IBM

AIX XL FORTRAN Compiler/6000

User's Guide

SC09-1257-00

First Edition (April 1990)

This edition applies to Version 1.1 of the IBM AIX XL FORTRAN Compiler/6000 and to all subsequent releases and modifications until otherwise indicated in new editions. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Canada Ltd
Information Development
Department 849
1150 Eglinton Ave East
North York, Ontario, Canada. M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks and Acknowledgements

Contents

Chapter 1. Introduction

This user's guide describes the IBM AIX XL FORTRAN Compiler/6000, and explains how to compile, link, and run programs written in XL FORTRAN. FORTRAN (FORmula TRANslation) is a high–level programming language primarily designed for applications involving numeric computations. FORTRAN is suited to most scientific, engineering, and mathematical applications.

The exceptional (XL) family of compilers provides consistency and high performance across multiple programming languages by sharing the same code optimization technology.

The XL FORTRAN compiler, Version 1.1, compiles programs written using the American National Standards Institute (ANSI) FORTRAN Language with selected RT PC VS FORTRAN and RT PC FORTRAN 77 extensions. The *American National Standard Programming Language FORTRAN*, ANSI X3.9–1978, details the ANSI FORTRAN language.

The XL FORTRAN compiler conforms to the Systems Application Architecture (SAA) definition of the FORTRAN language.

Note: Language extensions noted in this guide are extensions to the FORTRAN definition in *Systems Application Architecture Common Programming Interface FORTRAN Reference,* SC26–4357.

Who Should Use This Manual

This manual is for people who want to use the XL FORTRAN compiler, who are familiar with the IBM AIX Version 3 for RISC System/6000, and who have some previous programming experience. If you are not familiar with the operating system, refer to *AIX General Concepts and Procedures for IBM RISC System/6000*, SC23–2202.

How to Use This Book

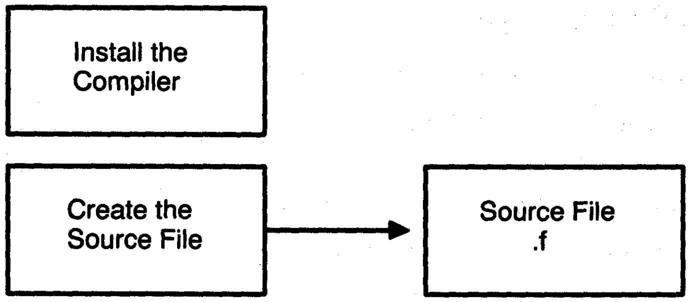
This book is not intended as a tutorial, but rather it explains the details for using the XL FORTRAN compiler. The manual is organized according to the steps necessary to compile, link, and run a program, and deals with more advanced topics in the later chapters.

If you have not used a compiler on the IBM AIX RISC System/6000 computer before, you might find it useful to read the first three chapters of this book before you proceed. After you become familiar with the system and the compiler, you can use this manual as a handy reference.

