IBM RT PC FORTRAN 77

FORTRAN 77

Programming Family

IBM

Personal
Computer
Software

59X7877

IBM RT PC FORTRAN 77

FORTRAN 77

Programming Family

Personal
Computer
Software

First Edition (November 1985)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

©Copyright International Business Machines Corporation 1985
©Copyright INTERACTIVE Systems Corporation 1984
©Copyright AT&T Technologies 1983

About This Book

This book, *IBM RT PC FORTRAN 77*, is a reference for this implementation of FORTRAN 77 on the IBM RT PC.

The IBM RT PC FORTRAN 77 Licensed Program Product implementation is based on the specifications outlined in the *American National Standard Programming Language FORTRAN* document (ANSI X3.9-1978) approved by the American National Standards Institute (ANSI) in 1978. This level of FORTRAN is commonly known as ***FORTRAN 77***.

The IBM RT PC FORTRAN 77 Licensed Program Product implementation is an adaptation of the version of FORTRAN 77 implemented in UNIX[1] System V produced by AT&T Bell Laboratories.

IBM RT PC FORTRAN 77 includes additional function defined in the *FORTRAN Military Standard* (MIL-STD-1753) published by the United States Department of Defense.

Functional specifics unique to this implementation are also discussed in this book.

Who Should Read This Book

The *IBM RT PC FORTRAN 77* book contains information for programmers to use in the design, writing, compilation, and execution of FORTRAN programs on the IBM RT PC.

You should have a working knowledge of some dialect of FORTRAN before using this book. Also, parts of this book assume some familiarity with the use of the AIX Operating System on the IBM RT PC.

How to Use This Book

Chapters 1 through 5 discuss the IBM RT PC FORTRAN 77 implementation. It is intended that you refer to these chapters for details on usage and functionality specific to this FORTRAN 77 implementation on the IBM RT PC. Refer to books dealing with ANSI Standard FORTRAN 77 for information on FORTRAN 77 specifics.

Chapters 6 through 8 discuss preprocessors for FORTRAN 77. If you are unfamiliar with the function and use of preprocessors, you may want to read the brief general explanation of preprocessors provided in Chapter 6, "Overview of Preprocessors." The chapter also generally discusses the two preprocessors, Ratfor and EFL, that are implemented for IBM RT PC FORTRAN 77.

---

1 Trademark of AT&T Bell Laboratories

If you are familiar with preprocessors and want detailed information on their use, you can read Chapter 7, "Ratfor — The Rational FORTRAN Preprocessor" for information on Ratfor and Chapter 8, "EFL — Extended FORTRAN Language" for information on EFL.

Organization

The *IBM RT PC FORTRAN 77* book is organized as follows:

- "Part 1. FORTRAN 77" consists of Chapters 1 through 5. This part of the book discusses the specifics of this FORTRAN 77 implementation on the IBM RT PC system.
- Chapter 1, "Differences Between IBM RT PC FORTRAN 77 and ANSI Standard FORTRAN 77," discusses differences between this FORTRAN 77 implementation and ANSI Standard FORTRAN 77. Both violations of the Standard and functional enhancements to the Standard are explained.
- Chapter 2, "Functions and Subroutines," explains the functions and subroutines that are part of the FORTRAN 77 libraries in this implementation. A quick reference alphabetical table of these functions and subroutines can be found at the end of the chapter.
- Chapter 3, "Compiling, Linking, Debugging, and Running a Program," describes the basics of compiling, linking, debugging, and running FORTRAN programs. It explains in detail the available compiling and linking options.
- Chapter 4, "Linking C With FORTRAN" describes how C Language programs can be linked to FORTRAN programs. It includes program examples and a discussion of argument passing between C Language and FORTRAN.
- Chapter 5, "AIX Operating System Commands for FORTRAN," explains the format and use of certain IBM RT PC AIX Operating System commands that you can use with FORTRAN program files.
- "Part 2. Ratfor and EFL — Two Preprocessors for FORTRAN" consists of Chapters 6, 7, and 8. This part of the book discusses general characteristics of preprocessors and the specific implementations of two FORTRAN preprocessors, Ratfor and EFL, that are included with the IBM RT PC FORTRAN 77 Licensed Program Product.
- Chapter 6, "Overview of Preprocessors," briefly explains the concept and implementation of preprocessors. It also provides overviews of Ratfor and EFL.
- Chapter 7, "Ratfor — The Rational FORTRAN Preprocessor," discusses the structure, statements, and use of Ratfor.
- Chapter 8, "EFL — Extended FORTRAN Language," discusses the structure, statements, and use of EFL.
- Appendix A, "Installing the IBM RT PC FORTRAN 77 Licensed Program Product," describes how to install the FORTRAN compiler on the IBM RT PC.

- Appendix B, "ASCII Character Codes," lists the decimal, octal, hexadecimal, and character representations for each ASCII standard character and for other characters supported on the IBM RT PC.

A glossary of terms and an index are provided at the end of this book for your information.

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

In addition to this book, the IBM RT PC FORTRAN 77 licensed program consists of the FORTRAN compiler and a library of object modules that make up the FORTRAN runtime library.

Highlighting Conventions

The highlighting conventions used in this book are as follows:

- Words printed in **lowercase boldface letters** in this book must be entered exactly as shown when they are part of a program. Many names of built-in functions and subroutines are used as part of their input formats. They appear in lowercase boldface letters in the *Format* sections of each explanation in "Function and Subroutine Directory" on page 2-5 and in other places in this book.

  For example, the format for the **getarg** subroutine is:

  call getarg (*i1*, *ch1*)

  The lowercase boldface letters indicate that both **call** and **getarg** must appear in program code exactly as they appear in the example.

- Items printed in *lowercase italic letters* are general names for items that you define and name in a program. Once such an item is defined, it retains its meaning for the entire discussion.

  For example, in the format for the **getarg** subroutine:

  call getarg (*i1*, *ch1*)

  the lowercase italic letters indicate that you replace the general names *i1* and *ch1* with variable names when entering the **getarg** subroutine in a FORTRAN program.

- Brackets indicate optional items.

  For example, the format for the **cmplx** function:

  cx1 = **cmplx** (*i1* [, *i2*])

  indicates that *i2* is an optional argument for the **cmplx** function.

- Punctuation marks that appear in lines of example code, such as commas, colons, slash marks, parentheses, and equal signs, must be entered exactly as shown.

- Blanks normally have no significance in the description of FORTRAN statements.

Terminology

The following terminology conventions are followed in this book:

- The word ***FORTRAN*** is used when discussing things specific to the IBM RT PC FORTRAN 77 licensed program implementation of the FORTRAN language on the IBM RT PC
- The phrase ***FORTRAN 77*** is used when discussing things specific to the ANSI standard known as FORTRAN 77
- The phrase ***FORTRAN 66*** is used when discussing things specific to the ANSI standard known as FORTRAN 66.

Related Information

While you are using this book, you may find references to one or more of the other books in the IBM RT PC library. The following paragraphs discuss some of these books.

- *IBM RT PC Bibliography and Master Index* provides brief descriptive overviews of the books and tutorial program that support the IBM RT PC hardware and the AIX Operating System. In addition, this book contains an index to the RT PC and AIX Operating System library.
- *IBM RT PC Using and Managing the AIX Operating System* describes using AIX Operating System commands, working with the file system, and developing shell procedures. This book also provides instructions for performing such system management tasks as adding and deleting user IDs, creating and mounting file systems, backing up the system, and repairing file system damage.
- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands.
- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.
- *IBM RT PC AIX Operating System Technical Reference* describes the system calls and subroutines that a C programmer uses to write programs. This book also provides information about the AIX file system, special files, miscellaneous files, and writing device drivers. (Available optionally)
- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)

- *IBM RT PC C Language Guide and Reference* provides guide information for writing, compiling, and running C language programs and includes reference information about C language data structures, operators, expressions, and statements. (Available optionally)

See IBM RT PC Bibliography and Master Index for order numbers of IBM RT PC publications and diskettes.

Ordering Additional Copies of This Book

To order additional copies of this publication (without program diskettes), use either of the following sources:

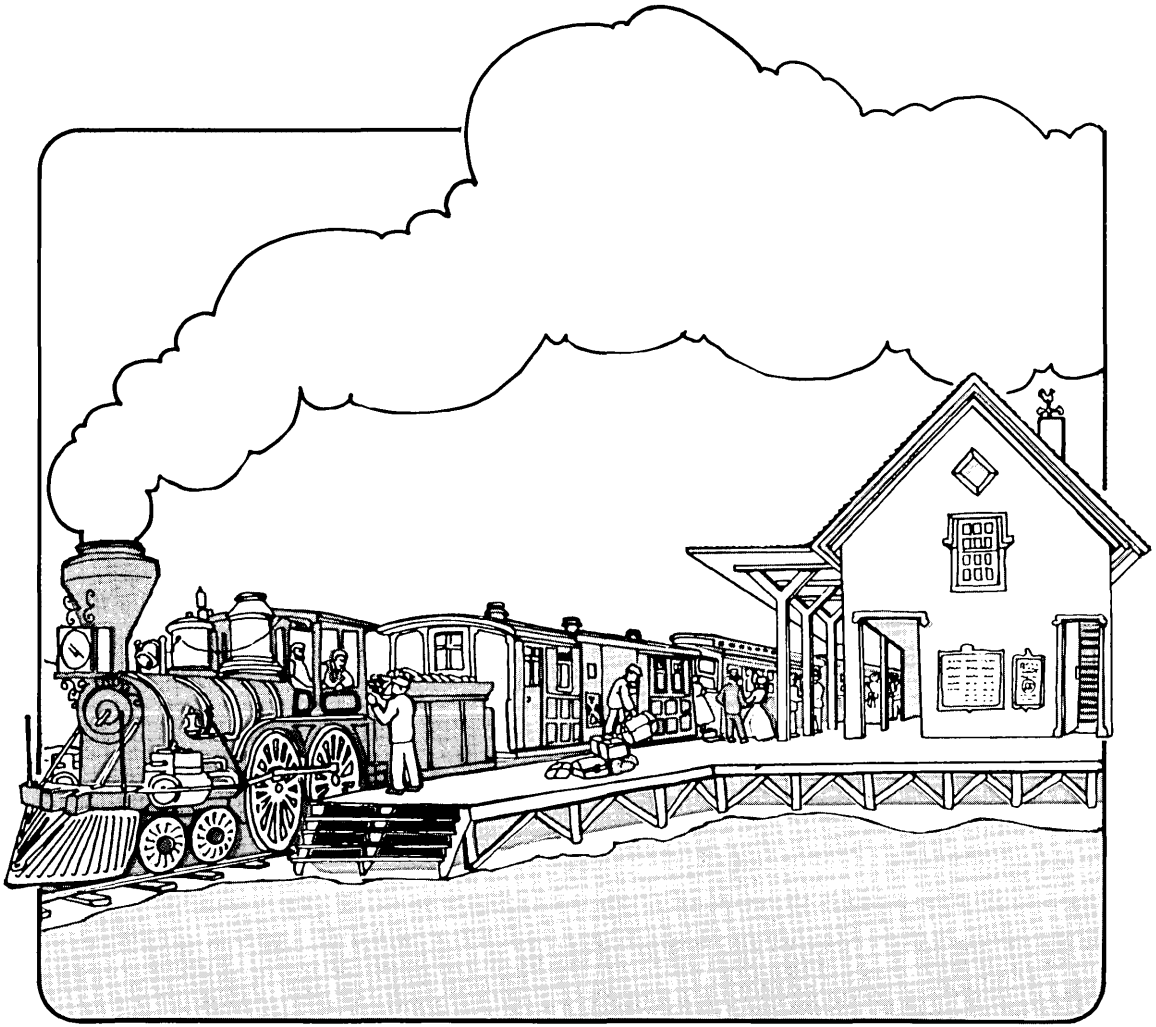
- To order from your IBM representative, use Order Number SV21-8027.
- To order from your IBM dealer, use Part Number 6280760.

A binder is included with the order.

Contents

Part 1. FORTRAN 77

Chapter 1. Differences Between IBM RT PC FORTRAN 77 and ANSI Standard FORTRAN 77



CONTENTS

About This Chapter

This chapter discusses the differences between this FORTRAN 77 implementation and ANSI Standard FORTRAN 77, and functional enhancements built into this version of FORTRAN 77.

The differences explained in this chapter consist largely of:

- Functional enhancements beyond the FORTRAN 77 standard
- Additional function that allows IBM RT PC FORTRAN 77 to communicate with C Language procedures
- Additional function that allows the compilation of FORTRAN programs written according to the 1966 ANSI Standard FORTRAN (FORTRAN 66).

In addition, this chapter discusses the file formats recognized by the I/O system.

Overview of the IBM RT PC FORTRAN 77 Compiler

The IBM RT PC FORTRAN 77 compiler is an implementation of the ANSI standard FORTRAN 77, with extensions. The standard defines two levels of FORTRAN 77:

- Full FORTRAN
- Subset FORTRAN.

IBM RT PC FORTRAN 77 can be viewed as a superset of Full FORTRAN.

Code generated by the IBM RT PC FORTRAN 77 compiler is compatible with calling sequences produced by C Language compilers and can drive C Language procedures produced by most C Language compilers.

Differences and Enhancements

The following sections describe differences between IBM RT PC FORTRAN 77 and ANSI Standard FORTRAN 77, and functional enhancements implemented in IBM RT PC FORTRAN 77.

Dummy Procedure Arguments

If any argument of a procedure is of type **character**, all dummy procedure arguments of that procedure must be declared in an **external** statement. A warning is printed if a dummy procedure is not declared external. Code is correct if there are no character arguments.

T and TL Formats

The **T** (absolute tab) and **TL** (leftward tab) format codes do not function in IBM RT PC FORTRAN 77 as defined in the ANSI Standard. These codes allow rereading or rewriting of part of the record that has already been processed.

IBM RT PC FORTRAN 77 uses seeks. If the unit is not one that allows seeks, such as a terminal, unpredictable results can occur.

A benefit of the implementation chosen is that there is no upper limit on the length of a record. Also, record lengths must be predeclared only where specifically required by FORTRAN or the operating system.

Logical*1 Data Type

A **logical*1** data type is implemented in this version of FORTRAN. The **logical*1** data type is a 1-byte logical quantity. It can be assigned the values **.true.** and **.false.**

Through equivalencing, other values can be assigned to **logical*1** quantities. As a result of equivalencing, a **logical*1** variable can hold any 8-bit quantity. All values other than 0 (zero) are **.true.**

When used in an expression, a **logical*1** variable behaves as a **logical*4** variable.

A **logical*1** variable can be printed as an integer or character in a formatted I/O statement. However, a **logical*1** variable is never implicitly converted to another data type.

In common storage, **logical*1** variables must not force halfword or fullword quantities to begin on a boundary other than a halfword or fullword boundary. In the following example:

```
logical*1 log1
integer*4 int
o
o
o
common log1(3), int
```

the **common** statement attempts to force an integer quantity to begin after 3 bytes. The integer requires 4 bytes of storage, a full word. Since fullword quantities must begin on fullword boundaries, the statement is invalid and is treated as an error by the compiler.

The following statement shows a valid use of a **logical*1** quantity in a **common** statement:

```
logical*1 log1
integer*4 int
o
o
o
common log1(4), int
```

Double Complex Data Type

The new type **double complex** is implemented in IBM RT PC FORTRAN 77. Each datum is represented by a pair of double-precision real variables.

A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

Pre-Connected Units and File Positions

The IBM RT PC FORTRAN 77 supports unit numbers 0 (zero) through 99. The AIX Operating System supports up to 20 open files per process.

Units 5, 6, and 0 are preconnected when a program starts. Unit 5 is connected to the standard input. Unit 6 is connected to the standard output. Unit 0 is connected to standard error. All units are connected for sequential formatted I/O.

All other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist and are not created unless their units are used without first executing an **open** statement. The default connection is for sequential formatted I/O.

The Standard does not specify the initial position of a file that is explicitly opened for sequential I/O. The IBM RT PC FORTRAN 77 I/O system attempts to position the file at the end, so that a write operation appends to the file and a read operation results in an end-of-file indication.

The **rewind** statement can be used to position a file at its beginning. The preconnected units 5, 6, and 0 are positioned as they come from the program's parent process.

Implicit Undefined Statement

In FORTRAN, the type of a variable that does not appear in a **type** statement is **integer** if its first letter is **i, j, k, l, m** or **n,** and **real** otherwise. The **implicit** statement can be used to override this rule.

The IBM RT PC FORTRAN 77 implementation permits an additional type, **undefined**. For example, the statement:

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism and instructs the compiler to issue a diagnostic for each variable that is used but does not appear in a **type** statement.

Specifying the **-u** compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures can call themselves, directly or through a chain of other procedures.

When recursive subroutines or functions are used, the IBM RT PC FORTRAN compiler cannot, in all cases, determine if static storage is required for a local variable. The following methods will force a variable to be static:

- Use the **save** command in the subroutine or function.
- Specify the variable in a common block subprogram.
- Explicitly type the variable as **static** in the subroutine or function.

Static/Automatic Storage

Two new keywords are recognized, **static** and **automatic**. These keywords can appear as types in **type** statements and in **implicit** statements.

Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables cannot appear in **equivalence, data,** or **save** statements.

Source Input Format

The Standard expects input to the compiler to be in 72-column format. Except in comment lines, the first 5 characters are the statement number, the next is the continuation character, and the next 66 characters are the body of the line.

If there are fewer than 72 characters on a line, the IBM RT PC FORTRAN 77 compiler pads it with blanks; characters after the seventy-second are ignored.

In order to make it easier to type FORTRAN programs, the IBM RT PC FORTRAN 77 compiler accepts input in variable length lines. A statement can consist of an initial line and up to 19 continuation lines.

In the Standard, there are only 26 letters – the 26 uppercase letters of the alphabet. Consistent with AIX Operating System system usage, the IBM RT PC FORTRAN 77 compiler expects lowercase input. By default, the compiler converts uppercase characters to lowercase, except inside character constants and Hollerith fields.

However, if the **U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them, and to have distinct variables differing only in case. Regardless of the setting of the option, keywords are recognized in lowercase only.

Warning:
The compiler expects keywords such as **if** and **else** to be entered in lowercase letters. By default, most uppercase letters are automatically converted to lowercase letters by the compiler. Therefore the compiler can compile FORTRAN programs containing keywords entered in uppercase letters. However, if you use the **U** option to compile a program containing keywords entered in uppercase letters, the compiler will not be able to properly interpret the keywords.

Names can have up to 127 characters, all of which are significant. The **x** compiler option causes the compiler to treat names longer than 6 characters as errors.

The include Statement

The statement:

```
include 'stuff'
```

is replaced by the contents of the file `stuff`.

Binary Initialization Constants

An **integer** variable can be initialized in a **data** statement by a non-decimal constant, denoted by a letter followed by a string within quotation marks.

If the letter is **b**, the string is binary and only zeroes and ones are permitted in the string. If the letter is **o**, the string is octal, with digits **0** through **7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0** through **9** and **a** through **f**.

Thus, the statements:

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of the array `a` to ten.

Dimensions

IBM RT PC FORTRAN 77 supports 20 array dimensions.

Floating-Point Format

IBM RT PC FORTRAN 77 supports the ANSI/IEEE floating-point format.

The following may be displayed in place of the expected floating-point number when writing information to a display screen or printer:

Display	Meaning
QNaN	Quite NaN.
SNaN	Signaling NaN.
+INF	Positive infinity.
−INF	Negative infinity.

Figure 1-1. Floating-Point Display

Character Strings

For compatibility with C Language usage, the following backslash escapes are recognized:

Escape	Meaning
\n	New-line
\t	Tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe (does not terminate a string)
\''	Quotation mark (does not terminate a string)
\\	Backslash
x	*x*, where *x* is any other character

Figure 1-2. Backslash Escapes

Standard FORTRAN 77 recognizes only one quotation character, ' (the single quotation mark or apostrophe). The IBM RT PC FORTRAN 77 compiler and I/O system recognize both the ' (single quotation mark) and the '' (double quotation mark).

If a string begins with one variety of quotation mark, the other can be embedded within it without using the repeated quotation mark or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to facilitate communication with C Language routines.

Hollerith Data

Standard FORTRAN 77 does not have Hollerith (*n*h) notation, although the Standard recommends implementing the Hollerith feature in order to improve compatibility with old programs.

In IBM RT PC FORTRAN 77, Hollerith data can be used in place of character string constants, and can also be used to initialize non-character variables in **data** statements.

Equivalence Statements

As a special case, FORTRAN 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. FORTRAN 77 does not permit this usage, since subscript lower bounds can now be different from 1.

The IBM RT PC FORTRAN 77 compiler permits single subscripts in **equivalence** statements, under the interpretation that missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip Do Loops

The FORTRAN 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in:

```
do 10 i = 2,1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop be performed at least once.

In order to accommodate old programs, although they are in violation of the 1966 Standard, IBM RT PC FORTRAN 77 supports a **-onetrip** compiler option that causes non-standard loops to be generated.

Commas in Formatted Input

When doing a formatted read of non-character variables, commas can be used as value separators in the input record, overriding the field lengths given in a **format** statement.

Thus, the format:

```
(i10, f20.10, i4)
```

reads the record:

```
-345,.05e-3,12
```

correctly.

Short Integers

The IBM RT PC FORTRAN 77 compiler accepts declarations of type **integer*2**. An expression involving only objects of type **integer*2** is of that type.

Additional Intrinsic Functions

IBM RT PC FORTRAN 77 supports the intrinsic functions specified in the FORTRAN 77 Standard and the bitwise Boolean operations defined in the *FORTRAN Military Standard.*

IBM RT PC FORTRAN 77 also implements additional built-in functions and subroutines that access the AIX Operating System.

For more information on the functions in the IBM RT PC FORTRAN 77 intrinsic function library, see the "Function and Subroutine Directory" on page 2-5.

Linking C Language with FORTRAN

You can write C Language procedures that call or are called by IBM RT PC FORTRAN 77 procedures. Chapter 4, "Linking C With FORTRAN" contains the following information on the interface of FORTRAN and C Language programs:

- Linking FORTRAN and C Language programs
- Conventions for the passing of data between FORTRAN and C Language programs
- The source code and output of a FORTRAN program and the C Language procedure called by the FORTRAN program.

File Formats

The following sections discuss the implementation of files in IBM RT PC FORTRAN 77.

Structure of FORTRAN Files

FORTRAN requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. In IBM RT PC FORTRAN 77, these files are implemented as ordinary files that are assumed to have the proper internal structure.

FORTRAN I/O is based on records. When a direct file is opened in a FORTRAN program, the record length of the records must be given. The record length is used by the FORTRAN I/O system to make the file look as if it is made up of records of the given length.

In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary AIX Operating System file system files. A read or write request on such a file reads or writes bytes until a specified condition is met, and is not restricted to a single record.

The requirements on sequential unformatted files make it unlikely that they will be read or written by any means except FORTRAN I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The FORTRAN I/O system breaks sequential formatted files into records while reading by using each new-line character as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system treats the record as being extended by blanks. On output, the I/O system writes a new-line character at the end of each record.

It is also possible for programs to write new-lines characters for themselves. This is an error, but the only effect is that the single record that appears to be written is treated as more than one record when being read or backspaced over.

The INed[1] editor may insert tab characters in a file being edited. The insertion of tab characters can be a problem in data files used as input to a FORTRAN program. You can use the AIX **untab** command to remove tab characters from a file.

---

1 INed is a registered trademark of INTERACTIVE Systems Corporation.

Portability Considerations

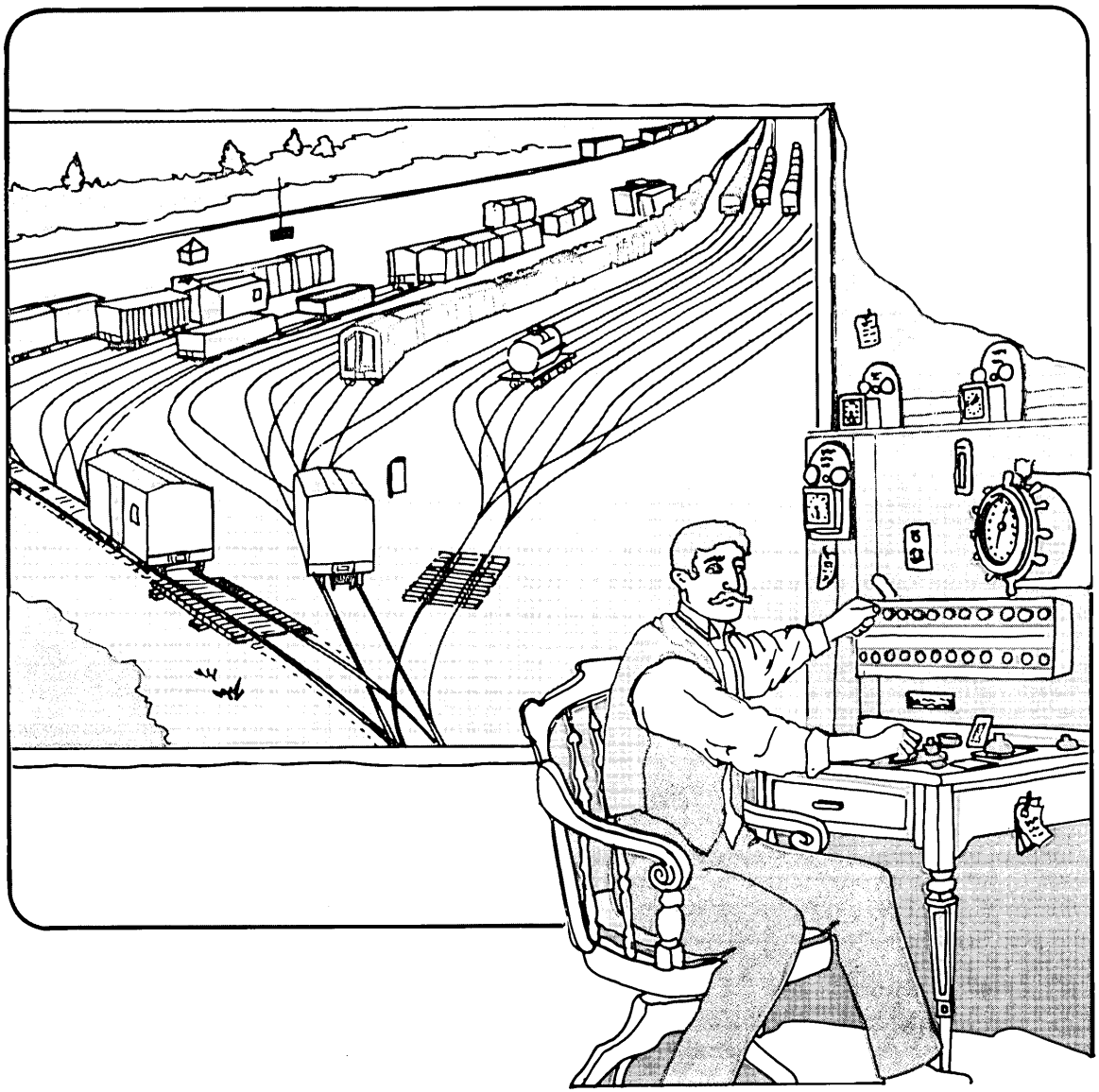
The IBM RT PC FORTRAN 77 I/O system uses the facilities of the standard C Language I/O library, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace.

Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect.

Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form that enables the program to reopen it. The AIX Operating System implementation attempts to determine the full path name.

Therefore there are two routines that depend on facilities of the operating system to provide these two services.

Chapter 2. Functions and Subroutines



CONTENTS

About This Chapter

This chapter contains a directory of the functions and subroutines that come as part of the IBM RT PC FORTRAN 77 libraries. The purpose and format of each built-in function and subroutine is given, and explanatory remarks are included.

An alphabetical listing of the functions and subroutines is included at the end of this chapter for quick reference use.

Generic and Specific Names

Some functions can take several different types of arguments. For example, the **abs** function takes a single argument that can be of type integer, real, double-precision, complex, or double complex.

The name of this type of function is considered a ***generic name***. A function with a generic name returns a value having the same data type as its argument or arguments (except for functions performing type conversion, nearest integer, and absolute value with a complex argument).

Some functions can take only one type of argument. The name of this type of function is considered a ***specific name***. The **iabs** function, for example, performs the same operation as the **abs** function. But the **iabs** function can take only an integer data type as its argument and returns only an integer value.

An ***intrinsic function*** is any pre-defined function included in the FORTRAN libraries. The functions discussed in this chapter are intrinsic to this implementation of FORTRAN 77. A function name used in an **intrinsic** statement must be the specific or generic name of an intrinsic function.

Only a specific intrinsic function name can be used as an actual argument when the corresponding dummy argument is an intrinsic function.

Function and Subroutine Directory

The following pages discuss the functions and subroutines included in the FORTRAN libraries accompanying this implementation of FORTRAN 77.

abort

Purpose

The **abort** function terminates the program that calls it.

Format

call abort ( )

Remarks

The **abort** function truncates all open files to the current position of the file pointer, closes all open files, then writes the following message to the standard error output:

