CONVEX Interlanguage Programming Guide

First Edition

CONVEX COMPUTER CORPORATION

Convex Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CONVEX Interlanguage Programming Guide

600

Order No. DSW-043

First Edition
July 1992

CONVEX Press
Richardson, Texas
United States of America

CONVEX Interlanguage Programming Guide

Order No. DSW-043

Printed in the United States of America

Revision Information for

CONVEX Interlanguage Programming Guide

Edition	Document No.	Description
First	720-005830-000	Initial release. Replaces *CONVEX Mixed-language programs Application Note*. (Document No. 720-005430-000).

Contents

Figures

Tables

How to use this guide

Purpose and audience

This guide is intended to aid the experienced programmer who is writing a program using two or more languages. This type of program is referred to as a *mixed-language program*. The three languages addressed are CONVEX Ada, CONVEX C, and CONVEX FORTRAN. This guide contains simple, useful examples that you can use as templates for constructing mixed-language programs.

It is assumed throughout that you are experienced in programming with Ada, C, or FORTRAN. For further discussion of these languages, please consult the section, "Associated documents," at the end of this preface.

Organization

This guide is organized into the following chapters and appendixes:

- Chapter 1 describes compiler, assembler, and loader options that apply to mixed-language programs.
- Chapter 2 lists data types that are compatible between each language.
- Chapter 3 describes the language interface and calling conventions of C, FORTRAN, and Ada.
- Chapter 4 shows how to access C constructs from FORTRAN.
- Chapter 5 shows how to access FORTRAN constructs from C.
- Chapter 6 shows how to access Ada contructs from FORTRAN.
- Chapter 7 shows how to access FORTRAN constructs from Ada.

- Chapter 8 shows how to access Ada constructs from C.
- Chapter 9 shows how to access C constructs from Ada.
- Chapter 10 discusses mixing input and output in Ada, C, and FORTRAN.
- Appendix A provides source code for unsupported C functions that you can call from FORTRAN.
- Appendix B discusses interfacing C++ with Ada, C, and FORTRAN.
- Appendix C describes register usage and stack layout for the compiler language interfaces.

Most of the examples appear in Chapter 4 through Chapter 9. If you are looking for a specific piece of code, you should refer to the "Figures" section which follows the "Contents" section.

Notational conventions

This section discusses notational conventions used in this book.

Command syntax

Consider this example:

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

where:

- `COMMAND` must be typed as it appears.
- *input_file* indicates a file name that must be supplied by the user.
- The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
- Either `a` or `b` must be supplied.
- [*output_file*] indicates an optional file name to be supplied by the user.

General conventions

In general, this book uses the following conventions:

- **Bold constant-width font** identifies user input in examples.
- *Italic* type face:
  - Designates user-supplied variables in a command-line example
  - Introduces new and important terms
  - Indicates document titles
- `Constant-width font` designates input and output, including:
  - Command names and options
  - System calls
  - Directives, program statements, machine op codes, printout examples, file names, directories, and error messages returned
- Horizontal ellipsis (...) shows a repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.
- References to man pages appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

Note

A note highlights supplemental information.

Caution

A caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

Current software revisions

Current revisions of the CONVEX compilers and operating system are as follows:

- ConvexOS V10.1
- CONVEX FORTRAN V7.0
- CONVEX C V4.3
- CONVEX Ada V2.1.1
- CONVEX Application Compiler V1.0
- CONVEX VECLIB V8.0
- CONVEX ALL (assembler, loader, and libraries) V1.1
- CONVEX C++ V2.1

The current revisions for tools used in application development are:

- CONVEX Performance Analyzer (CXpa) V1.3
- CONVEX Consultant V9.1
- CONVEX CXdb V1.2

Associated documents

The following documents, available from CONVEX Computer Corporation, are recommended for the application programmer:

- *CONVEX FORTRAN Guide* (DSW-038) describes the CONVEX FORTRAN compiler and the FORTRAN language and the CONVEX extensions to the language (V7.0, October 1991).
- *CONVEX Ada User's Guide* (DSW-146) describes the CONVEX Ada compiler and support tools and reviews basic Ada concepts (V2.0, December 1990).
- *CONVEX Ada Debugger User's Guide* (DSW-147) describes the `a.db` debugger which is capable of debugging mixed-language programs including Ada, C, and FORTRAN.
- *CONVEX Ada Debugger Reference Manual* (DSW-148) describes the commands that you can use in the `a.db` debugger.
- *CONVEX C Guide* (DSW-086) describes C language topics, CONVEX C extensions, and ANSI C features (V4.1, April 1991).

- *CONVEX Application Compiler User's Guide* (DSW-401) describes how to use the CONVEX Application Compiler (V1.0, 1991).
- *CONVEX VECLIB User's Guide* (DSW-132) describes the VECLIB software library of mathematical software and computational kernels for application programs involving arrays (V7.0, August 1991).
- *CONVEX Compiler Utilities User's Guide* (DSW-096) provides general information about the loader, assembler, and object file formats. (V1.0, September 1991).
- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251) explains the operation of CXpa and the steps needed to create and interpret a CXpa profile (V1.2, November 1990).
- *CONVEX CXdb Concepts* (DSW-471) contains an overview of CXdb and an explanation of how traditional and new debugging concepts are used in CXdb. (CXdb cannot debug programs that contain Ada source code.)
- *CONVEX CXdb Reference* (DSW-472) is a complete reference source for CXdb commands, parameters, concepts, and messages.
- *CONVEX CXdb User's Guide* (DSW-473) describes the CXdb user interface and explains how to use CXdb commands.
- *AT&T C++ Language System Selected Readings* contains papers about the C++ programming language, including "Type-safe Linkage for C++," which discusses how to use the C++ linkage specifier to interface with C routines.
- *AT&T C++ Language System Product Reference Manual* provides a complete definition of the C++ language supported by Release 2.1 of the C++ Language System.

Acknowledgments

The author wishes to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book

Command line options

1

This chapter lists the compiler, assembler, and loader options that can be useful in mixed-language programming. Some of these options are required, while others affect the way a compiler generates code for the application. You should be aware of the effect of these options so that you don't inadvertantly modify the behavior of your program, or its results.

Ada options

The following options are available with V2.1.1 of the CONVEX ada driver:

`-flink`
: This ada driver option causes FORTRAN libraries to be added during the final link step. The Ada `INFO` directive, `FLINK`, defines which libraries are to be loaded and the order in which they are linked. The ada driver automatically links in any C system libraries that are required by an Ada/C mixed-language program.

`-link` *arg*
: This option passes *arg* to the linker, `ld`, that is invoked by the ada driver. You can use `-link` to invoke linker options that are not accessible from the ada driver

`-od`
: This ada driver option causes the Ada compiler to generate code that disables integer overflow detection before a call to a routine that has been defined as having either a C or FORTRAN language interface. The previous state of integer overflow detection is restored after the call. This is the default action of the Ada compiler. Programs created by the C and FORTRAN compilers do not detect integer overflow by default.

`-oe`

This ada driver option prevents the Ada compiler from generating code that disables integer overflow detection before a call to a routine that has been defined as having either a C or FORTRAN language interface. This option provides slightly faster performance than the `-od` option.

C options

The following options are available with V4.3 of the CONVEX C compiler:

`-fi`
`-fn`

The `-fi` option translates floating-point values into IEEE format and performs floating-point operations in IEEE mode. It requires that the machine be equipped with IEEE support hardware. The `-fn` option translates floating-point values into native CONVEX format and performs floating-point operations in native mode. If no floating-point format is specified, the site default is used.

These options are useful when you are mixing C or FORTRAN code with Ada code. There are two versions of the Ada compiler: one that generates IEEE code and one that generates native floating-point code. For example, you must use the `-fi` option to compile your C code if you are mixing it with code generated by the IEEE version of the Ada compiler. You can use the `-v` Ada option to determine which version of the Ada compiler you have.

`-link` *arg*

This option causes *arg* to be passed to the linker, `ld`. If *arg* does not start with a dash (-) or starts with `-l`, it is added to the linker's file list; otherwise, it is added to the linker's flag list. For example,

```
-link -v3.2.8.5 -link -lmylib
```

passes the flag `-v3.2.8.5` to the linker causing it to set the version number of the executable. It also passes `-lmylib` in the file list, causing the linker to link in routines from the library `libmylib.a`.

`-re`

This option generates reentrant code for parallel or recursive invocation of routines. Each invocation of a routine has its own copy of local variables. Use this option to compile a routine called by parallelized regions of code such as Ada tasks.

`-lU77 -lF77 -lI77 -lD77 [-lf90] [-llfs] -lmathC[12]`
These options link in FORTRAN runtime libraries. Use these options only if you are mixing C and FORTRAN object files and the initial routine is the C main routine. The options must be specified in the indicated order. Include the `-lf90` option only if the FORTRAN source code uses the Fortran 90 features. Use the `-llfs` option only if you are using CONVEX FORTRAN V7.0 with ConvexOS V9.1. Use `-lmathC1` to access the C1 FORTRAN math libraries; use `-lmathC2` to access the FORTRAN math libraries for any other C Series machine.

FORTRAN options

The following options are available with V7.0 of the CONVEX FORTRAN compiler:

`-ansic`
This option links `/usr/lib/libc.a` to your program. The `libc.a` library is the extended POSIX library that includes the ANSI C library. This link allows mixed ANSI C and FORTRAN programs. This option is the default action of the compiler. This option also causes the compiler to generate signal 6 (SIGIOT) on multiple library aborts instead of signal 4 (SIGKILL), and prints "IOT" instead of "Illegal Instruction" when the compilation aborts.

`-cfc -i`*n* `-p8 -pd8 -r`*m*
These options affect the size of data types declared without specific size information, such as the declaration REAL instead of REAL*8. The compatibility between the data types of the three languages is dependent on how much memory each compiler allocates for each data type. For example, the FORTRAN LOGICAL data type defaults to 32 bits which is compatible with a `signed long int` in C. However, the `-i2` option reduces that storage to 16 bits which is compatible with a `signed short int` in C. To prevent such mistakes from occurring, you should always use a length override specification in the declaration of the variables that interact with source code in other languages. Refer to fc(1F) for more information.

`-fi`
`-fn`

The `-fi` option translates floating-point values into IEEE format and performs floating-point operations in IEEE mode. It requires that the machine be equipped with IEEE support hardware. The `-fn` option translate floating-point values into native CONVEX format and performs floating-point operations in native mode. If no floating-point format is specified, the site default is used.

These options are useful when you are mixing C or FORTRAN code with Ada code. There are two versions of the Ada compiler: one that generates IEEE code and one that generates native floating-point code. For example, you must use the `-fi` option to compile your C code if you are mixing it with code generated by the IEEE version of the Ada compiler. You can use the `-v` Ada option to determine which version of the Ada compiler you have.

`-link` *-arg*

This option passes *arg* to the linker, `ld`, where *arg* is an arbitrary linker option. You must delimit *arg* with quotation marks if a blank space appears in the argument. Note that a library linked with the FORTRAN compiler cannot contain a C main routine.

`-pcc`

This option links a backward-compatible `libc_old.a` with your program, allowing you to mix the traditional dialect of C with FORTRAN programs. This option overrides the default `-ansic` that allows you to mix ANSI C and FORTRAN object code.

`-re`

This option generates reentrant code for parallel or recursive invocation of routines. Each parallel invocation of a routine has its own copy of local variables. Arguments are passed on the stack instead of by argument packets. However, common variables and saved or data-initialized variables are still shared among invocations. If you compile a program with this option, you must initialize all local variables that are to remain unshared. You should also use this option to compile routines called by Ada tasks.

`-sa`

This option prevents the compiler from generating precompiled argument packets in the text segment. All arguments are placed on the stack. You should use this option only with FORTRAN routines that call Ada or C routines, not with your entire FORTRAN source, because it inhibits performance.

Other options

This section describes loader and assembler options that are sometimes useful when constructing mixed-language programs.

`-ada`

This assembler option permits special characters to be used in Ada symbols. It also recognizes assembler directives that only the Ada compiler uses.

`-A`*alias=symbol*

This linker option logically renames *alias* to *symbol*, causing all references of the name *alias* to resolve to the object symbol. It can prove useful if you have a C or FORTRAN program that calls a routine whose name does not conform to the format of symbols generated by the C or FORTRAN compilers. For example, if you have a routine called `test` that is called from C, include `-link -A_test=test` on the C compiler command line.

`-A`*file*

This linker option reads an alias list from *file*. The alias list is in the form *alias=symbol*, listed one to a line. Use this option with caution. Refer to the description of the `-A`*alias=symbol* option in this section for more details.

`-m`

This linker option forces the linker to perform multiple scans of the libraries if unresolved externals remain at the normal end of the link. You can use this option on the C command line without preceding it with the `-link` C option. It is useful when you are calling VECLIB routines from a C program.

Data types

2

This chapter discusses compatibility among Ada, C, and FORTRAN data types.

Integral data types

Table 1 lists the compatible integral data types between Ada and C.

Table 1
Ada and C integral data types

Ada	C
`x : boolean;`	`signed char x;`
`x : short_integer;`	`signed short x;`
`x : integer;`	`signed int x;`
`x : long_integer;`	`signed long long int x;`
`x : tiny_integer;`	`signed char x;`

You must use representation clauses in Ada to create types that are compatible with the unsigned integers in C. For example, to create a type in Ada that is compatible with the `unsigned char` data type in C, use the following code:

```
type UNSIGNED_CHAR is range 0 .. (2 ** 8) - 1;
for UNSIGNED_CHAR'SIZE use 8;
u_char : UNSIGNED_CHAR;
```

The first line defines a type that has the same range as the `unsigned char` type in C. The second line ensures that the same amount of storage is used. And the third line declares a variable. Other unsigned integral types can be constructed similarly.

Table 2 lists compatible integral data types between Ada and FORTRAN.

Table 2
Ada and FORTRAN integral data types

Ada	FORTRAN
x : BOOL;	LOGICAL*1 x
x : SHORT_INTEGER;	LOGICAL*2 x
x : INTEGER;	LOGICAL*4 x
x : LONG_INTEGER;	LOGICAL*8 x
i : TINY_INTEGER;	BYTE i
i : TINY_INTEGER;	INTEGER*1 i
i : SHORT_INTEGER;	INTEGER*2 i
i : INTEGER;	INTEGER*4 i
i : LONG_INTEGER;	INTEGER*8 i

Table 3 lists compatible integral types between C and FORTRAN.

Table 3
C and FORTRAN integral data types

C	FORTRAN
signed char x;	LOGICAL*1 x
signed short int x;	LOGICAL*2 x
signed int x;	LOGICAL*4 x
signed long long int x;	LOGICAL*8 x
signed char i;	INTEGER*1 i
signed short int i;	INTEGER*2 i
signed int i;	INTEGER*4 i
signed long long int i;	INTEGER*8 i

Unsigned C data types cannot be represented in FORTRAN.

Floating-point data types

Table 4 lists compatible floating-point data types between Ada and C.

Table 4
Ada and C floating-point data types

Ada	C
`a : SHORT_FLOAT;`	`float a;`
`a : FLOAT;`	`double b;`

Note that the `float` data type in Ada is not compatible with the `float` data type in C.

Table 5 lists compatible floating-point data types between Ada and FORTRAN.

Table 5
Ada and FORTRAN floating-point data types

Ada	FORTRAN
`a : SHORT_FLOAT;`	`REAL*4 a`
`b : FLOAT;`	`REAL*8 b`
none	`REAL*16 c`

Table 6 lists compatible floating-point types between C and FORTRAN.

Table 6
C and FORTRAN floating-point data types

C	FORTRAN
`float a;`	`REAL*4 a`
`double b;`	`REAL*8 b`
none	`REAL*16 c`

Complex data types

Because neither Ada nor C have complex data types, these types must be constructed using records and structs, respectively.

To construct a record in Ada that is compatible with the FORTRAN `COMPLEX*16` data type, use the following code:

```
type COMPLEX_16 is record
   REAL, IMAGINARY : FLOAT;
end record;
```

Alternatively, the `convexlib/complex.a` file contains a generic package that is based on a complex data type. This package provides arithmetic functions for a complex data type as well as some basic conversion routines. The section, "Complex return values" on page 72, contains an example that uses this package.

To create a struct in C that is compatible with the FORTRAN `COMPLEX*16` data type, use the following code:

```
typedef struct {
   double real, imaginary;
} complex16_t;
```

Data types in Ada and C that are compatible with the FORTRAN `COMPLEX*8` data type can be constructed similarly using `SHORT_FLOAT` and `float` as the data type of the respective fields.

Record types

The next example shows how to create a record in Ada that is compatible with a C record. The following C record is defined in `/usr/include/a.out.h`:

```
struct relocation_info {
   int           r_address;
   unsigned int  r_fill : 4;
   unsigned int  r_extern : 1;
   unsigned int  r_length : 2;
   unsigned int  r_pcrel  : 1;
   unsigned int  r_symbolnum : 24;
};
```

This C record includes bit fields that require explicit placement in this record.

Ada has several features that permit you to mimic the storage requirements of this record. One such feature is the record representation clause which specifies the order, position, and size of the fields in the record. Another feature is the length clause which specifies the amount of storage associated with a type. Using these clauses, you can provide additional information about a data type to represent any C record in Ada.

Three segments of Ada code are required to fully represent the C `relocation_info` struct:

- A type declaration that creates a compatible data type
- A record representation clause that declares storage requirements
- A length clause that specifies the amount of memory required

The following segment of Ada code creates a data type that is compatible with the previous C structure in `/usr/include/a.out.h`:

```
type RELOCATION_INFO is record
      R_ADDRESS   : INTEGER;
      R_FILL      : INTEGER range 0..(2**4)-1;
      R_EXTERN    : INTEGER range 0..1;
      R_LENGTH    : INTEGER range 0..3;
      R_PCREL     : INTEGER range 0..1;
      R_SYMBOLNUM : INTEGER range 0..(2**24)-1;
   end record;
```

The type declaration only specifies the possible values that the fields can have. Each range is determined from the number of bits that the C record allocates for each field.

Having created the data type, you must next declare the storage requirements of the `RELOCATION_INFO` type using the record representation clause:

```
for RELOCATION_INFO use
   record at mod 4;
      R_ADDRESS at 0 range 0 ..31;
      R_FILL    at 0 range 32..35;
      R_EXTERN  at 0 range 36..36;
      R_LENGTH  at 0 range 37..38;
      R_PCREL   at 0 range 39..39;
      R_SYMBOLNUM at 0 range 40..63;
   end record;
```

In this code segment, the range specifies the location in memory; that is, which bits each field is allocated in the record. The `at mod 4` phrase forces the record to be aligned on a 4-byte boundary which is the way the C compiler aligns the C record.

Finally, you must specify how much memory the record requires using the length clause:

```
for RELOCATION_INFO'SIZE
   use 8*SYSTEM.STORAGE_UNIT;
```

For more information on the Ada constructs demonstrated in this example, refer to sections 13.1 through 13.5 of the *ANSI Ada Reference Manual*. For more information on how the C compiler aligns structures, refer to the *CONVEX C Guide*.

The default language compatibility mode of the FORTRAN compiler does not support the definition of records. However, limited support for records exists in the VAX FORTRAN compatibility mode (`-vfc`). One limitation of the VAX record structure is that bit fields are not supported.

Following is an example of a FORTRAN structure that uses the VAX-compatible record structure.

```
STRUCTURE /INT_STRUCTURE/ INTEGER_FIELDS
   INTEGER*2   FIELD1
   BYTE        FIELD2
   BYTE        %FILL
   INTEGER*4   FIELD3
   BYTE        FIELD4
   BYTE        %FILL
END STRUCTURE
```

The `%FILL` declaration pads the structure. The compiler pads the structure the same way that the C and Ada compilers pad structures, so you do not have to worry about the explicit representation of the structure.

Array types

Mathematical matrices are represented using array notation within application programs for computational purposes.

Ada and C arrays are stored in row-major order, while FORTRAN arrays are stored in column-major order. This means that given a two-dimensional array in C, array elements are contiguous in memory based on the second subscript.

For example, in C, contiguous memory locations of array `a[2][3]` are accessed by the following sequence of array references:

```
a[0][0]
a[0][1]
a[0][2]
a[1][0]
a[1][1]
a[1][2]
```

Similarly, the contiguous memory locations of an array declared in Ada as

```
A : array (INTEGER range 3..4,
           INTEGER range -2..0) of INTEGER;
```

are accessed by the following sequence of array references:

```
A(3,-2)
A(3,-1)
A(3,0)
A(4,-2)
A(4,-1)
A(4,0)
```

In FORTRAN, contiguous memory locations of array `A(2,3)` with the default lower bound subscript of 1 are accessed by the following sequence of array references:

```
A(1,1)
A(2,1)
A(1,2)
A(2,2)
A(1,3)
A(2,3)
```

There are two ways to access an array declared in Ada or C from a FORTRAN routine. You can reverse the indices of the array, or you can use the ROW_WISE directive. Examples demonstrating these two methods are shown in Chapter 7 and Chapter 5.

The initial element of an array can be referenced using different indices in the three languages. Array subscripts always start at 0 in C. In FORTRAN, the default lower bound of a subscript is 1. Similarly, the lower bound of Ada array subscripts can be any integral or enumerated value. Consequently, when you subscript arrays that are accessed by more than one language, you must ensure that you are using the correct subscript value.

String types

In general, strings are arrays of characters. However, each of the three languages uses a different representation for strings. C strings are pointers to the first character in the string, and the last character is followed by a null value. In contrast, FORTRAN strings have a defined string length that is stored in addition to the characters. Finally, Ada strings require not only the string length, but also the lower and upper bounds so that it can check for array-out-of-bound errors. Consequently, it is not trivial to transfer a string directly from one language to another.

Language interface 3

This chapter describes C, FORTRAN, and Ada language interfaces including external symbol names, parameter passing methods, and function return methods. The chapters that follow this one provide numerous examples that apply the information contained in this chapter.

For additional information, refer to Appendix C for a description of how the compilers allocate registers and modify the call stack to call functions or subroutines.

C interface

This section describes the naming convention for external symbols, the parameter passing method, and the function return method of the CONVEX C compiler.

External symbols

C has simple naming convention for external symbols, which include global variables and function names. The C compiler prepends an underscore character ('_') to external symbols. Additionally, function names and global variables produced by the C compiler are unrestricted in length and case sensitive.

For example, the external symbols produced for the following code segment

```
int add( int num, int num2)
{
   extern int num3;
   return (num + num2 + num3);
}
```

are _add and _num3.

Parameter passing method

CONVEX C functions pass parameters by value. This means that only the contents of a parameter are passed to the called routine, not its address. The exception to this rule is arrays: to save time and stack space, arrays are passed by reference; only the address of the array is passed to the called routine.

However, C provides two operators you can use to work around this parameter passing method: the address operator (&) and the indirection operator (*). The address operator takes the address of its operand. The address operator is useful when you must call a FORTRAN routine from C. For example, to call a FORTRAN function that returns the sum of its INTEGER*4 arguments, use the following:

```
main()
{
  int arg1 = 4;
  int arg2 = 8;
  int retval;

  retval = add_int4_( &arg1, &arg2 );
}
```

The address operator is useful in this instance because FORTRAN passes its arguments by reference. The indirection operator is useful when you must pass the contents of a pointer. For example, to call an Ada function that expects two parameters passed by value and returns their sum, use the following:

```
main()
{
  int *arg1;
  int *arg2;
  int retval;

  *arg1 = 4;
  *arg2 = 8;
  retval = add_int4( *arg1, *arg2 );
}
```

Function return method

CONVEX C functions return integral data types, floating-point data types, and pointers in register S0. Functions that return other data types place the function result in a static area in the program's data segment and return a pointer to that location in register S0. (An exception to this is if you use the `-compat rrf=stack` C compiler option. In this case, the calling C routine allocates a location for the return value on the call stack prior to the argument list of the called function. This option affects only C functions that return a struct by value.)

FORTRAN interface

This section describes the naming convention for external symbols, the parameter passing method, and the function return method of the CONVEX FORTRAN compiler.

External symbols

The FORTRAN compiler prepends and appends underscores to subroutines, functions, and entry points. Named common blocks have two underscores prepended and one underscore appended. The external symbol for blank common blocks is `___blnk_`. The external symbol for the main program unit is `_MAIN__`. Additionally, function names and global variables produced by the FORTRAN compiler are unrestricted in length and are not case sensitive; the FORTRAN compiler translates uppercase names to lowercase.

For example, the external symbols generated for the following piece of code

```
SUBROUTINE SUB1( J, B )
REAL*8 A, B
INTEGER*4 I, J
COMMON A
COMMON /FOO/ I
J = I
B = A
RETURN
END
```

are `_sub1_`, `___blnk_`, and `__foo_`.

Parameter passing method

FORTRAN routines pass parameters by reference. However, CHARACTER parameters pass an additional argument by value. This additional argument is appended to the end of the parameter list.

For example, the FORTRAN compiler translates the parameter list of the following function

```
SUBROUTINE SUB( A, B, I, J )
CHARACTER A
CHARACTER*3 B
INTEGER I, J
RETURN
END
```

into

```
SUB( address-of-A, address-of-B, address-of-I,
     address-of-J, length-of-A, length-of-B )
```

This information is useful when you call a FORTRAN routine from Ada or C.

To help you when you call an Ada or C routine, FORTRAN provides two functions that you can use to modify how parameters are passed: the immediate value function (%VAL) and the reference function (%REF). These functions can only be used in an actual routine parameter list.