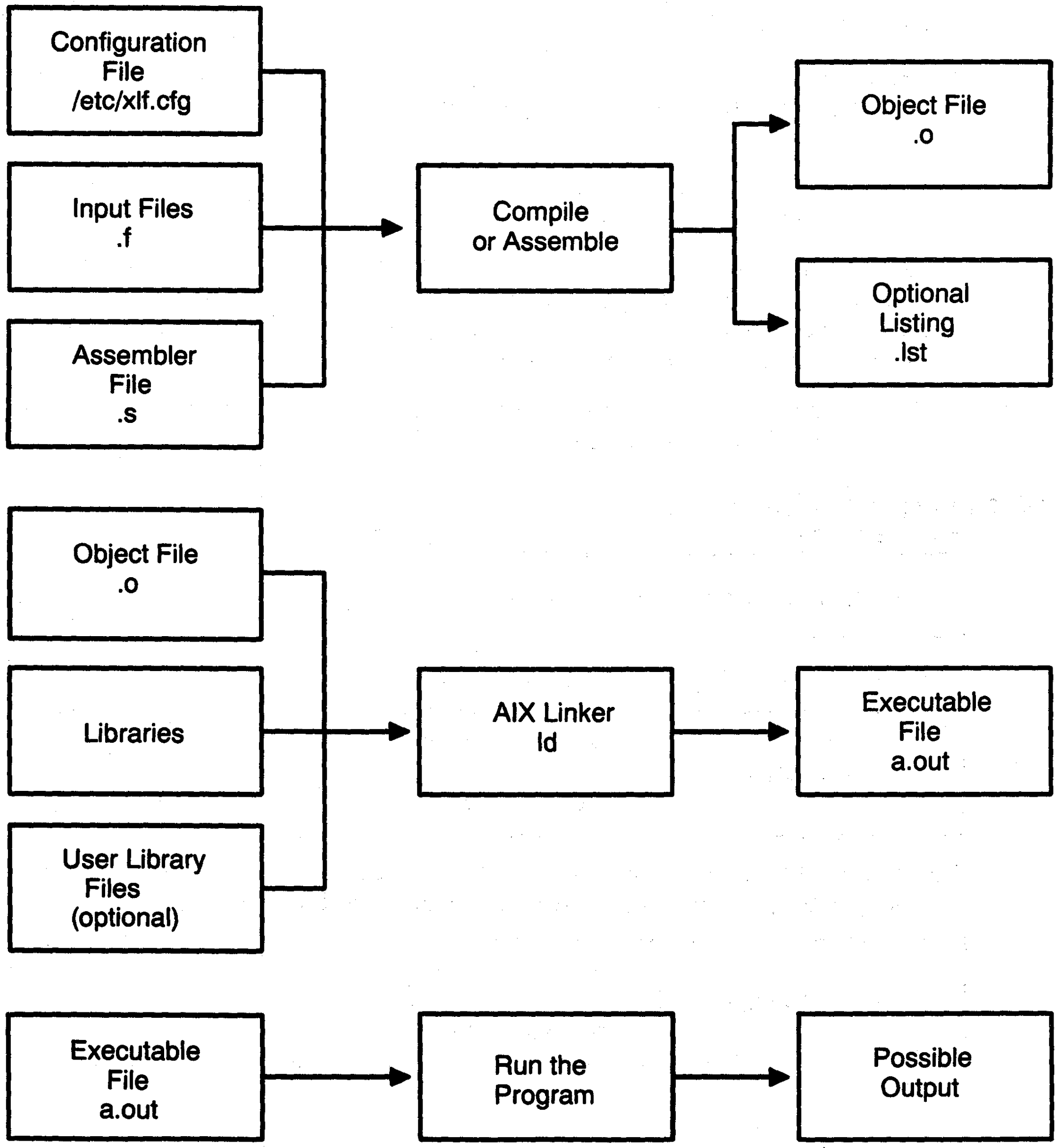
How This Book is Organized

The following diagram depicts the path that the XL FORTRAN compiler takes when you invoke it. The diagram also shows the location of related details within this guide.

Chapter 2



Chapter 3



How to Read the Syntax Diagrams

Throughout this book, syntax diagrams use the structure defined below:

- Syntax diagrams are read from left to right and from top to bottom, following the path of the line.

  The — symbol indicates the beginning of the diagram.

  The ➝ symbol indicates that the syntax is continued on the next line.

  The ▶— symbol indicates that the syntax is continued from the previous line.

  The —| symbol indicates the end of the diagram.

  Diagrams of syntactical units other than complete statements start with the ▶— symbol and end with the ➝ symbol.

- Keywords appear in the diagrams in uppercase; for example, **OPEN**, **COMMON**, and **END**. You must spell them exactly as shown.

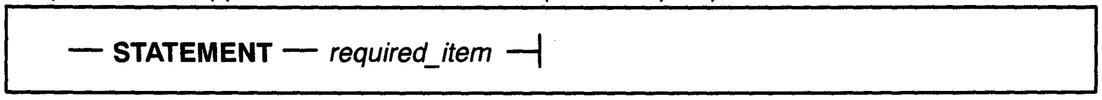
  Note: You can type keywords in uppercase, lowercase, or mixed case, and the compiler folds them into lowercase during compilation. However, if you specify the **MIXED** compiler option you must enter keywords in lowercase.

- Variables and user–supplied names appear in lowercase italics; for example, *array_element_name*. If one of these terms ends in *_list* it specifies a list of terms. A list is a nonempty sequence of the terms separated by commas. For example, the term *name_list* specifies a list of the term *name*.

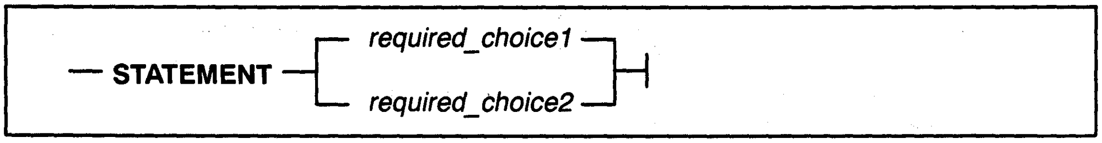
- Punctuation marks, parentheses, arithmetic operators, and other special characters must be entered as part of the syntax.

Required and Optional Items

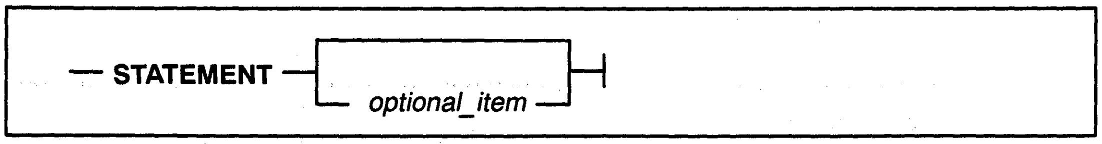
Required items appear on the horizontal line (the main path).



Branching shows two paths through the syntax.