```
"FORTRAN abort routine called".
```

abs

Purpose

The **abs** function and its related functions return an absolute value.

Format

integer *i1*, *i2*
real *r1*, *r2*
double precision *dp1*, *dp2*
complex *cx1*, *cx2*
double complex *dx1*, *dx2*

i2 = **iabs** (*i1*)
i2 = **abs** (*i1*)

r2 = **abs** (*r1*)

dp2 = **dabs** (*dp1*)
dp2 = **abs** (*dp1*)

cx2 = **cabs** (*cx1*)
cx2 = **abs** (*cx1*)

dx2 = **zabs** (*dx1*)
dx2 = **abs** (*dx1*)

Remarks

The **abs** function is a family of absolute value functions.

The generic form, **abs**, returns the type of its argument.

The **iabs** function returns the integer absolute value of its integer argument.

The **dabs** function returns the double-precision absolute value of its double-precision argument.

The **cabs** function returns the complex absolute value of its complex argument.

The **zabs** function returns the double-complex absolute value of its double-complex argument.

acos

Purpose

The **acos** intrinsic function and its related intrinsic function return an arccosine value.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **acos** (*r1*)

dp2 = **dacos** (*dp1*)
dp2 = **acos** (*dp1*)

Remarks

The **acos** function returns the arccosine as determined by its argument (either real or double-precision).

The **dacos** function returns the double-precision arccosine of its double-precision argument.

aimag

Purpose

The **aimag** function and its related function return the imaginary part of a complex argument.

Format

real *r1*
complex *cx1*
double precision *dp1*
double complex *dx1*

r1 = **aimag** (*cx1*)

dp1 = **dimag** (*dx1*)

Remarks

The **aimag** function returns the imaginary part of its single-precision complex argument.

The **dimag** function returns the double-precision imaginary part of its double-complex argument.

aint

The **aint** intrinsic function and its related intrinsic function return an integer part.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **aint** (*r1*)

dp2 = **dint** (*dp1*)
dp2 = **aint** (*dp1*)

Remarks

The generic form, **aint**, returns either a real value or a double-precision value depending on the type of its argument.

The **aint** function returns the truncated value of its real argument as a real value.

The **dint** function returns the truncated value of its double-precision argument as a double-precision value.

asin

Purpose

The **asin** intrinsic function and its related intrinsic function return an arcsine value.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **asin** (*r1*)

dp2 = **dasin** (*dp1*)
dp2 = **asin** (*dp1*)

Remarks

The generic form, **asin**, returns either a real value or a double-precision value depending on the type of its argument.

The **asin** function returns the real arcsine of its real argument.

The **dasin** function returns the double-precision arcsine of its double-precision argument.

atan

Purpose

The **atan** intrinsic function and its related intrinsic function return an arctangent value.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **atan** (*r1*)

dp2 = **datan** (*dp1*)
dp2 = **atan** (*dp1*)

Remarks

The generic form, **atan**, returns either a real value or a double-precision value depending on the type of its argument.

The **atan** function returns the real arctangent of its real argument.

The **datan** function returns the double-precision arctangent of its double-precision argument.

atan2

Purpose

The **atan2** intrinsic function and its related intrinsic function return an arctangent value of a quotient.

Format

real *r1*, *r2*, *r3*
double precision *dp1*, *dp2*, *dp3*

r3 = **atan2** (*r1*, *r2*)

dp3 = **datan2** (*dp1*, *dp2*)
dp3 = **atan2** (*dp1*, *dp2*)

Remarks

The generic form, **atan**, returns either a real value or a double-precision value depending on the type of its argument.

The **atan2** function returns the arctangent of *dp1* divided by *dp2* as a real value.

The **datan2** function returns the double-precision arctangent of its double-precision arguments.

bool

Purpose

The ***boolean intrinsic functions*** return the result of binary operations on their arguments.

Format

integer *i1*, *i2*, *i3*
real *r1*, *r2*, *r3*
double precision *dp1*, *dp2*, *dp3*

i3 = **and** (*i1*, *i2*)

r3 = **or** (*r1*, *r2*)

i2 = **xor** (*i1*, *r1*)

i2 = **not** (*i1*)

i3 = **lshift** (*i1*, *i2*)

i3 = **rshift** (*i1*, *i2*)

Remarks

The generic functions **and**, **or** and **xor** return the value of the binary operations on their arguments.

The **not** function is a unary function returning the one's complement of its argument.

The **lshift** and **rshift** functions return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument.

The boolean functions are generic; they are defined for all data types as arguments and return values. Where required, the compiler generates appropriate type conversions.

Note: Although defined for all data types, use of boolean functions on any but integer data produces unexpected results.

conjg

Purpose

The **conjg** intrinsic function and its related intrinsic function return a complex conjugate.

Format

complex *cx1*, *cx2*
double complex *dx1*, *dx2*

cx2 = **conjg** (*cx1*)

dx2 = **dconjg** (*dx1*)

Remarks

The **conjg** function returns the complex conjugate of its complex argument.

The **dconjg** function returns the double-complex conjugate of its double-complex argument.

cos

Purpose

The **cos** intrinsic function and its related intrinsic functions return a cosine.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*
complex *cx1*, *cx2*

r2 = **cos** (*r1*)

dp2 = **dcos** (*dp1*)
dp2 = **cos** (*dp1*)

cx2 = **ccos** (*cx1*)
cx2 = **cos** (*cx1*)

Remarks

The generic form, **cos**, returns the real, double-precision, or complex cosine depending on its type of argument.

The **cos** function returns the real cosine of its real argument.

The **dcos** function returns the double-precision cosine of its double-precision argument.

The **ccos** function returns the complex cosine of its complex argument.

cosh

Purpose

The **cosh** intrinsic function and its related intrinsic function return a hyperbolic cosine.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **cosh** (*r1*)

dp2 = **dcosh** (*dp1*)
dp2 = **cosh** (*dp1*)

Remarks

The generic form, **cosh**, returns either the real or the double-precision hyperbolic cosine depending on the type of its argument.

The **cosh** function returns the real hyperbolic cosine of its real argument.

The **dcosh** function returns the double-precision hyperbolic cosine of its double-precision argument.

dim

Purpose

The **dim** function and its related functions return the positive difference of two arguments.

Format

integer *i1*, *i2*, *i3*
real *r1*, *r2*, *r3*
double precision *dp1*, *dp2*, *dp3*

i3 = **idim** (*i1*, *i2*)
i3 = **dim** (*i1*, *i2*)

r3 = **dim** (*r1*, *r2*)

dp3 = **ddim** (*dp1*, *dp2*)
dp3 = **dim** (*dp1*, *dp2*)

Remarks

The **dim**, **ddim**, and **idim** functions return the positive difference between two arguments if *arg1* > *arg2*. If *arg1* <= *arg2*, the function returns 0.

dprod

Purpose

The **dprod** function returns a double-precision product.

Format

double precision *dp1*, *dp2*, *dp3*

dp3 = **dprod** (*dp1*, *dp2*)

Remarks

The **dprod** function returns the double-precision product of its two double-precision arguments.

exp

Purpose

The **exp** intrinsic function and its related intrinsic functions return an exponential value.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*
complex *cx1*, *cx2*

r2 = **exp** (*r1*)

dp2 = **dexp** (*dp1*)
dp2 = **exp** (*dp1*)

cx2 = **cexp** (*cx1*)
cx2 = **exp** (*cx1*)

Remarks

The generic function, **exp**, returns the real exponential function e^x of its real argument. The **exp** function becomes a call to the **dexp** function or the **cexp** function, depending on the type of its argument.

The **dexp** function returns the double-precision exponential function of its double-precision argument.

The **cexp** function returns the complex exponential function of its complex argument.

ftype

Purpose

The ***ftype functions*** perform explicit type conversions.

Format

integer *i1*, *i2*
real *r1*, *r2*
double precision *dp1*, *dp2*
complex *cx1*
double complex *dx1*
character*1 *ch1*

i1 = **int** (*r1*)
i1 = **int** (*dp1*)
i1 = **int** (*cx1*)
i1 = **int** (*dx1*)
i1 = **ifix** (*r1*)
i1 = **idint** (*dp1*)

r1 = **real** (*i1*)
r1 = **real** (*dp1*)
r1 = **real** (*cx1*)
r1 = **real** (*dx1*)
r1 = **float** (*i1*)
r1 = **sngl** (*dp1*)

dp1 = **dble** (*i1*)
dp1 = **dble** (*r1*)
dp1 = **dble** (*cx1*)
dp1 = **dble** (*dx1*)

cx1 = **cmplx** (*i1* [, *i2*])
cx1 = **cmplx** (*r1* [, *r2*])
cx1 = **cmplx** (*dp1* [, *dp2*])
cx1 = **cmplx** (*dx1*)

dx1 = **dcmplx** (*i1*[, *i2*])
dx1 = **dcmplx** (*r1*[, *r2*])
dx1 = **dcmplx** (*dp1*[, *dp2*])
dx1 = **dcmplx** (*cx1*)

i1 = **ichar** (*ch1*)
ch1 = **char** (*i1*)

Remarks

The **ftype** functions convert data from one data type to another. The functions associated with the **ftype** function are:

- **int**
- **ifix**
- **idint**
- **real**
- **float**
- **sngl**
- **dble**
- **cmplx**
- **dcmplx**
- **ichar**
- **char**

The **int** function converts its argument to integer form. The argument can be real, double-precision, complex, or double complex. If the argument is real or double-precision, **int** returns an integer with the largest value that does not exceed the value of the argument and with the same sign as the argument. For complex types, this rule is applied to the real part.

If converting an argument to integer exceeds the maximum **integer*4** value, the leftmost bit position (sign bit) is set to 1 and the remaining 31 bits are set to 0.

All conversions to **integer*2** variables are first converted internally to **integer*4**. The rightmost (least significant) 16 bits of the **integer*4** variable are then moved to the **integer*2** variable.

The **ifix** function converts only real arguments.

The **idint** function converts only double-precision arguments.

The **real** function converts its argument to real form. The argument can be integer, double-precision, complex, or double complex. If the argument is double-precision or double complex, as much precision as possible is kept. If the argument is one of the complex types, the real part is returned.

The **float** function converts only integer arguments.

The **sngl** function converts only double-precision arguments.

The **dble** function converts its argument to double-precision form. The argument can be integer, real, complex, or double complex. If the argument is a complex type, the real part is returned.

The **cmplx** function converts its argument or arguments to complex form. The arguments can be integer, real, double-precision, or double complex.

The **cmplx** function can have either one or two arguments. If there is only one argument, it is taken as the real part of the complex type, and an imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second part as the imaginary part.

The **dcmplx** function converts its argument or arguments to double complex form. The arguments can be integer, real, double-precision, or complex.

The **dcmplx** function can have either one or two arguments. If there is only one argument, it is taken as the real part of the complex type, and an imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part, and the second part as the imaginary part.

The **ichar** function converts its argument from a character to an integer depending on the character's position in the collating sequence.

The **char** function returns a character from the processor collating sequence. The character returned is dependent on the argument supplied. For example, if the argument is 5, the character in the fifth position in the processor collating sequence is returned. The **char** function uses the rightmost 8 bits of the integer argument. The remaining bits are ignored.

getarg

Purpose

The **getarg** subroutine returns a command line argument of the current process.

Format

character*N ch1*
integer *i1*

call getarg (*i1***, *ch1*)**

Remarks

The *i1* parameter specifies the argument to return. If *i1* is equal to 0, the program name is returned from the command line.

For example, if a program is invoked by the following:

```
prog arg1 arg2 arg3
```

a **getarg** subroutine of the form:

```
getarg(2, c)
```

returns the string `arg2` in the character variable `c`

For information on finding out the number of arguments entered on the command line, see "iargc" on page 2-26.

getenv

Purpose

The **getenv** subroutine returns the character string value of the specified environment variable contained in the **.profile** file of the current directory.

Format

character*N ch1*

call getenv ('*environment_name*'**,** *ch1***)**

Remarks

The **getenv** subroutine returns the character string value of the environment variable specified by the first parameter in the character variable of the second parameter. If no such environment variable exists, blanks are returned.

The following is an example of the **getenv** subroutine:

```
call getenv ('HOME', ch1)
```

iargc

Purpose

The **iargc** function returns the number of arguments entered on the command line.

Format

integer *i1*

i1 = **iargc ( )**

Remarks

The **iargc** function returns an integer. The integer is the number of arguments following the program name that have been entered on the command line.

For information on getting command line arguments, see "getarg" on page 2-24.

index

Purpose

The **index** function returns the location of a substring in a specified character string.

Format

character*N1 *ch1*
character*N2 *ch2*
integer *i1*

i1 = **index** (*ch1*, *ch2*)

Remarks

The **index** function returns the location of the substring specified by the *ch2* parameter in the string specified by the *ch1* parameter.

The returned location is the position at which substring *ch2* starts. If substring *ch2* is not present in the string *ch1* or if the length of *ch2* is greater than the length of *ch1*, a value of 0 is returned.

The entire declared length of *ch2* is used in the search. If the string assigned to *ch2* is shorter than the declared length of *ch2*, blanks fill the remaining character positions and are used as part of the search substring.

For example, the **index** function in the following section of code finds no matching substring and returns the value 0:

```
character*20 char1, char2
integer i
char1 = ''one two three four''
char2 = ''three''
i = index(char1,char2)
```

The string `char1` does not contain a match for the substring `char2`, which contains the character string `three` followed by the 15 blank characters required to fill the variable's declared length of 20 characters.

If the defined character length for `char2` in the example is changed to 5 characters (`character*5`), a match is found in `char1` beginning at character position 9 and the function in the example returns the value 9.

len

Purpose

The **len** function returns the length of a specified character string.

Format

character*N ch1*
integer *i1*

i1 = **len** (*ch1*)

Remarks

The **len** function returns the length of the string pointed to by the *ch1* parameter.

log

Purpose

The **log** intrinsic function and its related intrinsic functions return a natural logarithm.

Format

real *r1, r2*
double precision *dp1, dp2*
complex *cx1, cx2*

r2 = **alog** (*r1*)
r2 = **log** (*r1*)

dp2 = **dlog** (*dp1*)
dp2 = **log** (*dp1*)

cx2 = **clog** (*cx1*)
cx2 = **log** (*cx1*)

Remarks

The generic form, **log**, returns the real, double-precision, or complex logarithm depending on the type of its argument.

The **alog** function returns the real natural logarithm of its real argument.

The **dlog** function returns the double-precision natural logarithm of its double-precision argument.

The **clog** function returns the complex logarithm of its complex argument.

log10

Purpose

The **log10** intrinsic function and its related intrinsic functions return a common logarithm.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **alog10** (*r1*)
r2 = **log10** (*r1*)

dp2 = **dlog10** (*dp1*)
dp2 = **log10** (*dp1*)

Remarks

The generic form, **log10**, returns the real or double-precision common logarithm depending on its type of argument.

The **alog10** function returns the real common logarithm of its real argument.

The **dlog10** function returns the double-precision common logarithm of its double-precision argument.

max

Purpose

The **max** function and its related functions return a maximum value.

Format

integer *i1*, *i2*, *i3*, *i4*
real *r1*, *r2*, *r3*, *r4*
double precision *dp1*, *dp2*, *dp3*

i4 = **max** (*i1*, *i2*, *i3*)

r3 = **max** (*r1*, *r2*)
dp1 = **max** (*r1*, *r2*, *r3*)

i3 = **max0** (*i1*, *i2*)
r1 = **amax0** (*i1*, *i2*, *i3*)

i1 = **max1** (*r1*, *r2*)
r4 = **amax1** (*r1*, *r2*, *r3*)

dp3 = **dmax1** (*dp1*, *dp2*)

Remarks

The **max** function is a family of maximum-value functions.

The generic form, **max**, returns the type of its argument.

The **max0** function returns the integer form of the maximum value of its integer arguments.

The **amax0** function returns the real form of its integer arguments.

The **max1** function returns the integer form of its real arguments.

The **amax1** function returns the real form of its real arguments.

The **dmax1** function returns the double-precision form of its double-precision arguments.

mclock

Purpose

The **mclock** function returns time accounting information about the current process and its child processes.

Format

integer *i1*

i1 = **mclock ( )**

Remarks

The returned value is the sum of the current process's user time and the user and system times of all child processes. The unit of measure is one-sixtieth (1/60) of a second.

min

Purpose

The **min** function and its related functions return a minimum value.

Format

integer *i1*, *i2*, *i3*, *i4*
real *r1*, *r2*, *r3*, *r4*
double precision *dp1*, *dp2*, *dp3*

i4 = **min** (*i1*, *i2*, *i3*)
r3 = **min** (*r1*, *r2*)
dp1 = **min** (*r1*, *r2*, *r3*)

i3 = **min0** (*i1*, *i2*)
r1 = **amin0** (*i1*, *i2*, *i3*)

i1 = **min1** (*r1*, *r2*)
r4 = **amin1** (*r1*, *r2*, *r3*)

dp3 = **dmin1** (*dp1*, *dp2*)

Remarks

The **min** function is a family of minimum-value functions.

The generic form, **min**, returns the type of its argument.

The **min0** function returns the integer form of the minimum value of its integer arguments.

The **amin0** function returns the real form of its integer arguments.

The **min1** function returns the integer form of its real arguments.

The **amin1** function returns the real form of its real arguments.

The **dmin1** function returns the double-precision form of its double-precision arguments.

mod

Purpose

The **mod** intrinsic function and its related intrinsic functions return the remainder resulting from the division of the first parameter by the second parameter.

Format

integer *i1*, *i2*, *i3*
real *r1*, *r2*, *r3*
double precision *dp1*, *dp2*, *dp3*

i3 = **mod** (*i1*, *i2*)

r3 = **amod** (*r1*, *r2*)
r3 = **mod** (*r1*, *r2*)

dp3 = **dmod** (*dp1*, *dp2*)
dp3 = **mod** (*dp1*, *dp2*)

Remarks

The generic form, **mod**, returns the remainder in integer, real, or double-precision form depending on the type of its argument. The first parameter is divided by the second parameter.

The **amod** function returns the real remainder of the integer division of the two parameters.

The **dmod** function returns the double-precision whole number remainder of the integer division of the two parameters.

rand

Purpose

The **rand** and **irand** functions generate uniform random numbers. The **srand** subroutine provides the seed value for the random number generator.

Format

integer *i1*
real *r1*

call srand(*i1***)**
i1 = **irand( )**

call srand(*r1***)**
r1 = **rand( )**

Remarks

The **rand** and **irand** functions generate uniform random numbers. The **rand** and **irand** functions return the seed value each time it completes its operation.

The **srand** subroutine is used to provide the seed number specified by the argument for the random number generator.

The **irand** function returns a positive integer number greater than 0 and less than or equal to 32768.

The **rand** function returns a positive real number greater than 0 and less than 1.0.

round

Purpose

The **nint** function and its related functions return the nearest whole number value.

Format

integer *i1*
real *r1*, *r2*
double precision *dp1*, *dp2*

i1 = **nint** (*r1*)
i1 = **nint** (*dp1*)

r2 = **anint** (*r1*)
dp2 = **anint** (*dp1*)

dp2 = **dnint** (*dp1*)
i1 = **idnint** (*dp1*)

Remarks

The **anint** function returns the nearest whole real number to its real parameter.

The **dnint** function returns the nearest whole double-precision number to its double-precision parameter.

The **nint** function returns the nearest integer to its real or double-precision parameter.

The **idnint** function returns the nearest integer to its double-precision parameter.

sign

Purpose

The **sign** intrinsic function and its related intrinsic functions return the absolute value of first parameter with the sign of the second parameter.

Format

integer *i1*, *i2*, *i3*
real *r1*, *r2*, *r3*
double precision *dp1*, *dp2*, *dp3*

i3 = **isign** (*i1*, *i2*)
i3 = **sign** (*i1*, *i2*)

r3 = **sign** (*r1*, *r2*)

dp3 = **dsign** (*dp1*, *dp2*)
dp3 = **sign** (*dp1*, *dp2*)

Remarks

The generic form, **sign**, returns either an integer, real, or double-precision value depending on the type of its parameters.

The **isign** function returns the integer absolute value of the first parameter with the sign of the second parameter.

The **dsign** function returns the double-precision absolute value of the first parameter with the sign of the second parameter.

signal

Purpose

The **signal** subroutine allows a process to specify a function to be invoked upon receipt of a specific signal.

Format

integer *i1*
external *intfnc*

call signal (*i1*, *intfnc*)

Remarks

The *i1* parameter specifies the signal.

The *intfnc* parameter specifies the user-defined procedure to be invoked upon receipt of the specified signal.

The following signal values can be assigned to *i*:

Integer Value	Meaning
1	(sighup) System hangup
2	(sigint) System interrupt
3	(sigquit) Quit
4	(sigill) Illegal instruction (not reset when caught)
5	(sigtrap) Trace trap (not reset when caught)
6	(sigiot) IOT instruction
7	(sigemt) EMT instruction
8	(sigfpe) Floating-point exception
9	(sigfill) Kill (cannot be caught or ignored)

Figure 2-1 (Part 1 of 2). System Signals

Integer Value	Meaning
10	(sigbus) Bus error
11	(sigsegv) Segmentation violation
12	(sigsys) Invalid argument to system call
13	(sigpipe) Write on a pipe with no one to read it
14	(sigalrm) Alarm clock
15	(sigterm) Software termination signal
16	(sigusr1) User-defined signal 1
17	(sigusr2) User-defined signal 2
18	(sigcld) Death of a child process
19	(sigpwr) Power failure
27	(sigioint) I/O intervention required
28	(siggrant) Monitor mode granted
29	(sigretract) Monitor mode retracted
30	(sigsound) Sound acknowledge
31	(sigmsg) Data pending

Figure 2-1 (Part 2 of 2). System Signals

For more information on signals, see the **signal** subroutine in the *IBM RT PC AIX Operating System Commands Reference.*

sin

Purpose

The **sin** intrinsic function and its related intrinsic functions return the sine of the parameter.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*
complex *cx1*, *cx2*

r2 = **sin** (*r1*)

dp2 = **dsin** (*dp1*)
dp2 = **sin** (*dp1*)

cx2 = **csin** (*cx1*)
cx2 = **sin** (*cx1*)

Remarks

The generic form, **sin**, returns the real, double-precision, or complex sine depending on the type of its argument.

The **dsin** function returns the double-precision sine of its double-precision argument.

The **csin** function returns the complex sine of its complex argument.

sinh

Purpose

The **sinh** and **dsinh** intrinsic functions return the hyperbolic sine of the specified parameter.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **sinh** (*r1*)

dp2 = **dsinh** (*dp1*)
dp2 = **sinh** (*dp1*)

Remarks

The generic form, **sinh**, returns either the real or double-precision hyperbolic sine depending on the type of its parameter.

The **dsinh** function returns the double-precision hyperbolic sine of its double-precision parameter.

sqrt

Purpose

The **sqrt** intrinsic function and its related intrinsic functions return the square root of the specified argument.

Format

real *r1, r2*
double precision *dp1, dp2*
complex *cx1, cx2*

r2 = **sqrt** (*r1*)

dp2 = **dsqrt** (*dp1*)
dp2 = **sqrt** (*dp1*)

cx2 = **csqrt** (*cx1*)
cx2 = **sqrt** (*cx1*)

Remarks

The generic form, **sqrt**, returns the real, double-precision, or complex square root depending on the type of its argument.

The **dsqrt** function returns the double-precision square root of its double-precision argument.

The **csqrt** function returns the complex square root of its complex argument.

system

Purpose

The **system** subroutine allows you to issue an operating system command from FORTRAN.

Format

character*N ch1*

call system (*ch1***)**

Remarks

The **system** subroutine passes the *ch1* parameter to the operating system as input. The operating system accepts and executes the command specified by the *ch1* parameter as if it were typed from a terminal.

The current process pauses until the command is completed and control is returned from the operating system.

tan

Purpose

The **tan** and **dtan** intrinsic functions return the tangent of the specified argument.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **tan** (*r1*)

dp2 = **dtan** (*dp1*)
dp2 = **tan** (*dp1*)

Remarks

The generic form, **tan**, returns either the real or double-precision tangent of its argument. If the argument is a real value, the returned value is a real value. If the argument is a double-precision value, the returned value is a double-precision value.

The **dtan** function returns the double-precision tangent of its double-precision argument.

tanh

Purpose

The **tanh** and **dtanh** intrinsic functions return the hyperbolic tangent of the specified argument.

Format

real *r1*, *r2*
double precision *dp1*, *dp2*

r2 = **tanh** (*r1*)

dp2 = **dtanh** (*dp1*)
dp2 = **tanh** (*dp1*)

Remarks

The generic form, **tanh**, returns either the real or double-precision hyperbolic tangent of its argument. If the argument is a real value, the returned value is a real value. If the argument is a double-precision value, the returned value is a double-precision value.

The **dtanh** function returns the double-precision hyperbolic tangent of its double-precision argument.

Character String Comparison Functions

Purpose

The functions in this section compare two character strings and return a logical value. The character strings are compared according to their locations in the collating sequence for the supported character set.

Format

character*N* *ch1*, *ch2*
logical *l1*, *l2*, *l3*, *l4*

l1 = **lge** (*ch1*, *ch2*)

l2 = **lgt** (*ch1*, *ch2*)

l3 = **lle** (*ch1*, *ch2*)

l4 = **llt** (*ch1*, *ch2*)

Remarks

The **lge** function returns the value **.true.** if the two character strings are equal or if *ch1* follows *ch2* in the collating sequence. Otherwise, the function returns the value **.false.**.

The **lgt** function returns the value **.true.** if *ch1* follows *ch2* in the collating sequence. Otherwise, the function returns the value **.false.**.

The **lle** function returns the value **.true.** if the two character strings are equal or if *ch1* comes before *ch2* in the collating sequence. Otherwise, the function returns the value **.false.**.

The **llt** function returns the value **.true.** if *ch1* comes before *ch2* in the collating sequence. Otherwise, the function returns the value **.false.**.

If the character strings are of unequal length, the shorter string is padded with blanks to the length of the longer string for comparison purposes.

The character set and collating sequence supported on the IBM RT PC are shown in Appendix B, "ASCII Character Codes."

Bit Field Manipulation Functions and Subroutine

Purpose

The functions and the subroutine in this section manipulate bit fields.

Format

integer *i1*, *i2*, *i3*, *i4*, *i5*, *len*
logical *l1*

i3 = **ior** (*i1*, *i2*)

i3 = **iand** (*i1*, *i2*)

i3 = **ieor** (*i1*, *i2*)

i3 = **ishft** (*i1*, *i2*)

i4 = **ishftc** (*i1*, *i2*, *i3*)

i4 = **ibits** (*i1*, *i2*, *i3*)

l1 = **btest** (*i1*, *i2*)

i3 = **ibset** (*i1*, *i2*)

i3 = **ibclr** (*i1*, *i2*)

call mvbits(*i1*, *i2*, *i3*, *i4*, *i5*)

Remarks

The functions **ior**, **iand**, and **ieor** return the same results as the functions **and**, **or**, and **xor** defined in "bool" on page 2-14.

In the **ishft** and **ishftc** functions, *i1* specifies the integer to be shifted and *i2* specifies the shift count. *i2* > 0 indicates a left shift, *i2* = 0 indicates no shift, and *i2* < 0 indicates a right shift.

In the **ishft** function, zeros are shifted in. In the **ishftc** function, the rightmost *i3* bits are shifted circularly *i2* bits. If *i2* is greater than the machine word size, **ishftc** does not shift.

Bit fields are numbered from right to left. The rightmost bit is zero. The length of the *i3* field must be greater than zero.

The **ibits** function extracts a subfield of *i3* bits from *i1*, starting with bit position *i2* and extending left for *i3* bits. The resulting field is right-justified in the target variable, and the remaining bits in the target variable are set to zero.

The **btest** function tests the *i2*th bit of argument *i1*. The value of the function is **.true.** if the bit equals 1 and **.false.** if the bit equals 0.

The **ibset** function produces the value of *i1* with the *i2*th bit set to 1.

The **ibclr** function produces the value of *i1* with the *i2*th bit set to 0.

The **mvbits** subroutine moves *i3* bits, beginning at position *i2* of argument *i1*, to position *i5* of argument *i4*.

Floating-Point Status Subroutines

Purpose

The following two subroutines set and retrieve the status of floating-point operations.

Format