The immediate value function (%VAL) causes a parameter to be passed by value, while the reference function (%REF) causes a parameter to be passed by reference. You can use the reference function with any operand data type, but the the immediate value function can only be used with the LOGICAL, INTEGER*1, INTEGER*2, INTEGER*4, and REAL*4 data types. For example, to call a C function that returns the sum of its arguments, which are passed by value, use:

```
PROGRAM CALLC
INTEGER*4 I, J
INTEGER*4 IADD
I = 4
J = 8
PRINT *, IADD( %VAL( I ), %VAL( J ) )
END
```

Function return method

CONVEX FORTRAN functions of type INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8, LOGICAL*1, LOGICAL*2, LOGICAL*4,LOGICAL*8, REAL*4, or REAL*8 place the return value in register S0.

COMPLEX*4, COMPLEX*8, and REAL*16 functions allocate a location on the call stack prior to the parameter list to contain the address of the returned value. In other words, the FORTRAN compiler translates a COMPLEX*8 function into a subroutine in which the first argument is the address of the COMPLEX*8 return value. For example, the compiler translates

```
COMPLEX*8 FUNCTION IADDEM ( J1, J2 )
INTEGER*4 J1, J2
```

into

```
SUBROUTINE IADDEM( RETVAL, J1, J2 )
INTEGER*4 RETVAL      ! RETVAL = %REF(X)
                      ! where X is the return
                      ! value.
INTEGER*4 J1, J2
```

It is only necessary to understand this translation when you call a COMPLEX, INTEGER*8, or REAL*16 function from Ada or C.

Functions that return a CHARACTER type allocate two words of memory prior to the argument list of the called function. The first memory location contains the address of the character string to receive the return value, while the second contains the maximum length of the character string. In other words, the FORTRAN compiler translates a CHARACTER function into a subroutine in which the first argument is the address of the CHARACTER return value and the second argument is the length of the character string that is returned.

For example, the compiler translates

```
CHARACTER*5 FUNCTION CADDEM( J1, J2 )
INTEGER*4 J1, J2
```

into

```
SUBROUTINE CADDEM( RETVAL, 5, J1, J2 )
INTEGER*4 RETVAL    ! RETVAL = %REF(X) where X
                    ! is the return value
INTEGER*4 J1, J2
```

It is only necessary to understand this translation when you call a CHARACTER function from Ada or C.

Ada interface

This section describes the naming convention, parameter passing method, and function return method for external symbols of the CONVEX Ada compiler. It also descibes three pragmas that CONVEX Ada provides which permit you to interface Ada with routines in other languages.

External symbols

To access an Ada object, you must be able to modify the Ada source code. Ada has a complicated naming convention for its objects that can be accessed from another language.

For example, subprogram names usually have the following form:

_A_subnamellXcc.parent

where:

subname
is the subprogram name.

ll
is the line number of its definition.

X
is S if defined in the spec and B for the body.

cc
is the character number of its definition.

parent
is the name of the parent unit, without the _A_ prefix.

Consequently, when you change the location of the subprogram in its source file, the external symbol name changes. Fortunately, Ada provides pragmas that you can use to specify an unchanging external symbol name for a variable or function. These pragmas are:

- EXTERNAL_NAME
- INTERFACE
- INTERFACE_NAME

These pragmas insulate you from how the C and FORTRAN compilers generate link names for the external objects. As a result, you do not have to remember which underscore gets appended where with respect to routine link names that your Ada routine calls. Similarly, you can give an external name to an Ada object to make it accessible in another language.

pragma EXTERNAL_NAME

The EXTERNAL_NAME pragma allows you to specify an external symbol name, or linkage name, for an Ada variable or routine so that it can be accessed from another language. This pragma must occur as a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

The syntax for this pragma is as follows:

```
pragma EXTERNAL_NAME( Ada_object, link_name )
```

If *Ada_object* is a variable, it must be declared in a package specification. If it is a routine, it can be declared in either a library unit or a package specification.

Ada routines can recieve parameters from a C or FORTRAN routine by value or by reference. You must specify the particular method with the mode parameter: either `in` or `in out`. For example, suppose that a C routine calls an Ada function that returns the sum of its two parameters. A possible Ada function is the following:

```
package C_EXAMPLE is
  function ADD_LONG( ARG1: in INTEGER;
                     ARG2: in INTEGER ) return INTEGER;
  pragma EXTERNAL_NAME( ADD_LONG, "_add_long" );
end C_EXAMPLE;
```

```
package body C_EXAMPLE is
  function ADD_LONG( ARG1: in INTEGER;
           ARG2: in INTEGER ) return INTEGER is
  begin
    return ARG1 + ARG2;
  end ADD_LONG;
end C_EXAMPLE;
```

Note that the mode of both parameters is in. This mode specifies pass-by-value which conforms to C's parameter passing method. You should also note that the external name for the ADD_LONG function is in lowercase and has an underscore prepended to its name. This assumes that the C code calls an add_long function.

In contrast, suppose that a FORTRAN routine calls an Ada function that returns the sum of its two parameters. You could call the function in the C_EXAMPLE, above, but you would have to use the %VAL parameter function and change the external name.

Instead, you could create an Ada function similar to the following:

```
package F_EXAMPLE is
  function ADD_INT4( ARG1: in out INTEGER;
           ARG2: in out INTEGER ) return INTEGER;
  pragma EXTERNAL_NAME( ADD_INT4, "_add_int4_" );
end F_EXAMPLE;

package body F_EXAMPLE is
  function ADD_INT4( ARG1: in out INTEGER;
          ARG2: in out INTEGER ) return INTEGER is
  begin
    return ARG1 + ARG2;
  end ADD_INT4;
end F_EXAMPLE;
```

Note that the mode of both parameters is in out. This mode specifies pass-by-reference which conforms to FORTRAN's parameter passing method. You should also not that the external name for the ADD_INT4 function is in lowercase and has an underscore appended and prepended to its name. This assumes that the FORTRAN code calls an add_int4 function.

The two previous examples, C_EXAMPLE and F_EXAMPLE, are concerned with calling an Ada routine from C or FORTRAN. In such circumstances, you must use the EXTERNAL_NAME pragma to create an external symbol name that a C or FORTRAN routine can access. However, when you call a C or FORTRAN routine from Ada, you must use either the INTERFACE pragma or INTERFACE_NAME pragma. These pragmas place restrictions on the parameter mode depending on the language of the called routine.

pragma INTERFACE

This pragma permits you to call routines written in another language. This pragma can occur as a declarative item and must refer to a routine declared by an earlier declarative item of the same declarative part or package specification. It can also occur in a library unit after the subprogram declaration, but before any subsequent compilation unit.

The syntax for this pragma is as follows:

```
pragma INTERFACE( language, routine )
```

The *langauge* argument can be Ada, C, CALLQ, FORTRAN, or UNCHECKED.

- If *language* is Ada, the compiler generates a routine call as if it were to an Ada procedure, but it does not expect a matching procedure body.
- If *langauge* is C or FORTRAN, the compiler generates a call to *routine* concatenating the correct underscores.
- If *language* is UNCHECKED, the compiler generates a call as for an Ada procedure, but it does not expect a matching procedure body. Use the UNCHECKED argument when you are calling an assembly-language routine, or a language other than Ada, C, or FORTRAN.

CALLQ is used only by CONVEX; it is used in `convexlib/Fastmath.a` and `convexlib/math_lib.a`.

The following example informs the Ada compiler that all references to the routine `test` should generate calls to the link name `_test` because the C compiler prepends an underscore to all its external functions:

```
pragma INTERFACE( C, "test");
```

Similarly, the following example informs the Ada compiler that all references to the routine `ftest` should generate calls to the link name `_ftest_`, because the FORTRAN compiler prepends and appends an underscore to all FORTRAN routines:

```
pragma INTERFACE (FORTRAN, "ftest");
```

Consequently, one of the benefits of using the INTERFACE pragma is that you do not have to prepend and append underscores to C or FORTRAN routine names because the Ada compiler does this automatically.

Several restrictions are associated with the INTERFACE pragma:

- The return types are limited to scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS.
- The types of parameters for C routines must be scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS. Further, all parameters must have mode in.
- The types of parameters for FORTRAN routines must be the predefined type ADDRESS in SYSTEM.ADDRESS.
- The name of *routine* cannot be the same as an Ada keyword. (You can work around this restriction using the INTERFACE_NAME pragma, described in the following section.)

Examples using the INTERFACE pragma can be found in Chapter 6 and Chapter 8.

pragma INTERFACE_NAME

This pragma permits you to access external objects such as common blocks, C global variables, and routines written in other languages. This pragma can occur as a declarative item and must refer to an object declared by an earlier declarative item of the same declarative part or package specification. It can also occur in a library unit after the subprogram declaration, but before any subsequent compilation unit. The syntax for this pragma is as follows:

```
pragma INTERFACE_NAME( Ada_object, link_name)
```

The INTERFACE_NAME pragma replaces all occurrences of *Ada_object* with *link_name*. Unlike the INTERFACE pragma, you must either append and prepend any required underscores, or use constants such as C_PREFIX, FORTRAN_PREFIX, and

FORTRAN_SUFFIX declared in standard/language.a; there are no language conventions associated with the arguments of the INTERFACE_NAME pragma.

For example, to access the C errno variable, use the following code segment:

```
ERRNO : INTEGER;
pragma INTERFACE_NAME( ERRNO, "_errno" );
```

In addition, you can use this pragma to access a routine that has the same name as an Ada keyword:

```
procedure PRAGGMA( INTEGER : in SYSTEM'ADDRESS);
procedure INTERFACE( C, PRAGGMA );
procedure INTERFACE_NAME( PRAGGMA,
          language.C_PREFIX & "pragma" );
```

Two restrictions are associated with this pragma:

- *Ada_object* cannot be a loop variable, a constant, or an initialized variable.
- If *Ada_object* is a routine name, then it must have been specified in an INTERFACE pragma.

Examples using the INTERFACE_NAME pragma can be found in Chapter 6 and Chapter 8.

Parameter passing methods

Ada permits data to be passed by reference or by value using the formal argument mode. To pass a parameter by reference you must use the in out parameter mode; you must use the in parameter mode to pass a parameter by value. However, arrays and records must always be passed by address.

Function return method

Like C and FORTRAN, Ada returns integral and floating-point data types in register S0. However, it is not possible for C or FORTRAN routines to access other data types, including arrays or records, returned from an Ada function. One workaround to this limitation is to pass nonintegral data types to Ada routines by reference.

FORTRAN calls C

4

This chapter contains examples that demonstrate how to access C objects from FORTRAN. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter.

Each example includes the source code for C and FORTRAN; C source code appears in lower case, while FORTRAN source code appears in upper case. Command lines are provided so that you can reproduce the examples.

The examples in this chapter show you how to perform the following tasks:

- Pass integer and floating-point arguments to a C function
- Pass string arguments to a C function
- Pass complex arguments to a C function
- Pass Cray or Sun pointers to a C function
- Call C functions that return integer and floating-point values
- Call C functions that return string values
- Call C functions that return complex values

Integer and floating-point arguments

The example in Figure 1 shows how to pass INTEGER*4 and REAL*4 FORTRAN data types to a C function.

Figure 1
Passing integer and floating-point arguments from FORTRAN to C

```
        PROGRAM MAIN                    ! Source file is fcir.f
        INTEGER*4 I /0/
        REAL*4    A /0.0/
        CALL SUB1 (I,A)
        PRINT *, I, A
        END

void sub1_( int *j, float *b)           /* Source file is fcir2.c */
{
   *j = 1234567;
   *b = 9876.543;
}
```

You should note the following about Figure 1:

- Parameters b and j are declared as pointers because FORTRAN passes parameters by reference.
- An underscore is appended to the C function sub1_ because the C compiler only prepends an underscore, while the FORTRAN compiler prepends and appends an underscore.

The command lines to compile this program and its output are:

```
% cc -c fcir2.c
% fc -sa fcir.f fcir2.o
% a.out
     1234567    9876.543
```

The fc driver is used to load the program because the main program unit is written in FORTRAN.

String arguments

The example in Figure 2 shows how to pass the CHARACTER*n data type to a C function.

Figure 2
Passing string arguments from FORTRAN to C

```
        PROGRAM MAIN                          ! Source file is fcchar.f
        CHARACTER*11 CVAR1 /' '/              ! Declare two character variables.
        CHARACTER*6  CVAR2 /' '/
        CALL SUB1 (CVAR1, CVAR2)
        PRINT *, CVAR1, CVAR2
        END

#include <string.h>                          /* Source file is fcchar2.c */

void sub1_(char *str1, char *str2, int d1, int d2 )
{
   (void) strncpy( str1, "1234567890", d1);
   (void) strncpy( str2, "abcde", d2);
}
```

You should note the following about Figure 2:

- The character variables declared in the FORTRAN source file are 1 character longer than the strings that are copied into them in the C source file because C strings have a null terminator character. You can corrupt memory if you attempt to write a string into a space that is not large enough.
- The C function parameters, d1 and d2, correspond to the declared lengths of CVAR1 and CVAR2. They are not pointers because the FORTRAN compiler places their *values* on the call stack.

The command lines to compile this example and its output are:

```
% cc -c fcchar2.c
% fc -sa fcchar.f fcchar2.o
% a.out
1234567890abcde
```

Complex arguments

The example in Figure 3 shows how to pass a COMPLEX data type to a C function.

Figure 3
Passing complex arguments from FORTRAN to C

```
        PROGRAM MAIN                          ! Source file is fccplx.f
        COMPLEX*8 CX /(0.0,0.0)/
        CALL SUB1 (CX)
        PRINT *, CX
        END

void sub1_( struct { float r, i;} *cxs )   /* Source file is fccplx2.c  */
{
   cxs->r = 123.45;
   cxs->i = 678.90;
}
```

Note that the compatible data type in C for the COMPLEX data type in FORTRAN is a `struct`. The order of the components of the `struct` is important.

The command lines to compile this example and its output are:

```
% cc -c fccplx2.c
% fc -sa fccplx.f fccplx2.o
% a.out
 (  123.4500    ,  678.9000    )
```

Cray pointer and pointee arguments

You can use the Cray POINTER statement to manipulate the pointers of FORTRAN data types. The syntax for this statement is

```
POINTER( p, s )
```

where *p* is a pointer, and *s* is the name of a local variable or array. In Cray terminology, *s* is the pointee.

The example in Figure 4 shows how to pass a Cray pointer to a C routine.

Figure 4
Passing a Cray pointer to a C routine

```
        PROGRAM POINTER                              ! Source file is fcpntr.f
        REAL A
        POINTER(CPA, A)
        CALL HPALLOC( CPA, 8, IRETVAL, IAB )
        A = 10.0
        PRINT *,A
        CALL SUB(CPA)
        PRINT *,A
        END

void sub_(double **cray_ptr)                    /* Source file is fcpntr2.c */
{
   double *n;
   n = *(cray_ptr+1);
   *n = 6.0L;
}
```

You should note the following about Figure 4:

- In the Cray compatibility mode, `-cfc`, you must call `hpalloc` or `loc` to assign an address to the pointer, CPA, before you can access its value. (The `hpalloc` and `loc` functions are available only in the Cray compatibility mode. In all other compatibility modes, you must use the `%loc` function.)
- The variable A requires 8 bytes of storage because the Cray compatibility mode converts all REAL data types to REAL*8.

- FORTRAN passes the pointer by reference; consequently, the data type of the C function argument must be a pointer to a pointer to a `double` (a C `double` is compatible with a FORTRAN REAL*8).
- FORTRAN passes the Cray pointer as an 8-byte value; the first 4 bytes of the address are NULL. Consequently, you must increment the address of the argument by 1, and then dereference the result to obtain the address contained in the Cray pointer.

The command lines to compile this program and its output are:

```
% cc -c fcpntr2.c
% fc -sa -cfc fcpntr.f fcpntr2.o
% a.out
   10.0000000000000
    6.00000000000000
%
```

The `-cfc` option is included on the `fc` command line because this example uses the `hpalloc` function which is available only in the Cray compatibility mode.

The example in Figure 5 shows how to pass a Cray pointee to a C routine.

Figure 5
Passing a Cray pointee to a C function

```
        PROGRAM POINTEE                              ! Source file is fcpnte.f
        REAL*4 A
        POINTER(CPA, A)
        CALL HPALLOC( CPA, 8, IRETVAL, IAB )
        A = 10.0
        PRINT *,A
        CALL SUB(A)
        PRINT *,A
        END

void sub_(double *cray_pte)                     /* Source file is fcpnte2.c */
{
   *cray_pte = 6.0L;
}
```

You should note the following about Figure 5:

- In the Cray compatibility mode, `-cfc`, you must call `hpalloc` or `loc` to assign an address to the pointer, `CPA`, before you can access its value. (The `hpalloc` and `loc` functions are available only in the Cray compatibility mode. In all other compatibility modes, you must use the `%loc` function.)
- The variable `A` requires 8 bytes of storage because the Cray compatibility mode converts all `REAL` data types to `REAL*8`.
- FORTRAN passes the pointee by reference; consequently, the data type of the C function argument must be a pointer to a `double` (a C `double` is compatible with a FORTRAN `REAL*8`).

The command lines to compile this program and its output are:

```
% cc -c fcpnte2.c
% fc -sa -cfc fcpnte.f fcpnte2.o
% a.out
   10.0000000000000
    6.00000000000000
%
```

The `-cfc` option is included on the `fc` command line because this example uses the `hpalloc` function which is available only in the Cray compatibility mode.

Integer and floating-point return values

The example in Figure 6 shows how to call C functions that return REAL*4 and INTEGER*4 values.

Figure 6
Calling C functions that return integer and floating-point values from FORTRAN

```
        PROGRAM MAIN                              ! Source file is fcfunc.f
        REAL*4 ADDEM, B1 /43.0/, B2 /12.0/
        INTEGER*4 IADDEM, J1 /43/, J2 /12/
        PRINT *, ADDEM(B1,B2)
        PRINT *, IADDEM(J1,J2)
        END

float addem_( float *a1, float *a2 )          /* Source file is fcfunc2.c */
{
   return( *a1 + *a2 );
}

int iaddem_( int *i1, int *i2 )
{
   return( *i1 + *i2 );
}
```

Note that the two C functions addem_ and iaddem_ return float and int objects, the C data types compatible with the FORTRAN data types REAL*4 and INTEGER*4, respectively.

The command lines to compile this example and its output are:

```
% cc -c fcfunc2.c
% fc -sa fcfunc.f fcfunc2.o
% a.out
   55.00000
          55
```

String return values

The example in Figure 7 shows how to call C functions that return CHARACTER*n variables.

Figure 7
Calling C functions that return strings to FORTRAN

```
        PROGRAM MAIN                              ! Source file is fccfunc.f
        INTEGER VAL1, VAL2
        CHARACTER*10  DIGITS /'0123456789'/
        CHARACTER*30 MAKESTR
        PRINT 400, MAKESTR (DIGITS,VAL1,VAL2), VAL1, VAL2
400     FORMAT('String=',A,'|Val1=',I3,'|Val2=',I3,'|')
        END

#include <string.h>                              /* Source file is fccfunc2.c */

void makestr_(char cstr[], long int cstr_len, char *arg1, long int *num1,
              long int *num2, int arg1_len )
{
   (void) strcpy( cstr, "abcde       " );
   (void) strcat( cstr, arg1 );
   *num1 = cstr_len;
   *num2 = arg1_len;
}
```

You should note the following about Figure 7:

- A C function that is declared as type `void` simulates a FORTRAN function that returns CHARACTER variables.
- The first parameter of the C function contains the address of the CHARACTER variable.
- The second parameter of the C function contains the amount of space allocated for the CHARACTER variable in the FORTRAN source code. This parameter is passed in by value.
- The third parameter of the C function corresponds to first argument, DIGITS, that the FORTRAN main program passes to the C function.
- The `arg1_len` argument is added at the end of the C function parameter list to contain the length of the `arg1` argument; FORTRAN adds an argument that contains the length of a string for every argument of type CHARACTER.

The command lines to compile this example and its output are:

```
% cc -c fccfunc2.c
% fc -sa fccfunc.f fccfunc2.o
% a.out
String=abcde        0123456789|Val1= 30|Val2= 10|
```

This example demonstrates that `cstr_len` contains the declared length of the variable returned by MAKESTR and that `arg1_len` contains the declared length of DIGITS.

Complex return values

The example in Figure 8 shows how to call C functions that return COMPLEX*8 and COMPLEX*16 data types.

Figure 8
Calling C functions that return complex variables to FORTRAN

```
        PROGRAM MAIN                          ! Source file is fcxfunc.f
        COMPLEX*8  MAKECX8, ANS8
        COMPLEX*16 MAKECX16, ANS16
        REAL*4 B1 /5.0/, B2 /7.0/
        REAL*8 D1 /5.0/, D2 /7.0/
        ANS8 =  MAKECX8(B1,B2)
        ANS16 = MAKECX16(D1,D2)
        PRINT *, ANS8
        PRINT *, ANS16
        END

typedef struct {float r, i;} complex8_t;    /* Source file is fcxfunc2.c */
typedef struct {double r, i;} complex16_t;

void makecx8_( complex8_t *cx, float *a1, float *a2 )
{
   cx->r = *a1 + *a2;
   cx->i = *a1 - *a2;
}

void makecx16_( complex16_t *cx, double *a1, double *a2 )
{
   cx->r = *a1 + *a2;
   cx->i = *a1 - *a2;
}
```

You should note the following about Figure 8:

- A C function that is declared as type `void` simulates a FORTRAN function that returns `COMPLEX` variables.
- The first parameter of the C functions, `makecx8_` and `makecx16_`, contains the address of the `COMPLEX` variable that they are returning.
- The second parameter of the C functions corresponds to the first argument that the FORTRAN main program passes to them, `B1` and `D1`, respectively.
- The third parameter of the C functions corresponds to the second argument that the FORTRAN main program passes to them, `B2` and `D2`, respectively.

The command lines to compile this example and its output are:

```
% cc -c fcxfunc2.c
% fc -sa fcxfunc.f fcxfunc2.o
% a.out
 (  12.00000    , -2.000000    )
 (  12.0000000000000     , -2.00000000000000     )
```

C calls FORTRAN

5

This chapter contains examples that demonstrate how to access FORTRAN objects from C. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter.

Each example includes the source code for C and FORTRAN; C source code appears in lowercase while FORTRAN source code appears in uppercase. Command lines are provided so that you can reproduce the examples.

The examples in this chapter show you how to perform the following tasks:

- Access FORTRAN named and blank common blocks
- Pass integer and floating-point arguments to a FORTRAN function
- Pass string arguments to a FORTRAN function
- Pass complex arguments to a FORTRAN function
- Pass array arguments to a FORTRAN function
- Call FORTRAN functions that return integer and floating-point values
- Call FORTRAN functions that return string values
- Call FORTRAN functions that return complex values

Accessing common blocks

The example in Figure 9 shows how to access named and blank common blocks in FORTRAN.

Figure 9
Accessing FORTRAN common blocks in C

```
        PROGRAM MAIN                    ! Source file is fccmn.f
        REAL*8 A, X                     ! Use 4-byte or 8-byte data types only
        INTEGER*4 B, Y
        COMMON A, B
        COMMON /NAMED/ X, Y
        CALL SUB1 ( )
        PRINT *, A, B
        PRINT *, X, Y
        END

struct block { double a; int b; };          /* Source file is fccmn2.c */

void sub1_( )
{
   extern struct block __blnk_;          /* Define external structures */
   extern struct block _named_;

   __blnk_.a = 3.1415927;
   __blnk_.b = 61659;

   _named_.a = 2.7182818;
   _named_.b = 95616;
}
```

You should note the following about Figure 9:

- C structures are used to access common block areas defined in FORTRAN.
- C references the blank common area as `__blnk_`, because the C compiler only prepends an underscore to this data object. In contrast, the symbol that the FORTRAN compiler produces for the common block data is `___blnk_`.
- The symbol that FORTRAN produces for the NAMED common block is `__named_`. The C function refers to this symbol as `_named_` because the C compiler only prepends an underscore to external objects.

The command lines to compile this example and its output are:

```
% cc -c fccmn2.c
% fc fccmn.f fccmn2.o
% a.out
   3.14159270000000              61659
   2.71828180000000              95616
```

Integer and floating-point arguments

The example in Figure 10 shows how to pass the `int` and `float` data types to a FORTRAN routine.

Figure 10
Passing integer and floating-point parameters from C to FORTRAN

```
#include <stdio.h>                    /* Source file is cfir.c */
extern void sub1_( int *integer, float *real );

main()
{
   int     i = 0;
   float   a = 0.0;
   sub1_(&i,&a);
   (void) printf ("%d\n%f\n", i, a );
}
```

```
      SUBROUTINE SUB1( J, B )       ! Source file is cfir2.f
      INTEGER*4 J
      REAL*4 B
      J = 1234567
      B = 9876.543
      RETURN
      END
```

You should note the following about Figure 10:

- The FORTRAN subroutine `SUB1` is declared in the C source file as a function returning type `void`.
- Parameters `i` and `a` are passed as addresses because FORTRAN accesses parameters by reference.
- An underscore is appended to the C function name `sub1_` because the C compiler only prepends an underscore, while the FORTRAN compiler prepends and appends an underscore to a subroutine name.

The command lines to compile this program and its output are:

```
% fc -c cfir2.f
% cc cfir.c cfir2.o
% a.out
1234567
9876.542969
```

The cc driver loads this program because the main program unit is written in C.

String arguments

The example in Figure 11 shows how to pass string arguments to a FORTRAN routine.

