.4 Exponentiation Programmed Operators

The following runtimes perform exponentiation (**). In the argument column, the base type is listed, followed by the exponent type.

Table C-3: Exponentiation Routines

Runtime Name	Arguments	Result
mth$j_pow	j,j	j
mth$k_pow	k,k	k
mth$r_pow	r,r	r
mth$d_pow	d,d	d
mth$c_pow	c,c	c
mth$z_pow	z,z	z
mth$rj_pow_r	r,j	r
mth$dj_pow_d	d,j	d
mth$cj_pow_c	c,j	c
mth$zj_pow_z	z,j	z
mth$vj_pow	vj,vj	vj
mth$vk_pow	vk,vk	vk
mth$vr_pow	vr,vr	vr
mth$vd_pow	vd,vd	vd
mth$vc_pow	vc,vc	vc
mth$vz_pow	vz,vz	vz
mth$vrvj_pow_vr	vr,vj	vr
mth$vdvj_pow_vd	vd,vj	vd
mth$vcvj_pow_vc	vc,vj	vc
mth$vzvj_pow_vz	vz,vj	vz
mth$vjj_pow	vj,j	vj
mth$vkk_pow	vk,k	vk
mth$vrr_pow	vr,r	vr
mth$vdd_pow	vd,d	vd
mth$vcc_pow	vc,c	vc
mth$vzz_pow	vz,z	vz
mth$vrj_pow_vr	vr,j	vr
mth$vdj_pow_vd	vd,j	vd
mth$vcj_pow_vc	vc,j	vc
mth$vzj_pow_vz	vz,j	vz
mth$jvj_pow	j,vj	vj
mth$kvk_pow	k,vk	vk
mth$rvr_pow	r,vr	vr
mth$dvd_pow	d,vd	vd
mth$cvc_pow	c,vc	vc
mth$zvz_pow	z,vz	vz
mth$rvj_pow_vr	r,vj	vr
mth$dvj_pow_vd	d,vj	vd
mth$cvj_pow_vc	c,vj	vc
mth$zvj_pow_vz	z,vj	vz

C.1.5 Complex Programmed Operators

These runtimes perform complex multiplication and division. For division, the argument types are listed as: dividend, divisor.

Table C-4: Complex Programmed Operators

Function	Runtime Name	Argument	Result
Complex Division	mth$c_div	c,c	c
	mth$z_div	z,z	z
	$vc_div	vc,vc	vc
	mth$vz_div	vz,vz	vz
	mth$vcc_div	vc,c	vc
	mth$vzz_div	vz,z	vz
	mth$cvc_div	c,vc	vc
	mth$zvz_div	z,vz	vz
Complex Multiplication	mth$c_mul	c,c	c
	mth$z_mul	z,z	z
	mth$vc_mul	vc,vc	vc
	mth$vz_mul	vz,vz	vz
	mth$vcc_mul	vc,c	vc
	mth$vzz_mul	vz,z	vz
	mth$cvc_mul	c,vc	vc
	mth$zvz_mul	z,vz	vz

C.1.6 Vector Mask Programmed Operators

The runtime *mth$vm_lastnz* finds the position of the highest-order nonzero bit of the vector mask register.

This runtime is useful for conditional code, such as:

```
DO I=1,N
   IF(A(I)) B = A(I)
ENDDO
```

C.1.7 String-Manipulation Programmed Operators

The following are the string-manipulation programmed operators:

for$s_cat - string concatenation
for$s_copy - string copy
for$s_stop - STOP runtime
for$s_paus - PAUSE runtime
for$s_rnge - subscript out of range report
for$s_leq_l - string equal comparison
for$s_lne_l - string not equal comparison

C.1.8 Runtime Data Items

The following runtime labels are associated with data values used by the runtime libraries and the FORTRAN compiler.

The label *mth$vmones*, which is two long words of all 1s, is used to load the *vector mask* register.

The values -2 through 130 are available in 6 data types:

mth$h_indx	INTEGER*1
mth$i_indx	INTEGER*2
mth$j_indx	INTEGER*4
mth$k_indx	INTEGER*8
mth$r_indx	REAL*4
mth$r_indx	REAL*8

The following example shows a typical runtime code:

```
DO I = 1, N
   J(I) = I -1
ENDDO
```

C.2 FORTRAN I/O Library

This section summarizes the runtime entry points in the FORTRAN I/O library, *libI77.a*. The FORTRAN compiler generates calls to the I/O runtimes to implement I/O statements, such as, READ and WRITE. Runtimes from *libI77* are loaded automatically by the FORTRAN compiler (*fc*).