```
include '/usr/include/fpdc.h'
include '/usr/include/fpdt.h'

call fpgets(fpstat)

call fpsets(fpstat)
```

Remarks

The included file **fpdc.h** contains the data declarations (specification statements) for the two subroutines. The included file **fpdt.h** contains the data initializations (**data** statements) for the two subroutines.

The **fpgets** subroutine retrieves the floating-point process status and stores the results in a **logical** array called **fpstat**.

The **fpsets** subroutine sets the floating-point process status equal to the logical array **fpstat**.

The array **fpstat**, included by the file **fpdc.h**, contains logical values that can be set to enable or disable system checking for various floating-point errors.

The floating-point error checks are enabled when specified array elements in **fpstat** are set to **.true.**, as shown in Figure 2-2 on page 2-48.2:

Array Element	Error Check Enabled When .true.
fpstat(fpeall)	Floating-point error interrupts
fpstat(fpeyes)	Floating-point error detected
fpstat(fpinva)	Invalid floating-point number detected
fpstat(fpainv)	Invalid number error interrupt
fpstat(fpdivz)	Divide-by-zero detected
fpstat(fpadiv)	Divide-by-zero error interrupt
fpstat(fpover)	Overflow detected
fpstat(fpaove)	Overflow error interrupt
fpstat(fpundr)	Underflow detected
fpstat(fpaund)	Underflow error interrupt
fpstat(fpinxe)	Inexact result detected
fpstat(fpaine)	Inexact result error interrupt

Figure 2-2. fpstat Array Elements and Enabled Error Checks When .true.

A value of **.false.** for an element of **fpstat** disables the associated error interrupt, if any. All values are initialized to **.false.** at the beginning of a process.

The array element **fpstat(fpeall)** must be set to **.true.** to enable any floating-point error interrupts. If **fpstat(fpeall)** is **.false.**, no error interrupts are generated.

The floating-point process status is cumulative. It is changed only by setting specified elements in the array **fpstat** and calling **fpsets**.

The signal function **sigfpe** must be enabled to process floating-point errors. If error interrupts are enabled for a process and a signal catcher is not defined for that process, the process terminates. For more information on **sigfpe**, see "signal" on page 2-38.

Alphabetical List of Functions and Subroutines

Notation use:

- An * (asterisk) following a name indicates a subroutine. Subroutines must be invoked by a **call** statement.

Name	Use
abort*	Terminates the program that calls it.
abs	Returns an absolute value with the type of its argument.
acos	Returns the arccosine with the type of its real or double-precision argument.
aimag	Returns the imaginary part of its single-precision complex argument.
aint	Returns the truncated value of its real or double-precision value, with the type of the argument.
alog	Returns a real natural logarithm of its real argument.
alog10	Returns the real common logarithm of its real argument.
amax0	Returns the real form of the maximum value of its integer arguments.
amax1	Returns the real form of the maximum value of its real arguments.
amin0	Returns the real form of the minimum value of its integer arguments.
amin1	Returns the real form of the minimum value of its real arguments.
amod	Returns the real remainder of integer division of two real arguments.
and	Returns the value of the binary *and* operation on its two arguments.
anint	Returns the nearest whole number to its real or double-precision argument.
asin	Returns the real or double-precision arcsine, with the type of its argument.
atan	Returns the real or double-precision arctangent, with the type of its argument.
atan2	Returns the arctangent with the type of its real or double-precision arguments.

Figure 2-2 (Part 1 of 7). Alphabetical List of Functions and Subroutines

Name	Use
btest	Tests a specified bit in an integer, returns the value **.true.** if the bit equals 1 and **.false.** if the bit equals 0.
cabs	Returns the complex absolute value of its complex argument.
ccos	Returns the complex cosine of its complex argument.
cexp	Returns the complex exponential function of its complex argument.
char	Converts an argument from an integer to a character, based on the character's position in the collating sequence.
clog	Returns the complex natural logarithm of its complex argument.
cmplx	Converts one or two arguments of integer, real, double-precision, or double-complex type to complex type.
conjg	Returns the complex conjugate of its complex argument.
cos	Returns the cosine with the type of its real, double-precision, or complex argument.
cosh	Returns the hyperbolic cosine, with the type of its real or double-precision argument.
csin	Returns the complex sine of its complex argument.
csqrt	Returns the complex square root of its complex argument
dabs	Returns the double-precision absolute value of its double-precision argument.
dacos	Returns the double-precision arccosine of its double-precision argument.
dasin	Returns the double-precision arcsine of its double-precision argument.
datan	Returns the double-precision arctangent of its double-precision argument.
datan2	Returns the double-precision arctangent of its double-precision arguments.
dble	Converts an argument of integer, real, complex, or double-complex to double-precision type.
dcmplx	Converts one or two arguments of integer, real, double-precision, or complex type to double-complex type.
dconjg	Returns the double-complex conjugate of its double-complex argument.

Figure 2-2 (Part 2 of 7). Alphabetical List of Functions and Subroutines

Name	Use
dcos	Returns the double-precision cosine of its double-precision argument.
dcosh	Returns the double-precision hyperbolic cosine of its double-precision argument.
ddim	Returns the positive difference of its two double-precision arguments.
dexp	Returns the double-precision exponential function of its double-precision argument.
dim	Returns the positive difference of its two integer, real or double-precision arguments.
dimag	Returns the double-precision imaginary part of its double-complex argument.
dint	Returns the truncated value of its double-precision argument as a double-precision value.
dlog	Returns the double-precision natural logarithm of its double-precision argument.
dlog10	Returns the double-precision common logarithm of its double-precision argument.
dmax1	Returns the double-precision form of the maximum value of its double-precision arguments.
dmin1	Returns the double-precision form of the minimum value of its double-precision arguments.
dmod	Returns the double-precision whole number remainder of integer division of two double-precision arguments.
dnint	Returns the nearest whole double-precision number to its double-precision argument.
dprod	Returns the double-precision product of its two double-precision arguments.
dsign	Returns the absolute value of its first double-precision argument with the sign of its second double-precision argument.
dsin	Returns the double-precision sine of its double-precision argument.
dsinh	Returns the double-precision hyperbolic sine of its double-precision argument.

Figure 2-3 (Part 3 of 7). Alphabetical List of Functions and Subroutines

Name	Use
dsqrt	Returns the double-precision square root of its double-precision argument.
dtan	Returns the double-precision tangent of its double-precision argument.
dtanh	Returns the double-precision hyperbolic tangent of its double-precision argument.
exp	Returns the exponential function with the type of its real, double-precision, or complex argument.
float	Converts an argument of integer type to real type.
fpgets*	Retrieves the current floating-point process status and stores the results in a logical array.
fpsets*	Sets the floating-point process status equal to a logical array.
getarg*	Returns a specified command line argument of the current process.
getenv*	Returns the character string value of the specified environment variable.
iabs	Returns the integer absolute value of its integer argument.
iand	Returns the value of the binary *and* operation on its arguments.
iargc	Returns the integer number of arguments entered on the command line.
ibclr	Returns the value of the specified integer with a specified bit set to 0.
ibits	Returns a right-justified subfield of bits from an integer.
ibset	Returns the value of the specified integer with a specified bit set to 1.
ichar	Converts an argument from a character to an integer, based on the character's position in the collating sequence.
idim	Returns the positive difference of its two integer arguments.
idint	Converts an argument of double-precision type to integer type.
idnint	Returns the nearest integer to its double-precision argument.
ieor	Returns the value of the binary *xor* operation on its arguments.
ifix	Converts an argument of real type to integer type.
index	Returns an integer representing the location of a substring in a specified string.
int	Converts an argument of real, double-precision, complex, or double-complex type to integer type.

Figure 2-3 (Part 4 of 7). Alphabetical List of Functions and Subroutines

Name	Use
ior	Returns the value of the binary *or* operation on its arguments.
irand	Generates uniform random numbers; returns a positive integer number greater than 0 and less than or equal to 32768.
ishft	Shifts bits in a specified integer left or right, with zeros shifted in.
ishftc	Shifts the specified rightmost bits in an integer circularly left or right.
isign	Returns the absolute integer value of the first argument with the sign of the second argument.
len	Returns an integer representing the length of a specified character string.
lge	Returns the value **.true.** if its two character-string arguments are equal or if the first string follows the second in the collating sequence for the supported character set. Otherwise **.false.** is returned.
lgt	Returns the value **.true.** if its first character string argument follows its second character string argument in the collating sequence for the supported character set. Otherwise **.false.** is returned.
lle	Returns the value **.true.** if its two character string arguments are equal or if the first string comes before the second in the collating sequence for the supported character set. Otherwise **.false.** is returned.
llt	Returns the value **.true.** if its first character string argument comes before its second character string argument in the collating sequence for the supported character set. Otherwise **.false.** is returned.
log	Returns a natural logarithm with the type of its real, double-precision, or complex argument.
log10	Returns a common logarithm with the type of its real or double-precision argument.
lshift	Returns the value of the first argument shifted left the number of times specified by the second integer argument.
max	Returns the maximum value of its arguments, with the type of its integer, real, or double-precision arguments.
max0	Returns the integer form of the maximum value of its integer arguments.
max1	Returns the integer form of the maximum value of its real arguments.

Figure 2-3 (Part 5 of 7). Alphabetical List of Functions and Subroutines

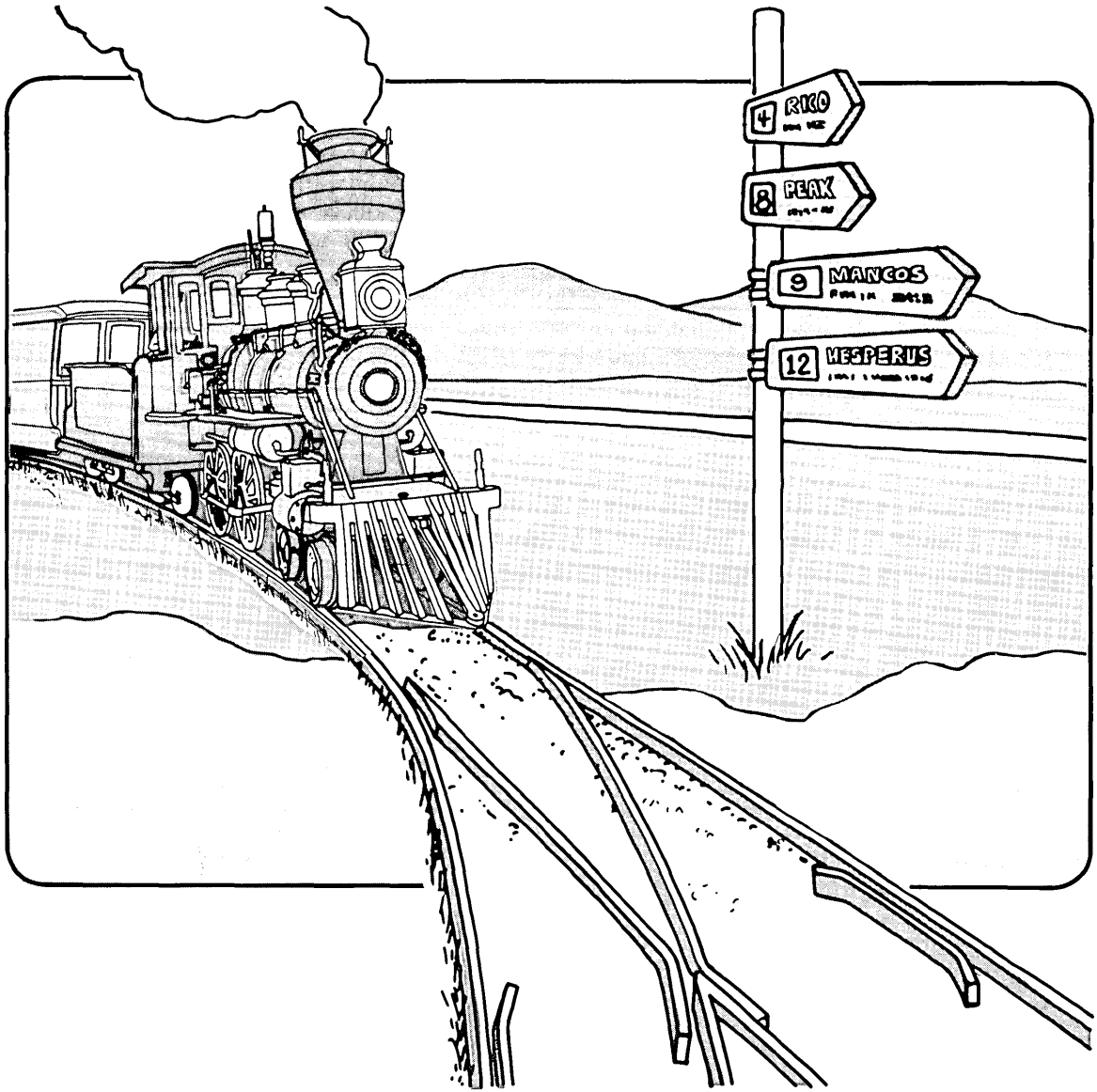
Name	Use
mclock	Returns the sum of the current process's user time and the user and system times of all child processes in sixtieths (1/60) of a second.
min	Returns the minimum value of its arguments, with the type of its integer, real, or double-precision arguments.
min0	Returns the integer form of the minimum value of its integer arguments.
min1	Returns the integer form of the minimum value of its real arguments.
mod	Returns the remainder of division of two integer, real, or double-precision arguments, with the type of the arguments.
mvbits*	Moves specified bits in one integer argument to specified bit locations in a second integer argument.
nint	Returns the nearest integer to its real or double-precision argument.
not	Returns the one's complement of its argument (the value of the binary *not* operation).
or	Returns the value of the binary *or* operation on its two arguments.
rand	Generates uniform random numbers; returns a positive real number greater than 0 and less than 1.0.
real	Converts an argument of integer, double-precision, complex, or double-complex type to real type.
rshift	Returns the value of the first argument shifted right the number of times specified by the second integer argument.
sign	Returns the absolute value of its first integer, real, or double-precision argument with the sign of its second integer, real or double-precision argument.
signal*	Specifies a function to be invoked upon receipt of a specific system signal.
sin	Returns the sine of its real, double-precision, or complex arguments.
sinh	Returns the real or double-precision hyperbolic sine of its real or double-precision argument.
sngl	Converts an argument of double-precision type to real type.
sqrt	Returns the real, double-precision, or complex square root of its real, double-precision, or complex argument.

Figure 2-3 (Part 6 of 7). Alphabetical List of Functions and Subroutines

Name	Use
srand*	Provides an integer or real seed number for the random number generator.
system*	Issues an AIX Operating System command from FORTRAN.
tan	Returns the real or double-precision tangent of its real or double-precision argument.
tanh	Returns the real or double-precision hyperbolic tangent of its real or double-precision argument.
xor	Returns the value of the binary *xor* operation on its two arguments.
zabs	Returns the double-complex absolute value of its double-complex argument.

Figure 2-3 (Part 7 of 7). Alphabetical List of Functions and Subroutines

Chapter 3. Compiling, Linking, Debugging, and Running a Program



CONTENTS

About This Chapter

This chapter describes:

- What you need to compile IBM RT PC FORTRAN 77 programs
- IBM RT PC FORTRAN 77 program names
- How to compile and link FORTRAN programs
- IBM RT PC FORTRAN 77 compiler options and their purposes
- The internal compilation process
- How to list compiler messages in a file
- How to produce listings with additional information, such as symbol table entries, external references, and full memory listings
- The AIX Operating System tool you can use to debug IBM RT PC FORTRAN 77 programs.

What You Need

Two things must happen before you can compile a FORTRAN program on your system:

1. The files that make up the FORTRAN compiler must be installed on your system. These files contain the procedures that compile and generate object code from your FORTRAN programs. The files are:

   - `/usr/bin/f77` — The FORTRAN compilation driver
   - `/usr/bin/ratfor` — The Rational FORTRAN (Ratfor) preprocessor
   - `/usr/bin/efl` — The Extended FORTRAN Language (EFL) preprocessor
   - `/usr/lib/f77pass1` — The FORTRAN compiler, pass 1
   - `/usr/lib/f77pass2` — The FORTRAN compiler, pass 2
   - `/usr/lib/f77passq` — The FORTRAN intermediate code optimizer
   - `/usr/lib/libF77.a` — The FORTRAN intrinsic function library
   - `/usr/lib/libI77.a` — The FORTRAN runtime I/O library.

   Instructions for installing the files that make up the FORTRAN compiler can be found in Appendix A, "Installing the IBM RT PC FORTRAN 77 Licensed Program." If a person or department maintains the computer system, contact them to verify that the FORTRAN compiler is installed.

2. Your user ID must have execute permission for all of these files. If a person or department maintains the system, contact them to verify that your user ID has the necessary permissions.

FORTRAN Program Names

A file containing FORTRAN code can have any file name valid to the AIX Operating System. The file name must end with an extension that the FORTRAN compiler can recognize.

If the FORTRAN compiler is sent a file with a name ending with the extension:

```
.f
```

the compiler recognizes the file as a FORTRAN source program and compiles the file. The resulting object program is stored in the current directory in a file with the same file name as the source file and an extension of **.o**. For instance, if you send a source file named:

```
test.f
```

to the FORTRAN compiler, the resulting object file is named:

```
test.o
```

If you send the FORTRAN compiler a file with the extension:

```
.r
```

the compiler recognizes the file as a Ratfor source program. A Ratfor source file is translated by the Ratfor preprocessor into a pure FORTRAN program. The resulting FORTRAN program is then automatically compiled by the FORTRAN compiler to produce an object file.

If you send the FORTRAN compiler a file with the extension:

```
.e
```

the file is recognized as an EFL source program. An EFL source file is translated by the EFL preprocessor into a pure FORTRAN program. The resulting FORTRAN program is then automatically compiled by the FORTRAN compiler to produce an object file.

The FORTRAN compiler also recognizes source files with **.c** and **.s** extensions. Source files with **.c** extensions are assumed to be C Language source programs and are sent to the C Language compiler for compilation. Source files with **.s** extensions are assumed to be Assembler Language source programs and are sent to the assembler for object code generation.

Also, a file name with the extension **.o** is recognized by the compiler as a pre-compiled object file. Object files are linked to the compiler output.

Compiling and Linking a FORTRAN Program

You use the `f77` and `ff77` commands to compile FORTRAN programs. Each command recognizes FORTRAN, Ratfor, EFL, C Language, and Assembler Language source files by their file extensions and sends each source file to the appropriate processor or compiler for compilation. The compiler also recognizes object files for linking by their **.o** extensions.

To compile a program, you type:

`f77` *options filenames*

and press **Enter**.

The *options* can be any of the FORTRAN compiler command line options discussed in "FORTRAN Compiler Options" on page 3-7.

The *filenames* can be the names of one or more FORTRAN, Ratfor, EFL, C Language, or Assembler Language source files, or pre-compiled object files for linking to the compiler output.

The `ff77` compile command automatically generates the **f** compiler option and performs the library linking needed for maximum optimization of object code.

The general form of the `ff77` command is:

`ff77` *options filenames*

For example, the command:

```
ff77  myfile.f
```

is equivalent to:

```
f77  -f  myfile.f
```

For more information on linking, see the **ld** command in the *AIX Operating System Commands Reference*. For more information on the **f** compiler option, see "The f Option" on page 3-8.

FORTRAN Compiler Options

The following sections describe *options* recognized by the FORTRAN compiler.

Note: Any option that you type as part of an `f77` command must begin with a – (dash).

The c Option

The **c** option:

- Suppresses link editing
- Produces an object file for each source file specified by *filenames*.

The C Option

The **C** option directs the compiler to generate code for the checking of subscript ranges at runtime.

The E Option

The **E** option directs the compiler to use the remaining characters in the argument list as EFL flag arguments when processing a file with the **.e** extension. For more information on EFL flag arguments, see the discussion of options for EFL flag arguments in Chapter 8, "EFL — Extended FORTRAN Language."

The f Option

The **f** option optimizes the resulting object code for processing directly to the floating-point accelerator (direct FPA). The resulting object file may run faster than an object file compiled without this option.

Note: The **f** option optimizes code that is run with the floating-point accelerator (FPA) card only. The FPA card must be installed on a machine on which you execute the resulting program.

The **f** option automatically links the optimized built-in math libraries to the object file that results from the compilation of a FORTRAN source program.

A special FORTRAN compile command, `ff77`, automatically generates the **f** option and performs the library linking needed for maximum optimization of object code. For information on the `ff77` command, see "Compiling and Linking a FORTRAN Program" on page 3-6.

For more information on linking, see the **ld** command in the *AIX Operating System Commands Reference.*

The F Option

The **F** option directs the compiler to:

- Apply the Ratfor and EFL preprocessors to relevant files
- Put the output of the preprocessors in files with the extension **.f**. (No object files are created.)

The g Option

The **g** option generates additional information that is needed in order to use the symbolic debugger, **sdb**, on the executable code.

The m Option

The **m** option instructs the compiler to send specified Ratfor and EFL source files to the M4 preprocessor before sending them to the Ratfor or EFL preprocessors.

The N Option

This option allows you to change the maximum size of one of the internal tables used by the compiler. It requires a *tableid* to identify a table plus a *value* to be used as the maximum size for that table. The compiler issues a diagnostic if one of the internal tables overflows during compilation.

The following list shows each *tableid* and its default *value*:

c The maximum depth of loops or **if** statements. The default is 20.

l The maximum number of computed **goto** statement numbers. The default is 125.

n The maximum number of variable or common block names. The default is 401.

p The maximum number of constants and internally generated code labels. The default is 600.

q The maximum number of equivalences. The default is 150.

s The maximum number of statement numbers. The default is 201.

x The maximum number of common block, function, and subroutine names. The default is 401.

You can set more than one table per compilation by specifying the **N** option for each table that you wish to set. For example, to set the maximum depth of loops to 40 and the maximum number of variable names to 800, you specify the **N** option as follows:

```
-Nc40 -Nn800
```

The o output Option

The **o** *output* option names the final output file *output* instead of the default name *a.out*.

The O Option

The **O** option optimizes the object code produced by the FORTRAN compiler.

The onetrip Option

The **onetrip** option ensures that **do** loops in the compiled program are executed at least once, if reached. In standard FORTRAN 77, **do** loops are not performed if the upper limit is smaller than the lower limit.

The **1** option performs the same function as this option.

The p Option

The **p** option prepares object files for profiling. For more information on profiling, see the *AIX Operating System Commands Reference* book.

The R Option

The **R** option directs the compiler to use the remaining characters in the argument list as Ratfor flag arguments when processing a file with the **.r** extension. For more information on Ratfor flag arguments, see the discussion of options for Ratfor flag arguments in Chapter 7, "Ratfor — The Rational FORTRAN Preprocessor."

The S Option

The **S** option directs the Assembler Language output of the compiler into files with file name extensions of **.s**. No **.o** files are created.

The u Option

The **u** option defines the default type of a variable as **undefined**. The option turns off the implicit typing of variables based upon the first letter of the variable name. When this option is used, all variables must be explicitly declared.

The U Option

The compiler normally interprets uppercase and lowercase letters, such as *a* and *A*, as identical letters. The **U** option causes the compiler to treat uppercase and lowercase letters as different letters, therefore making the compiler case-sensitive.

Warning:
The compiler expects keywords such as **if** and **else** to be entered in lowercase letters. The compiler normally interprets all letters in FORTRAN programs, both lowercase and uppercase, as lowercase letters. Therefore the compiler can compile FORTRAN programs containing keywords entered in uppercase letters. However, if you use the **U** option to compile a program containing keywords entered in uppercase letters, the compiler will not be able to properly interpret the keywords.

The v Option

The **v** option enables the verbose option of the compilation process. The verbose option displays the name and parameters of each program called during compilation.

The w Option

The **w** option suppresses warning messages produced by the compiler.

The w66 Option

The **w66** option suppresses warning messages related to FORTRAN 66 compatibility.

The x Option

The compiler allows names to have up to 127 characters, all of which are significant. The **x** option limits names to 6 characters. If the **x** option is used and the compiler encounters a name with more than 6 characters, an error message is issued.

The y Option

The **y** option specifies the rounding mode for floating-point constants. The general form of the **y** option is:

y*mode*

The parameter *mode* can have the following values:

Setting	Meaning
d	No floating-point constant rounding.
m	Round toward negative infinity.
n	Round to nearest. This is the default.
p	Round toward positive infinity.
z	Round toward 0 (zero).

Figure 3-1. Floating-Point Rounding Modes

The 1 Option

The **1** option performs the same function as the **onetrip** option. For more information, see "The onetrip Option" on page 3-10.

The 66 Option

The **66** option suppresses extensions that enhance FORTRAN 66 compatibility.

The Internal Compilation Process

Compilation begins after you type the `f77` command line and press **Enter**.

The following list discusses the flow of a file through the FORTRAN compiler:

1. The extension at the end of the file name is checked. If the extension is:
   - **.f**, the file is passed to the next step in the cycle
   - **.r**, the file is sent through the Ratfor preprocessor for translation into FORTRAN
   - **.e**, the file is sent through the EFL preprocessor for translation into FORTRAN
   - **.c**, the file is sent to the C Language compiler for object code generation
   - **.s**, the file is sent to the Assembler Language assembler for object code generation.
2. The file is sent through the FORTRAN parser for syntax and structure checking.
3. If errors in syntax or structure are found, the compilation stops. Error messages indicate lines of code that must be corrected before the program will compile properly.
4. If no errors in syntax or structure are found, the file is passed to the object code generator.
5. The object code generator outputs an object file with an **.o** extension.
6. The object file is passed to the linkage editor. The linkage editor adds to the object file the object code for:
   - FORTRAN library utilities used by the program
   - FORTRAN intrinsic functions used by the program, such as **sqrt**.

   Note: The linkage editor automatically appends built-in FORTRAN utilities and intrinsic functions to the object file. You can specifically link previously-compiled subroutines and functions to a file you want to compile by typing the names of the object code for those subroutines and functions after the name of the file being compiled by the `f77` command.

   For instance, to link the compiled object code `sub.o` to a program named `main.f` that you want to compile, you can enter the following command:

   ```
   f77 main.f sub.o
   ```

   For more information on the linkage editor, see the **ld** command (linkage editor command) in the *AIX Operating System Commands Reference* book.

7. The linkage editor outputs the object file with its appended pre-compiled object code into an executable file named **a.out**. The executable file can be renamed to create a command file with a unique name.

Listing Compiler Messages in a File

Messages produced by the FORTRAN compiler are sent to the device defined as the standard output device in your user profile. The usual setting for the standard output device is the screen. So messages produced by the FORTRAN compiler are normally sent to your screen.

You can ***re-direct*** messages generated by the FORTRAN compiler to a file. The following section discusses the method.

Note: Messages generated by the FORTRAN compiler are intended to be self-explanatory and therefore are not documented. You may also encounter messages generated by the linkage editor or the assembler.

Re-Directing Messages to a File

The FORTRAN compiler, by default, sends compile-time messages to standard output, which is usually the screen. You can use a AIX Operating System technique of re-directing standard output to channel compile-time message output to a file instead of to the standard output device.

For example, to re-direct messages generated by the compilation of the file `test.f` into a file named `testmsgs`, you can use the following command:

```
f77 text.f 2> testmsgs
```

The first part of the command, `f77 text.f`, invokes the FORTRAN compiler with the source file `text.f` as input. It is a regular syntactic form of the `f77` command.

The second part of the command, `2> testmsgs`, directs the output to the file `testmsgs`. The AIX Operating System recognizes `2` as a name for the standard output device. Therefore output sent to the standard output device is re-directed into `testmsgs`.

For more information on standard output devices, see *Installing and Customizing the AIX Operating System.* For more information on re-directing output, see the *AIX Operating System Commands Reference* book.

Informational Listings

You can use the AIX Operating System **dump** and **od** commands to create special listings of your FORTRAN object files.

For example, you can create a list of the symbol table entries by using the **t** option of the **dump** command. The **dump** command also has options that allow you to produce a listing in other data formats such as octal.

You can run an object file through the **od** filter command to produce an output file in octal, ASCII, hexadecimal, or decimal format.

For more information on the use of these commands, see *AIX Operating System Commands Reference.*

Debugging a FORTRAN Program

The AIX Operating System contains a debugging utility called ***sdb***, the ***symbolic debugger***. **sdb** works with FORTRAN and C Language programs.

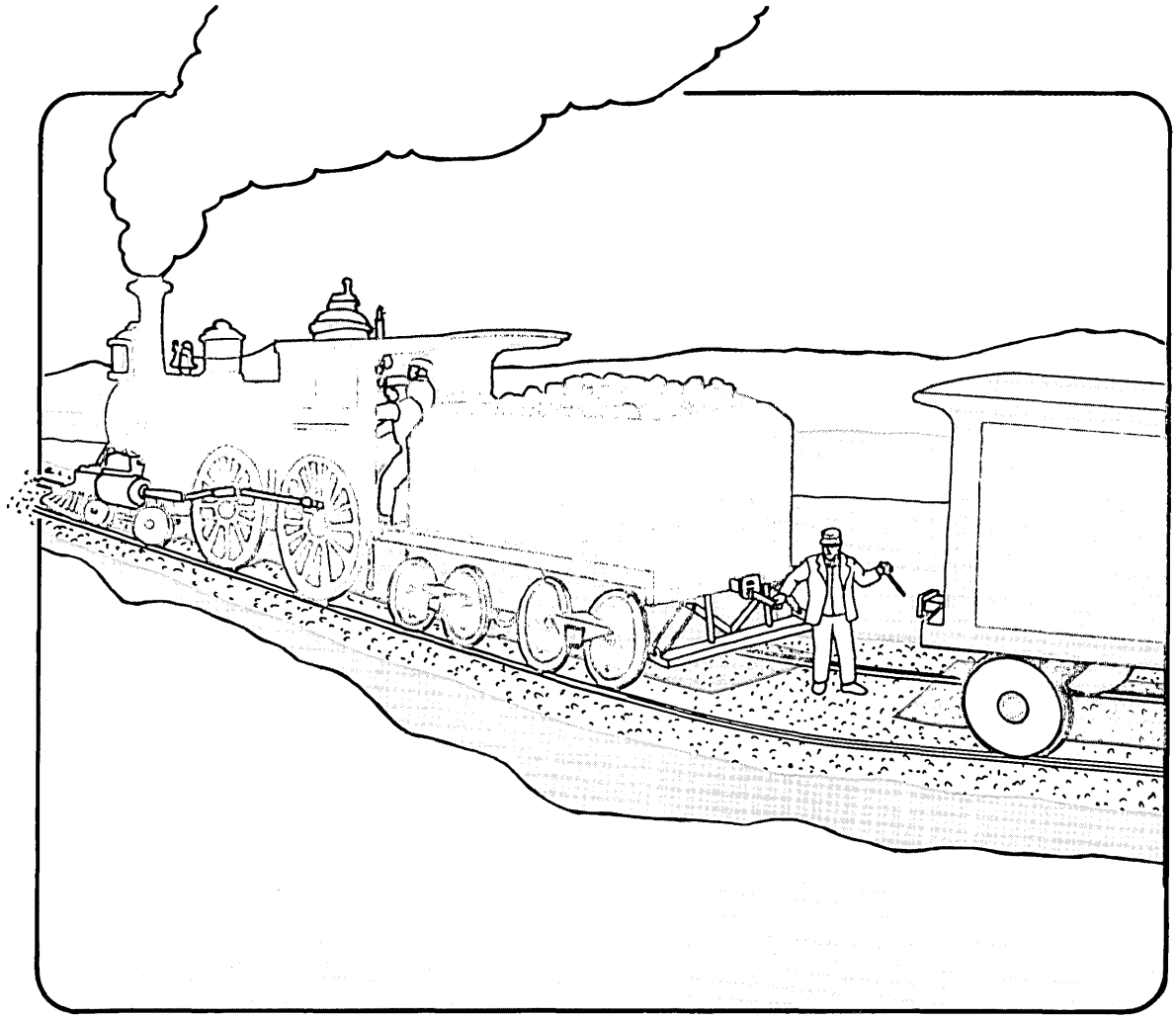
Using **sdb**, you can:

- Trap and examine the core image of a program that terminates abnormally
- Place breakpoints at selected statements to enable examination of program progress at certain intervals
- Step through program progress line-by-line.

sdb allows interaction with a program at the source language level. It can print lines of source code on the screen, allowing you to examine source code interactively.

For detailed information on how to use **sdb**, see *AIX Operating System Programming Tools and Interfaces.*

Chapter 4. Linking C With FORTRAN



CONTENTS

About This Chapter

IBM RT PC C Language programs can be linked with IBM RT PC FORTRAN 77 programs. This chapter describes:

- How to link a C Language program with a FORTRAN program
- Conventions for the passing of data between FORTRAN and C Language programs
- Source code examples and output of a FORTRAN program that calls a C Language routine.

How to Link a C Language Program to a FORTRAN Program

You can call a C Language routine from a FORTRAN program. C Language routines return a value. The C Language routine must be declared **external** in the FORTRAN program.

You link a C Language program to a FORTRAN program by including the name of the C Language source file or object module as part of the FORTRAN compile command (`f77`) you use to compile the FORTRAN source file.

If a file sent to the FORTRAN compiler ends with a **.c** extension, the file is automatically sent to the C Language compiler for compilation into an object module before linking.

For example, to link the C Language program `funct.c` to a FORTRAN program named `main.f`, you can enter the following compile command:

```
f77 main.f funct.c
```

The `funct.c` file is assumed to be a C Language source file. It is sent to the C Language compiler before the two programs are linked.

As a second example, assume that the C Language program `funct.c` was compiled into an object module named `funct.o`. To link the C Language object module to the FORTRAN program named `main.f`, you can enter the following compile command:

```
f77 main.f funct.o
```

The `funct.o` file is assumed to be an object module because of its **.o** extension. It is not sent for compilation, but is linked with the FORTRAN program.

The `f77` compile command uses the EIX Operating System linkage editor to output the FORTRAN object file with its appended C Language object code into a single executable file named **a.out**. For more information on the linkage editor, see the **ld** command (linkage editor command) in the *EIX Operating System Commands Reference* book.

Inter-Procedure Interface

The following sections discuss the conventions for procedure names, data representation, return values, and argument lists that must be considered when writing C Language procedures that call or are called by FORTRAN programs

Procedure Names

The name of a common block or a FORTRAN procedure has an underscore appended to it by the compiler to distinguish it from a C Language procedure or external variable with the same user-assigned name. FORTRAN library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Therefore the FORTRAN compiler will append an _ (underscore) to the name of the C Language routine declared **external**. In order for the linkage editor to recognize the C Language routine as the one referenced, an underscore must be appended to the name of the C routine.

Data Representations

The following is a table of corresponding FORTRAN and C declarations:

FORTRAN	C
integer*2 x	short int x;
integer x	long int x;
logical*4 x	long int x;
real x	float x;
double precision x	double x;
complex x	struct {float r, i;} x;
double complex x	struct {double dr, di;} x;
character*6 x	char x[6];

Figure 4-1. Corresponding FORTRAN and C Declarations

By the rules of FORTRAN, **integer**, **logical**, and **real** data occupy the same amount of memory.

Return Values

A function of type **integer**, **logical**, **real**, or **double precision** correctly returns the corresponding type, whether the called function is written in FORTRAN or in C. A **complex** or **double complex** function is passed an additional initial argument that points to the place where the return value is to be stored. Thus the declaration and invocation in FORTRAN:

```
complex  f, g
external f
o
o
o
g = f(A, B, C)
```

with A, B, and C as **real*4** values, would be received in a C Language routine as:

```
f_(rtnptr, Aptr, Bptr, Cptr)
float  *Aptr, *Bptr, *Cptr;
struct complex{
       float realdata;
       float imajdata;
};
struct complex *rtnptr;
```

The FORTRAN program expects the return value to be in the area pointed to by `rtnptr`.

A character-valued function is passed two extra initial arguments: a data return address and a return length. Thus the function declaration and invocation in FORTRAN:

```
character*15  g, h
external g
o
o
o
h = g(A, B, C)
```

where A, B, and C are **real*4** values, would be received in a C Language function as:

```
g_(rtnptr, length, aptr, bptr, cptr)
float *aptr, *bptr, *cptr;
char *rtnptr;
int length;   /* the expected length of the return value */
```

The FORTRAN program expects the return value to be in the area pointed to by `rtnptr`.

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

The statement:

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

is treated exactly as if it were the computed **goto**:

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

Argument Lists

FORTRAN arguments are passed by address. In addition, for every argument that is of type **character**, an argument giving the length of the value is passed. The string lengths are **long int** quantities passed by value.

The order of arguments is:

1. Extra arguments for **complex** and **character** functions (not generated for subroutine calls)
2. Address for each argument
3. A **long int** for each **character** argument.

Thus the call in the following procedure:

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, s, b(2))
```

actually passes four arguments. The fourth argument, the length of `s`, is passed last and can be ignored.

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

Source Code Examples

The following examples show a FORTRAN program that calls a C Language routine. The source code for the FORTRAN program `callcf` is immediately followed by the source code for the C Language routine `trble_`. The output produced by a run of `callcf` appears at the end of this section.

```
c the FORTRAN program callcf
      program callcf
      integer i
      character*5 char
      logical grins
      real r
      double precision d
      complex c
      double complex dc, dc2
      external trble
      i = 1
      char = 'Joe B'
      grins = .true.
      r = 5.0
      d = .9999d+2
      c = (5.0,5.0)
      dc = (.10d+2,.10d+2)
      dc2 = (.10d+2,.10d+2)
      write(*,1009)i,char,grins,r,d,c,dc
1009     format(i5,2x,a,l5,f10.2,f10.2,2f10.2,2g10.2)
      call trble(i,char,grins,r,d,c,dc2,dc)
      stop
      end
```

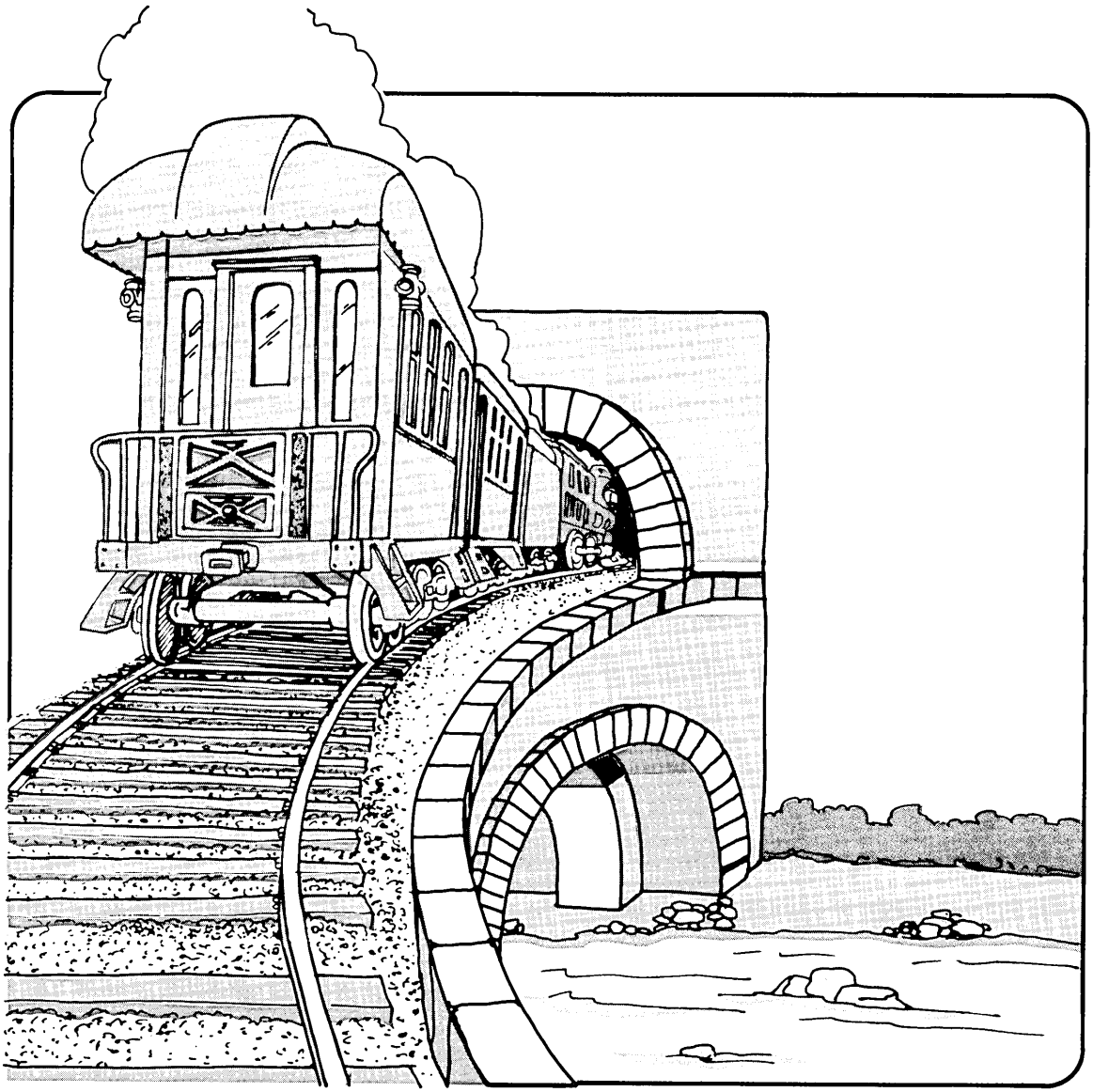
```
trble_(inti,chat,grin,rr,dr,comp,dcomp,try)
int *inti;
char *chat[5];
long *grin;
float *rr;
double *dr;
struct { float compr, compi } *comp;
struct { double dcompr, dcompi } *dcomp;
struct { double tryr, tryi } *try;
{
printf("\n\n\n");
printf(" %d = logical\n", *grin);
printf("%d = int\n", *inti);
printf("%s = char\n", chat);
printf("%f = r-real\n", *rr);
printf("%f = dr-dblprec\n", *dr);
printf(" %f comp\n ", (*comp).compr);
printf(" %f compi\n ", (*comp).compi);
printf(" %f  dcomp\n ", (*dcomp).dcompr);
printf(" %f  dcompi\n ", (*dcomp).dcompi);
printf(" %f  tryr\n ", (*try).tryr);
printf(" %10.2f  tryi \n", (*try).tryi);
}
```

Output:

```
       1  Joe B   T    5.00     99.99     5.00     5.00     10.     10.


 1 = logical
1 = int
Joe B = char
5.000000 = r-real
99.990000 = dr-dblprec
 5.000000 comp
  5.000000 compi
  10.000000  dcomp
  10.000000  dcompi
  10.000000  tryr
       10.00  tryi
```

Chapter 5. AIX Operating System Commands for FORTRAN



CONTENTS

About This Chapter

The IBM RT PC AIX Operating System contains special commands that perform certain functions on program and data files you use with FORTRAN. This chapter discusses the following IBM RT PC AIX Operating System commands that affect FORTRAN files:

- **asa**
- **fsplit**

This chapter assumes that you are familiar with the use and basic syntactic structure of commands on the IBM RT PC AIX Operating System.

asa

Purpose

The **asa** command interprets the output of FORTRAN programs that use ASA carriage control characters.

Format

asa [*files*]

Remarks

The **asa** command processes either the *files* whose names are given as arguments, or the standard input stream, if no file names are supplied.

The first character of each line is assumed to be a control character. The control characters and their meanings are:

Character	Meaning
<blank character>	Single new line before printing (blank character).
0	Double new line before printing.
1	New page before printing.
+	Overprint previous line.

Figure 5-1. Control Characters

Lines beginning with other than the defined control characters are treated as if they begin with a blank character.

The first character of a line is not printed. If any such lines appear, an appropriate diagnostic appears on the defined output for error messages.

Execution of the **asa** command causes the first line of each input file to start on a new page.

Example

To correctly view the output of FORTRAN programs which use ASA carriage control characters, you can use the **asa** command as a filter. For example, the following pipe:

```
a.out | asa > lpr
```

directs the output produced by the program `a.out`, properly formatted and paginated, to the line printer `lpr`. (For more information on filters, see the *AIX Operating System Reference*.)

FORTRAN output sent to the file `myfile` can be viewed using the following form of the **asa** command:

```
asa myfile
```

fsplit

Purpose

The **fsplit** command splits specified FORTRAN, Ratfor, or EFL source program files into several files.

Format

fsplit *options files*

Remarks

The **fsplit** command splits the specified *files* into separate files, with one procedure per file.

A procedure in this context includes the following program segments:

- Block data
- Function
- Main
- Program
- Subroutine.

You can specify one of the following *options* in the command line:

Option	Meaning
f	Input files are FORTRAN language files. This option is the default.
r	Input files are Ratfor files.
e	Input files are EFL files.
s	Strip FORTRAN input lines to 72 or fewer characters, and remove trailing blanks.

Figure 5-2. fsplit Options

A procedure recognized as *main* by **fsplit** is put in one of the following files, depending on the specified language option:

- *main.f*
- *main.r*
- *main.e*

Unnamed block data segments are put in one of the following files, depending on the specified language option:

- *blockdataN.f*
- *blockdataN.r*
- *blockdataN.e*

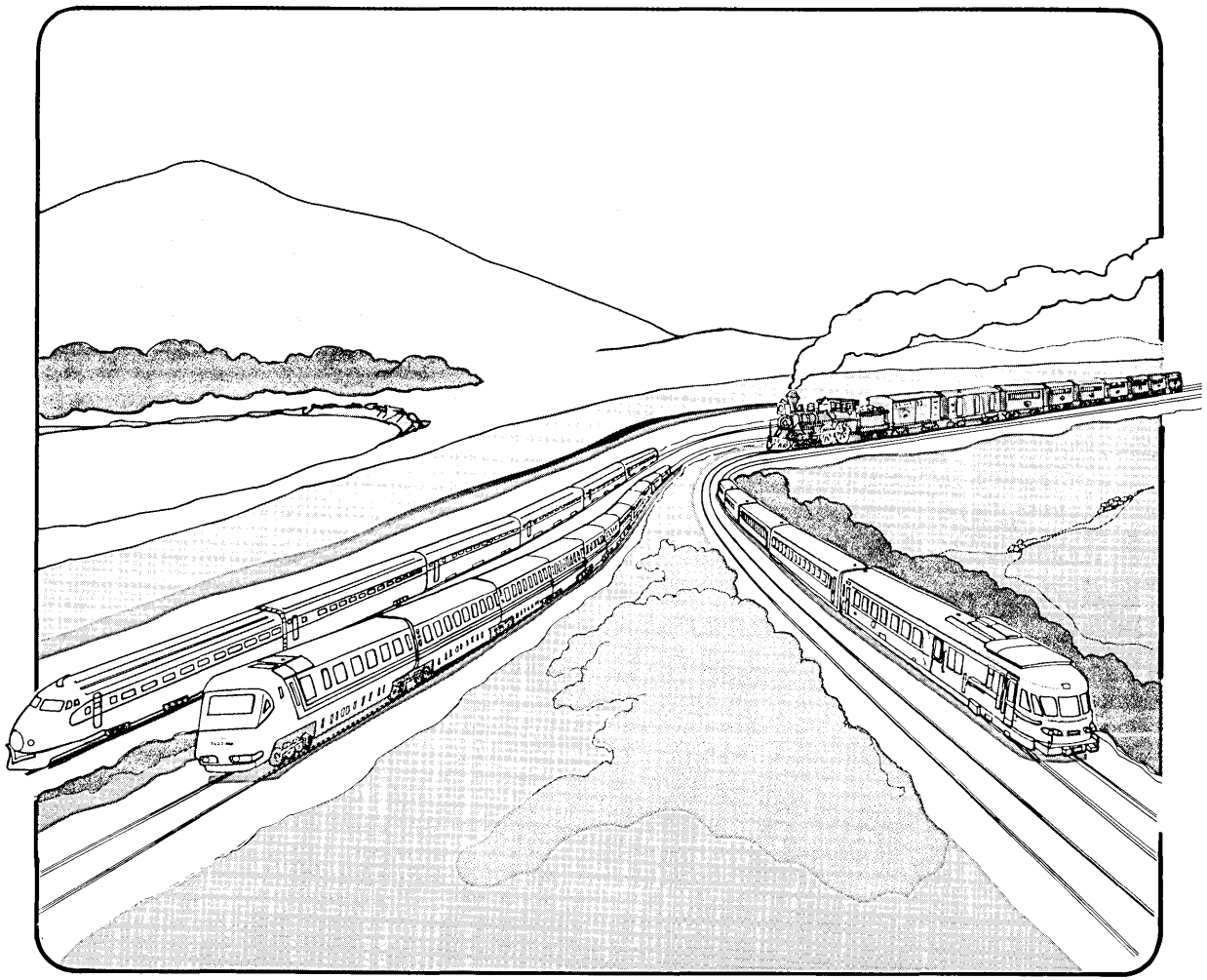
The *N* in each preceding file name example represents a unique integer value.