Figure 11
Passing string parameters from C to FORTRAN

```
#include <stdio.h>                          /* Source file is cfchar.c      */
extern void sub1_( char str1[], char str2[], int *arg3,
                   long len1, long len2 );

main ( )
{
   char str1[10];
   char str2[6];
   int num = 4;
   sub1_(str1,str2,&num,10,6);     /* Pass the character string  */
   str1[9] = '\0';                 /*     lengths by value.      */
   str2[5] = '\0';
   (void) printf ("%s %s\n",str1,str2);
}
```

```
      SUBROUTINE SUB1 (CVAR1,CVAR2,CVAR3)     ! Source file is cfchar2.f
      CHARACTER*10 CVAR1
      CHARACTER*6  CVAR2
      INTEGER CVAR3
      CVAR1 = '123456789'
      CVAR2 = 'abcde'
      RETURN
      END
```

You should note the following about Figure 11:

- Null characters are appended to the end of the strings because this is the convention understood by C library routines such as `printf`.
- The `sub1_` declaration and its invocation contain two additional parameters that are not present in the dummy argument list of subroutine `SUB1`. The `len1` argument specifies the declared length of `str1`, and `len2` specifies the declared length of `str2`.
- Before `str1` and `str2` can be displayed with `printf`, a null character (\0) must be added to the end of the strings. The `printf` function requires strings that are terminated by null characters. FORTRAN does not automatically append a null character at the end of a string.

The command lines to compile this program and its output are:

```
% fc -c cfchar2.f
% cc cfchar.c cfchar2.o
% a.out
123456789 abcde
```

Complex arguments

The example in Figure 12 shows how to pass structures that represent COMPLEX data types to a FORTRAN routine.

Figure 12
Passing a structure representing a complex data type from C to FORTRAN

```
#include <stdio.h>                    /* Source file is cfcmplx.c        */
typedef struct{ float  r, i;} complex8_t;
typedef struct{ double r, i;} complex16_t;

extern void sub1_( complex8_t *cx1, complex16_t *cx2);

main ( )
{
   complex8_t cx1;
   complex16_t cx2;
   sub1_ (&cx1, &cx2);
   (void)printf ("(%f,%f)\n",cx1.r,cx1.i);
   (void)printf ("(%f,%f)\n",cx2.r,cx2.i);
}
```

```
      SUBROUTINE SUB1 (CX1,CX2)     ! Source file is cfcmplx2.f
      COMPLEX*8 CX1
      COMPLEX*16 CX2
      CX1 = (1234.0,5678.0)
      CX2 = (8765.0,4321.0)
      RETURN
      END
```

Note that the compatible data type in C for the COMPLEX data type in FORTRAN is a `struct`. The order of its components is important.

The command lines to compile this program and its output are:

```
% fc -c cfcmplx2.f
% cc cfcmplx.c cfcmplx2.o
% a.out
(1234.000000,5678.000000)
(8765.000000,4321.000000)
```

Array arguments

The example in Figure 13 shows how to pass a two-dimensional array to a FORTRAN routine.

Figure 13
C source code that passes a two-dimensional array to FORTRAN

```
#include <stdio.h>                    /* Source file is cfarr.c */
extern void sub1_( int a[][3] );

main ( )
{
   int a[5][3];
   int i;
   sub1_(a);
   for (i=0; i<5; i++)
      (void)printf ("%3d %3d %3d\n",a[i][0],a[i][1],a[i][2]);
}
```

C stores arrays in row-major order, while FORTRAN stores arrays in column-major order. Consequently, FORTRAN source code that receives multidimensional C arrays must be modified to recognize this.

There are two ways to modify the FORTRAN source code: use the ROW_WISE directive or reverse the order of the indices. Figure 14 uses the ROW_WISE directive to force FORTRAN to access the array as a row-major array.

Figure 14
Modifying a FORTRAN array with the ROW_WISE directive

```
      SUBROUTINE SUB1 (A)             ! Source file is cfarr2a.f
C$DIR ROW_WISE (A)
      INTEGER A(5,3)
      DO I = 1, 5
         DO J = 1, 3
            A(I,J) = I*J
         ENDDO
      ENDDO
      RETURN
      END
```

The format of the ROW_WISE directive is as follows:

ROW_WISE(*array_name* [,...])

The following rules apply to using the ROW_WISE directive:

- Implicit array I/O, such as READ(5,*) A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears transposed when viewing it with a debugger.
- You cannot use the ROW_WISE directive with dummy arguments.

C does not have a pragma (directive) comparable to FORTRAN's ROW_WISE directive. Refer to the *CONVEX FORTRAN Optimization Guide* for information on the optimization aspects of this directive.

The second way to modify the FORTRAN source code is to reverse the order of the indices. This requires the array declaration to be modified as well as any loops that access the array's elements. This method is shown in Figure 15.

Figure 15
Modifying a FORTRAN array by reversing its indices

```
SUBROUTINE SUB1 (A)            ! Source file is cfarr2b.f
INTEGER A(3,5)
DO J = 1, 5
   DO I = 1, 3
      A(I,J) = I*J
   ENDDO
ENDDO
RETURN
END
```

The command lines to compile this program using both modifications of the FORTRAN source code are:

```
% fc -c cfarr2a.f
% fc -c cfarr2b.f
% cc cfarr.c cfarr2a.o
% a.out
  1   2   3
  2   4   6
  3   6   9
  4   8  12
  5  10  15
```

```
% cc cfarr.o cfarr2b.o
% a.out
  1   2   3
  2   4   6
  3   6   9
  4   8  12
  5  10  15
```

Integer and floating-point return values

The example in Figure 16 shows how to call FORTRAN functions that return `int` and `float` data types.

Figure 16
Calling integer and floating-point FORTRAN functions from C

```
#include <stdio.h>                    /* Source file is cffunc.c */
extern float addem_(float *a1, float *a2);
extern int iaddem_(int *i1, int *i2);

main ( )
{
   int i1 = 43;
   int i2 = 12;
   float a1 = 43.;
   float a2 = 12.;
   (void)printf("%d\n", iaddem_(&i1,&i2) );
   (void)printf("%f\n", addem_(&a1,&a2)  );
}
```

```
      INTEGER FUNCTION IADDEM (J1, J2)      ! Source file is cffunc2.f
      INTEGER J1, J2
      IADDEM = J1 + J2
      RETURN
      END

      REAL FUNCTION ADDEM(B1, B2)
      REAL B1, B2
      ADDEM = B1 + B2
      RETURN
      END
```

Note that the two function declarations, `addem_` and `iaddem_` return `float` and `int` objects, the C data types compatible with the default FORTRAN data types REAL and INTEGER, respectively.

The command lines to compile this program and its output are:

```
% fc -c cffunc2.f
% cc cffunc.c cffunc2.o
% a.out
55
55.000000
```

String return values

The example in Figure 17 shows how to call a FORTRAN function that returns a string.

Figure 17
Calling a string FORTRAN function from C

```
#include <stdio.h>                              /* Source file is cfcfunc.c */
extern void makestr_(char cstring[], long int cstr_len, char *arg1,
                     int arg2, long int arg1_len );

main()
{
   char cstring[20];
   char wxyz[] = " wxyz";
   int arg = 60;
   makestr_( cstring, 20L, wxyz, arg, sizeof(wxyz)-1 );
   (void)printf("%s\n", cstring );
}
```

```
      CHARACTER*20 FUNCTION MAKESTR (CVAR1,ARG)  ! Source is cfcfunc2.f
      CHARACTER*5 CVAR1
      INTEGER ARG
      MAKESTR = CVAR1//' edcba'//CHAR(0)
      RETURN
      END
```

You should note the following about Figure 17:

- Even though a FORTRAN *function* is being called, its return type in the C declaration of `makestr_` is type `void`.
- The first parameter of `makestr_` is a pointer to the character variable returned by the FORTRAN function.
- The second parameter of `makestr_` contains the amount of space allocated for the character variable returned by the FORTRAN function. This allocation occurs in the C main program.

- The third parameter of `makestr_` is a pointer to a string that MAKESTR uses.
- The fourth parameter of `makestr_` is not used in MAKESTR. It is present only to demonstrate that `arg1_len` is appended to the end of the actual parameter list.
- The last parameter of `makestr_` contains the length of its third parameter. In this example, it is computed using the `sizeof` macro, which returns the number of bytes in an array operand. A 1 is subtracted to strip the null (`\0`) character from the end of the string.
- The FORTRAN function, MAKESTR, concatenates two strings and appends the null character required by C using the CHAR() function.

The command lines to compile this program and its output are:

```
% fc -c cfcfunc2.f
% cc cfcfunc.c cfcfunc2.o -lF77
% a.out
 wxyz edcba
```

The concatenation operation in the MAKESTR function uses an intrinsic function called `for$s_cat`. Because this function is contained in `libF77.a`, you must include `-lF77` at the end of the cc command line to compile this example.

Complex return values

The example in Figure 18 shows how to call a FORTRAN function that returns a structure simulating the COMPLEX*8 FORTRAN data type.

Figure 18
Calling a complex FORTRAN function from C

```
#include <stdio.h>                    /* Source file is cfxfunc.c */
typedef struct { float r, i;} complex8_t;
extern void makecx_( complex8_t *cx, float *a1, float *a2);

main()
{
   float a1=43.;
   float a2=12.;
   complex8_t cx;
   makecx_(&cx,&a1,&a2);
   (void)printf("(%f,%f)\n", cx.r, cx.i );
}
```

```
      COMPLEX FUNCTION MAKECX (B1, B2)      ! Source file is cfxfunc2.f
      REAL B1, B2
      MAKECX = CMPLX (B1*2,B2*2)
      RETURN
      END
```

You should note the following about Figure 18:

- Even though a FORTRAN *function* is being called, its return type in the C declaration of makecx_ is type void.
- The first parameter of makecx_ is a pointer to the complex structure returned by the FORTRAN function.

The command lines to compile this program and its output are:

```
% fc -c cfxfunc2.f
% cc cfxfunc.c cfxfunc2.o
% a.out
(86.000000,24.000000)
```

FORTRAN calls Ada

6

This chapter contains examples that demonstrate how to access Ada objects from FORTRAN. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter.

The following topics are contained in this chapter:

- Elaboration
- Linking FORTRAN main program

The examples in this chapter show you how to perform the following tasks:

- Access Ada global variables using FORTRAN common blocks
- Pass integer and floating-point arguments to a FORTRAN routine
- Pass string arguments to a FORTRAN routine
- Pass complex arguments to a FORTRAN routine
- Pass array arguments to a FORTRAN routine
- Call FORTRAN functions that return integer and floating point types

Each example includes the source code for Ada and FORTRAN; Ada source code appears in mixed case while FORTRAN source code appears in uppercase. Command lines are provided so that you can reproduce the examples.

Elaboration

Elaboration is the process by which a declaration achieves its effect; elaboration occurs during program execution. For example, elaboration of

```
NUM : constant INTEGER := 5;
```

creates the NUM object and initializes it with the value 5.

Ada objects cannot be accessed before they have been elaborated. If you attempt to access an Ada object that has not been elaborated, the PROGRAM_ERROR exception will probably be raised.

The method of compiling and linking Ada routines with FORTRAN routines described in this chapter permits the Ada routines to require elaboration. The only way to create a program in which the main program unit is in FORTRAN is to call that unit from an Ada procedure. You cannot link a program that has a FORTRAN main program unit with the `fc` command line.

Linking FORTRAN main programs

The Ada source code below is an example of an Ada main program that calls a FORTRAN main program:

```
procedure AMAIN is
    procedure FMAIN;
    pragma INTERFACE( FORTRAN, FMAIN );
    pragma INTERFACE_NAME( FMAIN, "_MAIN__");
begin
    FMAIN;
end AMAIN;
```

The entry point in a FORTRAN main program is _MAIN__. Consequently, the pragma INTERFACE_NAME is required to replace all references to FMAIN with _MAIN__ in the Ada object file. Because the FORTRAN main program is called by an Ada main program, any elaboration that is required will be performed prior to the call to the FORTRAN main program.

There are a few problems with this method. Because the initial routine is in Ada, the Ada runtime system is created. By calling the _MAIN__ routine, you initiate a subset of the FORTRAN runtime system. One side effect is that the FORTRAN main program no longer has access to command line arguments with

the getarg and iargc FORTRAN library functions. You can work around this problem using the COMMAND_LINE package in convexlib.

The routines F_INIT and F_EXIT are C routines that initialize FORTRAN input and output. You must call the F_INIT routine before any FORTRAN I/O takes place, and you should call the F_EXIT routine only after all FORTRAN source code has executed. You should not mix FORTRAN and Ada I/O; that is, you should not perform both Ada and FORTRAN I/O on the same file. Refer to Chapter 10, "Mixing input and output," for more details.

You should remember that the Ada compiler disables integer overflow detection before a call to a FORTRAN routine by default. Any subsequent calls from that FORTRAN routine to an Ada routine will also have integer overflow detection disabled.

Accessing global variables

The following example shows how to access global objects in Ada. It consists of three source files: an Ada main that calls a FORTRAN main, and an Ada procedure. Figure 19 contains the Ada main program.

Figure 19
Accessing global variables in Ada, Ada main program

```
with ASUB;                                    -- Source file is amain.a
procedure AMAIN is
   procedure FMAIN;
   pragma INTERFACE(FORTRAN, FMAIN);
   pragma INTERFACE_NAME(FMAIN, "_MAIN__");
   procedure F_INIT;
   pragma INTERFACE(C, F_INIT);
   procedure F_EXIT;
   pragma INTERFACE(C, F_EXIT);
begin
   F_INIT;
   FMAIN;
   F_EXIT;
end AMAIN;
```

FMAIN is the name that this Ada routine uses to refer to the FORTRAN main program. Because FMAIN is a FORTRAN routine, you must use the pragma INTERFACE to indicate this to the Ada compiler. Moreover, the FORTRAN compiler generates

the global name _MAIN__ for all FORTRAN main programs. The pragma INTERFACE_NAME converts all references to FMAIN to the name that the FORTRAN compiler uses, _MAIN__.

Figure 20 contains the FORTRAN main program that is called by the Ada main program.

Figure 20
Accessing global variables in Ada, FORTRAN main program

```
      PROGRAM FMAIN          ! Source file is fmain.f

      COMMON /ADA_BLOCK/ A,B,C
      COMMON X,Y,Z
      INTEGER*4 A,X
      INTEGER*2 C,Z
      REAL*4    B,Y

      CALL ADA_SUB

      PRINT *, A,B,C
      PRINT *, X,Y,Z
      END
```

Figure 21 contains the Ada routine that is called from the FORTRAN main program.

Figure 21
Accessing global variables in Ada, Ada routine

```
package ASUB is                          -- Source file is asub.a
  type COM_RECORD is
    record
      I : INTEGER;
      J : SHORT_FLOAT;
      K : SHORT_INTEGER;
  end record;
  NAMED_COMMON, BLANK_COMMON : COM_RECORD;
  procedure ADA_SUB;
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub_" );
  pragma EXTERNAL_NAME( NAMED_COMMON, "__ada_block_" );
  pragma EXTERNAL_NAME( BLANK_COMMON, "___blnk_" );
end ASUB;

package body ASUB is
  procedure ADA_SUB is
  begin
    NAMED_COMMON.I := 4;
    NAMED_COMMON.J := 4.0;
    NAMED_COMMON.K := 3#222#;
    BLANK_COMMON.I := -4;
    BLANK_COMMON.J := -4.0;
    BLANK_COMMON.K := 3#111#;
  end ADA_SUB;
end ASUB;
```

You should note the following about the example in this section:

- The pragma EXTERNAL_NAME in source file asub.a defines the link names that are accessible to the FORTRAN main program. The external names that the Ada source code defines are _ada_sub_, __ada_block_, and ___blnk_. _ada_sub_ is referenced as ADA_SUB in the FORTRAN routine. The symbol __ada_block_ is referenced as the ADA_BLOCK named common block in FORTRAN. And the symbol ___blnk_ is referenced as a blank common block in FORTRAN.
- Common blocks are the only way that FORTRAN can share global variables between routines. If a global Ada variable is not accessible in a common block, then a FORTRAN routine cannot access it. Consequently, you must use the pragma EXTERNAL_NAME to create a link name that conforms to the FORTRAN common block naming convention.

- If there is more than one variable in the common block, you must use a record that includes all the variables in the common block. This example shows how FORTRAN can access six global Ada variables: three in a named common block (ADA_BLOCK), and three in the blank common block.
- FORTRAN aligns data types in common blocks the same way that Ada aligns compatible data types in records. Consequently, you do not need to pad the record with unused fields; both compilers automatically pad memory allocations.

The command lines to compile this example and its output are:

```
% ada asub.a amain.a
asub.a:
amain.a:
% fc -c fmain.f
% a.make amain fmain.o -flink
% a.out
           4    4.000000          26
          -4   -4.000000          13
```

The `a.make` utility is used to link this program. The `a.make` utility does not understand any dependencies between FORTRAN source files and Ada source files. Consequently, if you only modify FORTRAN source files you must remove the executable file to force `a.make` to link in any new FORTRAN object files.

The `-flink` option causes FORTRAN libraries to be added during the final link step.

Integer and floating-point arguments

The following example shows how to pass integer and floating-point data types to an Ada routine. This example also demonstrates how to avoid elaboration and start the program using the FORTRAN runtime system instead of Ada. Figure 22 contains the FORTRAN main program.

Figure 22
Passing integer and floating-point arguments from FORTRAN to Ada, FORTRAN main

```
      PROGRAM FMAIN    ! Source file is fair.f

      INTEGER*4 I
      INTEGER*8 LO
      REAL*4    A

      CALL ADA_SUB( I, LO, A )

      PRINT *, I, LO, A
      END
```

Figure 23 contains the Ada routine that is called from the FORTRAN main program. It also contains an Ada procedure that indirectly calls the FORTRAN main program.

Figure 23
Passing integer and floating-point arguments from FORTRAN to Ada, Ada routines

```
with SYSTEM;   use SYSTEM;                          -- Source file is fair.a
with LONG_INT; use LONG_INT;
package ASUB2 is
  procedure ADA_SUB( INT4 : in SYSTEM.ADDRESS;
                     LONG : in SYSTEM.ADDRESS;
                     REAL : in SYSTEM.ADDRESS );
  pragma external_name( ADA_SUB, "_ada_sub_" );
end asub2;

package body ASUB2 is
  procedure ADA_SUB( INT4 : in SYSTEM.ADDRESS;
                     LONG : in SYSTEM.ADDRESS;
                     REAL : in SYSTEM.ADDRESS ) is
     T_INT4 : INTEGER;
     T_LONG : LONG_INTEGER;
     T_REAL : SHORT_FLOAT;
     for T_INT4 use at INT4;
     for T_LONG use at LONG;
     for T_REAL use at REAL;
  begin
     T_INT4 := -4;
     T_LONG := -8;
     T_REAL := -4.0;
  end ADA_SUB;
end ASUB2;
```

Figure 23 (continued)
Passing integer and floating-point arguments from FORTRAN to Ada, Ada routines

```
with ASUB2;
procedure FAIR is
  procedure FMAIN;
  pragma interface(FORTRAN, FMAIN);
  pragma interface_name(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);
begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FAIR;
```

You should note the following about this example:

- FORTRAN passes integer and floating-point data types by address. Consequently, the corresponding formal parameters in the Ada routine must be type SYSTEM.ADDRESS with mode in. To access the values stored at those addresses, you must use an address clause (for NAME use at ADDRESS).
- The pragma EXTERNAL_NAME creates the link name for the Ada routine. In this case, _ada_sub_ conforms to the FORTRAN calling conventions.

The command lines to compile this program and its output are:

```
% fc -sa -c fair.f
% ada -M fair.a fair.o -flink
% a.out
          -4                          -8  -4.000000
```

Character and string arguments

All FORTRAN parameters are passed by address. However, parameters that are type CHARACTER require an additional argument that gives the length of the CHARACTER array that the FORTRAN array is passing. These additional arguments follow the last actual argument in the routine call. Each of these length arguments requires 4 bytes of storage. (Refer to the section "FORTRAN argument packets," on page 167 for more details.)

Not only must the length of a character string be included in the actual argument list, but character length arguments must be passed by value. The following example demonstrates one way to do this. Figure 24 contains the FORTRAN main program.

Figure 24
Passing string parameters from FORTRAN to Ada, FORTRAN main program

```
      PROGRAM FMAIN                                  ! Source file is fachar.f

      CHARACTER*10 STR
      INTEGER*4 NUM_SET

      STR = 'ZZZZZZZZZZ'
      NUM_SET = 5
      CALL ADA_SUB( STR, NUM_SET )

      PRINT *, STR, NUM_SET
      END
```

Figure 25 contains the Ada procedure that is called from the FORTRAN main program. It also contains the Ada procedure that invokes the FORTRAN main program.

Figure 25
Passing string parameters from FORTRAN to Ada, Ada routines

```
with SYSTEM;   use SYSTEM;                    -- Source file is fachar.a
package ASUB3 is
  procedure ADA_SUB( S_ADDR : in SYSTEM.ADDRESS;
                     D_ADDR : in SYSTEM.ADDRESS;
                     S_LEN  : in INTEGER );
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub_" );
end ASUB3;

package body ASUB3 is
  procedure ADA_SUB( S_ADDR : in SYSTEM.ADDRESS;
                     D_ADDR : in SYSTEM.ADDRESS;
                     S_LEN  : in INTEGER ) is
    NAME : STRING( 1..S_LEN );
    for NAME use at S_ADDR;
    NSET : INTEGER;
    for NSET use at D_ADDR;

 begin
    for I in 1..NSET loop
       NAME( I ) := 'A';
    end loop;
```

Figure 25 (continued)
Passing string parameters from FORTRAN to Ada, Ada routines

```
  end ADA_SUB;
end ASUB3;

with ASUB3;
procedure FACHAR is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
  pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);
begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FACHAR;
```

You should note the following about the example in this section:

- As you can see in Figure 24, the Ada routine, ADA_SUB, that the FORTRAN main program is calling has two actual parameters: the first is a string, and the second is an integer. However, in Figure 25, the Ada routine declares three parameters. The third argument corresponds to the length argument that the FORTRAN compiler automatically appends to the end of the actual parameter list.
- The first two parameters in the Ada routine, ADA_SUB, have type SYSTEM.ADDRESS because they correspond to the two arguments that the FORTRAN main program passes by reference. The third argument, S_LEN, is passed by value and requires four bytes, so it has type INTEGER.
- The statements

  ```
  NAME : STRING( 1..S_LEN );
  for NAME use at S_ADDR;
  ```

  declare an array that has the same length as the FORTRAN variable STR; the base address of this array is S_ADDR.

The command lines to compile this example and its output are:

```
% fc -sa -c fachar.f
% ada -M fachar.a fachar.o -flink
% a.out
AAAAAZZZZZ          5
```

Complex arguments

The example in this section shows how to pass complex arguments from FORTRAN to Ada. Because Ada does not have a complex data type, it must use a record to represent the complex FORTRAN data type. Figure 26 contains the FORTRAN main program.

Figure 26
Passing complex data types from FORTRAN to Ada, FORTRAN main program

```
      PROGRAM FMAIN     ! Source file is facmx.f

      COMPLEX*8 C
      COMPLEX*16 D

      C = (-1., 1.)
      D = (-3., -2.)
      PRINT *, C, D
      CALL ADA_SUB( C, D )
      PRINT *, C, D
      END
```

Figure 27 contains the Ada procedure that is called from the FORTRAN main program. It also contains the Ada procedure that invokes the FORTRAN main program.

Figure 27
Passing complex data types from FORTRAN to Ada, Ada routines

```
with SYSTEM; use SYSTEM;                          -- Source file is facmx.a
package ASUB4 is
  procedure ADA_SUB( C1_ADDR : in SYSTEM.ADDRESS;
                     C2_ADDR : in SYSTEM.ADDRESS );
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub_" );
end ASUB4;

package body ASUB4 is
  procedure ADA_SUB( C1_ADDR : in SYSTEM.ADDRESS;
                     C2_ADDR : in SYSTEM.ADDRESS ) is

     type COMPLEX16 is
       record
        RE, IM : FLOAT;
     end record;

     type COMPLEX8 is
       record
        RE, IM : SHORT_FLOAT;
     end record;
     CL1 : COMPLEX8;
     for CL1 use at C1_ADDR;
     CL2 : COMPLEX16;
     for CL2 use at C2_ADDR;
  begin
     CL1 := (5.0, 10.0 );
     CL2 := (4.0, 11.0 );
  end ADA_SUB;
end ASUB4;

with ASUB4;
procedure FACMX is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
  pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);

begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FACMX;
```

You should note the following about this example:

- Ada does not have a simple data type that is compatible with FORTRAN's complex data types. Ada must use records to represent the COMPLEX*8 and COMPLEX*16 data types. (CONVEX Ada defines a generic COMPLEX data type in convexlib/complex.a.) The order of the fields in the record is important.
- FORTRAN passes arguments by address. Consequently, the corresponding formal parameters in the Ada routine must be type SYSTEM.ADDRESS with mode in. To access the values stored at those addresses, you must use an address clause (for NAME use at ADDRESS).

The command lines to compile this program and its output are:

```
% fc -sa -c facmx.f
% ada -M facmx.a facmx.o -flink
% a.out
 ( -1.000000     ,  1.000000     )
 ( -3.00000000000000     , -2.00000000000000     )
 (  5.000000     ,  10.00000     )
 (  4.00000000000000     ,  11.0000000000000     )
```

Array arguments

The examples in this section show two ways to modify FORTRAN code so that its arrays are compatible with arrays stored in Ada: use the ROW_WISE directive or reverse the order of the indices. Ada stores arrays in row-major order, while FORTRAN stores arrays in column-major order. Consequently, FORTRAN source code that passes multidimensional arrays to Ada must be modified to recognize this. Figure 28 contains the Ada source code for the examples in this section.