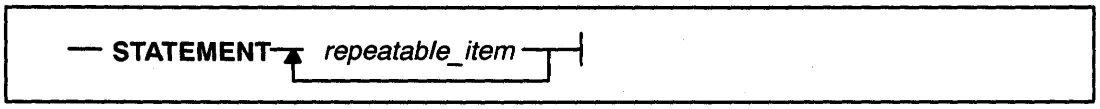


Optional items appear on the lower line of a branched path. The upper line is empty, indicating that you do not need to code anything for this syntax item.

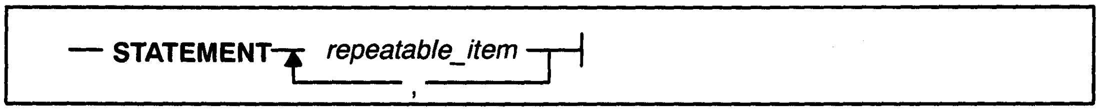


Repeatable Items

An arrow returning to the left below a line shows items that you can repeat.

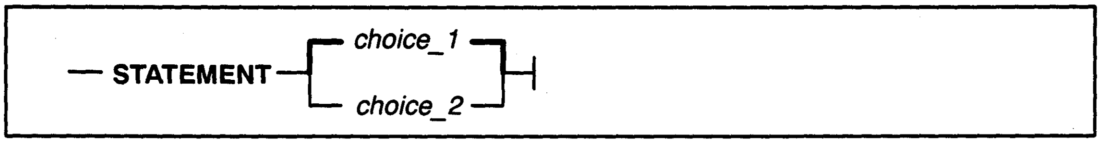


Punctuation on a repeat arrow must be placed between the repeated items.



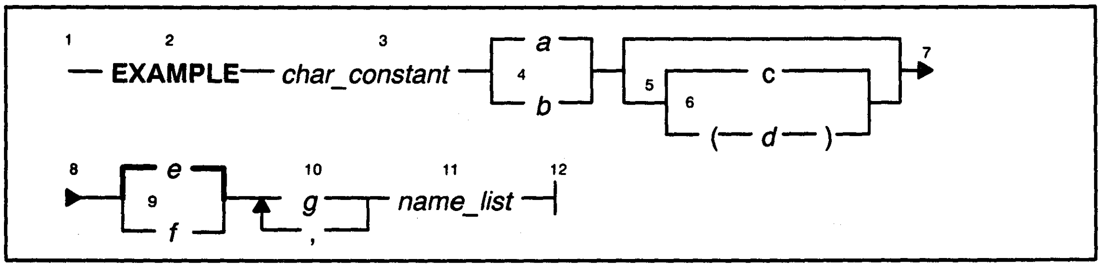
Default Items

A heavy line is the default path. Coding nothing for that item is the same as coding the default item.



Example of a Syntax Diagram

The following example of a fictitious statement shows how to use the syntax:



Interpret the diagram by following the numbers:

1. This is the start of the diagram.
2. Enter the keyword **EXAMPLE**.
3. Enter a value for *char_constant*.
4. Enter a value for *a* or *b*, but not for both.
5. This path is optional.
6. Enter a value for *c* or *d*, or no value. If you enter a value for *d*, you must include the parentheses.
7. The diagram is continued at 8.
8. The diagram is continued from 7.
9. Enter a value for *e* or *f*, or no value. If you do not enter a value, the default value *e* is used.
10. Enter at least one value for *g*. If you enter more than one value, you must put a comma between each.
11. Enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each.
12. This is the end of the diagram.

A Note About Examples

Examples in this book explain elements of the XL FORTRAN language. They are coded in a simple style. They do not try to conserve storage, check for errors, achieve fast run times, or demonstrate all possible uses of a language element.

Related Documentation

You might want to refer to the following publications for additional information:

IBM Publications:

Reference Manual for IBM AIX XL FORTRAN Compiler/6000, SC09–1258–00, describes the XL FORTRAN programming language as implemented on the IBM AIX RISC System/6000 computer.

Systems Application Architecture Common Programming Interface FORTRAN Reference, SC26–4357, describes the FORTRAN component of the common programming interface.

Non–IBM Publications:

American National Standard Programming Language FORTRAN, ANSI X3.9–1978

International Standards Organization Programming Language FORTRAN, ISO 1539–1980(E)

Federal Information Processing Standards Publication FORTRAN, FIPS PUB 69

Instrument Society of America Standard: Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output and Bit Manipulation, ANSI/ISA S61.1

Military Standard FORTRAN, DOD Supplement to ANSI X3.9–1978, MIL–STD–1753

ANSI/IEEE Standard for Binary Floating–Point Arithmetic, ANSI/IEEE Std 754–1985.

Chapter 2. The IBM AIX XL FORTRAN Compiler/6000

This chapter discusses the XL FORTRAN compiler features and summarizes the items to keep in mind when entering your FORTRAN source program. For details of the XL FORTRAN language refer to the *Reference Manual for IBM AIX XL FORTRAN Compiler/6000*, SC09–1258.