Other procedures *X* are put in one of the following files, depending on the specified language option for the procedure segment:

- *X.f*
- *X.r*
- *X.e*

Part 2. Ratfor and EFL — Two Preprocessors for FORTRAN

Chapter 6. Overview of Preprocessors



CONTENTS

About This Chapter

Ratfor, or Rational FORTRAN, and EFL, or Extended FORTRAN Language, are ***preprocessors*** for the FORTRAN language.

This chapter discusses:

- General definition of preprocessors
- Characteristics of the Ratfor preprocessor
- Characteristics of the EFL preprocessor
- Differences between Ratfor and EFL.

General Definition of Preprocessors

A ***preprocessor*** is basically a defined computer language. A preprocessor, like most other high-level computer languages, can be thought of as two separate parts:

- The language itself, the certain statements and structures that you use to write your source programs
- The language compiler, the interpreter that translates your source programs into machine-readable and machine-executable language.

However, a difference exists between preprocessors and high-level computer languages. Programs written in the language of a preprocessor are compiled into high-level computer language programs before being compiled into machine-readable code.

So a preprocessor is an implementation of a computer language that compiles into a high-level computer language.

IBM RT PC FORTRAN 77 comes with two different preprocessors:

- Ratfor
- EFL.

For instance, the relationship between Ratfor, FORTRAN, and machine language is illustrated in Figure 6-1 on page 6-5:

To get from a Ratfor program to machine language that your computer can understand and run, you:

1. Write the Ratfor program code.
2. Compile (preprocess) the Ratfor code with the Ratfor compiler to get corresponding FORTRAN code.
3. Compile the resulting FORTRAN code with the FORTRAN compiler to get the corresponding machine language (object code).

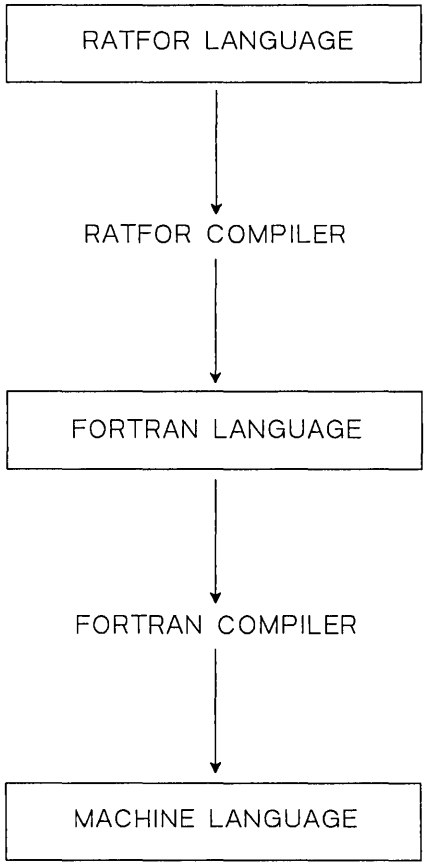


Figure 6-1. Relationship Between Ratfor, FORTRAN, and Machine Language

Characteristics of Ratfor

Ratfor's design gives you access to the full capabilities of FORTRAN in a logically structured environment. It contains statements controlling the flow of logic and structure controls — conditional branching and loops, for instance — that are not explicitly implemented in standard FORTRAN. Ratfor statements form a logical structure around specified FORTRAN statements and constructs to produce structured source code.

Ratfor statements and their functions are similar to flow-of-control statements in Algol, PL/I, Pascal, and other languages containing explicit statements that reflect the logical flow of control in a program.

Characteristics of EFL

EFL is a general purpose computer language intended to encourage portable programming. It has a uniform syntax, and data and control flow structuring.

The EFL compiler is more than a simple preprocessor. It attempts to diagnose syntax errors and to provide readable FORTRAN output. To achieve this goal, a sizable two-pass translator is implemented.

Thus the EFL language attempts to permit the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the FORTRAN environment.

The Differences Between Ratfor and EFL

There are a number of differences between Ratfor and EFL. EFL is a defined language, while Ratfor is the union of the special control structures and the language accepted by the underlying FORTRAN compiler. Ratfor running over standard FORTRAN is almost a subset of EFL.

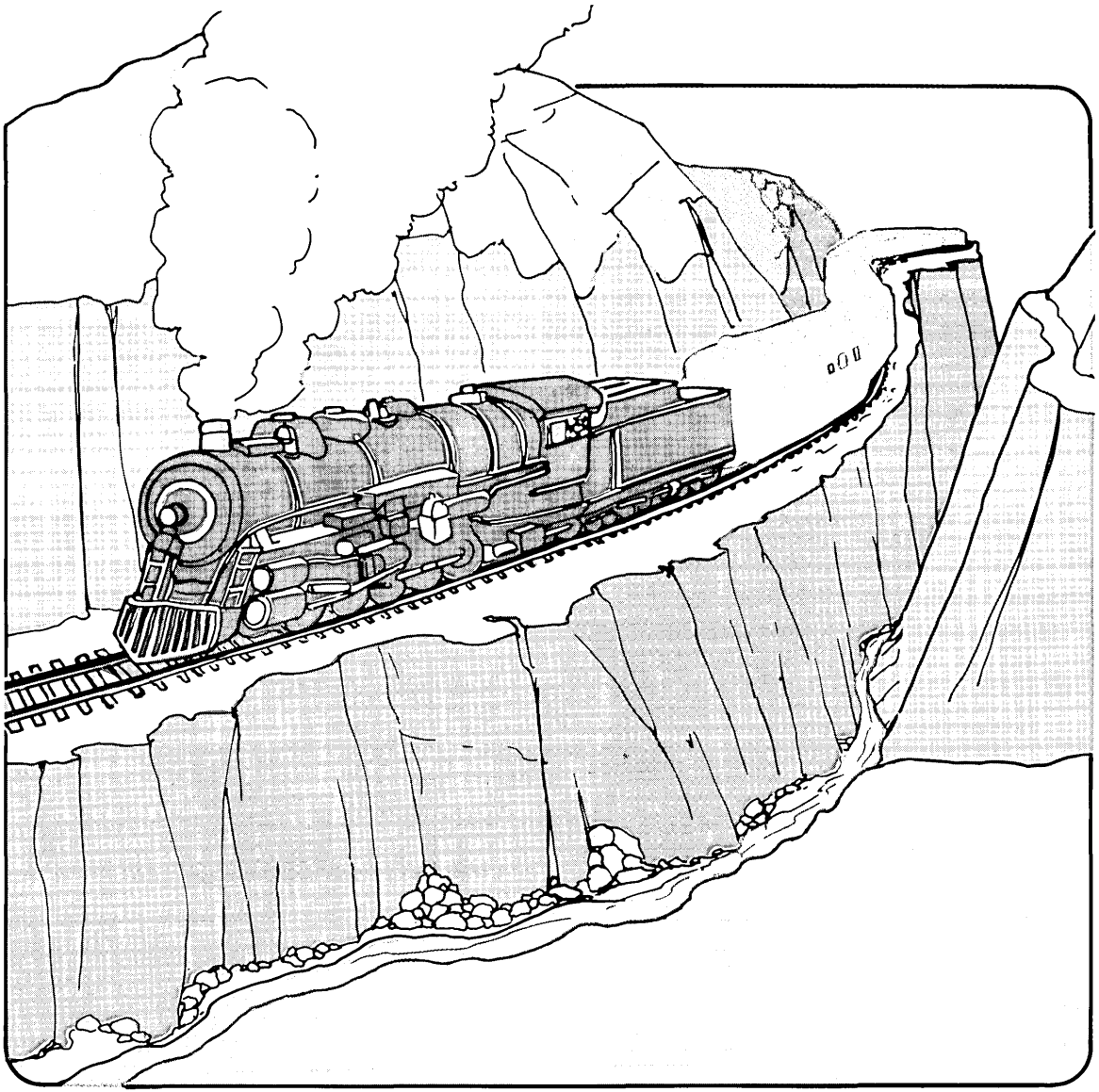
There a few incompatibilities between Ratfor and EFL. The syntax of the **for** statement is slightly different in the two languages. The three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The **initial** and **iteration** statements can be compound statements in EFL because of this change).

The input/output syntax is quite different in the two languages, and there is no **format** statement in EFL. There are no **assign** or assigned **goto** statements in EFL.

EFL permits more general forms for expressions, and provides a more uniform syntax. The FORTRAN/Ratfor restrictions on subscript and **do** expression forms are not implemented in EFL.

For more information on Ratfor, see Chapter 7, "Ratfor — The Rational FORTRAN Preprocessor." For more information on EFL, see Chapter 8, "EFL — Extended FORTRAN Language."

Chapter 7. Ratfor — The Rational FORTRAN Preprocessor



CONTENTS

About This Chapter

Ratfor, or Rational Fortran, is a preprocessor for the FORTRAN language. Ratfor source files are preprocessed into FORTRAN source code and then compiled into object code.

This chapter describes:

- The capabilities of Ratfor
- The syntactic structure of the language
- Ratfor statements
- General Ratfor conventions
- Usage considerations
- How to compile Ratfor source files.

The Capabilities of the Ratfor Preprocessor

Ratfor gives you access to the full capabilities of FORTRAN in a defined and structured form. It implements flow-of-control statements — conditional branches and loops — that are not explicitly implemented in standard FORTRAN. These statements and their functions are similiar to flow-of-control statements in Algol, PL/I, Pascal, and other languages containing statements that reflect the flow of logic in a program.

The statements and structural implementation of Ratfor include:

- A defined method of grouping statements
- **if-else** and **switch** statements for decision-making
- **while, for, do**, and **repeat-until** statements for looping
- **break** and **next** statements for controlling loop exits
- Free-form input (multiple statements/line, automatic continuation)
- Translation of conventional mathematical operators in their FORTRAN equivalents
- A **return**(*expression*) statement for functions
- A **define** statement for defining symbolic parameters
- An **include** statement for including external source files.

The relationship between Ratfor, FORTRAN, and machine language is illustrated in Figure 7-1 on page 7-5.

A Ratfor program is translated into machine language that your computer can run in the following way:

1. The Ratfor code is processed with the Ratfor preprocessor to get corresponding FORTRAN code.
2. The resulting FORTRAN code is compiled by the FORTRAN compiler to produce the corresponding machine language (object) code.

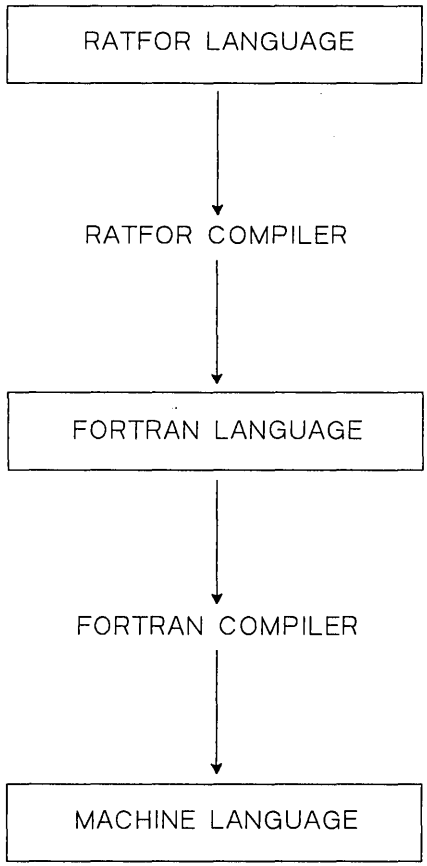


Figure 7-1. Relationship Between Ratfor, FORTRAN, and Machine Language

The Syntactic Structure of Ratfor

The following sections discuss the syntactic structure and elements of Ratfor.

Mathematical Operators

Standard mathematical operators, like >, are not understood by FORTRAN. The following figure shows Ratfor mathematical operators and their FORTRAN equivalents:

RATFOR	FORTRAN
>	.gt.
<	.lt.
> =	.ge.
< =	.le.
= =	.eq.
!=	.ne.
!	.not.
&	.and.
|	.or.

Figure 7-2. Ratfor and FORTRAN Mathematical Operators

Character Strings

Characters typed between a set of double quotation marks are treated as a single character string. Ratfor converts the string into the right number of **H**'s.

Free-form Input

Statements can be placed anywhere on a line. Long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until** statements.

Several statements can appear on one line if they are separated by semicolons.

Lines ending with any of the following characters:

```
=  +  -  *  ,  |     &  (  _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of a statement.

An all-numeric field preceding a statement is interpreted as a FORTRAN label and is placed in columns 1 through 5 upon output. Thus the following Ratfor example:

```
write(6, 100); 100 format('hello')
```

is converted into:

```
      write(6, 100)
100   format(5hhello)
```

Comment Lines

A # (number sign) in a line marks the beginning of a comment. Comments and code can appear on the same line. Characters following the number sign are interpreted as a comment.

Blank Lines

Blank lines are permitted anywhere in a Ratfor program. Blank lines are ignored by the compiler.

Character String Handling

Characters enclosed in matching single or double quotation marks are converted to **nH** but are otherwise unaltered by the compiler. The reformatting process, however, may split characters across card boundaries.

Using Special Characters Literally

Within character strings enclosed by quotation marks, the \ (backslash) serves as an escape character. Any special character immediately following the \ appears in its ASCII character representation.

For example, the backslashes preceding the special characters \ and ' in the following character string:

```
''\\\'''
```

produce the following output:

```
\'
```

Inhibiting the Processing of a Line

A line that begins with the % (percent character) is stripped of the % and moved one position to the left. This technique can be used for inhibiting the compiler interpretation of lines that must not be interpreted (such as device control instructions or an existing FORTRAN program).

Restricted Character Set Translations

The following character equivalencies are provided for input devices with restricted character sets:

[	{
$(	{
]	}
$)	}

Ratfor Statements

The following sections discuss:

- Statement grouping in Ratfor
- General format of statements in Ratfor
- The individual statements implemented in Ratfor.

Statement Grouping

Standard FORTRAN permits the grouping of statements into subroutines. In Ratfor, a group of statements are treated as a unit by enclosing the statements in a { (left brace) and } (right brace).

Several Ratfor statements can be enclosed in braces to be performed as a group or block wherever a single Ratfor statement can be used.

Statement Format

Statements can appear anywhere on a line. Several statements separated by semicolons can appear on a single line.

The following two examples of program code are treated identically by Ratfor:

Example 1:

```
if (x > 100)
    { call error("x>100"); err = 1; return }
```

Example 2:

```
if (x > 100) {
   call error("x>100")
   err = 1
   return
}
```

In the second example, no semicolon is needed at the end of each line. Ratfor assumes that there is one statement per line unless you specify otherwise.

If the statement that follows an **if** statement is a single statement, no braces are needed, as shown in the following example:

```
if (y <= 0.0 & z <= 0.0)
   write(6, 20) y, z
```

Notice that no continuation is indicated on the second line because the statement is implicitly not finished on the first line. In general, Ratfor continues lines when the preceding statement is implicitly not finished.

The break and next Statements

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration in a procedure.

The **break** statement causes an immediate exit from the **do** statement. It functions as a branch to the statement following the statements associated with the **do** statement.

The **next** statement is a branch to the bottom of the loop and causes the next iteration of the **do** loop to be performed.

The following example skips over negative values in an array x:

```
do i = 1, n {
   if (x(i) < 0.0)
       next
   process positive element
}
```

The **break** and **next** statements can be followed by an integer to indicate breaking or iterating a specified level of enclosing loop. The following example exits from two levels of enclosing loops:

```
break 2
```

The following two examples are equivalent:

```
break
```

```
break 1
```

The following example iterates the second enclosing loop:

```
next 2
```

The **break** statement exits immediately from the following statements:

- **do**
- **while**
- **for**
- **repeat – until**.

The **next** statement goes to the increment step of a **for** statement and to the test part of the following statements:

- **do**
- **while**
- **repeat – until.**

The define Statement

The **define** statement assigns constant values to symbol names.

The syntax of the **define** statement can take one of two forms. The following form is the simple form used to define a variable in the declarations at the beginning of a program:

```
define   name   value
```

The following form, also entered in the declarations, can be used to define a constant with a value that fills more than one line:

```
define   (name   valuevaluevaluevaluevaluevaluevalue
     valuevalue)
```

In both forms, the *name* is the name of the new constant and the *value* is the initial value of the variable.

Any string of alphanumeric characters can be defined as a name. Whenever the name occurs in the input delimited by non-alphanumeric characters, it is replaced by its value.

A defined name must begin with a letter and can be any length. Comments and trailing white spaces following the value are ignored.

The following example defines two variables, ROWS and COLS:

```
define ROWS 100
define COLS 50
dimension a(ROWS), b(ROWS, COLS)

if (i > ROWS | j > COLS) . . .
```

The definition for the variable ROWS can also be written as follows:

```
define(ROWS, 100)
```

The assigned value is everything after the comma up to the right parenthesis.

The following example, a routine named equal, defines four symbolic constants:

```
#  equal - compares str1 to str2;
#  return YES if equal, NO if not

define     YES       1
define     NO        0
define     EOS      -1
define     ARB     100

integer function equal(str1, str2)
integer str1(ARB), str2(ARB)
integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
     if (str1(i) == EOS)
     return(YES)
return(NO)
end
```

The do Statement

The **do** statement in Ratfor is similar to the **do** statement in FORTRAN, except that the Ratfor **do** statement uses no statement number.

The syntax for the Ratfor **do** statement is:

do *legal-FORTRAN-do-text*
    Ratfor statement

The *legal-FORTRAN-do-text* that follows the keyword **do** must be something that is legal in a FORTRAN **do** statement.

The *Ratfor statement* can be enclosed in braces. A single statement need not be enclosed in braces.

The following two sections of code show the same **do** statement written in Ratfor and then FORTRAN:

Ratfor code:

```
do i = 1, n {
     x(i) = 0.0
     y(i) = 0.0
     z(i) = 0.0
}
```

FORTRAN code:

```
      do 10 i = 1, n
      x(i) = 0.0
      y(i) = 0.0
      z(i) = 0.0
10    continue
```

In the Ratfor version, the statements associated with the **do** statement are enclosed in braces. The FORTRAN version uses the number 10 to mark the beginning and end of statements associated with the **do** statement.

The following Ratfor example sets all elements in an array x to 0.0:

```
do i = 1, n
   x(i) = 0.0
```

The following example sets the elements in a two-dimensional array m to 0:

```
do i = 1, n
   do j = 1, n
      m(i, j) = 0
```

The following example sets the upper triangle of m to -1, the diagonal to 0, and the lower triangle to +1:

```
do i = 1, n
   do j = 1, n
      if (i < j)
      m(i, j) = -1
      else if (i == j)
           m(i, j) = 0
       else
           m(i, j) = +1
```

In each case, the statement that follows the **do** statement is logically a single statement and thus needs no enclosing braces.

The for Statement

The **for** statement allows explicit initialization and increment steps.

The syntax of the **for** statement is:

```
for ( init ; condition ; increment )
      Ratfor statement
```

The *init* part is any single FORTRAN statement. This statement is always run once before the loop begins.

The *increment* part is any single FORTRAN statement. The *increment* is performed at the end of each pass through the loop, before *condition* is tested.

The *condition* part is any legal condition in a FORTRAN logical **if**.

The *init*, *condition*, or *increment* parts can be omitted. However, the semicolons that separate the three parts must always be present.

A non-existent condition is treated as always **true**. Therefore, the following **for** statement is an infinite loop:

```
for (;;)
```

The following **for** statement sets the variable `i` equal to 1 and runs as long as `i` is less than or equal to `n`:

```
for (i = 1; i <= n; i = i + 1) . . .
```

The following example contains two terminal conditions:

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
     term = -term * x**2 / float(i*(i-1))
     sin = sin + term
}
```

The following example contains decrements the counting variable `i`:

```
for (i = 80; i > 0; i = i - 1)
     if (lastname(i) != blank)
     break
```

The increment in a **for** statement need not be an arithmetic progression. The following program searches through an integer array `ptr` until a zero pointer is found, adding up elements from the array `values`:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
     sum = sum + value(i)
```

The if Statement

Ratfor provides an **if** statement with an **else** clause to handle the logical construction *if−then−else* common to most high−level languages.

The syntax of the Ratfor **if** statement is:

```
if (legal FORTRAN condition)
     Ratfor statement
else
     Ratfor statement
```

The *legal FORTRAN condition* is anything that can legally go into a FORTRAN logical **if**. The legality of the FORTRAN condition is not verified by Ratfor.

The *Ratfor statement* is any Ratfor or FORTRAN statement, or any collection of Ratfor or FORTRAN statements in braces.

The **else** part is optional.

The following is an example of the Ratfor *if−then−else* construct:

```
if (a < = b)
   { sw = 0; write(6, 1) a, b }
else
   { sw = 1; write(6, 1) b, a }
```

The preceding code writes out the variables `a` and `b` in ascending numerical order (lowest to highest), and then sets the variable `sw`.

The FORTRAN equivalent of this code is:

```
       if (a .gt. b) goto 10
      sw = 0
      write(6, 1) a, b
      goto 20
10     sw = 1
       write(6, 1) b, a
20     . . .
```

As previously stated, if the statement following an **if** or an **else** is a single statement, no braces are needed. The following example shows this implicit continuation:

```
if (a <= b)
   sw = 0
else
   sw = 1
```

Nested if Statements

The general structure of nested **if-else** statements in Ratfor is:

```
if ( . . . )
   - - -
else if ( . . . )
    - - -
     else if ( . . . )
         - - -
      else
         - - -
```

You can follow an **if** or **else** statement in Ratfor with any other Ratfor statement, including another **if** statement.

The following Ratfor example shows one **if** statement nested inside of the **else** clause of a preceding **if** statement:

```
if (x < 0)
   f = -1
else if (x > 100)
    f = +1
     else f = 0
```

In the preceding example, the variable f is set to:

- -1, if **x** is less than zero
- +1, if **x** is greater than 100
- 0, if neither of the two conditions is met.

Logically, the second **if-else** statement is a single statement. Therefore, nesting **if-else** statements is one way to write a multi-way branch in Ratfor. The test conditions are laid out in sequence, as conditions in the **if** clauses. Each **if** clause is followed by the code associated with it. The list of test conditions in **if** clauses is read until one test condition is satisfied. The code associated with the satisfied test condition is executed, and then the entire structure is exited. The final **else** clause handles the default case, where none of the conditions in the **if** clauses are met.

If there is no default action, the final **else** can be omitted, as in the following example:

```
if (x < 0)
   x = 0
else if (x > 100)
    x = 100
```

The following example contains two **if** clauses and only one **else** clause:

```
if (x > 0)
   if (y > 0)
      write(6, 1) x, y
   else
      write(6, 2) y
```

The nesting of these statements is resolved in Ratfor by associating the single **else** clause with the closest previous **if** clause. Thus in this case, the **else** clause goes with the inner **if** clause, as indicated by the indentation.

It is recommended that you explicitly resolve such nested cases by enclosing the associated **if** and **then** clauses in braces.

The following example uses braces to specify the desired association of nested clauses:

```
if (x > 0 {
   if (y > 0)
      write(6, 1) x, y
   else
      write(6, 2) y
}
```

The include Statement

The **include** statement inserts an external file into a program.

The syntax of the **include** statement is:

include *file*

The *file* is any type of file that can be validly included in a FORTRAN program. The specified file is read into the Ratfor input in place of the **include** statement.

One standard usage is to place **common** blocks on a file and include that file whenever a copy of it is needed:

```
subroutine x
    include commonblocks
    . . .
    end

subroutine y
    include commonblocks
    . . .
    end
```

This practice ensures that all copies of the **common** blocks are identical.

The Null Statement

A ; (semicolon) by itself on a line is a null statement.

Assume that `nextch` is a function that returns the next input character both as a function value and in its argument. A loop to find the first non-blank character can be written as follows:

```
while (nextch(ich) == iblank)
        ;
```

The null statement marks the end of the **while** statement. The looping continues until the first non-blank character is found.

The repeat – until Statement

The **repeat – until** statement always runs at least one time. The specified condition is tested after its associated statements are performed.

The syntax of the **repeat – until** statement is:

```
repeat
        Ratfor statement
until (legal FORTRAN condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If the condition is true, the loop is exited. If it is false, another pass is made.

The **until** part is optional.

The return Statement

Ratfor provides a **return** statement that returns a value from a function to a calling program.

The syntax of the **return** statement is:

`return(`*expression*`)`

The returned *expression* can be any valid FORTRAN expression.

For a function *F*, the statement *return(expression)* evaluates as follows:

```
{ F = expression; return }
```

The following example uses two **return** statements to signal the end of the function `equal`:

```
#  equal - compare str1 to str2;
#  return 1 if equal, 0 if not
integer function equal(str1, str2)
integer str1(100), str2(100)
integer i

for (i = 1; str1(i) == str2(i); i = i + 1
     if (str1(i) == -1)
     return(1)
return(0)
end
```

If no parenthesized expression following the **return** statement, a normal return is made.

The switch Statement

The **switch** statement implements multi-way branches that are selected on the basis of the value of an integer – valued expression.

The syntax for the **switch** statement is:

```
switch (expression) {
    case expr1:
     statements
    case expr2, expr3:
     statements
    . . .
    default:
     statements
}
```

Each **case** is followed by a list of integer expressions separated by commas. The *expression* inside **switch** is compared against the case expression *expr1, expr2,* and so on, until one case matches. When a match is found, the statements following the **case** are executed.

If no cases match *expression* and there is a **default** section, the statements in the default section are run. If no match is found and there is no **default** section, nothing is done.

As soon as a match is found and its associated block of statements is run, the entire **switch** statement is exited.

The while Statement

The Ratfor **while** statement causes one or more associated statements to be run while one or more conditions are true.

The syntax of the **while** statement is:

```
while (legal FORTRAN condition)
        Ratfor statement
```

The *legal FORTRAN condition* must be a condition that is legal in a FORTRAN logical **if** statement. The *Ratfor statement* can be a single Ratfor statement, or multiple statements enclosed in braces.

The condition in a **while** statement is tested before loop entry. If the condition is false, the **while** loop is not run and program control goes to the statement immediately following the loop.

The following example, which computes **sin**(x) to accuracy e using the Maclaurin series, combines two termination conditions:

```
real function sin(x, e)
     # Returns sin(x) to accuracy e, by computing
     # sin(x) = x - x**3/3! + x**5/5! -  . . .
     sin = x
     term = x
     i = 3
     while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
     }
     return
     end
```

Notice that if the routine is entered with term already smaller than e, the loop is not run. No attempt will be made to compute x**3 and thus a potential underflow is avoided.