Figure 28
Ada source that receives a two-dimensional array from FORTRAN

```
with SYSTEM;   use SYSTEM;                          -- Source file is faarr.a
package ASUB5 is
  type ARRAY_2D is
    array (1..4, 1..2) of SHORT_FLOAT;
  procedure ADA_SUB( ADA_ARRAY : in out ARRAY_2D );
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub_" );
end ASUB5;

package body ASUB5 is
  procedure ADA_SUB(  ADA_ARRAY : in out ARRAY_2D ) is

  begin
    for I in ADA_ARRAY'RANGE(1) loop
      for J in ADA_ARRAY'RANGE(2) loop
        ADA_ARRAY(I,J) := SHORT_FLOAT( I * J );
      end loop;
    end loop;
  end ADA_SUB;
end ASUB5;

with ASUB5;
procedure FAARR is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
  pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);
begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FAARR;
```

The first method uses the ROW_WISE directive ROW_WISE to force FORTRAN to access the array as a row-major array as shown in Figure 29.

Figure 29
Modifying a FORTRAN array with the ROW_WISE directive

```
        PROGRAM FMAIN    ! Source file is faarr.f

C$DIR ROW_WISE (ARRAY)
        INTEGER I,J
        REAL*4 ARRAY(4,2)

        CALL ADA_SUB( ARRAY )

        DO I=1,4
          PRINT *, ARRAY(I,1), ARRAY(I,2)
        END DO
        END
```

The format of the ROW_WISE directive is as follows:

ROW_WISE(*array_name* [,...])

The following rules apply to using the ROW_WISE directive:

- Implicit array I/O, such as READ(5,*) A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears transposed when viewed with a debugger.
- You cannot use the ROW_WISE directive with dummy arguments.

Ada does not have a pragma (directive) comparable to FORTRAN's ROW_WISE directive. Refer to the *CONVEX FORTRAN Optimization Guide* for information on the optimization aspects of this directive.

The second way to modify the FORTRAN source code is to reverse the order of the indices. This requires the array declaration to be modified as well as any loops that access the array's elements. This method is shown in Figure 30.

Figure 30
Modifying a FORTRAN array by reversing its indices

```
PROGRAM FMAIN     ! Source file is faarrf.f

INTEGER I,J
REAL*4 ARRAY(2,4)

CALL ADA_SUB( ARRAY )

DO J=1,4
  PRINT *, ARRAY(1,J), ARRAY(2,J)
END DO
END
```

Notice that the mode of array ADA_ARRAY in the Ada routine is `in out`. You must use this mode type because the array is being modified in the Ada routine.

The command lines to compile this program using both modifications of the FORTRAN source code and their output are:

```
% fc -sa -c faarr.f
% ada -M faarr.a faarr.o -flink
% a.out
   1.000000        2.000000
   2.000000        4.000000
   3.000000        6.000000
   4.000000        8.000000
% fc -sa -c faarrf.f
% ada -M faarr.a faarrf.o -flink
% a.out
   1.000000        2.000000
   2.000000        4.000000
   3.000000        6.000000
   4.000000        8.000000
```

Integer and floating-point return values

The example in this section shows how to call Ada functions that return INTEGER*2, INTEGER*4, REAL*4, and REAL*8 data types. Figure 31 shows the FORTRAN main routine.

Figure 31
Calling integer and floating point Ada functions from FORTRAN, FORTRAN main

```
      PROGRAM FMAIN    ! Source file is fafunc.f

      INTEGER*2 I, J, ADD_INT2
      INTEGER*4 K, L, ADD_INT4
      REAL*4    A, B, ADD_REAL4
      REAL*8    C, D, ADD_REAL8

      I = 4
      J = 4
      K = -10
      L = -20
      A = 4.5
      B = 5.4
      C = 9.8
      D = -12.3
      PRINT *, ADD_INT2(I,J)
      PRINT *, ADD_INT4(K,L)
      PRINT *, ADD_REAL4(A,B)
      PRINT *, ADD_REAL8(C,D)
      END
```

Figure 32 shows the Ada routines.

Figure 32
Calling integer and floating-point Ada functions from FORTRAN, Ada routines

```
with SYSTEM;  use SYSTEM;          -- Source file is fafunc.a

package ASUB7 is

  function ADD_INT2( ARG1_INT2: in SYSTEM.ADDRESS;
                     ARG2_INT2: in SYSTEM.ADDRESS ) return SHORT_INTEGER;
  function ADD_INT4( ARG1_INT4: in SYSTEM.ADDRESS;
                     ARG2_INT4: in SYSTEM.ADDRESS ) return INTEGER;
  function ADD_REAL4( ARG1_REAL4: in SYSTEM.ADDRESS;
                      ARG2_REAL4: in SYSTEM.ADDRESS ) return SHORT_FLOAT;
  function ADD_REAL8( ARG1_REAL8: in SYSTEM.ADDRESS;
                      ARG2_REAL8: in SYSTEM.ADDRESS ) return FLOAT;

  pragma EXTERNAL_NAME(ADD_INT2, "_add_int2_");
  pragma EXTERNAL_NAME(ADD_INT4, "_add_int4_");
  pragma EXTERNAL_NAME(ADD_REAL4, "_add_real4_");
  pragma EXTERNAL_NAME(ADD_REAL8, "_add_real8_");

end ASUB7;
```

Figure 32 (continued)
Calling integer and floating-point Ada functions from FORTRAN, Ada routines

```
package body ASUB7 is
  function ADD_INT2( ARG1_INT2: in SYSTEM.ADDRESS;
                     ARG2_INT2: in SYSTEM.ADDRESS ) return SHORT_INTEGER is
    N1 : SHORT_INTEGER;
    N2 : SHORT_INTEGER;
    for N1 use at ARG1_INT2;
    for N2 use at ARG2_INT2;
  begin
    return N1 + N2;
  end ADD_INT2;

  function ADD_INT4( ARG1_INT4: in SYSTEM.ADDRESS;
                     ARG2_INT4: in SYSTEM.ADDRESS ) return INTEGER is
    N1 : INTEGER;
    N2 : INTEGER;
    for N1 use at ARG1_INT4;
    for N2 use at ARG2_INT4;
  begin
    return N1 + N2;
  end ADD_INT4;

  function ADD_REAL4( ARG1_REAL4: in SYSTEM.ADDRESS;
                      ARG2_REAL4: in SYSTEM.ADDRESS ) return SHORT_FLOAT is
    N1 : SHORT_FLOAT;
    N2 : SHORT_FLOAT;
    for N1 use at ARG1_REAL4;
    for N2 use at ARG2_REAL4;
  begin
    return N1 + N2;
  end ADD_REAL4;

  function ADD_REAL8( ARG1_REAL8: in SYSTEM.ADDRESS;
                      ARG2_REAL8: in SYSTEM.ADDRESS ) return FLOAT is
    N1 : FLOAT;
    N2 : FLOAT;
    for N1 use at ARG1_REAL8;
    for N2 use at ARG2_REAL8;
  begin
    return N1 + N2;
  end ADD_REAL8;
end ASUB7;
with ASUB7;
procedure FAFUNC is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
```

Figure 32 (continued)
Calling integer and floating-point Ada functions form FORTRAN, Ada routines

```
   pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
   procedure F_INIT;
   pragma INTERFACE(C, F_INIT);
   procedure F_EXIT;
   pragma INTERFACE(C, F_EXIT);
begin
   F_INIT;
   FMAIN;
   F_EXIT;
end FAFUNC;
```

You should note that the Ada data types compatible with INTEGER*2, INTEGER*4, REAL*4, and REAL*8 are SHORT_INTEGER, INTEGER, SHORT_FLOAT, and FLOAT. These are the data types that the Ada functions must return to FORTRAN.

Ada functions that return the INTEGER*8 data type are addressed in the section "Complex return values," on page 72.

The command lines used to compile this program and its output are:

```
% fc -sa -c fafunc.f
% ada -M fafunc.a fafunc.o -flink
% a.out
       8
           -30
   9.900000
  -2.50000000000000
```

String return values

The example in this section shows how to call an Ada routine that returns a CHARACTER array. Ada returns a string as a hidden parameter that is appended to the actual arguments in a function call. However, FORTRAN returns strings with an address in the S0 register. Consequently, you cannot call an Ada function that returns a string in the usual way.

Figure 33 contains the FORTRAN main program.

Figure 33
Calling a string Ada function from FORTRAN, FORTRAN main program

```
      PROGRAM FMAIN                                    ! Source file is facfunc.f
      CHARACTER*10 S1, S2
      CHARACTER*20 NEW_S, ADA_FUNC
      S1 = 'ASDFG'
      S2 = 'QWERT'
      NEW_S = ADA_FUNC( S1, S2 )
      PRINT *, NEW_S
      END
```

Figure 34 contains the Ada routines.

Figure 34
Calling a string Ada function from FORTRAN, Ada routines

```
with SYSTEM;   use SYSTEM;              -- Source file is facfunc.a
package ASUB8 is
  procedure ADA_FUNC( S_ADDR : in SYSTEM.ADDRESS;
                      S_LEN  : in INTEGER;
                      ARG1_ADDR: in  SYSTEM.ADDRESS;
                      ARG2_ADDR: in  SYSTEM.ADDRESS;
                      ARG1_LEN : in  INTEGER;
                      ARG2_LEN : in  INTEGER );
  pragma EXTERNAL_NAME( ADA_FUNC, "_ada_func_" );
end ASUB8;

package body ASUB8 is
  procedure ADA_FUNC( S_ADDR : in SYSTEM.ADDRESS;
                      S_LEN  : in INTEGER;
                      ARG1_ADDR: in  SYSTEM.ADDRESS;
                      ARG2_ADDR: in  SYSTEM.ADDRESS;
                      ARG1_LEN : in  INTEGER;
                      ARG2_LEN : in  INTEGER ) is
    SOUT : STRING(1..S_LEN);
    for SOUT use at S_ADDR;
    ARG1 : STRING(1..ARG1_LEN);
    for ARG1 use at ARG1_ADDR;
    ARG2 : STRING(1..ARG2_LEN);
    for ARG2 use at ARG2_ADDR;
    I : INTEGER;
  begin
    SOUT := ARG1 & ARG2;
    I := S_LEN;
    while I > 1 loop
```

Figure 34 (continued)
Calling a string Ada function from FORTRAN, Ada routines

```
      SOUT(I) := 'Z';
      I := I - 3;
    end loop;
  end ADA_FUNC;
end ASUB8;

with ASUB8;
procedure FACFUNC is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
  pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);
begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FACFUNC;
```

You should note the following about this example:

- An Ada procedure (ADA_FUNC in this example) simulates a FORTRAN function that returns a CHARACTER array.
- The first parameter of the Ada procedure contains the address of the string that the Ada procedure creates. This address is created by the calling FORTRAN routine. Similarly, the second parameter, S_LEN, contains the amount of storage that the FORTRAN routine has allocated for the CHARACTER array.

The command lines used to compile this program and its output are:

```
% fc -sa -c facfunc.f
% ada -M facfunc.a facfunc.o -flink
% a.out
AZDFZ  Z  ZWEZT Z  Z
```

Complex return values

The return mechanism that Ada uses for records is not compatible with FORTRAN's calling conventions for COMPLEX*8, COMPLEX*16, or INTEGER*8. Consequently, you cannot call an Ada function that returns these data types in the usual way. Instead, you must create an Ada *procedure* in which the first parameter is the address of the record being returned. The example in this section demonstrates how to create an Ada procedure that FORTRAN can call as a function returning COMPLEX*8. Ada routines that return the COMPLEX*16 and INTEGER*8 can be created similarly.

Figure 35 contains the FORTRAN main program.

Figure 35
Calling a COMPLEX*8 Ada function from FORTRAN, FORTRAN main program

```
      PROGRAM FMAIN                                          ! faxfunc.f
      COMPLEX*8 C1, C2, ADA_FUNC
      C1 = (5.0, 10.0)
      C2 = (-4.0, -20.0)
      PRINT *, ADA_FUNC( C1, C2 )
      END
```

Figure 36 contains the Ada routines that are required.

Figure 36
Calling a COMPLEX*8 Ada function from FORTRAN, Ada routines

```
-- Source file is faxfunc.a
with MATH, COMPLEX_ARITH;
use MATH;
package C8 is new COMPLEX_ARITH(ARG=SHORT_FLOAT,REAL=SHORT_FLOAT);

with C8, SYSTEM;
use C8, SYSTEM;

package ASUB9 is
  procedure ADA_FUNC( C_ADDR : in SYSTEM.ADDRESS;
                      CX1_ADDR : in  SYSTEM.ADDRESS;
                      CX2_ADDR : in  SYSTEM.ADDRESS );
  pragma EXTERNAL_NAME( ADA_FUNC, "_ada_func_" );
end ASUB9;

package body ASUB9 is
  procedure ADA_FUNC( C_ADDR : in SYSTEM.ADDRESS;
                      CX1_ADDR: in  SYSTEM.ADDRESS;
                      CX2_ADDR: in  SYSTEM.ADDRESS ) is
```

Figure 36 (continued)
Calling a COMPLEX*8 Ada function from FORTRAN, Ada routines

```
    CX_OUT : C8.COMPLEX; for CX_OUT use at C_ADDR;
    CX1    : C8.COMPLEX; for CX1 use at CX1_ADDR;
    CX2    : C8.COMPLEX; for CX2 use at CX2_ADDR;
  begin
    CX_OUT := CX1 + CX2;
    return;
  end ADA_FUNC;
end ASUB9;

with ASUB9;
procedure FAXFUNC is
  procedure FMAIN;
  pragma INTERFACE(FORTRAN, FMAIN);
  pragma INTERFACE_NAME(FMAIN, "_MAIN__" );
  procedure F_INIT;
  pragma INTERFACE(C, F_INIT);
  procedure F_EXIT;
  pragma INTERFACE(C, F_EXIT);
begin
  F_INIT;
  FMAIN;
  F_EXIT;
end FAXFUNC;
```

You should note the following about this example:

- Even though FORTRAN is calling a *function*, the Ada routine is a *procedure*. This is the only way that Ada can return records to FORTRAN. Because FORTRAN cannot call functions that return records, this technique can only be used with the COMPLEX*8, COMPLEX*16, INTEGER*8, and REAL*16 FORTRAN data types.
- The first parameter of ADA_FUNC is the address of the value that is returned.

The command lines to compile this program and its output are:

```
% fc -sa -c faxfunc.f
% ada -M faxfunc.a faxfunc.o -flink
% a.out
 (  1.000000    , -10.00000    )
```

Ada calls FORTRAN 7

This chapter contains examples that demonstrate how to access FORTRAN objects from Ada. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter.

Each example includes the source code for Ada and FORTRAN; Ada source code appears in mixed case, while FORTRAN source code appears in upper case. Command lines are provided so that you can reproduce the examples.

The examples in this chapter show you how to perform the following tasks:

- Access FORTRAN named and blank common blocks
- Pass integer and floating-point arguments to a FORTRAN function
- Pass string arguments to a FORTRAN function
- Pass complex arguments to a FORTRAN function
- Pass array arguments to a FORTRAN function
- Call FORTRAN functions that return integer and floating-point values
- Call FORTRAN functions that return string values
- Call FORTRAN functions that return complex values

Accessing common blocks

To access FORTRAN common blocks in Ada you must use the INTERFACE_NAME pragma. The format of this pragma is as follows:

```
pragma INTERFACE_NAME( variable_name, "link_name");
```

This pragma replaces all occurrences of *variable_name* with an external reference to *link_name*. This pragma must apply to a variable declared earlier in the same package specification. You must declare the object as a scalar, array, record, access type, procedure, or function. The object cannot be a loop variable, constant, or an initialized variable. Figure 37 shows how to use this pragma to access FORTRAN common blocks.

Figure 37
Accessing FORTRAN common blocks in Ada

```
with TEXT_IO; use TEXT_IO;        -- Source file is afcmn.a
with SYSTEM;  use SYSTEM;

procedure AFCMN is
  package INT_IO is new INTEGER_IO(INTEGER);
  package REAL_IO is new FLOAT_IO (FLOAT);
  use INT_IO, REAL_IO;

  type BLANK is record
    M : FLOAT;
    N : INTEGER;
  end record ;

  procedure SUB1;
  pragma INTERFACE( FORTRAN, SUB1 );
  F_BLANK : BLANK;
  F_NAMED : BLANK;
  pragma INTERFACE_NAME( F_BLANK, "___blnk_" );
  pragma INTERFACE_NAME( F_NAMED, "__named_" );
begin
  SUB1;
  PUT(F_BLANK.M);
  PUT(F_BLANK.N);
  PUT_LINE("");
  PUT(F_NAMED.M);
  PUT(F_NAMED.N);
  PUT_LINE("");
end AFCMN;
```

Figure 37 (continued)
Accessing FORTRAN common blocks in Ada

```
      SUBROUTINE SUB1()            ! Source file is afcmn2.f
      REAL*8 A, X
      INTEGER*4 B, Y

      COMMON A, B
      COMMON /NAMED/ X, Y

      A = 3.1415927D0
      B = 61659
      X = 2.7182818D0
      Y = 95616
      RETURN
      END
```


You should note the following about Figure 37:

- The link name for objects in the blank common area is ___blnk_ and the link name for objects in a named common area is __*named*_ where *named* is the name of the named common area.
- You must use an Ada record to access the contents of a common area. The Ada compiler aligns the fields in the record in the same way that the FORTRAN compiler aligns the data in a common block; you must declare the fields in the same order and type as the objects in the common area that the record is simulating.

The command lines to compile this example and its output are:

```
% fc -sa -c afcmn2.f
% ada -M afcmn.a afcmn2.o -flink
% a.out
 3.14159270000000E+00       61659
 2.71828180000000E+00       95616
```

Integer and floating-point arguments

The example in Figure 38 shows how to pass INTEGER, LONG_INTEGER, and SHORT_FLOAT Ada data types to a FORTRAN routine.

Figure 38
Passing integer and floating-point arguments from Ada to FORTRAN

```
-- Source file is afir.a
WITH text_io; USE text_io;
WITH system;  USE system;
WITH long_int; USE long_int;

PROCEDURE afir IS
  PROCEDURE sub1( INTEGER      : in SYSTEM.ADDRESS;
                  LONG_INTEGER : in SYSTEM.ADDRESS;
                  SHORT_FLOAT  : in SYSTEM.ADDRESS );
  PRAGMA INTERFACE( fortran, sub1 );
  PACKAGE int_io IS NEW integer_io (INTEGER);
  PACKAGE real_io IS NEW float_io (SHORT_FLOAT);
  USE int_io, real_io;
  i  : INTEGER;
  lo : LONG_INTEGER;
  a  : SHORT_FLOAT;
BEGIN
  i := 0;
  lo := int_to_long(0);
  a := 0.0;

  sub1( i'address, lo'address, a'address );

  put(i);
  put_line("");
  put(long_to_int(lo));
  put_line("");
  put(a);
  put_line("");
END afir;

      SUBROUTINE SUB1( I, LO, A )     ! Source file is afir2.f
      INTEGER*4 I
      INTEGER*8 LO
      REAL*4 A
      I = 10
      LO = 145
      A = 432.98
      RETURN
```

You should note the following about Figure 38:

- The mode of the parameters in procedure `sub1` is `in`. The current implementation of CONVEX Ada does not permit a FORTRAN routine accessed by an Ada program to have parameters of mode `out` or `in out`. The FORTRAN routine can modify the values pointed to by the arguments, but not the arguments themselves.
- The `INTERFACE` pragma supports calls to FORTRAN. The second parameter is the name of the FORTRAN function or subroutine. The parameter types of the routine must be of type `SYSTEM.ADDRESS`. Furthermore, the result type of a FORTRAN function is limited to a scalar type.
- In the actual subroutine call, apply the address attribute to each parameter of `sub1`. You must pass the parameters by address because FORTRAN passes its routine parameters by address.
- The `long_int` package, included in the `convexlib` directory, contains a definition of the `LONG_INTEGER` data type. It also defines some functions, such as `int_to_long` and `long_to_int`, that pertain to this data type. Refer to `convexlib/long_int.a` for more details.

The command lines to compile this program and its output are:

```
% fc -sa -c afir2.f
% ada -M afir.a afir2.o -flink
% a.out
          10
         145
 4.32980E+02
%
```

The `ada` driver loads this program because the main program unit is written in Ada. The `-flink` Ada command line option adds FORTRAN runtime libraries to the link step.

Character and string arguments

As mentioned in previous examples, all FORTRAN parameters are passed by address. However, parameters that are type CHARACTER require an additional argument that gives the length of the CHARACTER array that is passed to the FORTRAN routine. These additional arguments follow the actual arguments in the routine call. Each length argument requires 4 bytes of storage. (Refer to the section, "FORTRAN argument packets," on page 167 for more detail.) For example,

```
SUBROUTINE SUB( I1, B, I2)
CHARACTER*3 B
INTEGER I1,I2
```

requires an Ada call similar to

```
SUB( I1'ADDRESS, B'ADDRESS, I2'ADDRESS, LEN_B )
```

As you can see, the actual routine call has one more argument than the subroutine definition; the fourth argument contains the length of the CHARACTER*3 variable.

Not only must the length of a character string be included in the actual argument list, but character length arguments must be passed by value. Figure 39 demonstrates one method of doing this.

Figure 39
Passing string parameters from Ada to FORTRAN

```
-- Source file is afchar.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

procedure AFCHAR is
  procedure SUB1( C1    : in SYSTEM.ADDRESS;
                  S1    : in SYSTEM.ADDRESS;
                  C2    : in SYSTEM.ADDRESS;
                  C1LEN : in SYSTEM.ADDRESS;
                  SLEN  : in SYSTEM.ADDRESS;
                  C2LEN : in SYSTEM.ADDRESS );
  pragma INTERFACE( FORTRAN, SUB1 );
  C1, C2  : CHARACTER;
  S1      : STRING(1..10);
  SLEN    : SYSTEM.ADDRESS;
  C1LEN, C2LEN : SYSTEM.ADDRESS;
begin
  SLEN := SYSTEM.MEMORY_ADDRESS( INTEGER( S1'LENGTH ) );
  C1LEN := SYSTEM.MEMORY_ADDRESS( 1 );
  C2LEN := SYSTEM.MEMORY_ADDRESS( 1 );

  SUB1( C1'ADDRESS, S1'ADDRESS, C2'ADDRESS, C1LEN, SLEN, C2LEN );

  PUT(C1);
  PUT_LINE(" a ");
  PUT(S1);
  PUT_LINE(" qwerty ");
  PUT(C2);
  PUT_LINE(" c ");
end AFCHAR;

      SUBROUTINE SUB1( C1, S, C2 )        ! Source file is afchar2.f
      CHARACTER C1, C2
      CHARACTER*(*) S
      C1 = 'A'
      S  = 'QWERTY_QWERTY'
      C2 = 'C'
      RETURN
      END
```

The variables SLEN, C1LEN, and C2LEN contain the values that the FORTRAN routine requires. SYSTEM.MEMORY_ADDRESS is a function that converts an INTEGER type to a SYSTEM.ADDRESS type. Thus, it is possible to pass a scalar value to a FORTRAN routine using this function.

The command lines used to compile this example and its output are:

```
% fc -sa -c afchar2.f
% ada -M afchar.a afchar2.o -flink
% a.out
A a
QWERTY_QWE qwerty
C c
%
```

The string QWERTY_QWERTY is truncated because the Ada routine passes a 10-character array to the FORTRAN function, which is not large enough to contain the entire string. ANSI FORTRAN specifies that when a string constant is assigned to a character array that is not large enough, the string constant becomes truncated.

Complex arguments

The example in Figure 40 shows how to pass structures that simulate COMPLEX*8 and COMPLEX*16 data types to a FORTRAN routine.

Figure 40
Passing a structure representing a complex data type from Ada to FORTRAN

```
-- source file is afcmplx.a

with TEXT_IO; use TEXT_IO;
with SYSTEM;

procedure AFCMPLX is
  package COMPLEX8_IO is new FLOAT_IO (SHORT_FLOAT);
  package COMPLEX16_IO is new FLOAT_IO (FLOAT);
  USE COMPLEX8_IO, COMPLEX16_IO;

  type COMPLEX8 is record
    REAL, IMAGINARY : SHORT_FLOAT;
  end record ;

  type COMPLEX16 is record
    REAL, IMAGINARY : FLOAT;
  end record ;

  procedure MODIFY_COMPLEX8 ( COMPLEX8 : in SYSTEM.ADDRESS );
  procedure MODIFY_COMPLEX16 ( COMPLEX16 : in SYSTEM.ADDRESS );
  pragma INTERFACE ( FORTRAN, MODIFY_COMPLEX8 );
```

Figure 40 (continued)
Passing a structure representing a complex data type from Ada to FORTRAN

```
   pragma INTERFACE ( FORTRAN, MODIFY_COMPLEX16 );

   C1 : COMPLEX8;
   C2 : COMPLEX16;

begin
   C1 := (5.0, 25.0);
   C2 := (5.0,25.0);


   MODIFY_COMPLEX8 ( C1'ADDRESS );
   MODIFY_COMPLEX16 ( C2'ADDRESS );

   PUT("("); PUT(C1.REAL); PUT(","); PUT(C1.IMAGINARY); PUT(")");
   PUT_LINE("");
   PUT("("); PUT(C2.REAL); PUT(","); PUT(C2.IMAGINARY); PUT(")");
   PUT_LINE("");
end AFCMPLX;
```

```
      SUBROUTINE MODIFY_COMPLEX8 ( C )      ! Source file is afcmplx2.f
      COMPLEX*8 C

      C = CMPLX ( (REAL(C)*(-2)),(AIMAG(C)*(-3)))
      RETURN
      END

      SUBROUTINE MODIFY_COMPLEX16 ( C )
      COMPLEX*16 C

      C = DCMPLX ( (DREAL(C)*(-2)),(DIMAG(C)*(-3)))
      RETURN
      END
```

Note that the compatible data type in Ada for complex data types in FORTRAN is a `record`. The order of its components is important.