The XL FORTRAN language comprises:

- FORTRAN 77:
  - The full ANSI FORTRAN 77 language (referred to as FORTRAN 77), defined in the document *American National Standard Programming Language FORTRAN*, ANSI X3.9–1978.
- The XL FORTRAN language extensions are primarily (though not exclusively):
  - IBM extensions specified in Systems Application Architecture FORTRAN Release 1.
  - Selected other extensions commonly available in earlier IBM FORTRAN compilers, and that the VS FORTRAN, RT PC FORTRAN 77, and RT PC VS FORTRAN compilers currently support.

Features of the XL FORTRAN Compiler

The XL FORTRAN compiler is an IBM Licensed Program that operates within the IBM AIX Version 3 for RISC System/6000 environment. The XL FORTRAN compiler supports all RISC System/6000 computer hardware configurations.

Language Support

The language level supported by XL FORTRAN is based on the ANSI FORTRAN 77 definition.

XL FORTRAN is also based on SAA FORTRAN, defined in the *Systems Application Architecture Common Programming Interface FORTRAN Reference*, SC26–4357. The *Reference Manual for IBM AIX XL FORTRAN Compiler/6000*, SC09–1258 outlines the extensions to SAA FORTRAN.

Compiler Features

The XL FORTRAN compiler provides the following support for engineering and scientific application development:

- Highly optimized object code
- VS FORTRAN, RT PC VS FORTRAN, and RT PC FORTRAN 77 compatibility (with some exceptions)
- Descriptive diagnostics
- **dbx** debugger support.

Compiler Options

You can invoke the compiler using the **xlf** command and control its actions with compiler options. The compiler sets the return code to indicate the completion status of the program compilation, and can also provide timing and resource usage data. For a discussion of compiler options and invocation refer to “Compiler Options” on page 16.

The compiler output listing has optional sections controlled by the compiler options you select. By default, XL FORTRAN produces no listing. A description of the listing format is in "Compiler Listings" on page 46.

Symbolic Debugger (dbx) Support

The XL FORTRAN compiler generates debug information tables compatible with **dbx** giving you use of the symbolic debugger for your FORTRAN programs.

Note: XL FORTRAN supports full source–level debugging with no optimization in effect. There is no guarantee that symbolic debugging can take place at the higher optimization level.

Source Code Conformance Flagging

If you specify the appropriate compiler option, the compiler flags XL FORTRAN source statements for conformance to the following FORTRAN language definitions:

- Full ANSI FORTRAN 77 (**FIPS** option)
- IBM Systems Application Architecture FORTRAN (**SAA** option).

Generated Code Optimization

The XL FORTRAN compiler gives you the ability to control the optimization of generated code through compiler options. See "Compiler Options" on page 16 for further information.

Online Compiler Help

The XL FORTRAN compiler provides online help that lists available command line options. Invoke this online help by using the **xlf** command with no arguments.

Migration Characteristics

The XL FORTRAN compiler aids migration by providing source code compatibility with various existing compilers, but it does not provide object code compatibility. See Appendix E, "Migration Considerations" for further information.

Also, the XL FORTRAN compiler provides two compiler options to allow you to flag (with warning messages) features that do not conform to certain FORTRAN definitions. See the "Source Code Conformance Flagging" section above for more information about this feature.

System Configuration

The XL FORTRAN compiler, its generated object programs, and the XL FORTRAN library run on all RISC System/6000 computer hardware configurations under IBM AIX Version 3 for RISC System/6000.

Compiler Installation

You install the compiler by using the AIX **installp** command while logged on as root or while in superuser mode. See the *AIX Commands Reference for IBM RISC System/6000*, SC23–2199 for the details of this command.

There is a file called **/usr/lpp/xlf/DOC/README.xlf** that you should examine for supplemental details. See the *Installation Instructions for IBM AIX XL FORTRAN Compiler/6000* for further information about installation.

Entering FORTRAN Source Programs

A variety of text editors are available to use under IBM AIX Version 3 for RISC System/6000. You can use any one of these to create FORTRAN source programs. You must create the source program with a suffix of **.f** .

For a complete description of the XL FORTRAN language and its structure, see the *Reference Manual for IBM AIX XL FORTRAN Compiler/6000*, SC09–1258.

FORTRAN Source Files

The following sections give information you need to enter your source files.

The XL FORTRAN Character Set

The XL FORTRAN character set consists of the letters A–Z, the letters a–z, the digits 0–9, and the following special characters:

```
Blank  !  "  $  %  '  (  )  *  +   ,  –  .  /  :   <  =  >  _
```

The characters are arranged in a collating sequence. The collating sequence is the arrangement of characters for a given system that determines their comparison status. XL FORTRAN uses ASCII (American National Standard Code for Information Interchange) to determine the ordinal sequence of characters. See Appendix B, "ASCII/EBCDIC Character Set" for the complete ASCII character set.

Names

A name (or symbolic name) is a sequence of 1 to 250 letters or digits, the first of which must be a letter. XL FORTRAN treats the currency symbol ($) and underscore character (_) as letters when you use them in a name, and you can use either one as the first character. Note that the use of $ as the first character in external names can cause unpredictable results in AIX shell procedures, because AIX uses $ as the first character in a shell variable name. Also, underscore (_) is reserved for system use and some compiler generated names, so you may want to avoid using it as your first character.