General Ratfor Conventions

The following sections discuss general Ratfor conventions related to:

- Error checking
- Keyword restrictions
- Character string specification.

Error Checking

The Ratfor compiler checks for certain syntax errors, including:

- Missing braces
- An **else** clause without a matching **if** clause
- Missing parentheses in statements.

Beyond the Ratfor compiler checks, errors are also reported by the FORTRAN compiler. Sometimes you may have to relate a FORTRAN diagnostic back to the Ratfor source.

Keyword Restrictions

Keywords, such as **if** and **while**, have specially defined meaning in Ratfor. It is recommended that you not use a keyword for anything but its defined purpose.

Hollerith Convention for Specifying Character Strings

The FORTRAN **nH** convention is not recognized by Ratfor. In Ratfor, character strings are specified by enclosing the characters in double or single quotation marks.

Implementation

The following is an outline of the Ratfor grammar:

```
program    : statement
     | program  statement
statement  : if ( . . . ) statement
     | if ( . . . ) statement else statement
     | while ( . . . ) statement
     | for ( . . . ;  . . . ;  . . . ) statement
     | do  . . .  statement
     | repeat statement
     | repeat statement  until( . . . )
     | switch ( . . . ) { case  . . . : program  . . .
                default: program }
     | return
     | break
     | next
     | digits  statements
     | { program }
     | anything unrecognizable
```

The observation that Ratfor knows no FORTRAN follows directly from the rule that says a statement is "anything unrecognizable." Most of FORTRAN falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code Generation

If the first thing on a source line is not a keyword (like **if** or **else**), the entire statement is copied to the output with appropriate character translation and formatting.

Leading digits are treated as a label.

Keywords cause slightly more complicated actions. For example, when **if** is recognized, two consecutive labels, L and $L+1$, are generated and the value of L is stacked. The condition is then isolated and the following code is generated:

```
if (.not. (condition)) goto L
```

The *statement* part of the **if** is then translated. When the end of the statement is encountered, the code:

```
L continue
```

is generated unless there is an **else** clause, in which case the generated code is:

```
   goto L+1
L  continue
```

In this latter case, the code:

```
L+1  continue
```

is produced after the *statement* part of the **else** clause.

Usage Considerations

FORTRAN syntax errors in Ratfor programs are detected by the FORTRAN compiler. The FORTRAN compiler then prints a message in terms of the generated FORTRAN code.

The Ratfor preprocessor checks for syntactic errors such as unbalanced parentheses and quotation marks.

Ratfor keywords are reserved and cannot be used for any other purpose in Ratfor programs.

A few standard FORTRAN constructions are not accepted by Ratfor. These constructions can be sent through the Ratfor preprocessor by protecting the line with a % in the first column. The preprocessor does not process the protected line.

Compiling Ratfor Source Files

The f77 command that compiles FORTRAN source files is also used to compile Ratfor files. The FORTRAN compiler recognizes a file with a **.r** extension as a Ratfor file. The file is translated by the Ratfor preprocessor into a FORTRAN program, then compiled by the FORTRAN compiler to produce an object file.

For example, to compile a Ratfor file named first.r, you type the following command:

```
f77  first.r
```

The general form of the command for compiling Ratfor files is:

f77 *options files*

The implemented *options* relevant to Ratfor are:

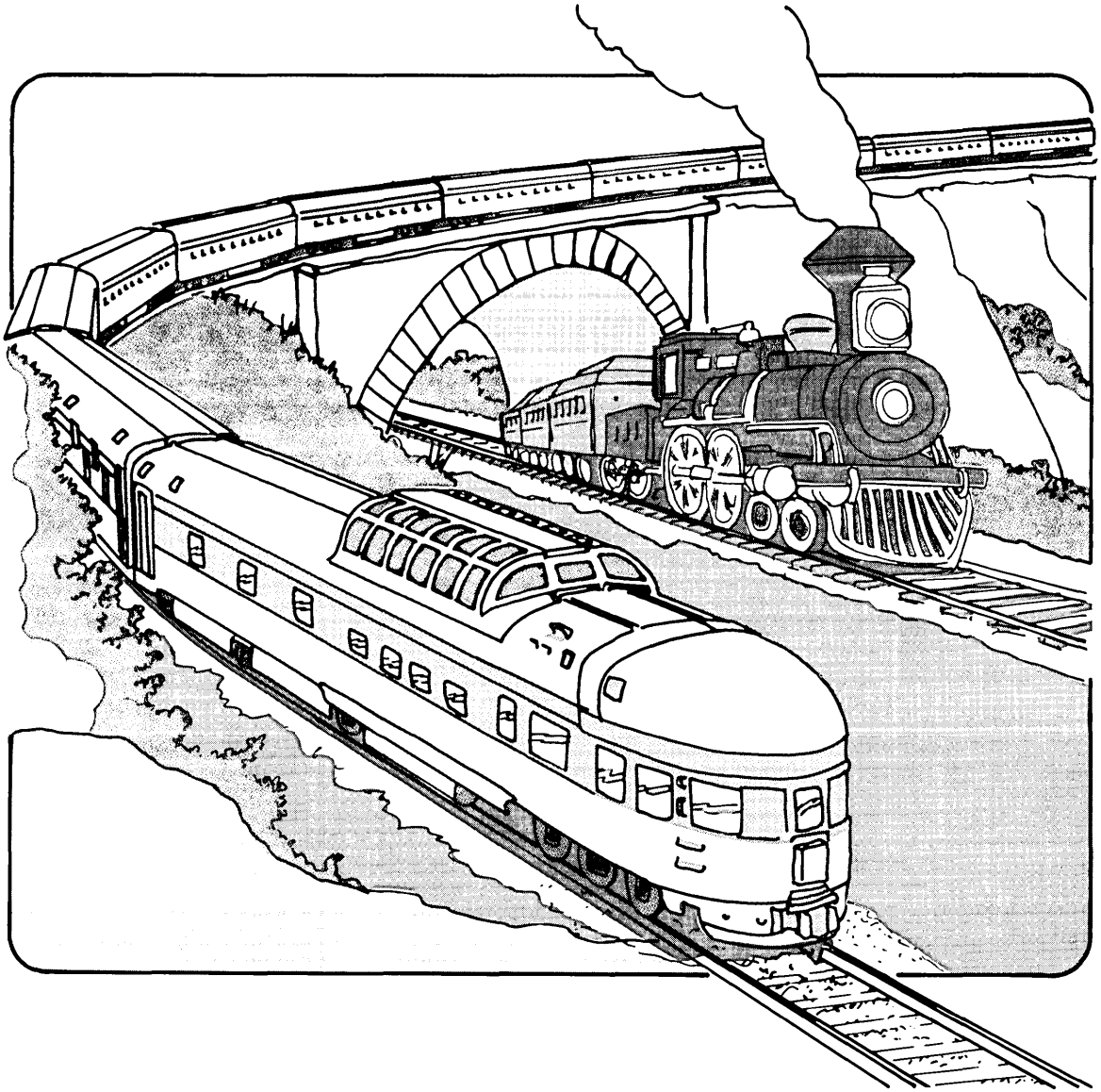
-c	Preprocess only; do not load
-F	Save intermediate FORTRAN files with **.f** extensions

Other flags are passed to the linker.

The *files* are Ratfor files that you want preprocessed, or any other type of file that the f77 command can recognize.

Files with names ending in **.r** are interpreted as Ratfor source files.

Chapter 8. EFL — Extended FORTRAN Language



CONTENT

About This Chapter

EFL, Extended FORTRAN Language, is a preprocessor for the FORTRAN language. The statements of EFL compile into FORTRAN code.

This chapter discusses:

- The terms and concepts of EFL
- Data types and variables
- Expressions
- Declarations
- EFL statements
- The EFL I/O system
- Subroutines and functions
- How to compile EFL source files
- General information on the EFL compiler
- Examples of EFL source code.

The discussions in this chapter assume a fair degree of familiarity with some high-level language implementation.

Capabilities of EFL

EFL is a general purpose computer language. It has a uniform syntax, and data and control flow structuring.

The EFL compiler attempts to diagnose syntax errors and to produce readable FORTRAN output.

The relationship between EFL, FORTRAN, and machine language is illustrated in Figure 8-1 on page 8-5.

An EFL source program in translated into machine language in the following way:

1. The EFL code is compiled with the EFL compiler to produce corresponding FORTRAN code.
2. The resulting FORTRAN is compiled by the FORTRAN compiler to produce the corresponding machine language (object) code.

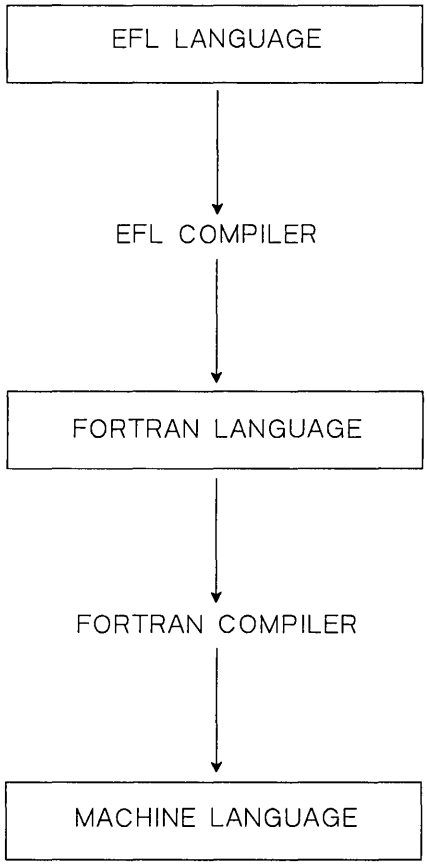


Figure 8-1. Relationship Between EFL, FORTRAN, and Machine Language

Notation and Highlighting

In examples and syntax specifications:

- A word in **boldface** type must be typed as is in an EFL program.
- A word in *italic* type must be replaced with an item of the specified type, such as an *expression*.
- Punctuation must be typed exactly as it appears in the syntax examples.
- A construct surrounded by brackets represents a list of one or more of the specified items, separated by commas.

  Thus, the notation:

  [*item*]

  can refer to any of the following:

  item
  item, item
  item, item, item
- The examples of EFL code appear in lowercase letters. The compiler, however, interprets lowercase and uppercase letters the same, except within strings.

Terms and Concepts

This section discusses the general structure and content of EFL.

Character Set

The following characters are legal in an EFL program:

Letters:	a b c d e f g h i j k l m
	n o p q r s t u v w x y z
Digits:	0 1 2 3 4 5 6 7 8 9
White space:	*blank tab*
Quotation marks:	' "
Number sign:	#
Continuation character:	_
Braces:	{ }
Parentheses:	( )
Other:	, ; : . + — * /
	= < > & | $

Letter case (upper or lower) is ignored except within strings. Therefore the letters **a** and **A** are interpreted the same, except within strings.

An ! (exclamation mark) can be used in place of a ~ (tilde).

[(left bracket) and] (right bracket) can be used in place of { (left brace) and } (right brace).

White Space

Outside of a character string or comment, a sequence of one or more spaces or tab characters acts as a single space.

Lines

In general, the end of a line marks the end of a statement. Exceptions to this general rule are discussed in "Continuation Lines."

The trailing portion of a line can be used for a comment.

Diagnostic messages are labeled with the line number where they are detected.

A line that begins with a % (percent sign) is not interpreted by the EFL preprocessor. It is copied through to the output, with the % removed but no other change.

If a sequence of lines constitutes a continued FORTRAN statement, the lines should be enclosed in braces.

Continuation Lines

Lines are continued explicitly with the _ (underscore character). If the last character of a line is an underscore, the end-of-line character and the initial blanks on the next line are ignored.

The _ (underscore) character should come before any comment on the same line.

Underscore characters inside of character strings are treated as literal characters.

Underscore characters are ignored elsewhere in EFL code.

Some lines are continued implicitly. A statement is continued if the last item on the line is an arithmetic or logical operator, a comma, a left brace, or a left parenthesis.

Some compound statements, such as **if-else**, are continued automatically. For more information, see individual statement explanations in "Statement Directory" on page 8-30.

Line Labels

A ***label*** is a name for a line. Labels are made up of characters, digits, or a combination of both, followed by a : (colon).

A line containing an executable statement can be preceded by a label, as in the following example:

```
        read(, x)
        if(x < 3) goto error
         . . .
error:  fatal('bad input')
```

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

Comments

A comment can appear at the end of any line. Comments remain in the source code as they are typed and are not translated by the compiler.

A comment is preceded by a # (number sign). Everything following the # to the end of the line is interpreted as the comment.

A # inside of a character string is interpreted literally and does not mark a comment.

A blank line is interpreted as a comment.

Reserved Words

Certain words are assigned a specific meaning by the EFL compiler. These ***reserved words*** must not be used for anything other than their defined purposes in EFL programs.

The following figure shows the reserved words that have special meaning in EFL:

array	exit	precision
automatic	external	procedure
break	false	read
call	field	readbin
case	for	real
character	function	repeat
common	go	return
complex	goto	select
continue	if	short
debug	implicit	sizeof
default	include	static
define	initial	struct
dimension	integer	subroutine
do	internal	true
double	lengthof	until
doubleprecision	logical	value
else	long	while
end	next	write
equivalence	option	writebin

Figure 8-2. Reserved Words in EFL

Character Strings

A ***character string*** is a sequence of characters surrounded by quotation marks.

A character string enclosed in single quotation marks can contain unprotected double quotation marks. A character string enclosed in double quotation marks can contain unprotected single quotation marks.

A character string can be more than a single line long.

The following examples are valid character strings:

```
'hello there'
"ain't misbehavin'"
```

Integer Constants

An ***integer constant*** is a sequence of one or more digits.

The following examples are valid integer constants:

```
0
57
123456
```

Floating-Point Constants

A ***floating-point constant*** contains a dot and can optionally contain an exponent field. An ***exponent field*** is a letter **d** or **e** followed by an optionally signed integer constant.

If I and J are integer constants and E is an exponent field, then a floating-point constant has one of the following forms:

```
I.
I.J
IE
I.E
I.JE
```

Arithmetic Operators

The binary arithmetic operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Figure 8-3. Arithmetic Operators in EFL

Exponentiation is right associative, thus:

```
a**b**c = a**(b**c)
```

The following shows examples of operations and their results:

```
8+2 = 10
8-2 = 6
8*2 = 16
8/2 = 4
8**2 = 8*8 = 64
```

Quotients are truncated toward zero, so:

```
8/3 = 2
```

Logical Operators

There are two logical operators in EFL:

- & (and)
- | (or)

Each of the operators can be used in two forms. The **and** operation can be written in one of two ways:

```
a & b

a && b
```

Likewise, the **or** operation can be written in one of two ways:

```
a | b

a || b
```

The use of a single operator (& or |) allows the compiler to evaluate the operands in any order. The lack of restriction on evaluation order at times speeds up program run time.

The use of double operators (&& or ||) forces a left-to-right evaluation of the operands. In an **and** operation, such as:

```
a && b
```

a is evaluated first. If a is false, the expression is false and b is not evaluated. If a is true, the expression is assigned the value of b.

In **or** operations, such as:

```
a || b
```

a is evaluated first. If a is true, the expression is true and b is not evaluated. If a is false, the expression is assigned the value of b.

Results of the two binary logical operations, **and** and **or**, are defined by the following truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Relational Operators

The following table shows the six relational operators in EFL:

EFL Operator	Meaning
<	Less than
< =	Less than or equal to

Figure 8-4 (Part 1 of 2). Relational Operators in EFL

EFL Operator	Meaning
= =	Equal to
=	Not equal to
>	Greater than
> =	Greater than or equal

Figure 8-4 (Part 2 of 2). Relational Operators in EFL

These operators are not associative.

Since the complex numbers are not ordered, the only relational operators that can take complex operands are = = and =. The character collating sequence is not defined.

Assignment Operators

The assignment operators are right-associative. The simple form of assignment is:

var = *expression*

The *var* is a scalar variable name, array element, or structure member of basic type. The *expression* on the right side is computed. The result of the computation is stored in the location named by *var*.

If the type of the result does not match the type of *var*, the type of the result is converted to the declared type of *var*.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. The expression *a op*=*b* is equivalent to *a* = *a op b*. The operator and equal sign must not be separated by blanks. Thus, the expression:

```
n+=2
```

adds 2 to n. In this form, the location of the left side is evaluated only once.

Repetition Operator

Inside of a list, an element of the form:

integer-constant-expression $ *constant-expression*

is equivalent to the appearance of the *constant-expression* a number of times equal to the *integer-constant-expression*. Therefore, the example:

```
(3, 3$4, 5)
```

is equivalent to:

```
(3, 4, 4, 4, 5)
```

Operator Precedence

In the following grouping of operators, all operators on a line have:

- Equal precedence
- Higher precedence than operators on later lines.

The grouping of operators in order of precedence is:

- > .
**
* / unary + - ++ --
+ -
< <= > >= == -=
& &&
| ||
$
= += -= *= /= **= &= |= &&= ||=

Figure 8-5. EFL Operator Precedence

A . (dot) is considered an operator when it is part of a structure element name, but not when it is a decimal point in a numeric constant.

Statements

Statements cause specific actions to be taken. Individual statements implemented in EFL are discussed in "Statement Directory" on page 8-30.

A statement is terminated by an end-of-line character or by a semicolon. Several statements can be written on a single line, if they are separated by semi-colons.

A line consisting only of a ; (semicolon) or ;; (semicolon following a semicolon) forms a ***null statement***.

Statements are frequently made up of other statements. A block, for example, is basically a statement composed of several statements.

Blocks

A ***block*** is an explicit group of statements that is essentially a compound statement. A block is an example of an executable statement; it is made up of declarative and executable statements. The statements making up a block are enclosed between a { (left brace) and a } (right brace).

A block is treated as an executable statement by the EFL compiler. Therefore a block can be used anywhere a statement is permitted.

Declaration statements and executable statements can be part of a block.

A block is not an expression and does not have a value.

A name defined in a block is defined throughout that block and in all deeper nested levels in which the name is not redefined or redeclared.

An example of a block is:

```
{
integer i     #  This variable is unknown outside the braces

big = 0
do i = 1,n
   if(big < a(i))
      big = a(i)
}
```

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure is given a name by which it is invoked.

Procedures are the basic program unit of an EFL program, and provide a means of segmenting a program into separately compilable and named parts.

The first procedure invoked during a program run is considered the main procedure and is assigned the null name.

Each procedure begins with a **procedure** statement and finishes with an **end** statement. For more information on the **procedure** and **end** statements, see the statement discussions in "Statement Directory" on page 8-30.

Files

A ***file*** is a sequence of lines. A file is compiled as a single unit and can contain one or more procedures.

Data Types and Variables

EFL supports a small number of ***scalar*** or basic data types. You can define objects made up of variables of basic type. Other aggregates can then be defined in terms of previously defined aggregates.

Basic Types

The basic types implemented in EFL are:

logical
integer
field*(m:n)*
real
complex
long real
long complex
character*(n)*

A **logical** type can have the value **true** or **false**.

An **integer** can be assigned any whole number value supported by the computer.

A **field** type is an integer restricted to a particular closed interval ([*m:n*]).

A **real** type is a floating-point approximation to a real or rational number. A **long real** type is a more precise approximation to a rational. **Real** types are represented as single-precision floating-point numbers. **Long real** types are represented as double-precision floating-point numbers.

A **complex** type is an approximation to a complex number and is represented as a pair of real quantities.

A **character** type is a fixed-length string of *n* characters.

The following FORTRAN and EFL types are equivalent:

FORTRAN	EFL
double precision	long real
function	procedure

Figure 8-6 (Part 1 of 2). Equivalent FORTRAN and EFL Types

FORTRAN	EFL
subroutine	**procedure** (*untyped*)

Figure 8-6 (Part 2 of 2). Equivalent FORTRAN and EFL Types

Constants

There is a notation for a constant of each basic type.

A **logical** type can be assigned one of two values:

```
true
false
```

An **integer** or **field** constant is a fixed point constant, optionally preceded by a plus or minus sign, as in the following examples:

```
17
-94
+6
0
```

A **long real** (or double-precision) constant is a floating-point constant containing an exponent field that begins with the letter **d**. A **real** (or single-precision) constant is any other floating-point constant. A **real** or **long real** constant can be preceded by a plus or minus sign. The following are valid **real** constants:

```
17.3
-.4
7.9c-6   (= 7.9X10)
14e9     (= 1.4x10)
```

The following are valid **long real** constants:

```
7.9d-6   (= 7.9x10)
5d3
```

A **character** constant is a character string with single or double quotation marks.

Variables

A ***variable*** is a quantity with a name and a location. At any time, the variable can also have a value. A variable is said to be ***undefined*** before it is initialized or assigned its first value, and after certain indefinite operations are performed.

A scalar variable name can be used in expressions. It can appear on the left or the right side of an assignment statement.

Each variable has the attributes of ***storage class***, ***scope***, and ***precision***.

Storage Class

The association of a name and a location is either ***transitory*** or ***permanent***.

Transitory association is achieved when arguments are passed to procedures. Other associations are ***static*** or permanent.

Scope of Names

The names of common areas are global, as are procedure names. These names can be used anywhere in a program.

Other names are local to the block in which they are declared.

To describe the scope of names, it is convenient to introduce the ideas of ***block*** and ***nesting level***.

The beginning of a program file is at nesting level zero. Options, definitions, and variable declarations at the beginning of a program file are also at level zero.

The text immediately following a **procedure** statement is at level 1.

After the declarations, a { (left brace) marks the beginning of a new block and increases the nesting level by 1. A } (right brace) drops the level by 1. Braces inside declarations do not mark blocks.

An **end** statement marks the end of the procedure, level 1, and the return to level 0.

A name (variable) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which the name is not redefined or redeclared.

Thus, a procedure might look like the following:

```
#  Block 0; beginning of level 0
procedure newprogram
real x
x = 2
 . . .
if(x > 2)
#  New block; beginning of level 1
   {
#  A different variable
   integer x
   do x = 1,7
      write(,x)
    . . .
#  End of new block; end of level 1
#  and return to block 0
   }
#  End of procedure; end of block 0
end
```

Precision

Floating-point variables are either of **normal** or **long** precision.

This attribute can be stated independently of the basic type.

Arrays

An ***array*** is basically a group of values of the same type. EFL permits the declaration of rectangular arrays of several dimensions.

An ***element*** of an array is denoted by the array name followed by integer values separated by commas and within parentheses. Each of the integer values, also called ***subscripts***, specifies an interval within one of the array's declared dimensions. Each of the integer values must lie within the corresponding interval. The intervals can include negative numbers.

The dimensionality of an array is specified by an **array** attribute in the array declaration. The general form of an array declaration is:

array(*b . . . b*)

The *b* dimensions can be a single integer expression or a pair of integer expressions separated by a colon. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to *n*, the number of bounds.

The integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are dummy arguments of the procedure; in this case, each bound must have the property that *upper-lower*+1 is equal to a dummy argument of the procedure. The compiler has limited ability to simplify expressions, but it recognizes cases such as (0:n-1).

The upper bound for the last dimension (*b*) can be marked by an * (asterisk) if the size of the array is not known.

The following example denotes the fifth element of array a:

```
a(5)
```

The following example denotes the element in the three-dimensional array b with 6 as its first dimension, -3 as its second dimension, and 4 as its third dimension:

```
b(6,-3,4)
```

Entire arrays can be passed as procedure arguments or in input/output lists. Other array references must be to individual elements.

Arrays can be initialized.

Unqualified names of arrays can appear only as procedure arguments and in input/output lists.

The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals can be constants for arrays that are local or common. A dummy argument array can have intervals that are of length equal to one of the other dummy arguments.

Structures

A ***structure*** is an aggregrate type made up of other types. The individual parts of a structure are called ***members***.

A structure declaration is of the form:

struct *structname* { *declaration statements*}

The *structname* is optional. If *structname* is present, it acts as a type name in the remaining statements within the scope of its declaration.

Each name that appears inside the *declaration statements* is a member of the structure and has a special meaning when used to qualify a variable declared with the structure type.

A name can appear as a member of several structures and can also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known.

The members of a structure can be of any type, including previously defined structures, or they can be arrays of previously defined structures.

Entire structures can be passed as arguments to procedures or used in input/output lists. Unqualified names of structures can appear only as procedure arguments and in input/output lists. Individual elements of structures can be referenced.

The following are valid structure declarations:

```
struct xx
       {
       integer a, b
       real x(5)
       }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

The following example is also a structure declaration:

```
struct tableentry
       {
       character(8) name
       integer hashvalue
       integer numberofelements
       field(0:1) initialized, used, set
       field(0:10) type
       }
```

A structure name followed by a dot and the name of a member of that structure is a reference to that specific member. If that member is itself a structure, the reference can be further qualified.

The following example references the member b in the structure a:

```
a.b
```

The following example references the fifth element of the array y, which is a member of the structure x:

```
x.y(5)
```

EFL defines a notation for dynamic structures that is similar to address types (with pointer and reference) in other high-level languages. The general form of the dynamic structure notation is:

leftside -> *structurename*

The *leftside* is a variable, array, array element, or structure member. The *structurename* is a defined structure. The type of the *leftside* must be one of the types in the structure declaration.

The expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location.

An element of such a structure is denoted using the dot operator. Thus the following example:

```
place(i) -> st.elt
```

refers to the elt member of the structure st starting at the i element of the array place

Expressions

Expressions are syntactic forms that yield a value. An expression can have one of the following forms, recursively applied:

primary

(expression)

unary-operator expression

expression binary-operator expression

Examples of expressions are:

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant and can be used anywhere a constant is required.

Expression Type Conversion

An expression of one precision or type can be converted to another by an expression of the form:

attributes (expression)

The *attributes* options are precision and basic types. Attributes are separated by white space.

An arithmetic value of one type can be converted to another arithmetic type. A character expression of one length can be converted into a character expression of another length. A logical expression cannot be converted into a nonlogical type.

As a special case, a quantity of complex or long complex type can be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the conversion.

The type of the result of a binary operation *A op B*, where *op* is an arithmetic operator, is determined by the types of its operands. The following figure shows the resultant types of arithmetic operations involving two operands, *A* and *B*:

Type of A	integer	real	long real	complex	long complex
Type of B					
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

Figure 8-7. Data Type of Result of Arithmetic Operations

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first converted to the type of the result.

Declarations

Declaration statements describe types and values of variables and procedures.

Syntax

A declaration statement is made up of attributes and variables. The two general forms of declaration statements are:

attributes variable-list

attributes { *declarations* }

In the first form, each name in the *variable-list* has the specified *attributes*. In the second form, each name in the *declarations* has the specified *attributes*. The *declarations* inside the braces are one or more declaration statements.

Examples of declarations are:

```
long real b(7,3)


common(cname)

    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

Scope

Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

A variable name can appear in more than one variable list as long as the attributes are not contradictory.