Also note that even though the mode of the actual parameters, `C1` and `C2`, is `in`, the formal parameters can be modified in the FORTRAN routines, and those changes update the actual parameters in the Ada program.

The command lines to compile this program and its output are:

```
% fc -sa -c afcmplx2.f
% ada -M afcmplx.a afcmplx2.o -flink
% a.out
(-1.00000E+01,-7.50000E+01)
(-1.00000000000000E+01,-7.50000000000000E+01)
%
```

Array arguments

The example in Figure 41 shows how to pass a two-dimensional array to a FORTRAN routine.

Figure 41
Ada source code that passes a two-dimensional array to FORTRAN

```
with TEXT_IO; use TEXT_IO;          -- Source file is afarr.a
with SYSTEM;  use SYSTEM;

procedure AFARR is
  procedure SUB1( I       : in SYSTEM.ADDRESS );
  pragma INTERFACE( FORTRAN, SUB1 );

  package INT_IO is new INTEGER_IO (INTEGER);
  use INT_IO;

  type MATRIX_2D is array (INTEGER'range , INTEGER'range ) of INTEGER;

  A : MATRIX_2D( 1..5, 1..3 );
begin

  SUB1( A'ADDRESS );

  for J in A'RANGE(1) loop
    PUT(A(J,1)); PUT(" "); PUT(A(J,2)); PUT(" "); PUT(A(J,3));
    PUT_LINE("");
  end loop;
end AFARR;
```

Ada stores arrays in row-major order, while FORTRAN stores arrays in column-major order. Consequently, FORTRAN source code that receives multiply dimensioned Ada arrays must be modified to recognize this.

There are two ways to modify the FORTRAN source code. Figure 42 uses the ROW_WISE directive to force FORTRAN to access the array as a row-major array.

Figure 42
Modifying a FORTRAN array with the ROW_WISE directive

```
        SUBROUTINE SUB1 (A)      ! source file is afarr2a.f
C$DIR ROW_WISE (A)
        INTEGER A(5,3)
        DO I = 1, 5
          DO J = 1, 3
            A(I,J) = I*J
          ENDDO
        ENDDO
        RETURN
        END
```


The format of the ROW_WISE directive is as follows:

ROW_WISE(*array_name* [,...])

The following rules apply to using the ROW_WISE directive:

- Implicit array I/O, such as READ(5,*) A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears transposed when viewing it with a debugger.
- You cannot use the ROW_WISE directive with dummy arguments.

Ada does not have a pragma (directive) comparable to FORTRAN's ROW_WISE directive. Refer to the *CONVEX FORTRAN Optimization Guide* for information on the optimization aspects of this directive.

The second way to modify the FORTRAN source code is to reverse the order of the indices. This requires the array declaration to be modified as well as any loops that access the array's elements. This method is shown in Figure 43.

Figure 43
Modifying a FORTRAN array by reversing its indices

```
SUBROUTINE SUB1 (A)      ! source file is afarr2b.f

INTEGER A(3,5)           ! indices are interchanged
DO J = 1, 5
  DO I = 1, 3
    A(I,J) = I*J
  ENDDO
ENDDO
RETURN
END
```

The command lines to compile this program using both modifications of the FORTRAN source code and their output are:

```
% fc -sa -c afarr2a.f
% fc -sa -c afarr2b.f
% ada -M afarr.a afarr2a.o -flink
% a.out
          1          2          3
          2          4          6
          3          6          9
          4          8         12
          5         10         15
% ada -M afarr.a afarr2b.o -flink
% a.out
          1          2          3
          2          4          6
          3          6          9
          4          8         12
          5         10         15
%
```

Integer and floating-point return values

The example in Figure 44 shows how to call FORTRAN functions that return real or integer data types.

Figure 44
Calling integer and floating-point FORTRAN functions from Ada

```
-- Source file is affunc.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

procedure AFFUNC is
  function ADD_INT2( ARG1_INT2: in SYSTEM.ADDRESS;
                     ARG2_INT2: in SYSTEM.ADDRESS ) return SHORT_INTEGER;
  function ADD_INT4( ARG1_INT4: in SYSTEM.ADDRESS;
                     ARG2_INT4: in SYSTEM.ADDRESS ) return INTEGER;
  function ADD_REAL4( ARG1_REAL4: in SYSTEM.ADDRESS;
                      ARG2_REAL4: in SYSTEM.ADDRESS ) return SHORT_FLOAT;
  function ADD_REAL8( ARG1_REAL8: in SYSTEM.ADDRESS;
                      ARG2_REAL8: in SYSTEM.ADDRESS ) return FLOAT;
  pragma INTERFACE ( FORTRAN, ADD_INT2 );
  pragma INTERFACE ( FORTRAN, ADD_INT4 );
  pragma INTERFACE ( FORTRAN, ADD_REAL4 );
  pragma INTERFACE ( FORTRAN, ADD_REAL8 );

  package SHORT_IO IS NEW INTEGER_IO (SHORT_INTEGER);
  package INT_IO   IS NEW INTEGER_IO (INTEGER);
  package REAL4_IO IS NEW FLOAT_IO   (SHORT_FLOAT);
  package REAL8_IO IS NEW FLOAT_IO   (FLOAT);
  USE SHORT_IO, INT_IO, REAL4_IO, REAL8_IO;

  I1,I2,I3  : SHORT_INTEGER;          J1,J2,J3  : INTEGER;
  A1,A2,A3  : SHORT_FLOAT;            B1,B2,B3  : FLOAT;
begin
  I1 := 2;   I2 := 4;
  J1 := 2;   J2 := 6;
  A1 := 4.0; A2 := 5.0;
  B1 := 4.0; B2 := 6.0;

  I3 := ADD_INT2( I1'ADDRESS, I2'ADDRESS );
  J3 := ADD_INT4( J1'ADDRESS, J2'ADDRESS );
  A3 := ADD_REAL4( A1'ADDRESS, A2'ADDRESS );
  B3 := ADD_REAL8( B1'ADDRESS, B2'ADDRESS );

  PUT( I3 ); PUT_LINE("");  PUT( J3 ); PUT_LINE("");
  PUT( A3 ); PUT_LINE("");  PUT( B3 ); PUT_LINE("");

end AFFUNC;
```

Figure 44 (continued)
Calling integer and floating-point FORTRAN functions from Ada

```
INTEGER*2 FUNCTION ADD_INT2 (K1, K2)    ! Source file is affunc2.f
INTEGER*2 K1, K2
ADD_INT2 = K1 + K2
RETURN
END

INTEGER*4 FUNCTION ADD_INT4 (L1, L2)
INTEGER*4 L1, L2
ADD_INT4 = L1 + L2
RETURN
END

REAL*4 FUNCTION ADD_REAL4 (C1, C2)
REAL*4 C1, C2
ADD_REAL4 = C1 + C2
RETURN
END

REAL*8 FUNCTION ADD_REAL8 (D1, D2)
REAL*8 D1, D2
ADD_REAL8 = D1 + D2
RETURN
END
```

FORTRAN functions that return REAL*16 and INTEGER*8 data types are addressed in the section "Complex return values" on page 91 in this chapter.

The command lines used to compile this program and its output are:

```
% fc -sa -c affunc2.f
% ada -M affunc.a affunc2.o -flink
% a.out
     6
           8
 9.00000E+00
 1.00000000000000E+01
%
```

String return values

The example in Figure 45 shows how to call a FORTRAN function that returns a string.

Figure 45
Calling a string FORTRAN function from Ada

```
-- Source file is affchar.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

procedure AFFCHAR is

  procedure MAKESTR( STR_out : in SYSTEM.ADDRESS;
                     STR_len : in SYSTEM.ADDRESS;
                     ARG1_ch : in SYSTEM.ADDRESS;
                     ARG2_ch : in SYSTEM.ADDRESS );
  pragma INTERFACE ( FORTRAN, MAKESTR );

  ARG1,ARG2 : CHARACTER;
  STR_retval: STRING(1..20);
  STR_LEN   : ADDRESS;
begin
  ARG1 := 'Z';
  ARG2 := 'z';
  STR_LEN := SYSTEM.MEMORY_ADDRESS( STR_retval'LENGTH );
  MAKESTR( STR_retval'ADDRESS, STR_LEN,ARG1'ADDRESS,ARG2'ADDRESS );

  PUT( STR_retval ); PUT_LINE("... asleep");

end AFFCHAR;


      CHARACTER*20 FUNCTION MAKESTR( CH1, CH2 )  ! Source is affchar2.f
      CHARACTER CH1, CH2
      MAKESTR = CH1 // CH2
      RETURN
      END
```

You should note the following about Figure 45:

- An Ada *procedure* is compatible with a FORTRAN *function* that returns a CHARACTER array.
- The first parameter of the MAKESTR call is a pointer to the character variable returned by the FORTRAN function.
- The second parameter of the MAKESTR call contains the amount of space allocated for the character variable returned by the FORTRAN function. This allocation occurs in the Ada routine that calls the FORTRAN function program.
- The third parameter of MAKESTR is the first argument of the FORTRAN function which is a CHARACTER data type in this example.
- The fourth parameter of MAKESTR is the second argument of the FORTRAN function.
- The value of STR_len is set using the SYSTEM.MEMORY_ADDRESS function because the type required by FORTRAN needs to be a value. But the Ada specification for the INTERFACE pragma requires a SYSTEM.ADDRESS type. Consequently, the MEMORY_ADDRESS function is used to pass the integer *value* as an address type.

The command lines used to compile this example and its output are:

```
% fc -sa -c affchar2.f
% ada -M affchar.a affchar2.o -flink
% a.out
Zz                    ... asleep
%
```

Complex return values

The example in Figure 46 shows how to call a COMPLEX FORTRAN function. In this example, the FORTRAN complex data type is received by an Ada record. You can use a similar technique to call FORTRAN routines that return INTEGER*8 and REAL*16 data types.

Figure 46
Calling complex FORTRAN functions from Ada

```
-- Source file is afxfunc.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

procedure afxfunc is
  type COMPLEX8 is record
    REAL, IMAGINARY : SHORT_FLOAT;
  end record;

  package COMPLEX8_IO is new FLOAT_IO ( SHORT_FLOAT );
  use COMPLEX8_IO;

  procedure ADD_CX8( CX8_out  : in SYSTEM.ADDRESS;
                     ARG1_CX8 : in SYSTEM.ADDRESS;
                     ARG2_CX8 : in SYSTEM.ADDRESS );
  pragma INTERFACE ( FORTRAN, ADD_CX8 );

  CX8_retval, C1, C2 : COMPLEX8;
begin
  C1 := (5.0, -7.0);
  C2 := (4.0, 14.0);
  ADD_CX8( CX8_retval'ADDRESS, C1'ADDRESS, C2'ADDRESS );

  PUT("("); PUT(CX8_retval.REAL); PUT(",");
  PUT(CX8_retval.IMAGINARY);  PUT_LINE(")");

end afxfunc;
```

```
      COMPLEX*8 FUNCTION ADD_CX8( B1, B2 ) ! Source is afxfunc.f
      COMPLEX*8 B1, B2
      ADD_CX8 = B1 + B2
      RETURN
      END
```

You should note the following about Figure 46:

- Even though a FORTRAN *function* is being called, the Ada specification requires that MAKECX is a *procedure*.
- The first parameter of MAKECX is a pointer to the complex structure returned by the FORTRAN function.

The command lines to compile this program and its output are:

```
% fc -sa -c afxfunc2.f
% ada -M afxfunc.a afxfunc2.o -flink
% a.out
( 9.00000E+00, 7.00000E+00)
%
```

C calls Ada

8

This chapter contains examples that demonstrate how to access Ada objects from C. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter. The following topics are contained in this chapter:

- Elaboration
- Linking C main programs

The examples in this chapter show you how to perform the following tasks:

- Access Ada global variables
- Pass integer and floating-point arguments to an Ada routine
- Pass array arguments to an Ada routine
- Pass record arguments to an Ada routine
- Call C functions that return integer and floating point types

Each example includes the source code for Ada and C; Ada source code appears in mixed case, while C source code appears in lowercase. Command lines are provided so that you can reproduce the examples.

Elaboration

Elaboration is the process by which a declaration achieves its effect; elaboration occurs during program execution. For example, elaboration of

```
NUM : constant INTEGER := 5;
```

creates the NUM object and initializes it with the value 5.

Ada objects cannot be accessed before they have been elaborated. If you attempt to access an Ada object that has not been elaborated, the PROGRAM_ERROR exception will probably be raised.

The method of compiling and linking Ada routines with C routines automatically elaborates the Ada routines when the program is executed. The only way to create a program in which the main program unit is in C is to call that unit from an Ada procedure. You cannot link a program that has a C main program unit with the cc command line.

Linking C main programs

The Ada source code below is an example of an Ada procedure that calls a C main program:

```
procedure AMAIN is
    procedure CMAIN;
    pragma INTERFACE( C, CMAIN );
    pragma INTERFACE_NAME( CMAIN, "_main");
begin
    CMAIN;
end AMAIN;
```

The entry point in all C main programs is _main. Consequently, the pragma INTERFACE_NAME is required to replace all references to CMAIN with _main in the Ada object file. Because the C main program is called by an Ada procedure, any elaboration that is required will be performed prior to the call to the C main program.

There are a few problems with this method. Because the initial routine is in Ada, the Ada runtime system is created. By calling the _main routine, you initiate a subset of the C runtime system. One side effect is that the C main program no longer has access to command line arguments or the environment. Access to the command line arguments and the environment is not provided with the parameters of the C main program, commonly named argc, argv, and envp. You can work around this problem using

the COMMAND_LINE package in the Ada library, convexlib. Finally, the program does not return the C main program's exit value to the environment.

You should also remember that the Ada compiler disables integer overflow detection before a call to a C function by default. Any subsequent calls from that C function to an Ada routine will also have integer overflow detection disabled. Refer to the description of the -od and -oe options in the section "Ada options" on page 1.

Accessing Ada global variables

The following example shows how to access global objects in Ada. It consists of three source files: an Ada main that calls a C main, and an Ada procedure. Figure 47 contains the Ada main program.

Figure 47
Accessing global variables in Ada, Ada main program

```
with ASUB;                    -- Source file is camain.a
procedure CAMAIN is
  procedure CMAIN;
  pragma INTERFACE( C, CMAIN );
  pragma INTERFACE_NAME( CMAIN, "_main");
begin
  CMAIN;
end CAMAIN;
```


CMAIN is the name that this Ada routine uses to refer to the C main program. Because CMAIN is a C function, you must use the pragma INTERFACE to indicate this to the Ada compiler. Moreover, the C compiler generates the global name _main for all C main programs. The pragma INTERFACE_NAME converts all references to CMAIN to the name that the C compiler uses, _main.

Figure 48 contains the C main program that is called by the Ada main program.

Figure 48
Accessing global variables in Ada, C main program

```
#include <stdio.h>      /* Source file is camain.c */

extern int x;
extern float y;
extern long long z;
void ada_func();

main()
{
  ada_func();
  (void) printf("\tx = %d\n", x );
  (void) printf("\ty = %g\n", y );
  (void) printf("\tz = %lld\n", z );
}
```

Figure 49 contains the Ada routine that is called from the C main program.

Figure 49
Accessing global variables in Ada, Ada function

```
with LONG_INT; use LONG_INT;   -- Source file is casub.a

package AFUNC is
  X : INTEGER;
  Y : SHORT_FLOAT;
  Z : LONG_INTEGER;
  procedure ADA_FUNC;
  pragma EXTERNAL_NAME( ADA_FUNC, "_ada_func" );
  pragma EXTERNAL_NAME( X, "_x" );
  pragma EXTERNAL_NAME( Y, "_y" );
  pragma EXTERNAL_NAME( Z, "_z" );
end AFUNC;

package body AFUNC is
  procedure ADA_FUNC is
  begin
    X := 4;
    Y := 4.0;
    Z := -1;
  end ADA_FUNC;
end AFUNC;
```

You should note that the pragma EXTERNAL_NAME defines the link names that are accessible to the C main program. The external names that the Ada source code defines are _ada_func, _x, _y, and _z. These names conform to the CONVEX C calling conventions described in Chapter 3, "Language interface."

The command lines to compile this example and its output are:

```
% ada casub.a camain.a
casub.a:
camain.a:
% cc -c camain.c
% a.make camain camain.o
% a.out
        x = 4
        y = 4
        z = -1
%
```

The a.make utility is used to link this program. The a.make utility does not understand any dependencies between C source files and Ada source files. Consequently, if you only modify C source files you must remove the executable file to force a.make to link in any new C object files.

Integer and floating-point arguments

The following example shows how to pass integer and floating-point data types to an Ada routine. Figure 50 contains the C main program.

Figure 50
Passing integer and floating-point arguments from C to Ada, C main

```
#include <stdio.h>       /* Source file is cair.c */
void ada_sub( int, int *, long long *, float * );

main()
{
  int j, i=3;
  long long lo;
  float a;

  ada_sub( i, &j, &lo, &a );

  (void) printf("%d, %lld, %f\n", j, lo, a );
}
```

Figure 51 contains the Ada routine that is called from the C main program.

Figure 52
Passing integer and floating-point arguments from C to Ada, Ada routines

```
with LONG_INT, SYSTEM;                          -- Source file is cair.a
use LONG_INT, SYSTEM;

-- Ada routine called from C follows
package CASUB2 is
  procedure ADA_SUB( I    : in INTEGER;
                     INT4 : in SYSTEM.ADDRESS;
                     LONG : in SYSTEM.ADDRESS;
                     REAL : in SYSTEM.ADDRESS );
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub" );
end CASUB2;

package body CASUB2 is
  procedure ADA_SUB( I    : in INTEGER;
                     INT4 : in SYSTEM.ADDRESS;
                     LONG : in SYSTEM.ADDRESS;
                     REAL : in SYSTEM.ADDRESS ) is
     T_INT4 : INTEGER;
     T_LONG : LONG_INTEGER;
     T_REAL : SHORT_FLOAT;
     for T_INT4 use at INT4;
     for T_LONG use at LONG;
     for T_REAL use at REAL;
  begin
     T_INT4 := I;
     T_LONG := -8;
     T_REAL := -4.0;
  end ADA_SUB;
end CASUB2;

-- Ada main routine follows
with CASUB2;
procedure CAIR is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CAIR;
```

You should note the following about this example:

- C passes integer and floating-point data types by value. However, when you modify the parameters in a procedure, you must pass them by address. In the C source code, i is passed by value, while the remaining

parameters are passed by address; only the value of `i` is used in the Ada procedure, while the other parameters are modified. Consequently, `I` can be used directly in the Ada procedure, while the other parameters must be accessed indirectly using address clauses (`for NAME use at ADDRESS`).

- The pragma `EXTERNAL_NAME` creates the link name for the Ada routine. In this case, `_ada_sub` conforms to the C calling conventions.

The command lines to compile this program and its output are:

```
% cc -c cair.c
% ada -M cair.a cair.o
% a.out
3, -8, -4.000000
%
```

String arguments

The example in this section shows how to pass strings, also called character arrays, to an Ada routine. Figure 52 contains the C main program.

Figure 51
Passing a string parameter from C to Ada, C main program

```
#include <stdio.h>    /* Source file is cachar.c */
void ada_sub( char [], int );

main()
{
  char str[] = "zzzzzzzzzz";
  int  num_set = 5;

  ada_sub( str, num_set );

  (void) printf("%s\n", str );
}
```

Figure 53 contains the Ada main program that calls the C main program. It also contains the Ada procedure that is called by the C main program.

Figure 53
Passing a string parameter from C to Ada, Ada routines

```
with SYSTEM, C_STRINGS;                          -- Source file is cachar.a
use  SYSTEM, C_STRINGS;

package CASUB3 is
  procedure ADA_SUB( C_STR : in out C_STRING;
                     NSET :   in INTEGER );
  pragma EXTERNAL_NAME( ADA_SUB, "_ada_sub" );
end CASUB3;

package body CASUB3 is
  procedure ADA_SUB( C_STR : in out C_STRING; NSET :   in INTEGER ) is
  begin
     for I in 1..NSET loop
        C_STR( I ) := 'A';
     end loop;
  end ADA_SUB;
end CASUB3;

-- Ada main routine follows
with CASUB3;
procedure CACHAR is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CACHAR;
```


You should note that the data type that Ada uses to represent C strings is C_STRING. This data type is defined in the C_STRINGS package which contains a number of conversion functions that you can use to convert between the STRING, A_STRING, and C_STRING data types.

The command lines to compile this example and its output are

```
% cc -c cachar.c
% ada -M cachar.a cachar.o
% a.out
AAAAAzzzzz
%
```

Record arguments

The example in this section shows how to pass records from C to Ada. Figure 54 contains the C main program.

Figure 54
Passing record parameters from C to Ada, C main program

```
#include <sys/sysinfo.h>                        /* Source file is carec.c */
#include <stdio.h>
#include <stdlib.h>

struct system_information sysinfo;
void print_sysinfo( struct system_information * );

main()
{
   if( getsysinfo( SYSINFO_SIZE, &sysinfo ) == -1 ){
      perror("System call failed:");
      exit(1);
   }
   print_sysinfo( &sysinfo );
}
```

The C main program makes a system call with the `getsysinfo` routine. For more information on this routine, refer to the getsysinfo(2) man page in the ConvexOS online man pages.

Figure 55 contains the Ada procedure that is called from the C main program. It also contains the Ada procedure that invokes the FORTRAN main program.

Figure 55
Passing record parameters from C to Ada, Ada routines

```
-- Source file is carec.a

with TEXT_IO, LONG_INT, SYSTEM;
use TEXT_IO, LONG_INT;

package CASUB4 is
  type UCHAR is range 0..(2**8)-1; for UCHAR'SIZE use 8;
  type USHORT is range 0..(2**16)-1; for USHORT'SIZE use 16;
  package INT_IO is new INTEGER_IO(USHORT); use INT_IO;
  package CHAR_IO is new INTEGER_IO(UCHAR); use CHAR_IO;
  type SYS_INFO is record
    SERIAL_NO : USHORT;
    CPU_TYPE  : UCHAR;
    CPU_NUM   : UCHAR;
```

Figure 55 (continued)
Passing record parameters from C to Ada, Ada routines

```
    FLAGS      : LONG_INTEGER;
  end record;

  procedure PRINT_SYSINFO( INFO: in SYS_INFO );
  pragma EXTERNAL_NAME( PRINT_SYSINFO, "_print_sysinfo");
end CASUB4;

package body CASUB4 is
  procedure PRINT_SYSINFO( INFO: in SYS_INFO ) is
  begin
    PUT("Serial number of this machine is ");
    PUT(INFO.SERIAL_NO);
    PUT_LINE("");
  end PRINT_SYSINFO;
end CASUB4;

-- Ada main routine follows
with CASUB4;
procedure CAREC is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CAREC;
```


You should note that CONVEX Ada passes records by address. Consequently, you must pass C records to Ada using a pointer to the C structure.

The command lines to compile this program and its output are:

```
% cc -c carec.c
% ada -M carec.a carec.o
% a.out
Serial number of this machine is   8201
%
```

Array arguments

This section shows how to pass arrays from C to Ada. In Ada there are two types of arrays: constrained and unconstrained. A constrained array has its array boundaries defined at compile time, while an unconstrained array has its array boundaries defined during runtime. This section shows how to pass an array to either a constrained or an unconstrained array type.

Constrained array

The next example shows how to pass an array to an Ada function that requires a constrained array. Figure 56 contains the C main program.

Figure 56
Passing a constrained array to C from Ada, C main program

```
int a[5][3];    /* Source file is caarr.c */

void print_array( int a[5][3] );

main()
{
   int i,j;

   for( i=0; i<5; i++ )
      for( j=0; j<3; j++ )
         a[i][j] = (i+1)*(j+1);
   print_array( a );
}
```

Figure 57 contains the Ada routines used in this example

Figure 57
Passing a constrained array from C to Ada, Ada routines

```
-- Source file is caarr.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

package CASUB5 is
  type MATRIX_2D is array (1..5, 1..3 ) OF INTEGER;
  package INT_IO is new INTEGER_IO (INTEGER); use INT_IO;

  procedure PRINT_ARRAY( I_ARR : in MATRIX_2D );
  pragma EXTERNAL_NAME( PRINT_ARRAY, "_print_array" );
end CASUB5;
```

Figure 57 (continued)
Passing a constrained array from C to Ada, Ada routines

```
package body CASUB5 is
  procedure PRINT_ARRAY( I_ARR : in MATRIX_2D ) is
  begin
    for J in I_ARR'RANGE(1) loop
      PUT(I_ARR(J,1)); PUT(" ");
      PUT(I_ARR(J,2)); PUT(" ");
      PUT(I_ARR(J,3));
      PUT_LINE("");
    end loop;
  end PRINT_ARRAY;
end CASUB5;

-- Ada main program follows
with CASUB5; use CASUB5;
procedure CAARR is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CAARR;
```

You should note that the C array is passed by address. You must pass constrained arrays by address.