C.2.1 I/O Operation

FORTRAN file I/O to a logical unit consists of the following:

- Open (this may be implicit) (OPEN)
- Series of I/O statements (READ, WRITE, PRINT, ACCEPT, ENCODE, DECODE)
- Optional close (CLOSE)

Each I/O transfer (READ, WRITE, PRINT, ACCEPT, ENCODE, DECODE) is implemented as a sequence of operations:

- Initialize I/O transfer
- Series of I/O transmissions
- Terminate transfer

For example, the I/O statement:

```
READ (5,100) a, b, c
```

is compiled into the following runtime references:

```
for$s_rsfe ; start read sequential formatted external
for$do_fio ; do formatted I/O for a
for$do_fio ; do formatted I/O for b
for$do_fio ; do formatted I/O for c
for$e_rsfe ; end read sequential formatted external
```

If the I/O list is empty, as in READ (5,100), the *for$do_fio* calls are not used. But, the end-io-call, *for$e_rsfe* in this example, is always required.

C.2.2 I/O Runtime Naming Convention

The type of I/O transfer is defined by the following attributes:

read or write:	r or w
sequential or direct:	s or d
formatted, unformatted, list-directed, namelist:	f, u, l, or n
external or internal:	e or i

Many of the runtime names are constructed using the letters listed above. For example, the runtime *for$s_rsfe* initializes I/O for a read-sequential-formatted-external transfer. The following two types of I/O are invalid:

ui—unformatted/internal	dl—direct/list-directed
dn—direct/namelist	ni—namelist/internal

C.2.3 I/O List Initialization

These runtimes prepare the unit for I/O. The following operations are performed:

- If not currently open, opens file with default file attributes, e.g., RECORDTYPE
- Initializes logical record buffer
- Compiles runtime formats
- Saves ERR and END flags
- Checks for various error conditions, e.g., illegal unit number, formatted I/O to file open for unformatted access

Entry points for I/O list initialization:

```
for$s_{r,w}s{f,u,l,n}e
for$s_{r,w}d{f,u}e
for$s_{r,w}s{f,l}i
for$s_{r,w}dfi
for$s_encode
for$s_decode
```

C.2.4 I/O List Element Transmission

These runtimes perform the actual I/O transmission. There are two levels of buffering in the I/O runtimes: logical record buffering and physical record buffering. The record buffers are filled and flushed, as required by the I/O transmission runtimes. Each unformatted I/O statement transfers a single logical record; formatted and list-directed I/O statements may transfer multiple records. The physical record size corresponds to the file system block size.

Each call transmits a single value, except for arrays. Arrays are transmitted with a single runtime call. If referenced in column major order, arrays in implied DO-lists are also transmitted with a single runtime. Formatted transfer of complex values generates two runtime calls—one for the real part and one for the complex part. List-directed transfer of complex values generates a single runtime reference.

Entry points for I/O list element transmission:

```
for$do_{f,u,l}io
```

C.2.5 I/O List Termination

These runtimes complete the I/O operation. This may require flushing the logical record buffer and completion of formatting, if any elements remain in the format string.

for$e_{r,w}s{f,l,u}e
for$e_{r,w}d{f,u}e
for$e_{r,w}s{f,l}i
for$e_{r,w}dfi
for$e_encode
for$e_decode

C.2.6 Auxiliary I/O Operations

The following runtimes implement the auxiliary I/O statements. These statements perform I/O initialization (OPEN), perform I/O termination (CLOSE), position files (BACKSPACE, ENDFILE, REWIND), and return information about a file or unit (INQUIRE). The auxiliary I/O runtimes, along with the FORTRAN statements that they implement, are listed below.

I/O Runtime	I/O Statement
for$back	BACKSPACE
for$close	CLOSE
for$end	ENDFILE
for$inqu	INQUIRE
for$open	OPEN
for$rew	REWIND

D

Problem Reporting

D.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.

D.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 D–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.14 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University (RETURN)
> 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 FORTRAN User's Guide
Document No. 720-000030-203, Eighth Edition

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are welcome and help us continue 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
In Texas	(214)952-0379
Other continental locations	1(800)952-0379
Outside continental U.S.	Contact local CONVEX office

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)