XL FORTRAN folds all letters in a source program from uppercase to lowercase unless they are in a character context[1]. If you specify the **MIXED** compiler option, XL FORTRAN does not fold the source program, and symbolic names are distinct if you specify them in a different case.

Keywords

A keyword is a sequence of characters that, in certain contexts, identifies a language construct. XL FORTRAN does not reserve any sequence of characters in all contexts. You can write keywords in uppercase, lowercase, or mixed case, but XL FORTRAN folds them to lowercase. If you specify the **MIXED** compiler option, the compiler does not fold the source program, and you must write keywords in lowercase.

Statements

A FORTRAN statement is a sequence of syntactic items. Statements form program units – a sequence of statements and optional comment lines that constitute a main program or

[1] A character context means characters within character constants, Hollerith constants, format–item lists in FORMAT statements, and comments.

subprogram. You must write each statement according to the source input format specified by your selected compiler options (**FIXED** or **FREE**). The default input format is **FIXED**.

Fixed–Form Input Format

In fixed–form input format, each line is a sequence of 72 characters. Columns 73 and beyond are not significant to the compiler, and you can use them for identification, sequencing, or any other purpose.

An initial line contains a statement label, if you desire, in columns 1 to 5, a blank or zero in column position 6, and the characters representing the statement in columns 7 to 72. If there is no statement label, leave columns 1 to 5 blank.

The text of any statement except the **END** statement and the **EJECT**, **INCLUDE**, and **@PROCESS** compiler directives can continue on the following line. A continuation line contains blanks in columns 1 to 5, and any character in the XL FORTRAN character set other than a blank or zero in column position 6. XL FORTRAN allows columns 1 to 5 to contain characters, but the compiler ignores them. You can use up to 99 continuation lines for a single statement.

A comment indicator (`C`, `c`, or `*`) in column 1 will cause the compiler to treat the line as a comment. Comment lines do not affect the executable program, and you can use them to provide documentation. They can have either of two forms:

- `C`, `c`, or `*` in column 1 and, optionally, any characters you can use in a character constant in columns 2 through 72
- Blanks in columns 1 through 72.

Note that a `D` in column 1 will also cause the compiler to treat the line as a comment line if you do not specify the **DLINES** compiler option.

A comment line can appear anywhere in the program unit before the **END** statement. There is no restriction on the number of comment lines you can use.

An exclamation point (!) initiates an inline comment except when it appears in a character context, or if it appears in column 6 (where it is treated as a continuation character). The comment extends to the end of the source line. An **END** statement can contain a comment that you initiate with an exclamation point (!). An **@PROCESS** compiler directive cannot contain an inline comment.

```
c
c This is a fixed–form example
c
      DO 10 I=1,10
        WRITE(6,*)'this is the index',I   ! with an inline comment
10    CONTINUE
```

Free–Form Input Format

In free–form input format, the first character of the statement (after a label, if there is one) must be alphabetic. The maximum length of a free–form statement is 6600 characters (equivalent to 100 fixed–form lines), excluding the continuation characters and the statement labels. The statement continuation character is a minus sign (–) and it appears at the end of every line to be continued.

An initial line can start in any position, and can contain (as the leftmost entry on a line) a statement label. XL FORTRAN ignores leading and imbedded blanks in a statement label.

The text of any statement, except the **END** statement and the **EJECT**, **INCLUDE**, and **@PROCESS** compiler directives can continue on the following line. You indicate a line you want continued with a minus sign terminating the line. It must be the last nonblank character that is not part of a comment. The statement text of a continuation line can start in any position. You can have up to 99 continuation lines in a single statement.

A comment line cannot be continued, and must not follow a continued line. It begins with a double quotation mark (") in column 1, or is a blank line.

An exclamation point (!) initiates an inline comment except when it appears in a character context. The comment extends to the end of the source line, and XL FORTRAN processes it as if it were blank characters (including the delimiter). An **END** statement can contain an inline comment that you initiate with !. An **@PROCESS** compiler directive cannot contain an inline comment.

The minus sign for continuation must precede the ! delimiter on continuation lines. You can intersperse ! commentary with free–form source lines, but you must use the – to continue any line. If you want to continue a character context, you cannot follow the – signifying continuation by an inline comment.

```
"
" This is a free—form example
"
 DO 10 I=1,10
 WRITE(6,*)'this is —
      the index',I
10 CONTINUE
```

Tabs

A tab character placed anywhere in columns 1 to 6 will tab to column 7. Therefore, you cannot tab continuation lines in fixed–form input. XL FORTRAN treats any other tab characters, except for those in a character context, as blanks.

For example, if you assume the @ is the system–generated tab character, the code segment:

```
C@Example of tab input lines
@I=0
10@CONTINUE
```

is equivalent to:

```
C     Example of tab input lines
      I=0
10    CONTINUE
```

after resolution of the tabs.