The command lines used to compile this example and its output are:

```
% cc -c caarr.c
% ada -M caarr.a caarr.o
% a.out
          1          2          3
          2          4          6
          3          6          9
          4          8         12
          5         10         15
%
```

Unconstrained array

When you pass an array to an Ada routine that expects an unconstrained array, you must include a dope vector which contains the boundary information of the array indices. The format of a dope vector for an n-dimensional array is as follows:

```
typedef struct {
   int element_size;
   int low_bound_n;
   int high_bound_n;
   int size_of_array_without_index_n;
   .
   .
   .
   int low_bound_2;
   int high_bound_2;
   int size_of_array_without_indices_3_to_n;
   int low_bound_1;
   int high_bound_1;
   int size_of_array;
} n_dim_dope_vector;
```

The dope vector is used at runtime to verify that array references do not exceed array boundaries. When you specify array boundary information in the dope vector, be sure to specify boundaries that you are using in the Ada source code.

The next example shows how to pass an array to an Ada procedure that expects an unconstrained array. Figure 58 contains the C main program.

Figure 58
Passing an unconstrained array from C to Ada, C main program

```
#include <stdio.h>                    /* Source file is caarr2.c */

int a[5][3][2];

typedef struct {
  int element_size;
  int lo_bound3;
  int high_bound3;
  int size_first_two_indices;
  int lo_bound2;
  int high_bound2;
  int size_first_index;
```

Figure 58 (continued)
Passing an unconstrained array from C to Ada, C main program

```
   int lo_bound1;
   int high_bound1;
   int array_size;
} dope_3d;
void print_array( int arr[5][3][2], dope_3d *, char * );

main()
{
   int i,j,k;
   char title[] = "ADA";
   dope_3d dope = {sizeof(a[0][0][0]), /* element size */
                   5,                  /* lower bound of third index */
                   6,                  /* upper bound of third index */
                   sizeof(a[5][3]),  /* size of array without third index */
                   3,                  /* lower bound of second index */
                   5,                  /* upper bound of second index */
                   sizeof(a[5]),   /* size of array with only first index */
                   5,                  /* lower bound of first index */
                   9,                  /* upper bound of first index */
                   sizeof(a)};         /* size of entire array */

   for( i=0; i<5; i++ )
      for( j=0; j<3; j++ )
         for( k=0; k<2; k++ )
            a[i][j][k] = (i+1)*(j+1)*(k+1);

   print_array( a, &dope, title );
}
```

Figure 59 contains the Ada routines.

Figure 59
Passing an unconstrained array from C to Ada, Ada routines

```
-- Source file is caarr.a
with TEXT_IO, SYSTEM, C_STRINGS;
use TEXT_IO,  SYSTEM, C_STRINGS;

package CASUB6 is
  type MATRIX_3D is array (INTEGER range <>,INTEGER range  ,
                           INTEGER range <> ) OF INTEGER;
  package INT_IO is new INTEGER_IO (INTEGER); use INT_IO;

  procedure PRINT_ARRAY( I_ARR : in MATRIX_3D; C_STR : in out C_STRING );
  pragma EXTERNAL_NAME( PRINT_ARRAY, "_print_array" );
```

Figure 59 (continued)
Passing an unconstrained array from C to Ada, Ada routines

```
end CASUB6;

package body CASUB6 is
  procedure PRINT_ARRAY( I_ARR : in MATRIX_3D; C_STR : in out C_STRING ) is
  begin
    PUT_LINE("ADA:");
    for K in 5..6 loop
      for I in 5..9 loop
        for J in 3..5 loop
          PUT(I_ARR(I,J,K));
        end loop;
        PUT_LINE("");
      end loop;
      PUT_LINE("");
    end loop;
  end PRINT_ARRAY;
end CASUB6;

-- Ada main program follows
with CASUB6; use CASUB6;
procedure CAARR2 is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CAARR2;
```

You should note the following about this example:

- The array is passed by address.
- The address of the dope vector immediately follows the address of the array. (The third argument is included only to show that the dope vector does *not* come at the end of the parameter list.)
- The function declaration of `print_array` in C has three parameters, but the function declaration of `PRINT_ARRAY` in Ada has only two parameters. The unconstrained array parameter in Ada assumes that both the address of the array and the address of the dope vector are passed to the Ada routine.

The command lines used to compile this example and its output are:

```
% cc -c caarr2.c
% ada -M caarr2.a caarr2.o
% a.out
ADA:
          1          2          3
          2          4          6
          3          6          9
          4          8         12
          5         10         15

          2          4          6
          4          8         12
          6         12         18
          8         16         24
         10         20         30

%
```

Integer and floating-point return values

This section shows how to call Ada functions that return the C types `short`, `long` (which is the same as `int`), `float`, and `double`. Figure 60 contains the C main program.

Figure 60
Calling integer and floating-point Ada functions from C, C main program

```
#include <stdio.h>                          /* Source file is cafir.c */
short add_short( short, short );
long  add_long ( long, long );
float add_float( float, float );
double add_double( double, double );

main()
{
  short a=1, b=2;       long  c=3, d=4;
  float e=5, f=6;       double g=7, h=8;

  (void) printf("%d + %d = %d\n", a, b, add_short(a,b) );
  (void) printf("%ld + %ld = %ld\n", c, d, add_long(c,d) );
  (void) printf("%f + %f = %f\n", e, f, add_float(e,f) );
  (void) printf("%f + %f = %f\n", g, h, add_double(g,h) );
}
```

Figure 61 contains the Ada routines.

Figure 61
Calling integer and floating-point Ada functions from C, Ada routines

```
-- Source file is cafir.a
with SYSTEM;   use SYSTEM;
package CASUB7 is
  function ADD_SHORT( ARG1: in SHORT_INTEGER;
                      ARG2: in SHORT_INTEGER ) return SHORT_INTEGER;
  function ADD_LONG ( ARG1: in INTEGER;
                      ARG2: in INTEGER ) return INTEGER;
  function ADD_FLOAT( ARG1: in SHORT_FLOAT;
                      ARG2: in SHORT_FLOAT ) return SHORT_FLOAT;
  function ADD_DOUBLE( ARG1: in FLOAT;
                      ARG2: in FLOAT ) return FLOAT;
  pragma EXTERNAL_NAME( ADD_SHORT, "_add_short" );
  pragma EXTERNAL_NAME( ADD_LONG, "_add_long" );
  pragma EXTERNAL_NAME( ADD_FLOAT, "_add_float" );
  pragma EXTERNAL_NAME( ADD_DOUBLE, "_add_double" );
end CASUB7;

package body CASUB7 is

  function ADD_SHORT( ARG1: in SHORT_INTEGER;
                      ARG2: in SHORT_INTEGER ) return SHORT_INTEGER is
  begin
    return ARG1 + ARG2;
  end ADD_SHORT;

  function ADD_LONG ( ARG1: in INTEGER;
                      ARG2: in INTEGER ) return INTEGER is
  begin
    return ARG1 + ARG2;
  end ADD_LONG;

  function ADD_FLOAT( ARG1: in SHORT_FLOAT;
                      ARG2: in SHORT_FLOAT ) return SHORT_FLOAT is
  begin
    return ARG1 + ARG2;
  end ADD_FLOAT;

  function ADD_DOUBLE( ARG1: in FLOAT;
                      ARG2: in FLOAT ) return FLOAT is
  begin
    return ARG1 + ARG2;
  end ADD_DOUBLE;
end CASUB7;
```

Figure 61 (continued)
Calling integer and floating-point Ada functions from C, Ada routines

```
with CASUB7;
procedure CAFIR is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CAFIR;
```

The command lines used to compile this program and its output are:

```
% cc -c cafir.c
% ada -M cafir.a cafir.o
% a.out
1 + 2 = 3
3 + 4 = 7
5.000000 + 6.000000 = 11.000000
7.000000 + 8.000000 = 15.000000
%
```

Ada calls C

9

This chapter contains examples that demonstrate how to access C objects from Ada. Chapter 3, "Language interface," contains useful information relating to the examples in this chapter.

The examples in this chapter require only that you can modify Ada source code. That is, the techniques illustrated in this chapter do not require you to modify C source code.

Each example includes the source code for Ada and C; Ada source code appears in mixed case, while C source code appears in lowercase. Command lines are provided so that you can reproduce the examples.

The examples in this chapter show you how to perform the following tasks:

- Access C global variables
- Pass integer and floating-point arguments to a C void function
- Pass string arguments to a C void function
- Pass complex arguments to a C void function
- Pass array arguments to a C void function
- Call C functions that return integer and floating-point values
- Call C functions that return string values
- Call C functions that return records

Accessing global variables

To access C global variables in Ada you must use the INTERFACE_NAME pragma. The format of this pragma is as follows:

```
pragma INTERFACE_NAME( variable_name, "link_name");
```

This pragma replaces all occurrences of *variable_name* with an external reference to *link_name*. This pragma must apply to a variable declared earlier in the same package specification. You must declare the object as a scalar, array, record, access type, procedure, or function. The object cannot be a loop variable, constant, or an initialized variable. Figure 62 shows how to use this pragma to access C global variables.

Figure 62
Accessing C global variables

```
with TEXT_IO; use TEXT_IO;      -- Source file is accmn.a
with SYSTEM;  use SYSTEM;

procedure ACCMN is
  package INT_IO is new INTEGER_IO(INTEGER);
  package REAL_IO is new FLOAT_IO (FLOAT);
  use INT_IO, REAL_IO;

  type STUFF is record
    M : FLOAT;
    N : INTEGER;
  end record ;

  procedure SUB1;
  pragma INTERFACE( C, SUB1 );
  C_VAR : STUFF;
  pragma INTERFACE_NAME( C_VAR, "_c_var" );
begin
  SUB1;
  PUT(C_VAR.M);
  PUT(C_VAR.N);
  PUT_LINE("");
end ACCMN;

/* Source file is accmn.c */
struct {
  double x;
  int y;
} c_var;

void sub1()
{
  c_var.x = 5.0;
  c_var.y = 3;
}
```

You should note the following about Figure 62:

- The link name for C global variables has the format *_var_name* where *var_name* is the name of the variable.
- You must use an Ada record to access the contents of a C struct. The Ada compiler aligns the fields in the record in the same way that the C compiler aligns the fields in the C structure; you must declare the fields in the same order using compatible data types.

The command lines to compile this example and its output are:

```
% cc -c accmn.c
% ada -M accmn.a accmn.o
% a.out
 5.00000000000000E+00           3
```

Integer and floating-point arguments

The example in Figure 63 shows how to pass INTEGER, LONG_INTEGER, and SHORT_FLOAT Ada data types to a C function.

Figure 63
Passing integer and floating-point arguments from Ada to C

```
-- Source file is acir.a
WITH text_io; USE text_io;
WITH system;  USE system;
WITH long_int; USE long_int;

PROCEDURE ACIR IS
  PROCEDURE sub1( I      : in SYSTEM.ADDRESS;
                  I_LONG : in SYSTEM.ADDRESS;
                  A_FLOAT  : in SYSTEM.ADDRESS );
  PRAGMA INTERFACE( C, sub1 );
  PACKAGE int_io IS NEW integer_io (INTEGER);
  PACKAGE real_io IS NEW float_io (SHORT_FLOAT);
  USE int_io, real_io;
  I  : INTEGER;
  I_LONG : LONG_INTEGER;
  A_FLOAT  : SHORT_FLOAT;
BEGIN
  I := 0;
  I_LONG := int_to_long(0);
  A_FLOAT := 0.0;

  sub1( I'ADDRESS, I_LONG'address, A_FLOAT'address);

  PUT(I);
  PUT_LINE("");
  PUT(long_to_int(I_LONG));
  PUT_LINE("");
  PUT(A_FLOAT);
  PUT_LINE("");
END ACIR;
```

Figure 63 (continued)
Passing integer and floating-point arguments from Ada to C

```
void sub1( int *x, long long *y, float *z )      /* Source file is acir.c */
{
   *x = 10;
   *y = 145;
   *z = 432.98;
}
```

You should note the following about Figure 63:

- The mode of the parameters in procedure `sub1` is `in`. The current implementation of CONVEX Ada does not permit a C function accessed by an Ada program to have parameters of mode `out` or `in out`. The C routine can modify the values pointed to by the arguments passed in, but not the arguments themselves.
- The `INTERFACE` pragma supports calls to C. The second parameter is the name of the C function.
- The parameter types and return type of the function must be scalar, access, or type `SYSTEM.ADDRESS`.
- In the actual function call, the `ADDRESS` attribute is applied to each parameter of `sub1`.
- The `long_int` package, included in the `convexlib` directory, contains a definition of the `LONG_INTEGER` data type. It also defines some functions, such as `int_to_long` and `long_to_int`, that pertain to this data type. Refer to `convexlib/long_int.a` for more details.

The command lines to compile this program and its output are:

```
% cc -c acir.c
% ada -M acir.a acir.o
% a.out
         10
        145
 4.32980E+02
```

The ada driver loads this program because the main program unit is written in Ada. The C libraries are automatically linked in by the linker.

Character and string arguments

The example in Figure 64 shows how to pass character and string arguments to a C routine.

Figure 64
Passing string parameters from Ada to C

```
-- Source file is acchar.a
with TEXT_IO, SYSTEM;
use  TEXT_IO, SYSTEM;
procedure ACCHAR is
  procedure SUB1( C1    : in SYSTEM.ADDRESS;
                  S1    : in SYSTEM.ADDRESS;
                  C2    : in SYSTEM.ADDRESS );
  pragma INTERFACE( C, SUB1 );
  C1, C2  : CHARACTER;
  S1      : STRING(1..20);
begin
  S1(10) := ascii.nul;
  SUB1( C1'ADDRESS, S1'address, C2'ADDRESS );
  PUT(C1);
  PUT_LINE(" a ");
  PUT(S1);
  PUT_LINE(" qwerty ");
  PUT(C2);
  PUT_LINE(" c ");
end ACCHAR;

void sub1( char *ch1, char s[], char *ch2 )    /* Source file is acchar.c */
{
   *ch1 = 'A';
   (void) strcpy( s, "QWERTY_QWERTY");
   *ch2 = 'C';
}
```

The command lines to compile this example and its output are:

```
% cc -c acchar.c
% ada -M acchar.a acchar.o
% a.out
A a
QWERTY_QWE qwerty
C c
```

Record arguments

The example in Figure 65 shows how to pass an Ada record to a C function.

Figure 65
Passing records from Ada to C

```
with TEXT_IO, LONG_INT, SYSTEM;                    -- source file is acrec.a
use TEXT_IO, LONG_INT;

procedure ACREC is
  type UCHAR is range 0..(2**8)-1; for UCHAR'SIZE use 8;
  type USHORT is range 0..(2**16)-1; for USHORT'SIZE use 16;
  package INT_IO is new INTEGER_IO(USHORT); use INT_IO;
  type SYS_INFO is record
    SERIAL_NO : USHORT;
    CPU_TYPE  : UCHAR;
    CPU_NUM   : UCHAR;
    FLAGS     : LONG_INTEGER;
  end record;
  procedure GET_SYSINFO ( info : in SYSTEM.ADDRESS );
  pragma INTERFACE ( C, GET_SYSINFO );
  INFO : SYS_INFO;
begin
  GET_SYSINFO( INFO'ADDRESS );
  PUT("Serial number of this machine is ");
  PUT(INFO.SERIAL_NO);
  PUT_LINE("");
end ACREC;


#include <sys/sysinfo.h>                    /* Source file is acrec.c */
#include <stdio.h>

void get_sysinfo( struct system_information *sysinfo )
{
   if( getsysinfo( SYSINFO_SIZE, sysinfo ) == -1 ){
      perror("System call failed:");
      exit(1);
   }
}
```

Note that the INFO Ada record is passed by address to the C function.

The C function makes a system call with the getsysinfo routine. For more information on this routine, refer to the getsysinfo(2) man page in the ConvexOS online man pages.

The command lines to compile this program and its output are:

```
% cc -c acrec.c
% ada -M acrec.a acrec.o
% a.out
Serial number of this machine is   8201
```

Array arguments

The example in Figure 66 shows how to pass a two-dimensional array to a C function.

Figure 66
Passing an array from Ada to C

```
with TEXT_IO; use TEXT_IO;          -- Source file is acarr.a
with SYSTEM;  use SYSTEM;

procedure acarr is
  procedure SUB1( AN_ARRAY      : in SYSTEM.ADDRESS );
  pragma INTERFACE( C, SUB1 );

  package INT_IO is new INTEGER_IO (INTEGER); use INT_IO;

  type MATRIX_2D is array (INTEGER range <>, INTEGER range ) OF INTEGER;

  A : MATRIX_2D( 1..5, 1..3 );
begin

  SUB1( A'ADDRESS );

  for J in A'RANGE(1) loop
    PUT(A(J,1));
    PUT(" ");
    PUT(A(J,2));
    PUT(" ");
    PUT(A(J,3));
    PUT_LINE("");
  end loop;
end acarr;

void sub1( int array[5][3] )                    /* Source file is acarr.c */
{
   int i,j;
   for( i=0; i<5; i++ )
      for( j=0; j<3; j++ )
         array[i][j] = (i+1)*(j+1);
}
```

Ada and C store arrays in row-major order. Consequently, you do not need to modify an array before it is passed to C. Remember that an Ada array can have any integral base index. The Ada arrays in this example have a base index of 1, while the C arrays have a base index of 0.

Like records, you must pass arrays to C by reference. This is a requirement of the Ada compiler. The pragma INTERFACE does not permit you to pass arrays or records by value.

The command lines used to compile this program and its output are:

```
% cc -c acarr.c
% ada -M acarr.a acarr.o
% a.out
          1          2          3
          2          4          6
          3          6          9
          4          8         12
          5         10         15
```

Integer and floating-point return values

The example in Figure 67 shows how to call C functions that return real or integer data types.

Figure 67
Calling integer and floating-point C functions from Ada

```
-- Source file is acfunc.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

procedure ACFUNC is
  function ADD_INT2( arg1: in SYSTEM.ADDRESS;
                     arg2: in SYSTEM.ADDRESS ) return SHORT_INTEGER;
  function ADD_INT4( arg1: in INTEGER;
                     arg2: in INTEGER ) return INTEGER;
  function ADD_REAL4( ARG1_REAL4: in SHORT_FLOAT;
                      ARG2_REAL4: in SHORT_FLOAT ) return SHORT_FLOAT;
  function ADD_REAL8( ARG1_REAL8: in FLOAT;
                      ARG2_REAL8: in FLOAT ) return FLOAT;
  pragma INTERFACE ( C, ADD_INT2 );
  pragma INTERFACE ( C, ADD_INT4 );
  pragma INTERFACE ( C, ADD_REAL4 );
```

Figure 67 (continued)
Calling integer and floating-point C functions from Ada

```
   pragma INTERFACE ( C, ADD_REAL8 );
   package SHORT_IO IS NEW INTEGER_IO (SHORT_INTEGER);
   package INT_IO   IS NEW INTEGER_IO (INTEGER);
   package REAL4_IO IS NEW FLOAT_IO   (SHORT_FLOAT);
   package REAL8_IO IS NEW FLOAT_IO   (FLOAT);
   USE SHORT_IO, INT_IO, REAL4_IO, REAL8_IO;

   I1,I2,I3   : SHORT_INTEGER;
   J1,J2,J3   : INTEGER;
   A1,A2,A3   : SHORT_FLOAT;
   B1,B2,B3   : FLOAT;
begin
   I1 := 2;    I2 := 4;
   J1 := 2;    J2 := 6;
   A1 := 4.0; A2 := 5.0;
   B1 := 4.0; B2 := 6.0;

   I3 := ADD_INT2( I1'ADDRESS, I2'ADDRESS );
   J3 := ADD_INT4( J1, J2 );
   A3 := ADD_REAL4( A1, A2 );
   B3 := ADD_REAL8( B1, B2 );
   PUT( I3 ); PUT_LINE("");
   PUT( J3 ); PUT_LINE("");
   PUT( A3 ); PUT_LINE("");
   PUT( B3 ); PUT_LINE("");
end ACFUNC;

short add_int2( short *a, short *b )          /* Source file is acfunc.c */
{
   return *a + *b;
}

int add_int4( int a, int b )
{
   return a + b;
}

float add_real4( float a, float b)
{
   return a + b;
}

double add_real8( double a, double b)
{
   return a + b;
}
```

Note that the parameters of the ADD_INT2 function are type SYSTEM.ADDRESS, while the parameters of the other functions are INTEGER, SHORT_FLOAT, and FLOAT. This is because the pragma INTERFACE permits only 32-bit or 64-bit scalar values to be passed. Consequently, when you pass SHORT_INTEGER variables to a C function, you must pass them by address.

The command lines used to compile this program and its output are:

```
% cc -c acfunc.c
% ada -M acfunc.a acfunc.o
% a.out
     6
           8
 9.00000E+00
 1.00000000000000E+01
```

String return values

The example in Figure 68 shows how to call a C function that returns a string.

Figure 68
Calling C functions that return strings from Ada

```
with TEXT_IO; use TEXT_IO;                    -- Source file is acfchar.a
with SYSTEM;  use SYSTEM;

procedure ACFCHAR is
  type STRING20 is access STRING(1..20);
  function MAKESTR(  ch1_addr : in SYSTEM.ADDRESS;
                     ch2_addr : in SYSTEM.ADDRESS ) return STRING20;
  pragma INTERFACE ( C, MAKESTR );

  CH1,CH2    : CHARACTER;
  STR_OUT    : STRING20;
begin
  CH1 := 'Z';
  CH2 := 'z';

  STR_OUT     := MAKESTR( CH1'ADDRESS, CH2'ADDRESS );

  PUT( STR_OUT.all );
  PUT_LINE("... asleep");
end ACFCHAR;
```

Figure 68 (continued)
Calling C functions that return strings from Ada

```
#include <stdlib.h>                          /* Source file is acfchar.c */

char *makestr( char *ch1, char *ch2 )
{
   char *ch = (char *)malloc(20);
   ch[0] = *ch1;
   ch[1] = *ch2;
   ch[2] = ch[3] = ' ';
   ch[4] = '\0';
   return ch;
}
```

You should note the following about Figure 68:

- The parameters of MAKESTR must be of type SYSTEM.ADDRESS because the scalar representation of a CHARACTER is only 8 bits; the pragma INTERFACE requires scalar parameters to have either a 32-bit or 64-bit representation.
- The MAKESTR function returns an access type to an array that has already been dimensioned. The pragma INTERFACE does not permit the result type of access to be an unconstrained array.
- The C function includes a call to the `malloc` system function. Calls to the `malloc` system function *are* compatible with the Ada runtime system because the Ada runtime system itself uses `malloc`.

The command lines used to compile this example and its output are:

```
% cc -c acfchar.c
% ada -M acfchar.a acfchar.o
% a.out
Zz  ... asleep
```

Record return values

Figure 69 shows how to call a C function that returns a record.

Figure 69
Calling C functions that return records from Ada

```
with TEXT_IO, LONG_INT, SYSTEM;                    -- source file is acrec.a
use TEXT_IO, LONG_INT;

procedure ACRFUNC is
  type UCHAR is range 0..(2**8)-1; for UCHAR'SIZE use 8;
  type USHORT is range 0..(2**16)-1; for USHORT'SIZE use 16;
  package INT_IO is new INTEGER_IO(USHORT); use INT_IO;
  type SYS_INFO is record
    SERIAL_NO : USHORT;
    CPU_TYPE  : UCHAR;
    CPU_NUM   : UCHAR;
    FLAGS     : LONG_INTEGER;
  end record;
  type SYS_INFO_A is access SYS_INFO;

  function GET_SYSINFO return SYS_INFO_A;
  pragma INTERFACE ( C, GET_SYSINFO );

  INFO : SYS_INFO_A;
begin
  INFO := GET_SYSINFO;
  PUT("Serial number of this machine is ");
  PUT(INFO.SERIAL_NO);
  PUT_LINE("");
end ACRFUNC;

#include <sys/sysinfo.h>                    /* Source file is acrec.c */
#include <stdio.h>

struct system_information *get_sysinfo( )
{
   struct system_information *sysinfo;
   sysinfo = (struct system_information *)malloc(SYSINFO_SIZE);
   if( getsysinfo( SYSINFO_SIZE, sysinfo ) == -1 ){
      perror("System call failed:");
      exit(1);
   }
   return sysinfo;
}
```

You should note the following about Figure 69:

- The return value of the GET_SYSINFO function is an access type.
- The pragma INTERFACE permits only records of type access to be returned. You cannot return a record by value.

The command lines to compile this program and its output are:

```
% cc -c acrfunc.c
% ada -M acrfunc.a acrfunc.o
% a.out
Serial number of this machine is   8201
```

Mixing input and output 10

This chapter discusses issues related to performing input and output in Ada, C, and FORTRAN. In general:

- FORTRAN routines that perform I/O should not be called from an application that does not have a FORTRAN main program unit.
- C routines that perform I/O should not be called from an application that does not have a C main function.
- Ada routines that perform I/O should not be called from an application that has initialized the FORTRAN or C runtime system.

Caution

Output behavior is *undefined* when you open a file in one language and write or read from it using routines written in other languages.

In many cases, mixed I/O is unnecessary. For example, CONVEX Ada provides access to many C-like I/O functions in the files `unix.a`, `os_files.a`, `close_all.a`, and `tty.a` that are in the standard library. Similarly FORTRAN provides many functions that emulate C library calls; these functions are described in the intro(3F) man page.

FORTRAN functions useful with C and Ada I/O

If you cannot avoid mixing FORTRAN I/O with C or Ada, you may find the following routines useful:

`f_init`—Initialize FORTRAN I/O. Call this from C or Ada when calling FORTRAN routines that perform I/O. This routine is especially useful when an Ada or C application must call VECLIB routines. Refer to Chapter 6, "FORTRAN calls Ada," for some examples that use this routine.