Initialization

You can assign an initial value to a name of a nonargument variable in a declaration statement.

For example, the following declaration statement defines an integer `k` and assigns `k` an initial value of 2:

```
integer k = 2
```

Attributes

The following sections discuss various attributes that can be declared for EFL data items.

Basic Types

The following are basic types in declarations:

logical
integer
field*(m:n)*
character*(k)*
real
complex

The quantities *k*, *m*, and *n* denote integer constant expressions with the properties $k > 0$ and $n > m$.

The **array** attribute defines the dimensionality of an array. The basic form of the **array** attribute in an array declaration is:

array*(b . . . b)*

An array declaration can have several *b* dimensions. For more information on arrays, see "Arrays" on page 8-20.

The **structure** attribute declares a user-defined data aggregate. The basic form of the **structure** attribute in a structure declaration is:

struct *structname* { *declaration statements*}

The *structname* is the name of the defined structure. The *declaration statements* define the members of the structure. For more information on structures, see "Structures" on page 8-22.

Precision

Variables of floating-point type (**real** or **complex**) can be declared **long** to ensure they have higher precision than ordinary floating-point variables. The default precision is **short**.

Common

Certain objects called ***common areas*** have external scope and can be referenced by any procedure that has a declaration for the name using a **common** attribute. The general form of the **common** attribute in a declaration statement is:

common (*commonareaname*)

All variables declared with a particular **common** attribute are in the same block, and the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

External Procedure Names

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it must be declared with the **external** attribute. The general form of the **external** attribute in a declaration statement is:

external *name*

The *name* is the name of the externally declared procedure.

If a name is declared with the **external** attribute and is a dummy argument of a procedure, the name is associated with a procedure identifier passed as an actual argument at each call. If the name is not a dummy argument, then the name is the actual name of a procedure as it appears in the corresponding **procedure** statement.

Variable List

The elements of a variable list in a declaration consist of:

- A name
- An optional dimension specification
- An optional initial value specification

The name follows the usual rules for a variable name.

The dimension specification has the same form and meaning as the parenthesized list in an **array** attribute.

The initial value specification is an = (equal sign) followed by a constant expression. If the name is an array, the right side of the equal sign can be a parenthesized list of constant expressions, or repeated elements or lists. The total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If a name is used in the context of a procedure invocation, it is assumed to be a procedure name. Otherwise a name is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name.

The association of letters and types can be given in an **implicit** statement. For more information on the **implicit** statement, see the statement entry in "Statement Directory" on page 8-30.

Statement Directory

To increase the legibility of EFL programs, some statement forms can be broken without an explicit continuation. A ■ (square) in the syntax of the statement represents a point where the end of a line is ignored.

Assignment Statements

An expression that is a simple assignment or a compound statement is a statement:

```
a = b
a = sin(x)/6
x *= y
```

The backspace Statement

The **backspace** statement causes the pointer to the current record in the specified unit to back up one record, so that the next read operation re-reads the previous record and the next write operation over-writes it.

The general form of the **backspace** statement is:

backspace(*unit*)

The *unit* is an integer expression defining the device containing the data. See "The Input/Output System" on page 8-43 for more information on units.

The **backspace** statement can also be used as an integer expression that yields non-zero if an error is detected.

The break Statement

A **break** statement transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
   {
   do a computation
   if( finished )
     break

   }
```

More general forms permit controlling a branch out of more than one construct:

```
break 3
```

transfers control to the statement following the third loop and/or **select** surrounding the statement.

It is possible to specify which type of construct (**for, while, repeat, do,** or **select**) is to be counted. The statement:

```
break while
```

breaks out of the first surrounding **while** statement.

Either of the statements:

```
break 3 for
```

```
break for 3
```

transfers to the statement after the third enclosing **for** loop.

The call Statement

The **call** statement invokes a subroutine.

The general form of the **call** statement is:

call *subroutine*

The *subroutine* is the name of a subroutine defined within the scope of the current process.

The define Statement

EFL has a simple macro substitution facility. A name can be defined to be equal to a string; whenever that name appears in the program, the string replaces it.

A name is given a value in a **define** statement like:

```
define count     n += 1
```

Wherever the name **count** appears in the program, it is replaced by the statement:

```
n += 1
```

A **define** statement must appear alone on a line.

The general form of the **define** statement is:

define *name* *rest-of-line*

Trailing comments are part of the string.

The do Statement

EFL has a loop form for ranging over an ascending arithmetic sequence:

do *variable* = *expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to:

```
t2 = expression-2
t = expression-3
for(variable = expression-1 , variable <= t2, variable += t3)
      statement
```

The **do** variable cannot be changed inside of the loop, and *expression-1* must not exceed *expression-2.*

The sum of the first hundred positive integers could be computed by:

```
n = 0
do i = 1, 100
   n += i
```

The end Statement

The **end** statement terminates an EFL procedure.

The general form of the **end** statement is:

end

The endfile Statement

The **endfile** statement causes a file to be marked so that the record most recently written is the last record on the file. Attempts to read past the last record result in errors.

The general form of the **endfile** statement is:

endfile(*unit*)

The *unit* is an integer expression defining the device containing the data. For more information on units, see "The Input/Output System" on page 8-43.

The **endfile** statement can be used as an integer expression that yields non-zero if an error is detected.

The for Statement

The general form of the **for** statement is:

for (*initial-statement* , □ *logical-expression* , □*iteration-statement*) □*body-statement*

If the *initial-statement* is true, the *logical-expression* is evaluated. The *body-statement* and then the *iteration-statement* are run, and the *logical-expression* is re-evaluated. The *body-statement* and the *iteration-statement* are re-run as long as the *logical-expresion* is true.

This form is useful for general arithmetic iterations, and for various pointer-type operations. For example, the sum of integers from 1 to 100 is computed by the following **for** statement:

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

The same computation can also be done by the single statement:

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
     ;
```

Note that the body of the **for** loop is a null statement in this case.

The goto Statement

The general form of the **goto** statement is:

goto *label*

After executing this statement, the next statement performed is the one following the given label.

The Computed goto Statement

A **goto** statement of the following form:

goto(*label*) , *expression*

passes control to the statement marked by the label whose position in the list is equal to the *expression*. The expression must be of type integer, a positive value, and no larger than the number of labels in the list.

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in the following form:

```
go to xyz
```

Inside of a **select**, the case labels of that block can be used as labels, as in the following example:

```
select(k)
    {
    case 1:
          error(7)
    case 2:
          k - 2
          goto case 4

    case 3:
          k = 5
          goto case 4

    case 4:
          fixup(k)
          goto default
    default:
          prmsg("ouch")
    }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

The if Statement

The simplest of the test statements is the **if** statement, of the form:

if (*logical-expression*) □*statement*

The *logical-expression* is evaluated. If it is true, then the *statement* is executed.

The if-else Statement

The general form of the **if-else** statement is:

if (*logical-expression*) ❑ *statement-1* ❑**else**❑*statement*

If the expression is true then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements can be an **if-else** statement, so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** clause applies to the nearest preceding un-**else**d **if** statement.

The following example shows the use of the **if-else** statement as a series of sequential tests:

```
if(x==1)
   k = 1
else if (x==3 | x==5)
     k = 2
else
     k = 3
```

The implicit Statement

The **implicit** statement defines the default association of letters and types.

The general form of the **implicit** statement is:

implicit *(letter-list) type*

The *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign).

The following are the default type and letter associations:

```
implicit (a-h, o-z) real
implicit (i-n) integer
```

The include Statement

It is possible to insert the contents of a file at a point in the source text, by referencing the file in an **include** statement. For example, the following **include** statement inserts the contents of the file `joe` into a program:

```
include joe
```

No statement or comment can follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file.

The initial Statement

An initial value can be specified for a simple variable, array, array element, or member of a structure using an **initial** statement of the form:

initial *var* = *val*

The *var* can be a variable name, array element specification, or member of structure. The *val* follows the same rules as for an initial value specification in other declaration statements.

The next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration; the next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat . . . until**, or the increment of a **do**. Elaborations similar to those for **break** are available.

The following are examples of **next** statements:

```
next
next 3
next 3 for
next for 3
```

A **next** statement ignores **select** statements.

The procedure Statement

Each procedure begins with a statement of one of the forms:

procedure

attributes **procedure** *procedurename*

attributes **procedure** *procedurename*()

attributes **procedure** *procedurename* (*name*)

The first example specifies the main procedure. In the other examples, the *attributes* option specifies precision and type. This option can be omitted. The precision and type of a procedure can also be declared in a declaration statement.

If no type is declared, the procedure is considered a subroutine and no value can be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call.

Each *name* inside the parentheses in the last example is a dummy argument of the procedure.

A procedure is invoked by an expression of one of the forms:

procedurename ()

procedurename (*expression*)

procedurename (*expression-1*, . . . , *expression-n*)

The *procedurename* is either the name of a variable declared **external**, the name of a function known to the EFL compiler, or the name of a procedure as it appears in a **procedure** statement.

If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure.

Each *expression* in the examples is an actual argument.

The following example invokes the procedure `f` with `x` as an argument:

```
f(x)
```

The following example invokes the procedure `work` with an empty argument list:

```
work()
```

The following example invokes the procedure g with multiple arguments:

```
g(x, y+3, 'xx')
```

When a procedure is invoked, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and dummy arguments are checked for agreement.

If an actual argument is a variable name, array element, or structure member, the called procedure can use the corresponding dummy argument as the left side of an assignment or in an input list; otherwise it can only use the value.

After the dummy and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is returned as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure. The attributes in the calling procedure must agree with the attributes declared for the function in its procedure.

In the special case of a generic function, the type of the result is also affected by the type of the argument.

The read Statement

The **read** statement transmits data in the form of lines of characters. The general form of the **read** statement is:

read(*unit* , *formatted-input-list*)

The *unit* is an integer expression defining the device containing the data. If the unit is omitted, the standard input unit is used.

The *formatted-input-list* is an iolist with or without format specifiers in which each of the expressions is a variable name, array element, or structure member.

Each statement moves one or more records. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. Numbers are translated into decimal notation.

For more information on units, iolists, and format specifiers, see "The Input/Output System" on page 8-43.

The readbin Statement

The **readbin** statement transmits data quickly in a machine-dependent binary format. The general form of the **readbin** statement is:

readbin(*unit* , *binary-input list*)

The statement moves one unformatted record between storage and the device.

The *unit* is an integer expression defining the device containing the data. The *binary-input-list* is an iolist without format specifiers in which each of the expressions is a variable name, array element, or structure member.

For more information on units, iolists, and format specifiers, see "The Input/Output System" on page 8-43.

The repeat Statement

The general form of the **repeat** statement is:

repeat ␣ *statement*

The *statement* is run repeatedly. A test must be included in the statement to stop the looping.

The repeat...until Statement

The general form of the **repeat...until** statement is:

repeat ␣ *statement* ␣**until** (*logical-expression*)

The *statement* is first run. Then the *logical-expression* is evaluated. The *statement* is re-run as long as the *logical-expression* is false. When the *logical-expression* is true, control goes to the first statement following the **repeat...until** statement.

The statement always runs at least once.

An **until** clause refers to the nearest preceding **repeat** statement.

The return Statement

The last statement of a procedure is followed by a return of control to the caller. If such a return is desired from another point in the procedure, a **return** statement of the following form:

```
return
```

can be executed.

Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

The rewind Statement

The **rewind** statement moves a device to its beginning, so that the next input statement reads the first record.

The general form of the **rewind** statement is:

rewind (*unit*)

The *unit* is an integer expression defining the device containing the data. For more information on units, see "The Input/Output System" on page 8-43.

The **rewind** statement can be used as an integer expression that yields non-zero if an error is detected.

The select Statement

The general form of the **select** statement is: as a **select** statement, which has the general form:

select (*expression*) □ *block*

The *expression* is evaluated. Based on the evaluation, one of the statements in the *block* is run.

The *block* is a group of statements enclosed in braces. Within the block, two forms of labelled statements are recognized:

case *constant* : *statement*

default : *statement*

There can be several statements with **case** labels in the block, but only one statement with a **default** label.

A statement with the **case** label is run if the expression in the **select** statement is equal to the *constant*. If the expression does not equal a case-labelled constant, control goes to the statement with the **default** label. If there is no default-labelled statement, control goes to the first statement following the block.

Execution of a statement with a **case** or a **default** label continues until another **case** or **default** label is detected. Then control goes to the first statement following the block.

The following example shows a **select** statement containing two cases and one default:

```
select(x)
    {
    case 1:
        k = 1
    case 3,5:
        k = 2
    default:
        k - 3
    }
```

The while Statement

The general form of the **while** statement is:

while (*logical-expression*) □ *statement*

If the expression is true, the statement is run and the test is performed again. If the expression is false, control goes to the next statement.

The write Statement

The **write** statement transmits data in the form of lines of characters. The general form of the **write** statement is:

write(*unit* , *formatted-output-list*)

The *unit* is an integer expression. If the unit is omitted, the standard output unit is used.

The *formatted-output-list* is an iolist with or without format specifiers.

Each statement moves one or more records (lines). The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. Numbers are translated into decimal notation.

For more information on units, iolists, and format specifiers, see "The Input/Output System" on page 8-43.

The writebin Statement

The **writebin** statement transmits data quickly in a machine-dependent binary format. The general form of the **writebin** statement is:

writebin(*unit* , *binary-output list*)

The *unit* is an integer expression defining the output device to which the data is sent. The *binary-output-list* is an iolist without format specifiers.

The statement writes one unformatted record from storage to the output device.

For more information on units, iolists, and format specifiers, see "The Input/Output System" on page 8-43.

The Input/Output System

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**).

These forms can be used either as a primary with an **integer** value or as a statement.

If an error occurs when one of these forms is used as a statement, the result is undefined and is usually treated as a fatal error. If the statements are used in a context where they return a value, they return zero if no error occurs.

For the input forms, a negative value indicates end-of-file and a positive value indicates an error.

Input/output statements resemble procedure invocations but do not yield a value. If an error occurs, the program stops.

Input/Output Units

Each I/O statement refers to a ***unit***, identified by a small positive integer.

A unit in EFL is essentially the same as a unit in FORTRAN. Two special units are defined by EFL, the ***standard input unit*** and the ***standard output unit***. These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into ***records*** that are roughly equivalent to a line. Records can be read or written in a fixed sequence. Each transmission moves an integral number of records. Transmission proceeds from the first record until the end-of-file indicator is reached.

Iolists

An ***iolist*** specifies a set of values to be written or a set of variables into which values are to be read. An iolist is a list of one or more ***ioexpressions*** of the form:

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an ioexpression can also have the forms:

ioexpression : format-specifier
: format-specifier

A ***do-specification*** looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

Format Specifiers

The following are valid ***format-specifiers***. The quantities *w*, *d*, and *k* must be integer constant expressions.

Specifier	Usage
i(*w*)	Integer with *w* digits
f(*w*,*d*)	Floating-point number of *w* characters, *d* of them to the right of the decimal point
e(*w*,*d*)	Floating-point number of *w* characters, *d* of them to the right of the decimal point with the exponent field marked with the letter **e**
l(*w*)	Logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	Character string of width equal to the length of the datum
c(*w*)	Character string of width *w*
s(*k*)	Skip *k* lines
x(*k*)	Skip *k* spaces

If no format is specified for an item in a formatted input/output statement, a default form is used.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

Input/Output Expressions

The EFL input/output statements can be used as integer primaries that have a non-zero value if an error occurs during the input or output.

Subroutines

The following sections discuss various aspects of subroutines in EFL.

Subroutine Call

A procedure invocation that returns no value is known as a ***subroutine call.*** Such an invocation is a statement.

Examples of subroutine calls are:

```
work(in, out)
run()
```

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the dummy argument. The procedure can reference the values in the object, and assign to it.

Otherwise, the value of the actual argument is associated with the dummy argument. In this case, the procedure cannot attempt to change the value of that dummy argument.

If the value of one of the arguments is changed in the procedure, the corresponding actual argument cannot be associated with another dummy argument or with a **common** element that is referenced in the procedure.

Execution and Return Values

After actual and dummy arguments are associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed.

If the procedure is a function (has a declared type) and a **return**(*value*) is executed, the value is first converted to the correct type and precision and then returned to the invoking procedure.

Functions

A number of functions are known to EFL and need not be declared. The compiler knows the types of these functions. Some of them are ***generic***; they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max**. The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that can take different numbers of arguments in different calls.

If any of the arguments are long real, the result is long real. If any of the arguments are real, the result is real. Otherwise, the arguments and the result must be integer.

Examples of the minimum and maximum functions are:

```
min(5, x, -3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument.

For integer and real arguments, the type of the result is identical to the type of the argument. For complex arguments, the type of the result is the real value of the same precision.

Generic Functions

The following generic functions take arguments of real, long real, or complex type and return a result of the same type:

Function	Purpose
sin	Sine function
cos	Cosine function
exp	Exponential function (*e*).

Figure 8-8 (Part 1 of 2). EFL Generic Functions

Function	Purpose
log	Natural (base *e*) logarithm
log10	Common (base 10) logarithm
sqrt	Square root function

Figure 8-8 (Part 2 of 2). EFL Generic Functions

In addition, the **atan** and **atan2** functions accept only real or long real arguments.

The **sign** functions take two arguments of identical type: **sign**$(x,y) = sgn(y)|x|$.

The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

Compiling EFL Source Files

The `f77` command that compiles FORTRAN source files is also used to compile EFL files. The FORTRAN compiler recognizes a file with an **.e** extension as an EFL file. The file is translated by the EFL compiler into a FORTRAN program, then compiled by the FORTRAN compiler to produce an object file.

For example, to compile an EFL file named `first.e`, you type the following command:

```
f77  first.e
```

The general form of the command for compiling EFL files is:

`f77` *options files*

The *options* are those options recognized by the FORTRAN compiler.

The *files* are EFL files that you want compiled, or any other type of file that the `f77` command can recognize.

Files with names ending in **.e** are interpreted as EFL source files.

The Compiler

The EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described previously except for **long complex** numbers.

The following sections discuss different aspects of the EFL compiler.

Diagnostics

The EFL compiler diagnoses syntax errors. It gives the line and file name (if known) in which an error is detected. Warnings are given for variables that are used but not explicitly declared.

Quality of FORTRAN Produced

To the extent possible, the variable names that appear in the EFL program are used in the FORTRAN code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded **goto** and **continue** statements are used.

The following is the FORTRAN procedure produced by the EFL compiler for the matrix multiplication example presented in "Matrix Multiplication" on page 8-53:

```
      subroutine matmul(a, b,    c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do 3 i = 1, m
     do 2 j = 1, p
        c(i, j) = 0
        do 1 k = 1, n
       c(i, j) = c(i, j)+a(i, k)*b(k, j)
1       continue
2     continue
3      continue
      end
```

The following is the FORTRAN procedure for the tree walk example presented in "Walking a Tree" on page 8-55:

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c nextstate values
c initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth = first
1     if (stapth .le. 0) goto  9
      curode = staame(2, stapth)
      goto  7
2     if(tree(1, curode) .ne. const1(1)) goto 3
      call outval(tree1(4, curode))
c a leaf
      stapth = stapth-1
      goto  4
3     call outch(1h()
c a binary operator node
      staame(1, stapth) = 2
      stapth = stapth+1
      staame(1, stapth) = 1
      staame(2, stapth) = tree(2, curode)
4     goto  8
```

```
5       call outch(tree(1, curode))
        staame(1, stapth) = 3
        stapth = stapth+1
        staame(1, stapth) = 1
        staame(2, stapth) = tree(3, curode)
        goto  8
6       call outch(1h)
        stapth = stapth-1
        goto  8
7       if (staame(1, stapth) .eq. 3) goto  6
        if (staame(1, stapth) .eq. 2) goto  5
        if (staame(1, stapth) .eq. 1) goto  2
8       continue
        goto  1
9       continue
        end
```

Compiler Restrictions

The following paragraphs describe the major restrictions imposed on the implementation of EFL by adherence to FORTRAN design concepts.

External Names

External names (procedure and common block names) must be no longer than 6 characters in FORTRAN. Further, an external name is global to the entire program. Therefore EFL can support block structure within a procedure but can have only one level of external name if the EFL procedures are to be compilable separately, as are FORTRAN procedures.

Procedure Interface

The FORTRAN standards, in effect, permit arguments to be passed between FORTRAN procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in FORTRAN. A procedure name can only be passed as an argument or be invoked; it cannot be stored.

Recursion

Standard FORTRAN and EFL procedures are not recursive. However, the IBM RT PC implementation of FORTRAN 77 does allow recursive invocation of procedures.

Storage Allocation

The definition of the FORTRAN standard does not specify the lifetime of variables. It is possible, however, to simulate stack or heap storage by using common blocks.

Examples

This section shows and discusses several EFL programs and program segments.

File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line


while(read( , line) == 0)
     write( , line)
end
```

Since **read** returns zero until the end-of-file character or a read error is reached, this program keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the mxn matrix a by the nxp matrix b to give the mxp matrix c. The calculation obeys the formula $c = a*b$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)

do i = 1,m
do j = 1,p
   {
   c(i,j) = 0
   do k - 1,n
      c(i,j) += a(i,k) * b(k,j)
   }
end
```

Searching a Linked List

Assume we have a list of pairs of numbers (x, y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a specified value of x and returns the corresponding y value.

```
define LAST   0
define NOTFOUND  -1

integer procedure val(list, first, x)

#  list is an array of structures
#  Each structure contains a thread index value, an x, and a y value.

struct xxxy
       {
       integer nextindex
       integer x, y
       } list(*)
integer first, p, arg

for(p = first , p =LAST && list(p).<=x , p = list(p).nextindex)
   if(list(p).x == x)
      return( list(p).y )

   return(NOTFOUND)
   end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (`list(p).x > x`). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the `list(p)` reference. Therefore, the && operator is used. The next element in the chain is found by the iteration statement `p=list(p).nextindex`.

Walking a Tree

As an example of a more complicated problem, assume that we have an expression tree stored in a common area and that we want to print out an infix form of the tree.

Each node is either a leaf (containing a numeric value) or a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk can be implemented by the following simple pseudocode:

```
if this node is a leaf
   print its value
otherwise
   print a left parenthesis
   print the left node
   print the operator
   print the right node
   print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation.

The following procedure calls a procedure `outch` to print a single character and a procedure `outval` to print a value:

```
procedure walk(first)  # print out an expression tree

integer first         # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
   {
   character(1) op
   integer leftp, rightp
   real val
   } tree(100) # array of structures

struct xxxy
   {
   integer nextstate
   integer nodep
   } stackframe(100)

define NODE  tree(currentnode)
define STACK     stackframe(stackdepth)

#  nextstate values
define DOWN  1
define LEFT  2
define RIGHT     3

#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
     currentnode = STACK.nodep
     select(STACK.nextstate)
       {
       case DOWN:
        if(NODE.op == "")  # a leaf
            {
            outval( NODE.val )
            stackdepth -= 1
            }
        else{ # a binary operator node
            outch("(")
            STACK.nextstate = LEFT
            stackdepth += 1
            STACK.nextstate = DOWN
            STACK.nodep = NODE.leftp
            }

       case LEFT:
        outch( NODE.op )
        STACK.nextstate = RIGHT
        stackdepth += 1
        STACK.nextstate = DOWN
        STACK.nodep = NODE.rightp

       case RIGHT:
        outch(")")
        stackdepth -= 1
       }
end
```

Portability

The output of the EFL compiler is intended to be acceptable to any standard FORTRAN compiler.

Primitives

Certain EFL operations cannot be implemented in portable FORTRAN, so a few machine-dependent procedures must be provided in each environment.

Character String Copying

The subroutine **eflase** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copies. If the target string is longer, its end is padded with blanks. The calling sequence is:

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first `lb` characters from `b` to the first `la` characters of `a`.

Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is:

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string `a` of length `la` precedes the string `b` of length `lb`. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

Appendix A. Installing the IBM RT PC FORTRAN 77 Licensed Program Product

This appendix explains how to install the IBM RT PC FORTRAN 77 Licensed Program Product. If a person or a department maintains the computer system, contact them to find out if the FORTRAN 77 Licensed Program Product is already installed.

Prerequisites to Installation

You must have the IBM RT PC FORTRAN 77 Licensed Program Product diskettes that comes with this book to install FORTRAN on the computer system. The diskette is packaged in the plastic envelope in the back of the binder.

The Virtual Resource Manager and the AIX Operating System must be installed on the system before FORTRAN can be installed. If a person or a department maintains the computer system, contact them to verify that these necessary system components are installed.

To install FORTRAN, you must have superuser authority or be a member of the system group. For more information on superuser authority, see *IBM RT PC Using and Managing the AIX Operating System.*

Before beginning the installation, make certain that other users are logged off of the system. If other users are working on the system during the installation process, problems can result.

Installing FORTRAN from Usability Services

To install FORTRAN from Usability Services, follow the steps listed below:

1. Make sure that no one else is using the system and no user programs are running.
2. From the WINDOWS window, select TOOLS from the Window Types pane.
3. Select OPEN from the command bar. A TOOLS window appears on the screen.
4. From the TOOLS window:
   - Select CUSTOMIZATION. The command bar changes.
   - Select OPEN from the command bar. The Customizations Tools Group appears on the screen.
5. From the Customization Tools Group:
   - Select INSTALL. The command bar changes.
   - Select RUN from the command bar. A pop-up containing the choices for INSTALL appears.
6. Make the choices and press **Do** in each pop-up. Pressing **Do** in the last remaining pop-up on the screen runs the command with your choices.
7. Follow the prompts on the display screen.

Installing FORTRAN from the standalone shell

To install FORTRAN from the standalone shell, follow the steps listed below. If a message occurs during the procedure, see the *IBM RT PC Messages Reference* (Part No. SV21-8002) for details.

1. Make sure that no one else is using the system and that no user programs are running. If the system is not in a quiet state, problems can occur as you install the files that make up the FORTRAN 77 licensed program.
2. Log in as superuser or as a member of the system group. You will then see the # prompt.
3. Type the **installp** command after the # prompt, as follows:

   ```
   # installp
   ```

4. The following message appears to remind you to make sure that the system is quiet:

   ```
   000-123  Before you continue, you must make sure there is no other
            activity on the system.  You should have just restarted the system,
            and no other users should be logged on.  Refer to your messages
            reference book for more information.