Nonsignificant Blanks

You can position as many blanks as you want in a statement or comment to improve readability. You can even imbed blanks within keywords or names, because the compiler ignores them. XL FORTRAN retains blanks inserted in character or Hollerith constants, and treats them as blanks within the data.

Statement Labels

A statement label is a sequence of one to five digits, one of which must be nonzero, that you can use to identify statements in a FORTRAN program unit. You can label a fixed–form statement by placing a statement label anywhere in columns 1 through 5 of its initial line. The compiler ignores statement labels that appear on continuation lines.

Statement labels on free–format input lines must be the first nonblank characters (digits) on an initial line. You do not need blanks between the statement label and the first nonblank character following.

You must not give the same label to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels. You can label any statement, but you can only refer to executable statements and **FORMAT** statements using statement labels. You must place the statement making the reference and the statement you want to reference in the same program unit.

Chapter 3. Compiling, Linking, and Running Programs

After you create the source file, there are three phases to preparing and running the program:

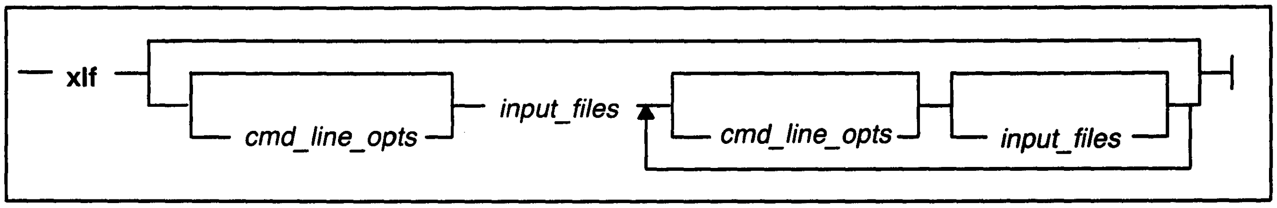
1. Compiling
2. Linking
3. Running.

This chapter discusses the details of using the XL FORTRAN compiler. It outlines how to invoke the compiler, how to specify the available compiler options, how to invoke the linkage editor, and how to run your compiled program.

Note: Before using this compiler, it is important that the programmer first read the descriptions of the compiler options to understand the correct operation and limitations of this product.

Invoking The Compiler

To compile a source program, use the **xlf** command, which has the form:



You can specify *cmd_line_opts* in the following ways:

- –q*option*.
- Option flags (usually a single letter preceded by –).

All options specified on the command line are in effect for all specified source files. (See "Specifying Options on the Command Line" on page 17 for the syntax.)

A description of *input_files* is in "Input Files" on page 16.

The flags and options you can use appear in the tables starting at "Summary of the XL FORTRAN Compiler Options" on page 18.

The **xlf** command invokes the XL FORTRAN compiler. It compiles the FORTRAN source files, sends any **.s** files to the assembler, and then links the resulting object files with any object files and any libraries specified on the command line in the order indicated. It then produces a single executable file called **a.out** by default. You can use the **–o** flag to specify an alternative name for the resulting executable file.

The **xlf** command executes the following sequence of programs. Each program executes, and then sends the results to the next step in the sequence.

1. The process may call **xlfentry** (the compiler).

   The compiler consists of the following four phases:

   - Phase 1: Front end parsing and semantics handling
   - Phase 2: Optimization

- Phase 3: Register Allocation
- Phase 4: Final Assembly.

2. The process calls the assembler for **.s** files, if needed.
3. It then calls the linkage editor **ld** (if you have not specified the **–c** compiler option).

Environment Variables

The message catalogs must be installed before the compiler will compile your program. Before you invoke the compiler, the following two environment variables must be set and exported:

LANG	Specifies the national language for message and help files.
NLSPATH	Specifies the path name of message and help files.

You can set the environment variable from the Bourne shell, Korn shell, or the C shell. To determine which shell is in use, issue the AIX **echo** command:

```
echo $SHELL
```

The Bourne shell path is `/bin/sh`, Korn shell is `/bin/ksh`, and C shell is `/bin/csh`.

To set the environment variables from the Bourne or Korn shell, use the following commands:

```
LANG=En_US
NLSPATH=/usr/lpp/msg/%L/%N:/usr/lpp/msg/En_US/%N
export LANG NLSPATH
```

To set the variables system wide, so all users will have access to them, add the lines above to the file **/etc/profile**. To set them for a specific user only, add them to the file **.profile** in the user's home directory. This will set the environment variables the next time the user logs in.

To set the environment variables from the C shell, use the following commands:

```
setenv LANG En_US
setenv NLSPATH /usr/lpp/msg/%L/%N:/usr/lpp/msg/En_US/%N
```

In the C shell, you cannot set environment variables system wide. To set them for a specific user only, add the lines above to the file **.cshrc** in the user's home directory. This will set the environment variables the next time the user logs in.

`En_US` is the national language code for United States English. You can substitute any other valid national language code for `En_US` provided the message catalogs associated have been installed.