`f_exit`—Close access to FORTRAN I/O files. You should only call this when you are immediately exiting from your application. It is better to explicitly close each file that you opened in FORTRAN instead of calling this function. Leave all other files alone and return to the C or Ada calling routine.

`for$getfp`—This library routine returns the file pointer associated with a FORTRAN I/O unit number. You can use this routine to pass a file pointer to a C routine to modify certain attributes of a file using the `fcntl` system call.

`for$getfd`—This library routine returns the file descriptor associated with the FORTRAN I/O unit number.

Known mixed I/O behavior

You should remember the following points if you choose to perform I/O in a language other than the one in which the main program unit is written:

- FORTRAN unit numbers 0, 5, and 6 do not affect the C streams `stderr`, `stdin`, and `stdout`. In contrast, because Ada uses C routines to perform I/O, you could make a call to a C routine that uses `freopen` to redirect error output to a location other than `text_io.standard_error`.
- Reopening the FORTRAN unit numbers 0, 5, and 6 does not affect the C streams `stderr`, `stdin`, or `stdout`.
- In programs with C or Ada main programs, when data is read from unit 5 in a FORTRAN routine, it is expected to be in file `fort.5`. (Normally, data read from unit 5 is read from the keyboard.) Trying to read unit 5 in the absence of the `fort.5` file causes the program to abort with an I/O error.

  Similarly, data written to unit 6 (the standard output) with the `PRINT` or `WRITE` statement is written to the file `fort.6` rather than to the screen.

  There are two solutions for this behavior:

  – Call the `f_init` routine prior to performing any I/O in FORTRAN.

  – Set the `FOR005` environment variable to `SYS\$INPUT`, and set the `FOR006` environment variable to `SYS\$OUTPUT`.

FORTRAN-callable C I/O routines

Source code for the following C routines is included in the next several pages. These routines are FORTRAN callable, hence the underscore appended to the file names.

- `fc_params.h`
- `fc_close_.c`
- `fc_open_.c`
- `fc_read_.c`
- `fc_write_.c`

These routines are provided only for your convenience. They do not constitute a product supported by CONVEX, so please use them with discretion.

fc_params.h

The `fc_params.h` file defines global values that the functions in this chapter use.

```
/*=============================================================*/
/*                                                             */
/* FILE:        fc_params.h                                    */
/*                                                             */
/* PURPOSE:     Define the global values.                      */
/*                                                             */
/* AUTHOR:      John P. Kole, Convex Computer Corporation      */
/*              January 29, 1991                               */
/*                                                             */
/*=============================================================*/

#define FC_ERR_IFV 1;           /* Invalid IFCNO value.        */
#define FC_ERR_FAO 2;           /* IFCNO already open.         */
#define FC_ERR_OPN 3;           /* Open failed (see IOS).      */
#define FC_ERR_FNO 4;           /* IFCNO not open.             */
#define FC_ERR_CLS 5;           /* Close failed (see IOS).     */
#define FC_ERR_WRT 6;           /* Write failed (see IOS).     */
#define FC_ERR_EOF 7;           /* End of file.                */
#define FC_ERR_RED 8;           /* Read failed (see IOS).      */

extern int errno;

#define ifcno_min 0
#define ifcno_max 255

#define opntbl_size ifcno_max+1

/* The End */
```

fc_close_.c

Use the fc_close_ function to close a file that has been opened by the fc_open_ function.

```
/*============================================================*/
/*                                                            */
/* PROGRAM:     fc_close_.c                                   */
/*                                                            */
/* PURPOSE:     To close a file opened via fc_open.           */
/*                                                            */
/* FORMAT:      INTEGER IFCNO, IOS, IERR                      */
/*              ...                                           */
/*              CALL FC_CLOSE (IFCNO, IOS, IERR)              */
/*                                                            */
/* ARGUMENTS:   IFCNO - File number.                          */
/*              IOS   - System error code when IERR != 0.     */
/*              IERR  - See RETURNS heading below.            */
/*                                                            */
/* RETURNS:     IERR  - Error code as follows:                */
/*                      0          - No error occurred.       */
/*                      FC_ERR_IFV - Invalid IFCNO value.     */
/*                      FC_ERR_FNO - IFCNO not open.          */
/*                      FC_ERR_CLS - Close failed (see IOS).  */
/*                                                            */
/* AUTHOR:      John P. Kole, Convex Computer Corporation     */
/*              January 29, 1991                              */
/*                                                            */
/*============================================================*/
#include <stdio.h>

#include "fc_params.h"

extern FILE *opntbl[opntbl_size];

fc_close_ (ifcno, ios, ierr)

int     *ifcno;
int     *ios;
int     *ierr;

{

/*------------------------------------------------------------*/
/* If this file number is out of range, give an error.        */
/*------------------------------------------------------------*/
        if (*ifcno < ifcno_min || *ifcno > ifcno_max) {
            *ios = 0;
            *ierr = FC_ERR_IFV;
            return;
            }
```

```
/*---------------------------------------------------------------*/
/* If this file number is not open, give an error.               */
/*---------------------------------------------------------------*/
        if (opntbl[*ifcno] == NULL) {
            *ios = 0;
            *ierr = FC_ERR_FNO;
            return;
            }

/*---------------------------------------------------------------*/
/* Close the file.                                               */
/*---------------------------------------------------------------*/
        if (fclose(opntbl[*ifcno]) != NULL) {
            *ios = errno;
            *ierr = FC_ERR_CLS;
            }
        opntbl[*ifcno] = NULL;

/*---------------------------------------------------------------*/
/* Everything OK; clear error codes and return.                  */
/*---------------------------------------------------------------*/
        *ios = 0;
        *ierr = 0;
        return;
}

/*[The End]*/
```

fc_open_.c

Use the fc_open_ function to open a file from FORTRAN.

```
/*================================================================*/
/*                                                                */
/* PROGRAM:     fc_open_.c                                        */
/*                                                                */
/* PURPOSE:     To open a file from FORTRAN.                      */
/*                                                                */
/* FORMAT:      INTEGER IFCNO, ICREAT, IOS, IERR                  */
/*              CHARACTER*(*) CFILE                               */
/*              CFILE = "filename"                                */
/*              ...                                               */
/*              CALL FC_OPEN (IFCNO, CFILE, ICREAT, IOS, IERR)    */
/*                                                                */
/* ARGUMENTS:   IFCNO - File number.                              */
/*              CFILE - Filename to be opened. If this is a       */
/*                      zero length string, then a file named     */
/*                      fc_fort.N will be opened, where N will    */
/*                      be the unit number.                       */
/*                      If the file exists, it will be opened     */
/*                      for append. If the file does not exist,   */
/*                      it will be created.                       */
/*              ICREAT- Create mode as follows:                   */
/*                      0  - Create (or truncate to zero length)  */
/*                           while opening for read or write.     */
/*                      !0 - Do not create while opening for      */
/*                           read or write. The file must exist.  */
/*              IOS    - System error code when IERR != 0.        */
/*              IERR   - See RETURNS below.                       */
/*                                                                */
/* RETURNS:     IERR   - Error code as follows:                   */
/*                      0           - No error occurred.          */
/*                      FC_ERR_IFV  - Invalid IFCNO value.        */
/*                      FC_ERR_FAO  - IFCNO already opened.       */
/*                      FC_ERR_OPN  - Open failed.                */
/*                                                                */
/* AUTHOR:      John P. Kole, Convex Computer Corporation         */
/*              January 29, 1991                                  */
/*                                                                */
/*================================================================*/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "fc_params.h"

extern FILE *opntbl[opntbl_size] = {(opntbl_size)*NULL};
```

```
fc_open_ (ifcno, cfile, icreat, ios, ierr, il_cfile)

int     *ifcno;
char    *cfile;
int     *icreat;
int     *ios;
int     *ierr;

int     il_cfile;

{
        char *mfile;
        int fs;
        struct stat buf;
        char mode[] = "??";

/*---------------------------------------------------------------*/
/* If this file number is out of range, give an error.           */
/*---------------------------------------------------------------*/
        if (*ifcno < ifcno_min || *ifcno > ifcno_max) {
            *ios = 0;
            *ierr = FC_ERR_IFV;
            return;
            }

/*---------------------------------------------------------------*/
/* If this file number is already open, return error.            */
/*---------------------------------------------------------------*/
        if (opntbl[*ifcno] != NULL) {
            *ios = 0;
            *ierr = FC_ERR_FAO;
            return;
            }

/*---------------------------------------------------------------*/
/* Allocate an area and move the caller's filename into it and   */
/* append a null to the end (assuring no spaces in filename).    */
/*---------------------------------------------------------------*/
        if (il_cfile > 0) {
          mfile = malloc (il_cfile+1);    /* Area for name + null */
          strncpy (mfile,cfile,il_cfile);/* Move user's filename */
          *(mfile + il_cfile) = '\040';  /* Terminate with space */
          *(index (mfile,'\040')) = '\0';/* 1st space to null    */
          }
        else {
          mfile = malloc (9);             /* Area for name * null */
          (void)sprintf(mfile,"fc_fort.%d",*ifcno);
          }
```

```
/*-----------------------------------------------------------------*/
/* Set the open "mode."                                            */
/*-----------------------------------------------------------------*/
        if (*icreat == 0) {
            strncpy (mode, "w+", 2);
            }
        else {
            strncpy (mode, "r+", 2);
            }

/*-----------------------------------------------------------------*/
/* Open the file; creating if it is not there.                     */
/*-----------------------------------------------------------------*/
        if ((opntbl[*ifcno] = fopen (mfile, mode)) == NULL) {
            *ios = errno;
            *ierr = FC_ERR_OPN;
            return;
            }

/*-----------------------------------------------------------------*/
/* Everything OK; clear error codes and return.                    */
/*-----------------------------------------------------------------*/
        *ios = 0;
        *ierr = 0;
        return;
}

/*[The End]*/
```

fc_read_.c

Use the fc_read_ function to read raw data from a file that has been opened with the fc_open_ function.

```
/*============================================================*/
/*                                                            */
/* PROGRAM:     fc_read_.c                                    */
/*                                                            */
/* PURPOSE:     Read raw data from a file opened via fc_open. */
/*                                                            */
/* FORMAT:      INTEGER IFCNO, ILEN, IOS, IERR                */
/*              <Any Data Type> ARRAY(*)                      */
/*              ...                                           */
/*              CALL FC_READ (IFCNO, ARRAY, ILEN, IOS, IERR)  */
/*                                                            */
/*                                                            */
/* ARGUMENTS:   IFCNO - File number.                          */
/*              ARRAY - Buffer (data) to be read into.        */
/*              ILEN  - Number of bytes to be read.           */
/*              IOS   - System error code when IERR != 0.     */
/*              IERR  - See RETURNS heading below.            */
/*                                                            */
/* RETURNS:     IERR  - Error code as follows:                */
/*                      0          - No error occurred.       */
/*                      FC_ERR_IFV - Invalid IFCNO value.     */
/*                      FC_ERR_FNO - IFCNO not open.          */
/*                      FC_ERR_WRT - Read failed (see IOS).   */
/*                                                            */
/* AUTHOR:      John P. Kole, Convex Computer Corporation     */
/*              Convex Computer Corporation                   */
/*              January 29, 1991                              */
/*                                                            */
/*============================================================*/
#include <stdio.h>

#include "fc_params.h"

extern FILE *opntbl[opntbl_size];

fc_read_ (ifcno, array, ilen, ios, ierr)
        int     *ifcno;
        char    array[];
        int     *ilen;
        int     *ios;
        int     *ierr;
{
        int     read;
```

```
/*-------------------------------------------------------------*/
/* If this file number is out of range, give an error.         */
/*-------------------------------------------------------------*/
        if (*ifcno < ifcno_min || *ifcno > ifcno_max) {
            *ios = 0;
            *ierr = FC_ERR_IFV;
            return;
            }

/*-------------------------------------------------------------*/
/* If this file number is not open, give an error.             */
/*-------------------------------------------------------------*/
        if (opntbl[*ifcno] == NULL) {
            *ios = 0;
            *ierr = FC_ERR_FNO;
            return;
            }

/*-------------------------------------------------------------*/
/* Read ILEN bytes from the file only if ILEN > zero.          */
/*-------------------------------------------------------------*/
        if (*ilen > 0) {
            read = fread (array, 1, *ilen, opntbl[*ifcno]);
            if (read != *ilen) {
                if (read == 0) {
                    *ierr = FC_ERR_EOF;
                    *ios = 0;
                    }
                else {
                    *ierr = FC_ERR_RED;
                    *ios = errno;
                    }
                return;
                }
            }

/*-------------------------------------------------------------*/
/* Everything OK; clear error codes and return.                */
/*-------------------------------------------------------------*/
        *ios = 0;
        *ierr = 0;
        return;
}

/*[The End]*/
```

fc_write_.c

Use the fc_write_ function to write raw data to a file that has been opened with the fc_open_ function.

```
/*=============================================================*/
/*                                                             */
/* PROGRAM:     fc_write_.c                                    */
/*                                                             */
/* PURPOSE:     Write raw data to a file opened via fc_open.   */
/*                                                             */
/* FORMAT:      INTEGER IFCNO, ILEN, IOS, IERR                 */
/*              <Any Data Type> ARRAY(*)                       */
/*              ...                                            */
/*              CALL FC_WRITE (IFCNO, ARRAY, ILEN, IOS, IERR)  */
/*                                                             */
/*                                                             */
/* ARGUMENTS:   IFCNO - File number.                           */
/*              ARRAY - Buffer (data) to be written.           */
/*              ILEN  - Number of bytes to be written.         */
/*              IOS   - System error code when IERR != 0.      */
/*              IERR  - See RETURNS heading below.             */
/*                                                             */
/* RETURNS:     IERR  - Error code as follows:                 */
/*                      0          - No error occurred.       */
/*                      FC_ERR_IFV - Invalid IFCNO value.     */
/*                      FC_ERR_FNO - IFCNO not open.          */
/*                      FC_ERR_WRT - Write failed.            */
/*                                                             */
/* AUTHOR:      John P. Kole, Convex Computer Corporation      */
/*              January 29, 1991                               */
/*                                                             */
/*=============================================================*/
#include <stdio.h>

#include "fc_params.h"

extern FILE *opntbl[opntbl_size];

fc_write_ (ifcno, array, ilen, ios, ierr)
        int     *ifcno;
        char    array[];
        int     *ilen;
        int     *ios;
        int     *ierr;
{
        int     written;
```

```
/*-------------------------------------------------------------*/
/* If this file number is out of range, give an error.         */
/*-------------------------------------------------------------*/
        if (*ifcno < ifcno_min || *ifcno > ifcno_max) {
            *ios = 0;
            *ierr = FC_ERR_IFV;
            return;
            }

/*-------------------------------------------------------------*/
/* If this file number is not open, give an error.             */
/*-------------------------------------------------------------*/
        if (opntbl[*ifcno] == NULL) {
            *ios = 0;
            *ierr = FC_ERR_FNO;
            return;
            }

/*-------------------------------------------------------------*/
/* Write ILEN bytes to the file.                               */
/*-------------------------------------------------------------*/
        written = fwrite (array, 1, *ilen, opntbl[*ifcno]);
        if (written != *ilen) {
            *ios = errno;
            *ierr = FC_ERR_WRT;
            return;
            }

/*-------------------------------------------------------------*/
/* Everything OK; clear error codes and return.                */
/*-------------------------------------------------------------*/
        *ios = 0;
        *ierr = 0;
        return;
}

/*[The End]*/
```

C++ examples

B

This appendix describes the language interface of C++ and provides some examples that show you how to access objects in other languages as well as access C++ objects from other languages.

For more information on the C++ language interface, refer to section 7.4 in AT&T's C++ *Language Reference Manual* and Chapter 6, "Type-safe Linkage for C++," in *UNIX SYSTEM V AT&T C++ LANGUAGE SYSTEM RELEASE 2.1 Selected Readings*.

This appendix is not self-contained; it primarily shows examples. The language interface for C++ is essentially the same as that for C. Consequently, much of the discussion for C in the previous chapters is applicable to C++.

Language interface

CONVEX C++ is a preprocessor that converts C++ source code into C code that is then compiled by CONVEX C. Consequently, the language is similar to C. For instance, the following items are the same:

- Data types
- Function parameter passing methods
- Function return methods
- Input and output

CONVEX C++ does include a `complex` data type. The `complex.h` header file defines a type that is compatible with FORTRAN's `COMPLEX*16` data type, while the `complex8.h` header file defines a type that is compatible with FORTRAN's `COMPLEX*8` data type.

Linkage specifier

Like Ada, C++ generates symbols that do not simply have underscores prepended and appended to a function name. Function symbol names usually have return type and parameter data types encoded in them. Fortunately, C++ provides a *linkage specifier* that causes C++ to generate a symbol name that conforms to the C language interface.

The two formats of the linkage specifier are:

```
extern string-literal declaration
extern string-literal { declaration-list }
```

where:

string-literal
: can be `"C"` or `"C++"` to indicate whether a declaration should have C or C++ linkage. The default linkage for a declaration is C++.

declaration
: is a function or variable declaration

declaration-list
: is a list of function and/or variable declarations.

The *AT&T C++ Reference Manual* provides the following information about the linkage specifier:

> Linkage specifications nest. A linkage specification does not establish scope. A linkage-specification may only occur in file scope. A linkage specification for a class applies to non-member functions and objects declared within it. A linkage specification for a function also applies to functions and objects declared within it. A linkage declaration with a string that is unknown to implementation is an error.
>
> If a function has more than one linkage specification, they must agree; that is, they must specify the same string-literal. A function declaration without a linkage specification may not precede the first linkage specification for that function. A function may be declared without a linkage specificaiton after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

At most, one of a set of overloaded functions with a particular name can have C linkage.

Examples

The examples in this chapter show you how to perform the following tasks:

- Access a global variable in C from C++
- Access FORTRAN common blocks in C++
- Pass integer and floating-point arguments to C++ from Ada
- Pass an array argument from C++ to Ada
- Pass a complex data type from C++ to FORTRAN
- Initialize FORTRAN I/O in a C++ main program
- Pass a complex data type from FORTRAN to C++
- Call C++ functions that return complex data types to a FORTRAN main program

Accessing a global variable in C

The example in this section shows you how to access a C global variable from C++. Figure 70 contains the source code for the C routine.

Figure 70
Accessing a C global variable in C++, C routine

```
struct {                    /* Source file is accmn.c */
  double x;
  int y;
} c_var;

void sub1()
{
  c_var.x = 5.0;
  c_var.y = 3;
}
```

Figure 71 contains the source code for the C++ main program.

Figure 71
Accessing a C global variable in C++, C++ main program

```
// Source file is cppccmn.C
#include <iostream.h>

extern "C" {
  struct {
    double x;
    int y;
  } c_var;
  void sub1( void );
}


main()
{
  sub1();
  cout << c_var.x << "\n" << c_var.y << "\n";
}
```

Note that the linkage specifier contains two declarations: `c_var` and `sub1`. The `c_var` declaration is required because the C++ source code prints out information contained in the `c_var` structure. The `sub1` function declaration is required because the C++ source code calls the `sub1` C function.

The command lines to compile this program and its output are:

```
% cc -c accmn.c
% CC cppccmn.C accmn.o
CC  cppccmn.C:
cc      cppccmn.c accmn.o  -lC
% a.out
5
3
```

Accessing FORTRAN common variables

The example in this section shows you how to access FORTRAN common variables in C++. The FORTRAN source code is shown in Figure 72.

Figure 72
Accessing FORTRAN common blocks in C++, FORTRAN main routine

```
      PROGRAM MAIN              ! Source file is fcppcmn.f
      REAL*8 A, X
      INTEGER*4 B, Y
      COMMON A, B
      COMMON /NAMED/ X, Y
      CALL SUB1 ( )
      PRINT *, A, B
      PRINT *, X, Y
      END
```

The C++ source code is shown in Figure 73.

Figure 73
Accessing FORTRAN common blocks in C++, C++ routine

```
// Source file is fcppcmn2.C
struct block { double a; int b; };

extern "C" {
  void sub1_( void );
  extern struct block __blnk_;
  extern struct block _named_;
}

void sub1_ ()
{

  __blnk_.a = 3.1415927;
  __blnk_.b = 61659;

  _named_.a = 2.7182818;
  _named_.b = 95616;
}
```

This example is derived from the section "Accessing common blocks" on page 76.

You should note the following about this example:

- C++ structures are used to access common block areas defined in FORTRAN.

- C++ references the blank common area as `__blnk_` because the C compiler only prepends an underscore to this data object. In contrast, the symbol that the FORTRAN compiler produces for common block data is `___blnk_`.
- The symbol that FORTRAN produces for the named common block is `__named_`. The C++ function refers to this symbol as `_named_` because the C compiler only prepends an underscore to external objects.

The command lines to compile this program and its output are:

```
% CC -c fcppcmn2.C
CC  fcppcmn2.C:
cc  -c  fcppcmn2.c
% fc -sa fcppcmn.f fcppcmn2.o -lC
% a.out
   3.14159270000000                61659
   2.71828180000000                95616
```

Because the main routine is FORTRAN, the linker must be invoked with the `fc` command line. Consequently, you must specify the C++ libraries on the `fc` command line with the `-lC` option.

Integer and floating-point arguments

The example in this section is derived from the section "Integer and floating-point arguments" on page 116. Figure 74 contains the Ada source file.

Figure 74
Passing integer and floating-point arguments from Ada to C++, Ada main program

```
-- Source file is acppir.a
WITH text_io; USE text_io;
WITH system;  USE system;
WITH long_int; USE long_int;

PROCEDURE ACPPIR IS
  PROCEDURE sub1( I      : in SYSTEM.ADDRESS;
                  I_LONG : in SYSTEM.ADDRESS;
                  A_FLOAT  : in SYSTEM.ADDRESS );
  PRAGMA INTERFACE( C, sub1 );
  PACKAGE int_io IS NEW integer_io (INTEGER);
  PACKAGE real_io IS NEW float_io (SHORT_FLOAT);
  USE int_io, real_io;
  I  : INTEGER;
  I_LONG : LONG_INTEGER;
  A_FLOAT  : SHORT_FLOAT;
BEGIN
  I := 0;
  I_LONG := int_to_long(0);
  A_FLOAT := 0.0;

  sub1( I'ADDRESS, I_LONG'address, A_FLOAT'address);

  PUT(I);
  PUT_LINE("");
  PUT(long_to_int(I_LONG));
  PUT_LINE("");
  PUT(A_FLOAT);
  PUT_LINE("");
END ACPPIR;
```

Figure 75 contains the C source file.

Figure 75
Passing integer and floating-point arguments from Ada to C++, C++ routine

```
// Source file is acppir.C
extern "C" void sub1( int *x, long long *y, float *z);

void sub1( int *x, long long *y, float *z )
{
  *x = 10;
  *y = 145;
  *z = 432.98;
}
```

You should note the following about this example:

- The mode of the parameters in procedure `sub1` is `in`. The current implementation of CONVEX Ada does not permit a C function accessed by an Ada program to have parameters of mode `out` or `in out`. This restriction only affects the declaration of the C function in Ada. Parameters that are changed in the C or C++ function remain changed in the Ada program, as this example demonstrates, because they are passed by reference.
- The `INTERFACE` pragma supports calls to C. The second parameter is the name of the C++ function that has C linkage. The parameter types and return type of the function must be scalar, access, or type `SYSTEM.ADDRESS`.
- In the actual function call, apply the `ADDRESS` attribute to each parameter of `sub1` so that you can modify them in the C++ function.
- The `long_int` package, included in the `convexlib` directory, contains a definition of the `LONG_INTEGER` data type. It also define some functions, such as `int_to_long` and `long_to_int`, that pertain to this data type. Refer to `convexlib/long_int.a` for more details.

The command lines for this example and its output are:

```
% CC -c acppir.C
CC  acppir.C:
cc  -c  acppir.c
% ada -M acppir.a acppir.o -lC
% a.out
         10
        145
 4.32980E+02
```

Because the main routine is Ada, the linker must be invoked with the `ada` command line. Consequently, you must specify the C++ libraries on the `ada` command line with the `-lC` option.

Array arguments

The example in this section is derived from the section "Array arguments" on page 104. Figure 76 contains the C++ source code.

Figure 76
Passing a constrained array from C++ to Ada, C++ main program

```
// Source file is cppaarr.C
int a[5][3];

extern "C" void print_array( int a[5][3] );

main()
{
   int i,j;

   for( i=0; i<5; i++ )
      for( j=0; j<3; j++ )
         a[i][j] = (i+1)*(j+1);

   print_array( a );
}
```

Figure 77 contains the Ada source code.

Figure 77
Passing a constrained array from C++ to Ada, Ada routines

```
-- Source file is cppaarr.a
with TEXT_IO; use TEXT_IO;
with SYSTEM;  use SYSTEM;

package CPPASUB5 is
  type MATRIX_2D is array (1..5, 1..3 ) OF INTEGER;
  package INT_IO is new INTEGER_IO (INTEGER); use INT_IO;

  procedure PRINT_ARRAY( I_ARR : in MATRIX_2D );
  pragma EXTERNAL_NAME( PRINT_ARRAY, "_print_array" );
end CPPASUB5;

package body CPPASUB5 is
  procedure PRINT_ARRAY( I_ARR : in MATRIX_2D ) is
  begin
    for J in I_ARR'RANGE(1) loop
      PUT(I_ARR(J,1));
      PUT(" ");
      PUT(I_ARR(J,2));
      PUT(" ");
      PUT(I_ARR(J,3));
      PUT_LINE("");
    end loop;
  end PRINT_ARRAY;
end CPPASUB5;

-- Ada main program follows
with CPPASUB5; use CPPASUB5;
procedure CPPAARR is
  procedure CMAIN;
  pragma INTERFACE(C, CMAIN);
  pragma INTERFACE_NAME(CMAIN, "_main" );
begin
  CMAIN;
end CPPAARR;
```

Note that the C++ array is passed by address. You must pass constrained arrays by address.