            Do you want to continue with this command?  (y or n)
   ```

5. Type **y** and press **Enter** to continue with the **installp** command. The following prompt appears:

   ```
   Insert the program diskette into diskette drive
   ''dev/rfd0'' and then press Enter.
   ```

6. Insert the IBM RT PC FORTRAN 77 licensed program diskette into the diskette drive and press **Enter**. The following prompt appears:

   ```
   The program ''FORTRAN 77 Compiler''
   will be installed.

   Do you want to do this?  (y or n)
   ```

7. Type **y** to indicate that you wish to continue with the installation. Then press **Enter**.
8. If a version of this program is already installed on the system, the following message appears:

```
You are about to install version ''01.00.000'' of this program.
This version is the same as or older than the version currently
on your system.  Do you want to do this?  (y/n)
If you type y and press Enter, the
installation process begins.
Please mount volume 1 on /dev/rfd0
```

Your program diskette should already be in the diskette drive (**/dev/rfd0**). Type **y** and press **Enter** to continue the installation (`Type return` is the same as `Press Enter`).

As installation continues, files are listed on the screen as they are copied to the fixed disk.

When installation is complete, the following messages appear:

```
The installation process has completed.


Your operating system will now restart.
```

9. Log off as superuser or as a member of the system group.

   If you want to return to Usability Services, press **Ctrl-D**.

The installation is now completed. You can now begin using the IBM RT PC FORTRAN 77 Licensed Program Product.

Appendix B. ASCII Character Codes

This appendix lists the decimal, octal, hexadecimal, and character representations for each ASCII standard character and for other characters supported on the IBM RT PC.

Decimal Value	Octal Value	Hex Value	Character Value	Decimal Value	Octal Value	Hex Value	Character Value
000	000	00	NUL	043	053	2B	+
001	001	01	SOH	044	054	2C	,
002	002	02	STX	045	055	2D	—
003	003	03	ETX	046	056	2E	.
004	004	04	EOT	047	057	2F	/
005	005	05	ENQ	048	060	30	0
006	006	06	ACK	049	061	31	1
007	007	07	BEL	050	062	32	2
008	010	08	BS	051	063	33	3
009	011	09	HT	052	064	34	4
010	012	0A	LF	053	065	35	5
011	013	0B	VT	054	066	36	6
012	014	0C	FF	055	067	37	7
013	015	0D	CR	056	070	38	8
014	016	0E	SO	057	071	39	9
015	017	0F	SI	058	072	3A	:
016	020	10	DLE	059	073	3B	;
017	021	11	DC1	060	074	3C	<
018	022	12	DC2	061	075	3D	=
019	023	13	DC3	062	076	3E	>
020	024	14	DC4	063	077	3F	?
021	025	15	NAK	064	100	40	@
022	026	16	SYN	065	101	41	A
023	027	17	ETB	066	102	42	B
024	030	18	CAN	067	103	43	C
025	031	19	EM	068	104	44	D
026	032	1A	SUB	069	105	45	E
027	033	1B	ESC	070	106	46	F
028	034	1C	FS	071	107	47	G
029	035	1D	GS	072	110	48	H
030	036	1E	RS	073	111	49	I
031	037	1F	US	074	112	4A	J
032	040	20		075	113	4B	K
033	041	21	!	076	114	4C	L
034	042	22	"	077	115	4D	M
035	043	23	#	078	116	4E	N
036	044	24	$	079	117	4F	O
037	045	25	%	080	118	50	P
038	046	26	&	081	121	51	Q
039	047	27	'	082	122	52	R
040	048	28	(	083	123	53	S
041	051	29	)	084	124	54	T
042	052	2A	*	085	125	55	U

Decimal Value	Octal Value	Hex Value	Character Value	Decimal Value	Octal Value	Hex Value	Character Value
086	126	56	V	129	201	81	ü
087	127	57	W	130	202	82	é
088	130	58	X	131	203	83	â
089	131	59	Y	132	204	84	ä
090	132	5A	Z	133	205	85	à
091	133	5B	[	134	206	86	å
092	134	5C	\	135	207	87	ç
093	135	5D	]	136	210	88	ê
094	136	5E	^	137	211	89	ë
095	137	5F	_	138	212	8A	è
096	140	60	`	139	213	8B	ï
097	141	61	a	140	214	8C	î
098	142	62	b	141	215	8D	ì
099	143	63	c	142	216	8E	Ä
100	144	64	d	143	217	8F	Å
101	145	65	e	144	220	90	É
102	146	66	f	145	221	91	æ
103	147	67	g	146	222	92	Æ
104	150	68	h	147	223	93	ô
105	151	69	i	148	224	94	ö
106	152	6A	j	149	225	95	ò
107	153	6B	k	150	226	96	û
108	154	6C	l	151	227	97	ù
109	155	6D	m	152	230	98	ÿ
110	156	6E	n	153	231	99	Ö
111	157	6F	o	154	232	9A	Ü
112	160	70	p	155	233	9B	¢
113	161	71	q	156	234	9C	£
114	162	72	r	157	235	9D	¥
115	163	73	s	158	236	9E	₨
116	164	74	t	159	237	9F	ƒ
117	165	75	u	160	240	A0	á
118	166	76	v	161	241	A1	í
119	167	77	w	162	242	A2	ó
120	170	78	x	163	243	A3	ú
121	171	79	y	164	244	A4	ñ
122	172	7A	z	165	245	A5	Ñ
123	173	7B	{	166	246	A6	ª
124	174	7C	|	167	247	A7	º
125	175	7D	}	168	250	A8	¿
126	176	7E	~	169	251	A9	⌐
127	177	7F	⌂	170	252	AA	¬
128	178	80	Ç	171	253	AB	½

Decimal Value	Octal Value	Hex Value	Character Value
172	254	AC	¼
173	255	AD	¡
174	256	AE	«
175	257	AF	»
176	260	B0	▒
177	261	B1	▒
178	262	B2	▓
179	263	B3	─
180	264	B4	┤
181	265	B5	╡
182	266	B6	╢
183	267	B7	╖
184	270	B8	╕
185	271	B9	╣
186	272	BA	║
187	273	BB	╗
188	274	BC	╝
189	275	BD	╜
190	276	BE	╛
191	277	BF	┐
192	300	C0	└
193	301	C1	┴
194	302	C2	┬
195	303	C3	├
196	304	C4	─
197	305	C5	┼
198	306	C6	╞
199	307	C7	╟
200	310	C8	╚
201	311	C9	╔
202	312	CA	╩
203	313	CB	╦
204	314	CC	╠
205	315	CD	═
206	316	CE	╬
207	317	CF	╧
208	320	D0	╨
209	321	D1	╤
210	322	D2	╥
211	323	D3	╙
212	324	D4	╘
213	325	D5	╒
214	326	D6	╓
215	327	D7	╫
216	330	D8	╪
217	331	D9	┘
218	332	DA	┌
219	333	DB	▒
220	334	DC	▄
221	335	DD	▌
222	336	DE	▐
223	337	DF	▀
224	340	E0	α
225	341	E1	ß
226	342	E2	Γ
227	343	E3	π
228	344	E4	Σ
229	345	E5	σ
230	346	E6	µ
231	347	E7	τ
232	350	E8	Φ
233	351	E9	Θ
234	352	EA	Ω
235	353	EB	δ
236	354	EC	∞
237	355	ED	∅
238	356	EE	∈
239	357	EF	∩
240	360	F0	≡
241	361	F1	±
242	362	F2	≥
243	363	F3	≤
244	364	F4	⌠
245	365	F5	⌡
246	366	F6	÷
247	367	F7	≈
248	370	F8	°
249	371	F9	∙
250	372	FA	·
251	373	FB	√
252	374	FC	n
253	375	FD	2
254	376	FE	■
255	377	FF	

Figures

Glossary

absolute value. The numeric value of a real number regardless of its sign (positive or negative).

access method. The way records in files are referred to by the system. The reference can be sequential (records are referred to one after another in the order in which they appear in the file), or it can be random (the individual records can be referred to in any order).

address. A number that identifies the location of data in storage.

addressing. A means of identifying storage locations.

allocate. To assign a resource, such as a disk file or a diskette file, to perform a specific task.

All Points Addressable (APA) display. A display that allows each pel to be individually addressed. An APA display allows for images other than ASCII characters to be displayed. Contrast with *character display*.

alphabetic. Pertaining to a set of letters a through z.

alphanumeric. Pertaining to the set of letters a through z and the digits 0 through 9.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment.

American National Standards Institute. An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

ANSI. See *American National Standards Institute.*

a.out. An output file produced by default for certain instructions. By default, this file is executable and contains information for the symbolic debugger.

argument. A parameter passed between a calling program and a procedure.

arithmetic data. Data of the following types: integer, real, double-precision, complex, or double-complex.

arithmetic operator. One of the symbols: + (addition), - (subtraction), * (multiplication), / (division), ** (exponentiation).

array. A sequence of data items collectively identified with one unique symbolic name and data type.

array declarator. A symbolic name and number of dimensions in an array. The number of dimensions determines the number and configuration of array elements.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

ASCII. See *American National Standard Code for Information Interchange.*

Assembler Language. A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

assignment statement. A FORTRAN statement that assigns a value to a variable.

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

binary operator. A symbol representing an operation to be performed on two data items, arrays, or expressions. The four types of binary operators are numeric, character, logical and relational.

bit. Either of the binary digits 0 or 1.

blank common. A single group of storage locations that is accessible to subprograms without being specified as subprogram arguments.

block. A group of records that is recorded or processed as a unit. Same as *physical record.*

block data subprogram. A nonexecutable program unit used to provide initial values for variables and array elements in named common blocks. A block data subprogram has a block data statement as its first statement.

boundary alignment. The position in main storage of a fixed-length field (such as halfword or doubleword) on an integral boundary for that unit of information. For example, a word boundary is a storage address evenly divisible by four.

breakpoint. A place in a computer program, usually specified by an instruction, where execution can be interrupted by external intervention or by a monitor program.

byte. The amount of storage required to represent one character; a byte is 8 bits.

call. (1) To activate a program or procedure at its entry point. Compare with *load.*

cancel. To end a task before it is completed.

carriage return. A keystroke generally indicating the end of a command line.

character. A letter, digit, or other symbol.

character operator. The concatenation operator, //

character position. On a display, each location that a character or symbol can occupy.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character storage unit. The amount of storage required to hold 1 character of data. This implementation uses 1 byte of storage for 1 character of data.

character string. A string or group of characters that contains numerical digits, alphabetical letters, or special characters.

character substring. A contiguous portion of a character string.

character variable. The name of a character data item whose value is assigned and/or changed while the program is running.

close. To end a task.

code. (1) Instructions for the computer. (2) To write instructions for the computer; to *program.* (3) A representation of a condition, such as an error code.

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing.

command. A request to perform an operation or execute a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command name. (1) The first or principal term in a command. A command name does not include parameters, arguments, flags, or other operands. (2) The full name of a command when an abbreviated form is recognized by the computer (for example, print working directory for pwd).

command synonym. A user-assigned alias for a command name.

comment line. A character sequence within the program code that is used to provide program documentation. A comment line does not affect an executable program in any way.

common block. A storage area that can be referenced by more than one program unit.

compilation time. The time during which a source program is translated from a high-level language to a machine language program.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compiler. A program that translates instructions written in a high-level programming language into machine language.

compiler directing statement. A statement controlling what the compiler does rather than what the user program does.

complex data. A processor approximation to the value of a complex number. A complex number is represented as an ordered pair of real numbers occupying a total of 8 consecutive bytes of storage.

concatenate. (1) To link together. (2) To join two character strings.

condition. An expression in a program or procedure that can be evaluated to a value of either true or false when the program or procedure is running.

consecutive processing. The processing of records in the order in which they exist in a file. Same as *sequential processing.*

constant. A data item with a value that does not change. It is either an arithmetic constant, a logical constant, or a character constant.

continuation line. A line used to contain portions of a FORTRAN 77 statement that exceed the columns available in the initial line for statement syntactic items. A statement can have up to 19 continuation lines.

current record. The record currently available to the program.

data item. A constant, variable, array element, or character substring.

data type. The properties and internal representations that characterize data and function. The basic data types are integer, real, complex, logical, double-precision, and character.

debug. (1) To detect, locate, and correct mistakes in a program. (2) To find the cause of problems detected in software.

debugger. A device used to detect, trace, and eliminate mistakes in computer programs or software.

defined. Pertaining to the definition status of a program entity. A defined entity has a value that does not change until the entity becomes undefined or redefined with a different value. An entity must be defined before it can be referenced.

digit. Any of the numerals from 0 through 9.

direct access file. A file containing information that can be accessed in nonsequential order.

double-precision. A processor approximation to the value of a real number that occupies 8 consecutive bytes of storage and can assume a positive, negative, or zero value. The precision is greater than that of type real.

dummy argument. A variable, array, dummy procedure, or * (asterisk) that appears in the argument list of a subprogram or statement function and is associated with an actual argument from the calling program or function

reference. A statement function dummy argument can only be a variable.

dummy variable. A variable whose value is not used in a program but whose name is necessary to use in order to read desired data values.

EFL. FORTRAN preprocessor; a computer language that is translated into FORTRAN before translation into machine language.

element. See *array element.*

embedded blanks. Blanks that are surrounded by other characters.

endfile record. A record written by an **endfile** statement that can occur only as the last record of a file.

entry point. An address or label of the first instruction performed upon entering a computer program, a routine, or a subroutine. A program can have several different entry points, each corresponding to a different function or purpose.

executable program. A collection of program units that consists of exactly one main program and any number (including none) of subprograms.

executable statement. A statement that causes some action to be taken by the program; for example, a calculation, test, or transfer of control.

exponent. A number, indicating the power to which another number (the base) is to be raised.

exponential notation. A notation for real values that uses an **E** to separate the mantissa and the exponent.

exponentiation. The operation in which a value is raised to a power.

expression. A notation that represents a value. An expression is formed from operands, operators, and parentheses. Expressions can be arithmetic, logical, character, or relational.

external file. A file that is available to a program through an external device such as a disk drive, card reader, or tape drive.

external function. See *function subprogram.*

external procedure. An executable program unit that is not a main program. An external procedure can be written in FORTRAN or in another language. When written in FORTRAN, an external procedure is either a function subprogram or a subroutine subprogram.

field. A portion of a record that is read on input or written on output under control of a single edit descriptor.

field width. The number of characters in a field.

file. A sequence of records. An internal file is located in internal storage. An external file is located on an external I/O device.

file name. The name used by a program to identify a file.

floating-point value. A numerical value that can contain decimal positions.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) To arrange such things as characters, fields, and lines.

formatted I/O. The input or output statements that use **format** statements to describe the spacing.

formatted record. A sequence of any characters that can be represented in the processor. The length of a formatted record is measured in characters.

FORTRAN (formula translation). A high-level programming language used primarily for scientific, engineering, and mathematical applications.

full path name. The name of a directory, sub-directory, or file expressed in tree-structure notation. The full path name begins with the root directory.

function. A subprogram that returns a single value to the main program.

function subprogram. A subprogram whose first statement is a **function** statement.

generic function. A function that returns a value of the same type as its input argument.

global. Pertains to information available to more than one program or subroutine.

global entity. An entity in one of the following classes: common block, external function, subroutine, main program, or block data subprogram.

hex. See *hexadecimal.*

hexadecimal. Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

high-level language. An English-like computer language that has to be converted into machine language before it can be executed.

high-order. Most significant; leftmost.

home directory. (1) A directory associated with an individual user. (2) The user's current directory on login or after issuing the cd command with no argument.

I/O. See *input/output.*

I/O list. A list of variables in an input or output statement specifying which data is to be read or written.

ID. Identification.

IF expressions. Expressions within a procedure, used to test for a condition.

implied-DO list. An indexing specification, appearing in an input/output statement or a **data** statement, that has a list of data elements as its range.

initial line. The first line of a FORTRAN 77 statement. If the statement exceeds the initial line, you can use up to 19 continuation lines.

initialize. To give an initial value to a variable.

initially defined. Pertaining to the definition status of an entity. An entity is initially defined if it is assigned a value in a **data** statement.

input. Data to be processed.

input device. Physical devices used to provide data to a computer.

input file. A file that is opened for reading.

input list. A list of variables to which values are assigned from input data.

input/output (I/O). The information that a program reads or writes.

instruction. A statement that specifies an operation to be performed by the computer, along with the values or locations of operands, if any exist. This statement represents the programmer's request to the processor to perform a specific operation.

integer. A positive or negative whole number; that is, an optional sign followed by a number that does not contain a decimal point.

integer data. An exact representation of an integer value that occupies 2 or 4 consecutive bytes of storage and can assume a positive, negative, or zero integral value.

integer value. A value that contains no fractional portion.

internal file. A file defined on information stored in the internal memory of the computer.

intrinsic function. A function used so frequently that its code is included in a library available to the compiler.

K-byte. 1024 bytes.

keyword. A specified sequence of characters that are significant to the FORTRAN compiler in a particular context.

left-justified. No blanks on the left side.

library. A collection of functions, calls, subroutines, or other data.

library function. A function whose code is included in a library that is available to the compiler.

library subroutine. A subroutine whose code is included in a library available to the compiler.

licensed application program. A licensed program used to perform a particular data processing task, such as a distribution management application or a construction management application.

licensed programs. Software programs that remain the property of the manufacturer, for which customers pay a license fee.

linkage editor. A program that resolves cross-references between separately assembled object modules, then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linkage editor simply makes it relocatable. (Also called "link editor.")

list-directed I/O. The input or output statements that do not use **format** statements to describe desired spacing.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

load. To move data or programs into storage.

load module. A file that is suitable for loading into main storage for execution; it is usually the output of a linkage editor.

loader. A program that reads run files into main storage, thus preparing them for execution.

local entity. An entity in one of the following classes: variable, statement function, intrinsic function, or dummy procedure.

local variable. A variable used in a subprogram that is not an argument or a variable in common.

logical data. A representation of one of the two values *true* or *false*; occupies 1 or 4 consecutive bytes of storage.

logical expression. An expression consisting of logical operators and/or relational operators that can be evaluated to a value of either true or false.

logical operator. One of the following symbols: **.not.** (logical negation), **.and.** (logical conjunction), **.or.** (logical inclusive disjunction), **.eqv.** (logical equivalence), or **.neqv.** (logical nonequivalence).

logical value. A value that is either *true* or *false*.

loop. A sequence of instructions performed repeatedly until an ending condition is reached.

low-order. Least significant; rightmost.

machine language. The binary language understood by computers.

main program. A program unit that is not a subprogram. It can have a **program** statement as its first statement. The main program is the program unit that receives control from the operating system to begin execution of a FORTRAN 77 program.

megabyte. One million bytes.

memory. The storage available for the variables and constants needed in a program.

named common. A group of storage locations that is accessible to subprograms by name without being specified as subprogram arguments.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

nested DO loop. A **do** loop that is completely contained within another **do** loop.

new-line character. A control character that causes the print or display position to move to the first position on the next line.

nonexecutable program unit. A block data subprogram.

nonexecutable statement. A statement that is not part of the execution sequence. Nonexecutable statements can specify characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units; and specify entry points within subprograms.

null. Having no value, containing nothing.

null character. The character hex 00, used to represent the absence of a printed or displayed character.

null character string. Two consecutive single quotation marks that specify a character constant of no characters.

numeric. Pertaining to any of the digits 0 through 9.

numeric operator. A symbol representing an operation to be performed on numeric data.

numeric storage unit. The amount of storage required to hold an integer, real, or logical numeric value. A double-precision or complex numeric value uses 2 numeric storage units in a storage sequence.

object code. Output from the assembler. For a single program, object code consists of directly executable machine code. For programs that must be linked, object code consists of relocatable machine code.

object module. A set of instructions in machine language. The object module is produced by an assembler from a subroutine or source module and can be input to the linker. The object module consists of object code. See *module.*

object program. A program in machine language form.

operand. An instruction field that represents data (or the location of data) to be manipulated or operated upon. Not all instructions require an operand field.

operation. A specific action (such as move, add, multiply, load) that the computer performs when requested.

operator. A symbol representing an operation to be done.

output. The result of processing data.

output devices. Physical devices used by a computer to present data to a user.

output file. A file that is opened for writing.

output list. A list of variables from which values are written to a file or device.

overwrite. To write output into a storage or file space that is already occupied by data.

pad. To fill unused positions in a field with dummy data, usually zeros or blanks.

parameter. A named entity, fixed for a particular use, that is used to determine the values of other entities.

path name. A complete file name specifying all directories leading to that file.

pipe. To direct the data from one process to another process.

port. To transfer programs from one computer to another.

position. The location of a character in a series, as in a record, a displayed message, or a computer printout.

precision. The degree of accuracy of a number.

preprocessor. A computer language that is translated into a high-level computer language before translation into machine language.

procedure. A subroutine, external function, statement function, or intrinsic function.

process. (1) A sequence of discrete actions required to produce a desired result. (2) An entity receiving a portion of the processor's time for executing a program. (3) An activity within the system begun by entering a command, running a shell program, or being started by another process.

program. A document containing a set of instructions, conforming to a particular programming language syntax. Programs perform processes and are represented by process objects when active (i.e., when they are executed).

program diskette. The diskette on which a software program product is recorded.

program product. A licensed program for which a fee is charged.

program unit. A sequence of statements and optional comment lines that constitutes a main program or a subprogram.

prompt (n.). A displayed request for information or operator action.

quotient. The quantity that is the result of a division operation.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

Ratfor. FORTRAN preprocessor; a computer language that is translated into FORTRAN before translation into machine language.

real data. A processor approximation to the value of a real number that occupies 4 consecutive bytes of storage and can assume a positive, negative, or zero value.

real number. A number containing a decimal point.

record. A sequence of related values or characters treated as a unit. A record can be a formatted record, an unformatted record, or an endfile record.

relational expression. A expression that compares the values of two arithmetic expressions or two character expressions, resulting in a value of *true* or *false.*

relational operator. One of the following symbols used to compare two arithmetic expressions: **.lt.** (less than), **.le.** (less than or equal to), **.eq.** (equal to), **.ne.** (not equal to), **.gt.** (greater than), **.ge.** (greater than or equal to).

required parameter. A parameter having no value automatically supplied. The user must provide a value.

restore. Return to an original value or image. For example, to restore a library from diskette.

right-justified. No blanks on the right side.

root. The user who can operate without the restrictions designed to prevent data loss or damage to the system.

rounding. A technique that approximates a value.

run. To cause a program, utility, or other machine function to be performed.

scale factor. A number indicating the position of the decimal point in a real number.

scientific notation. A notation for real values that expresses the value as a number between 1 and 10 multiplied by a power of 10.

scope. The extent to which a given symbolic name or statement label can affect a program.

scratch file. A file, usually used as a work file, that exists until the program that uses it ends.

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file.

sequential access file. A file containing information that can be accessed only in a sequential order.

sequential processing. The processing of records in the order in which they exist in a file.

shell. See *shell program.*

shell program. The command interpreter providing the user with an interface to the system kernel.

shell procedure. A series of commands combined in a file that carry out a particular function when the file is run or when the file is specified as an argument to the sh command. Shell procedures are frequently called shell scripts.

source module. The statements or codes that form input to the assembler.

source program. A program in a high-level language form.

source statement. A statement written in a programming language.

special character. A character other than an alphabetic or numeric character. For example, *, , and % are special characters.

specification statement. A statement that specifies the nature of the values to be stored in a variable.

standard input. The source of data going into a process. Standard input generally comes from the display station unless redirection or piping is used, in which case standard input can be a file or the data from a process.

standard output. The destination of data coming from a process. Standard output generally comes from the display station unless redirection or piping is used, in which case standard output can be a file or another process.

statement. A sequence of syntactic items that is the basic unit of a FORTRAN program. Each statement begins with a keyword except for assignment statements and statement function statements.

statement function. A procedure specified by a single statement that is similar in form to an arithmetic assignment statement. It is classified as a nonexecutable statement.

statement label. A number having from 1 to 5 decimal digits that can be used to identify a statement.

status. (1) The current condition or state of a program or device. For example, the status of a printer. (2) The condition of the hardware or software, usually represented in a status code.

storage. (1) The location of saved information. (2) In contrast to memory, the saving of information on physical devices such as disk or tape. See *memory.*

store. To place information onto a diskette where it is available for retrieval and updating.

subexpression. An expression surrounded by parentheses.

subprogram. A program unit that is either a block data subprogram or an external procedure.

subroutine. (1) A sequenced set of statements that can be used in one or more computer programs and at one or more points in a computer program. (2) A routine that can be part of another routine.

subroutine subprogram. A subprogram whose first statement is a **subroutine** statement. A subroutine is referenced with the **call** statement.

subscript. One or more integer or real expressions, enclosed in parentheses and used with an array name to identify a particular array element.

substring. A contiguous part of a character string.

symbolic debugger (sdb). An operating system command that debugs programs written in Assembler Language, as well as programs written in certain other high-level languages.

symbolic name. A sequence of alphanumeric characters, the first of which must be alphabetic, used to identify a global or local entity.

syntax. The rules for the construction of a command or program.

system. The computer and its associated devices and programs.

truncate. To shorten a field or statement to a specified length.

two's complement. Representation of negative binary numbers. Formed by subtracting each digit of the number from zero, then adding one to the result.

type declaration. The specification of the type and, optionally, the length of a variable or function in a specification statement.

unformatted record. A sequence of values in a processor-dependent form. The length of an unformatted record is measured in processor-dependent units and depends on the output list used when it is written, the processor, and the external storage medium.

utility. A service; in programming, a program that performs a common service function.

valid. (1) Allowed. (2) True, in conforming to an appropriate standard or authority.

value. (1) A string or quantity associated with a name. (2) In programming, the contents of a storage location.

variable. A data item whose value can change. Contrast with *constant.*

word. A contiguous series of 32 bits (four bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

zero suppression. The substitution of blanks for leading zeros in a number. For example, 00057 becomes 57 when using zero suppression.

Index

F

G

H

I

L

M

N

O

P

R

S

T

V

X

Z

Notes:

Notes:

Notes:

Notes:

Book Title

SV21-8027-0
Order No.

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name _____

Company name _____

Street address _____

City, State, ZIP _____

No postage necessary if mailed in the U.S.A.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

PLEASE DO NOT STAPLE

Tape

The IBM RT PC
Family

Reader's Comment Form

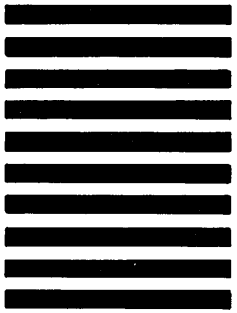
FORTRAN 77 SC23-0818-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

©IBM Corp. 1985
All rights reserved.

International Business
Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Printed in the
United States of America

59X7877