These environment variables are initialized when the Operating System is installed, and may be different from the ones that you want to use with the compiler. To determine the national language code in use, issue the AIX **echo** command:

```
echo $LANG
echo $NLSPATH
```

Configuration File

The configuration file is a file that specifies information that the compiler uses when you invoke it. The default configuration file is provided at installation and stored as **/etc/xlf.cfg**.

The configuration file contains the following attributes:

use	Values for attributes are taken from the named stanza in addition to the default stanza **DEFLT**.

crt	Path name of the object file passed as the first parameter to the linkage editor.
mcrt	Path name of the object file passed as the first parameter to the link editor if you have specified the **–p** option.
gcrt	Path name of the object file passed as the first parameter to the link editor if you have specified the **–pg** option.
xlf	The absolute file name of the compiler.
xlfopt	List of options that if seen on the command line, are directed to the compiler.
as	The absolute file name of the assembler.
asopt	List of options that if seen on the command line, are directed to the assembler.
ld	The absolute file name of the linkage editor.
ldopt	List of options that if seen on the command line, are directed to the linkage editor.
options	A string of option flags, separated by commas, to be processed by **xlf** as if these options were entered on the command line.
fsuffix	The suffix for FORTRAN source programs. The default is **f**.
osuffix	The suffix for object files. The default is **o**.
ssuffix	The suffix for assembler files. The default is **s**.
libraries	Flags separated by commas to be passed to the linkage editor. It specifies the libraries used by the linkage editor at link–edit time for both profiling and non–profiling.
libraries2	Flags separated by commas to be passed to the linkage editor. It specifies the libraries used by the linkage editor at link–edit time. It should only include libraries for which the profiled version exists in the "proflibs" stanza.
proflibs	Flags separated by commas to be passed to the linkage editor when profiling flags are specified. It specifies the profiling libraries used by the linkage editor at link–edit time. It should only include libraries for which the non–profiled version exists in the "libraries2" stanza.

The following is a typical configuration file:

```
* The "libraries2" stanza should ONLY include libraries for which
* profiled versions exist in the "proflibs" stanza – all others go
* in "libraries".

* standard xlf compiler

xlf:  use        = DEFLT
      crt        = /lib/crt0.o
      mcrt       = /lib/mcrt0.o
      gcrt       = /lib/gcrt0.o
      libraries  = –lxlf
      libraries2 = –lc,–lm
      proflibs   = –lc_p,–lm_p

* common definitions
```

```
DEFLT: xlf     = /usr/lpp/xlf/bin/xlfentry
       as      = /bin/as
       ld      = /bin/ld
       options = —estart,—T512,—H512
```

Input Files

The input files to **xlf** are:

- Source Files – **.f**

  All **.f** files are source files for compilation. The **xlf** command sends all source files to the compiler in the order in which they appear. If it cannot find a specified source file, XL FORTRAN produces an error message and the **xlf** command proceeds to the next file if one exists.

- Object Files – **.o**

  All **.o** files are object files. The **xlf** command sends all **.o** files to the linkage editor **ld** at link–edit time unless you specify the **–c** option. After it compiles all the source files, the compiler link–edits the resulting **.o** files with any **.o** files that you specify in the input file list, and produces a single executable output file.

- Assembler Files – **.s**

  The **xlf** command sends all the **.s** files to the assembler (**as**). The assembler output is object files that are sent to the linkage editor at link time.

Output Files

The output files produced include the following:

- Executable Files – **a.out**

  If you do not specify the **–c** compiler option, XL FORTRAN produces an executable file in the current directory. Its default name is **a.out**. If you want to name the executable file explicitly, use the compiler option flag **–o***filename*. If you specify **–c**, XL FORTRAN does not produce an executable file.

- Object Files – *filename***.o**

  If you do specify the **–c** compiler option, the XL FORTRAN compiler produces an object file for each of the **.f** source files, and the assembler produces an object file for each of the **.s** source files. No executable file is produced. The object files have the same prefix name as the source file, and appear in the current directory.

- Listing Files – *filename***.lst**

  By default, no listing is produced unless you specify one or more listing–related compiler options. The listing file has the same file name as the source prefix, but with an extension of **.lst**. XL FORTRAN places listing files in the current directory.

Compiler Options

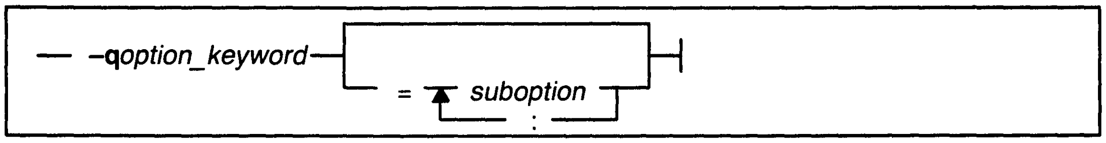
XL FORTRAN provides compiler options to change any of the compiler's default settings. You can specify options on the command line, and they remain in effect for all compilation units in the file, unless the compiler directive **@PROCESS** overrides them. Any options that you can specify on the command line can also appear in the configuration file, and remain in effect for all compilations unless the command line or compiler directive options override them.