The command lines for this example and its output are:

```
% CC -c cppaarr.C
CC  cppaarr.C:
cc  -c  cppaarr.c
% ada -M cppaarr.a cppaarr.o -lC
% a.out
        1          2          3
        2          4          6
        3          6          9
        4          8         12
        5         10         15
```


Even though the main routine is in C++, the linker must be invoked with the ada command line. This is because all programs that contain Ada source code must be linked with the ada driver. Consequently, the only way to create a program in which the main program unit is in C++ is to call that unit from an Ada procedure, as shown in this example.

There are a few problems with this approach. Because the initial routine is in Ada, the Ada runtime system is created. By calling the C++ entry point, _main, you initiate a subset of the C++ runtime system. One side effect is that the C++ main program no longer has access to command line arguments or the environment. Access to the command line arguments and the environment are not provided with the parameters of the C main program, commonly named argc, argv, and envp. You can work around this problem using the COMMAND_LINE package in the Ada library, convexlib. Finally, the exit value of the C++ main routine is not passed on to the calling Ada routine.

You should also remember that the Ada compiler disables integer overflow detection before a call to a C++ function by default. Any subsequent calls from that C function to an Ada routine will also have integer overflow detection disabled. For information on Ada compiler options that affect integer overflow, refer to the description of the -od and -oe options in the section "Ada options" on page 1.

Note that the -lC option is included on the ada command line. This option specifies that the linker should include the C++ libraries in its search for object code.

Passing a complex argument from C++

The example in this section is derived from the section "Complex arguments" on page 44. The C code has been modified to use the complex C++ data type and the C linkage specifier. The FORTRAN source code has been modified similarly.

Figure 78 contains the C++ source code.

Figure 78
Passing a complex data type from C++ to FORTRAN, C++ main program

```
// Source file is cppfcx.C
#include <complex.h>

extern "C" void sub1_( complex * , complex * );

main()
{
  complex cx1, cx2;

  cx1 = complex(5.0,5.0);
  sub1_( &cx1, &cx2 );

  cout << cx1 << "\n";
  cout << cx2 << "\n";
}
```

Figure 79 contains the FORTRAN source code.

Figure 79
Passing a complex data type from C++ to FORTRAN, FORTRAN routine

```
      SUBROUTINE SUB1 (CX1,CX2)       ! Source file is cppfcx2.f
      COMPLEX*16 CX1, CX2
      CX1 = (1234.0,5678.0)
      CX2 = (8765.0,4321.0)
      RETURN
      END
```

The command lines to compile this example and its output are:

```
% fc -c cppfcx2.f
% CC cppfcx.C cppfcx2.o -lcomplex
CC  cppfcx.C:
cc     cppfcx.c cppfcx2.o  -lcomplex -lC
% a.out
( 1234, 5678)
( 8765, 4321)
```

The CC command line includes the `-lcomplex` option because this example uses routines in the C++ complex arithmetic library which is named `libcomplex.a`.

Passing a complex data type to C++

The example in this section is derived from the section "Complex arguments" on page 30. The C code has been modified to use the complex C++ data type and the C linkage specifier. The FORTRAN source code has been modified similarly. Figure 80 contains the C++ source code:

Figure 80
Passing a complex data type from FORTRAN to C++, C++ routine

```
#include <complex.h>        // Source file is fcppcx2.C
#include <iostream.h>

extern "C" void sub1_( complex *cx, int *ix, float *fx );

void sub1_( int *num )
{
  *num = 123;
}

void sub1_( float *num )
{
  *num = 678.9;
}

void sub1_( complex *cx, int *ix, float *fx )
{

  sub1_( ix );
  sub1_( fx );

  *cx = complex( (float)*ix, *fx );
}
```

Figure 81 contains the FORTRAN source code.

Figure 81
Passing a complex data type from FORTRAN to C++, FORTRAN main program

```
      PROGRAM MAIN                          ! Source file is fcppcx.f
      COMPLEX*16 CX1 /(0.0,0.0)/
      INTEGER*4 IIX
      REAL*4 FFX
      CALL SUB1( CX1, IIX, FFX )
      WRITE (6,'(''(''G10.4'',''G10.4'')'' I3 G10.4)') CX1, IIX, FFX
      END
```

Note that the C++ source code contains three functions that have overloaded the `sub1_` function name. The linkage specifier permits only one of these functions to have a C linkage. This example demonstrates that one overloaded function can be used by another language.

The command lines used to compile this program and its output are:

```
% CC -c fcppcx2.C
CC  fcppcx2.C:
cc  -c  fcppcx2.c
% fc -sa fcppcx.f fcppcx2.o -lcomplex -lC
% a.out
( 123.0     , 678.9     )123 678.9
```

Complex return types

The example in this section demonstrates how to access C++ functions that return complex data types to a FORTRAN main program. Figure 82 contains the C++ source code.

Figure 82
Calling C++ functions that return complex types to FORTRAN, C++ source code

```
// Source file is fcppxf16.C
#include <complex.h>

extern "C" void makecx16_( complex *cx, double *a1, double *a2 );

void makecx16_( complex *cx, double *a1, double *a2 )
{
   *cx = complex( *a1, *a2 );
}


// Source file is fcppxf8.C
#include <complex8.h>

extern "C" void makecx8_( complex8 *cx, float *a1, float *a2 );

void makecx8_( complex8 *cx, float *a1, float *a2 )
{
   *cx = complex8( *a1, *a2 );
}
```

Figure 83 contains the FORTRAN source code.

Figure 83
Calling C++ functions that return complex types to FORTRAN, FORTRAN main program

```
PROGRAM MAIN                          ! Source file is fcppxf.f
COMPLEX*16 MAKECX16, ANS16
COMPLEX*8  MAKECX8,  ANS8
REAL*4 B1 /5.0/, B2 /7.0/
REAL*8 D1 /5.0/, D2 /7.0/

ANS8 = MAKECX8( B1, B2 )
ANS16= MAKECX16( D1, D2 )

PRINT *, ANS8
PRINT *, ANS16
END
```

You should note the following about this example:

- A C++ function that is type `void` simulates a FORTRAN function that returns COMPLEX variables.
- The first parameter of the C++ functions, `makecx8_` and `makecx16_`, contains the address of the COMPLEX variable that they are returning.
- The second parameter of the C++ functions corresponds to the first argument that the FORTRAN main program passes to them.

The command lines used to compile this program and its output are:

```
% CC -c fcppxf16.C fcppxf8.C
CC  fcppxf16.C:
CC  fcppxf8.C:
cc  -c  fcppxf16.c fcppxf8.c
fcppxf16.c:
fcppxf8.c:
% fc -sa fcppxf.f fcppxf16.o fcppxf8.o \
     -lcomplex -lC
% a.out
 (  5.000000     ,  7.000000     )
 (  5.00000000000000     ,  7.00000000000000     )
```

Mixed input and output

The example in this section demonstrates how to initialize the FORTRAN input and output system by calling the `f_init` routine. This routine conforms to the C calling conventions, but it is found in `libU77.a`. Figure 84 contains the FORTRAN source code:

Figure 84
Mixing C++ and FORTRAN input and output, FORTRAN source code

```
      SUBROUTINE SUB                    ! Source file is cppfio2.f
      PRINT *, 'Printed from the FORTRAN subroutine'
      END
```

Figure 85 contains the C++ source code:

Figure 85
Mixing C++ and FORTRAN input and output, C++ source code

```
// Source file is cppfio.C
#include <iostream.h>

extern "C" {
   void f_init( void );
   void sub_( void );
}

main()
{
   f_init();
   cout << "Printed from the C++ main routine\n" << flush;
   sub_();
   cout << "Printed from the C++ main routine again\n";
}
```

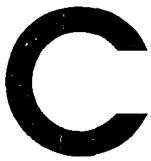
Note that the `f_init` function is declared in the C linkage specifier. Also, the `flush` argument is required.

The command lines used to compile this example and its output are:

```
% fc -c cppfio2.f
% CC cppfio.C cppfio2.o -lU77 -lF77 -lI77 \
    -lD77 -lmathC2
CC  cppfio.C:
cc     cppfio.c cppfio2.o  -lU77 -lF77 -lI77
-lD77 -lmathC2 -lC
% a.out
Printed from the C++ main routine
Printed from the FORTRAN subroutine
Printed from the C++ main routine again
```

Note that the options `-lU77`, `-lF77`, `-lI77`, `-lD77`, and `-lmathC2` are included on the `CC` command line. These are the FORTRAN libraries. The `-lmathC2` option is required because FORTRAN I/O calls FORTRAN math functions to perform calculations. If you compile this code on a CONVEX C1 Series machine, you must use the `-lmathC1` option instead of the `-lmathC2` option.

Register interface



This appendix describes calling conventions that you should be familiar with when using the assembler. It is not necessary to know this information if you are only programming in Ada, C, or FORTRAN. Specifically, this appendix discusses the following topics:

- Special address registers, including the stack pointer, argument pointer, and frame pointer
- Interprocedural call instructions
- General calling conventions
- Runtime-stack layout
- FORTRAN argument packets

Special address registers

Some address registers are used for special functions by the hardware and the compilers to implement function and procedure calls. The address registers A0, A6, and A7 are used as the stack-pointer (SP), argument-pointer (AP), and frame-pointer (FP) registers, respectively.

Stack pointer

The psh and pop machine instructions manipulate data on the top of a runtime stack by incrementing and decrementing the SP. Although the other address registers can be used as index registers for load and store instructions, the SP cannot. The hardware uses a 0 in the index register field of a load or store instruction to indicate that no index register is to be used. This encoding prevents the use of the stack pointer (A0) as an index register. Compilers generate automatic (local) storage on top of the runtime stack by directly adjusting the SP.

Argument pointer

By convention, the code generated by CONVEX compilers uses address register A6 as an argument-pointer (AP) register. Before calling a subprogram (subroutine or function), the generated code loads into the AP the base address of the argument list to be passed to the called routine. This argument list can be located either on the runtime stack or anywhere in the program address space. When possible, the FORTRAN compiler builds packets of subprogram arguments at compile time to increase runtime efficiency. The called routine references the arguments passed to it by using the AP as an index register.

Frame pointer

The call instructions (`call` and `calls`) and the return instructions (`rtn` and `rtnc`) use address register A7 as a frame pointer in order to maintain linkage information for subprogram calls. Each `call` or `calls` instruction causes the contents of several registers to be saved on the top of the runtime stack (determined by the contents of SP), and the value of FP is updated to point to the saved context. The compiler-generated code uses the FP as an index register to refer to automatic storage. The program can find the previous routine context blocks by retrieving subsequent frame pointers out of the saved context block pointed to by the current frame pointer.

In general, a function uses scalar register S0 to return its result value to the calling routine. The return (`rtn`) instruction does not alter the value of S0, so the return value is available to the calling routine.

Compiler-generated code

Other than the SP, AP, and FP registers, the compiler-generated code makes no other assumptions about registers being preserved across procedure calls. The CONVEX C compiler (`cc`) ignores the `register` keyword in its language except when a function contains the `asm` keyword. The CONVEX compilers explicitly save whatever intermediate information exists in registers before calling a routine.

Interprocedural call instructions

The interprocedural call instructions `call`, `calls`, and `callq` store information on and retrieve information from the runtime stack. Each of these three instructions push current state

information on the runtime stack and branch to a destination address contained within the instruction. For the return, the instructions keep track of the called routine's virtual address.

callq

The `callq` instruction, or fast call, pushes the current value of the program-counter register on to the runtime stack. The value in the stack-pointer register is decreased by 4 to account for the pushed value. None of the other registers is saved by the `callq` instruction.

To return, the routine executes a `rtnq` instruction, which removes the program-counter (PC) value (the top word of the runtime stack), and places the return address into the PC where execution continues after the subroutine call.

The CONVEX compilers never use the `callq` instruction for calling user-written functions; `callq` is used only for calling short library routines that perform housekeeping functions.

calls

The `calls` instruction is the most frequently used instruction for interprocedural calling. Like the `callq` instruction, the `calls` instruction pushes the value of the return address onto the runtime stack. It also pushes the value of the processor status word (PSW), the value of the frame-pointer register (A7), and the value of the argument-pointer register (A6).

After these 4 words have been pushed on the stack, the frame-pointer register (A7) is updated to contain the current value of the stack pointer. The called routine can then access the last stack frame on the runtime stack using the frame-pointer register. To return control to its caller, the called routine executes a `rtn` instruction.

call

Like the `calls` instruction, the `call` instruction saves the current values of the program counter, processor status word, frame pointer, and argument pointer on the runtime stack. The `call` instruction also saves the current values of the other address and scalar registers, except A0 and S0.

The CONVEX compilers currently do not generate any `call` instructions, but use the faster `calls` instruction instead.

General calling conventions

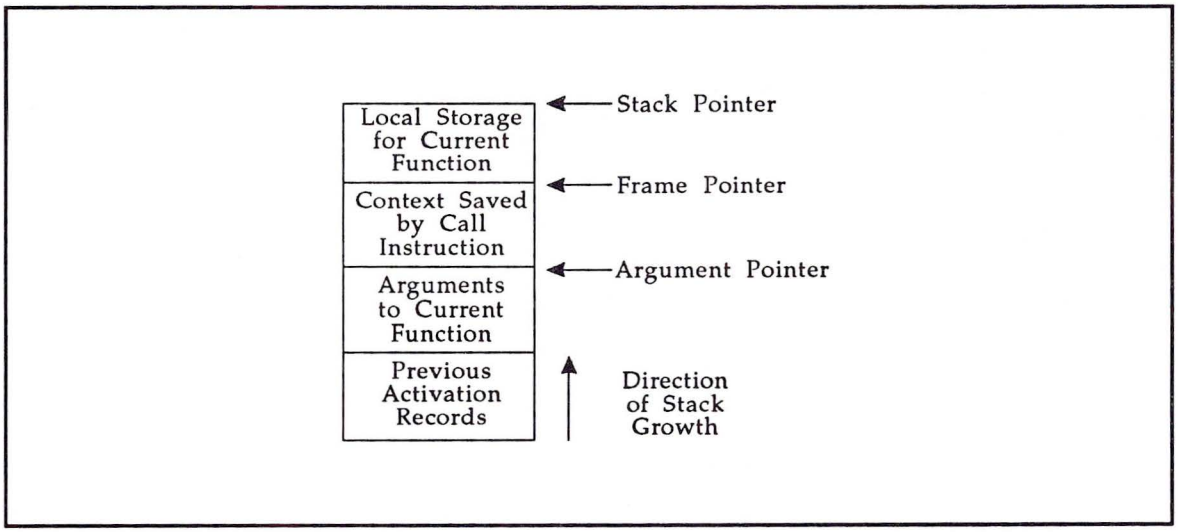
This section describes the general calling conventions used with CONVEX functions and subroutines. When function or subroutine calls are executed, the current state of several hardware registers used by the calling routine must be preserved. The contents of these registers are pushed onto the runtime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

CONVEX compilers do not preserve the value of every register across a function call. Only those registers required to maintain the state of the runtime stack are preserved. Called routines that can restore the frame-pointer register to its original state are allowed to modify any register passed to them.

Function or subroutine stack layout

Figure 86 illustrates the top of the runtime stack. Although the stack grows downward in the address space, this diagram shows the stack as growing upward on the page. The stack-pointer register contains the address of the topmost location on the runtime stack. The frame-pointer register contains the address of the last frame pushed on the runtime stack by a `call` or `calls` instruction. The argument-pointer register contains the address of the arguments passed to the current routine.

Figure 86
Top of the runtime stack



Function or subroutine calling sequences

When a function is called, the compiler generates code according to one of the following two sequences.

Sequence 1

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

2. Update the argument-pointer register. The updated register should point to the first argument in the argument list. (The first argument in the list is the last one pushed.)

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

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

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

Sequence 2

For FORTRAN, the arguments are precompiled when possible (only when the `-sa` option is not used), so the calling sequence is reduced to the following:

1. Load the address of the argument packet into the argument pointer.

2. Call the subroutine (using the `calls` instruction).

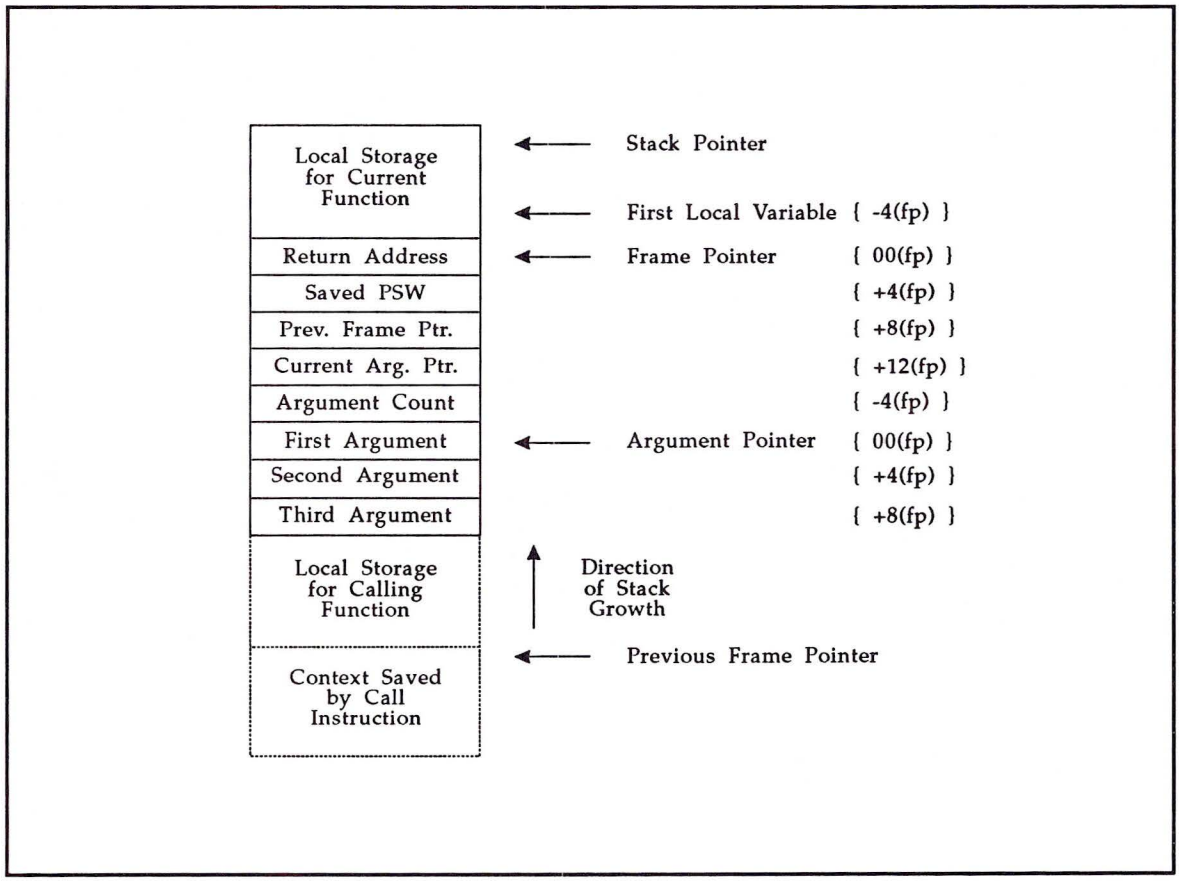
Conventions that apply to function calls are listed below:

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

- The frame-pointer register points to the context block pushed by the caller when calling the child routine. The called child routine references the local storage it has allocated on the runtime stack by referencing negative offsets from the frame pointer.
- The called routine references arguments passed to it by its parent by referencing positive offsets from the argument-pointer register: 0(ap) corresponds to the first argument, 4(ap) corresponds to the second argument, and so on. The word with an address of -4 relative to the argument pointer contains a count of the number of arguments passed to the routine.

Figure 87 illustrates the layout of the stack as seen by a routine after it has been called, and after it has allocated some storage for local variables on the top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 87
Stack layout



Called routines return to their parents by:

1. Placing a return value in register S0.

2. Executing the `rtn` instruction.

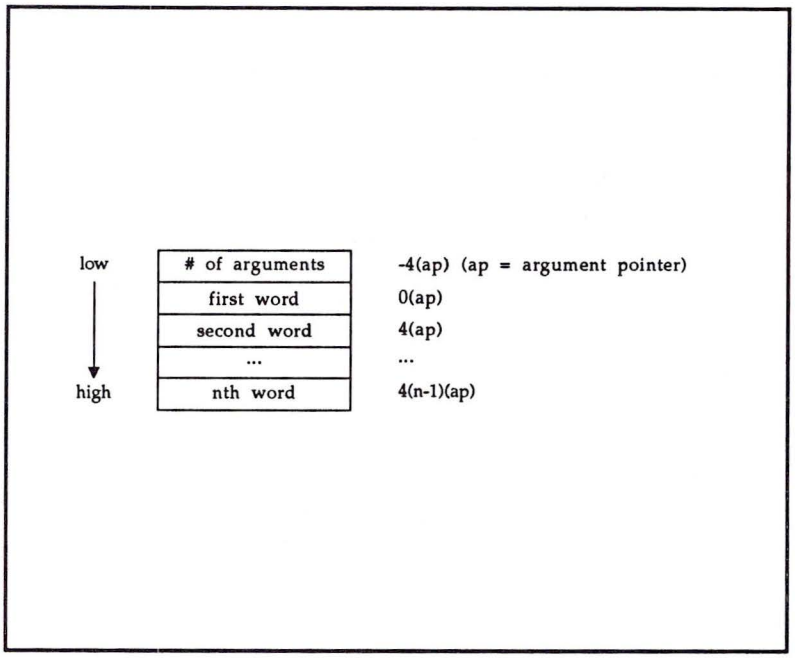
When the computer executes the `rtn` instruction, automatic storage allocated by the called routine is automatically deallocated. This instruction also restores the PSW register and the frame pointer to their previous states, and then returns control to the location immediately following the `calls` instruction that called the routine. After control returns to the parent, the stack-pointer register points to the location that contains the pushed argument count. The parent routine adds a fixed offset to the value in the stack pointer to remove the argument count and any pushed arguments. The value added is the total number of bytes pushed prior to the call. Finally, before the parent can access any of its own arguments, it must reload its own argument-pointer register from the current frame on the stack. This value is offset +12 bytes from the recently restored frame pointer.

FORTRAN argument packets

This section includes additional information on FORTRAN argument packet calling conventions.

An argument packet is a sequence of word (4-byte) entries. Only precompiled argument lists are allocated static memory space. When possible, the compiler creates argument list entries at compile time, but it cannot precompile an argument list if any argument is a dummy argument, an array element with nonconstant subscripts, or the if the `-sa` compiler option is used. Figure 88 shows the layout of an argument packet in memory.

Figure 88
Argument packet: example 1



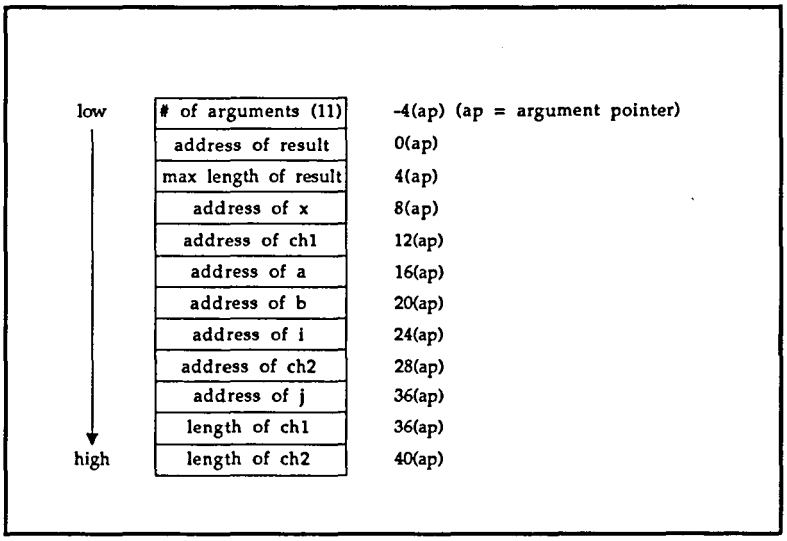
The first word is normally the address of the first argument, the second word is the address of the second argument, and so on. For character arguments, an extra by-value word containing the length of the character entity is added to the end of the list. For each character argument there is one extra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

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 contains two words: the first is the address of the character string to receive the value of the function and the second is its maximum length.

Figure 89 shows the argument packet layout resulting from the following FORTRAN code:

```
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
```

Figure 89
Argument packet: example 2



Index

G

I

L

M

N

O

P

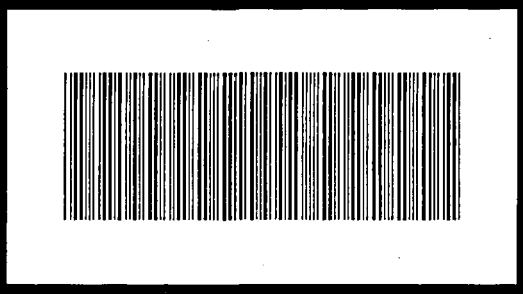
R

S

T

U

V



Order Number
DSW-043

Document Number
720-005830-000