Refer to "Summary of the XL FORTRAN Compiler Options" on page 18 and "Detailed Descriptions of the FORTRAN Options" on page 23 for a description of these options.

Specifying Options on the Command Line

There are two methods for specifying compiler options on the command line:

- **–q***option*



  You can specify options on the command line using the **–q***option* format. You can have multiple **–q***options* in the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **–q** in lowercase. You can specify any **–q***option* before or after the file name.

  For example:

  ```
  xlf –qlist –qcharlen=1000 file.f
  xlf file.f –qFIPS –qSTAT
  ```

  Some of the listed options allow you to specify suboptions. These suboptions are indicated on the command line with an equal sign following the **–q***option_keyword.* Suboptions must be separated with a colon (:).

- Single and Multi–letter Flags

  The FORTRAN, C, and Pascal compilers use a number of common conventional flags. These flags are single letters and the XL FORTRAN compiler supports them. Each language has its own set of flags also.

  XL FORTRAN also supports flags directed to the AIX **ld** command. XL FORTRAN passes on those flags directed to **ld** at link time. All single letter flags are case sensitive.

The following items apply for command line options:

- You can specify flags that do not take arguments without separating blanks (with the exception of **–pg** which could be confused as **–p** and **–g**).

  For example:

  ```
  xlf –Ocv file.f
  ```

  has the same effect as

  ```
  xlf –O –c –v file.f
  ```

  You cannot run the **–o** flag together with other flags, because the **–o** option requires an argument.

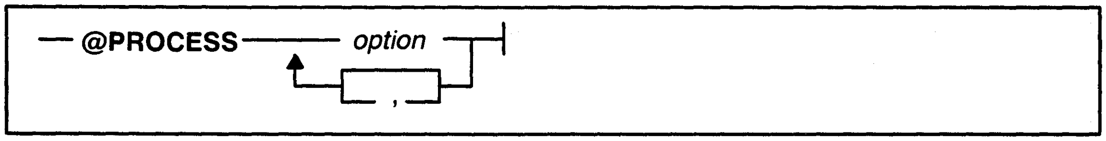
  ```
  xlf –otest –cv test.f
  ```

- You cannot specify flags such as **–qlist** on the command line without separating blanks. (For example, `–qlistqsource` is not valid.)
- Command line options can also follow the file name.

Specifying Options in the Source File

You can use the **@PROCESS** compiler directive in the source file to modify the options specified on the command line and in the configuration file, or to change the default setting temporarily if no command line or configuration file options are in effect.

The syntax of this compiler directive is:



@PROCESS can start in column 1 of the source statement, and you can specify options in columns 9 through 72. If the compiler directive does not start in column 1, it must start at or after column 7 in fixed–form, similar to other FORTRAN statements. In free–form input format, the **@PROCESS** compiler directive can start in any column. Do not place a statement label on the **@PROCESS** compiler directive. You cannot specify an inline comment on the **@PROCESS** compiler directive.

Separate options in the statement with commas or blanks. Option settings you designate with the **@PROCESS** compiler directive are effective only for the compilation unit in which the statement appears. If the file has more than one compilation unit, the option setting is reset to the configuration file setting, to the command line setting, or to the default setting before compiling the next unit.

You can place the **@PROCESS** compiler directive anywhere before the **END** statement. You cannot use it in a continuation line. The **SOURCE** and **NOSOURCE** options are the only options that can appear in an **@PROCESS** compiler directive that occurs after the first statement of the program unit. All other options are only valid when they appear before the first statement of the program unit.

Summary of the XL FORTRAN Compiler Options

The following tables show the compiler options available in the XL FORTRAN compiler that you can enter in the FORTRANsource code using the **@PROCESS** compiler directive, on the command line using the **–q** flags and the single or multi–letter flags, or in the configuration file.

The tables show:

- The option syntax for the **@PROCESS** compiler directive.

  (The uppercase letters in the option keyword represent the valid abbreviation for the option keyword. For example: `OPT` is a valid abbreviation for `OPTimize`.)

- The equivalent **–q** form on the command line, or the corresponding single letter flag, if there is one.

- The default value of that option if you do not specify it on the command line, in an **@PROCESS** compiler directive, or in the configuration file.

- A brief description of the option's effect during compilation.

A detailed description for each compiler option follows the tables.

Note: Before using this compiler, it is important that the programmer first read the descriptions of the compiler options to understand the correct operation and limitations of this product.

Options Describing the Input to the Compiler

Compiler option (@PROCESS Syntax)	Command line flag	Default	Description
FREE FIXED	–k	FIXED	Specifies format of input source program.
MIXED NOMIXED	–U	NOMIXED	Specifies case sensitivity.
DLINES NODLINES	–D	NODLINES	Specifies whether lines with a **D** in column 1 will be compiled or treated as comments.
UNDEF NOUNDEF	–u	NOUNDEF	Indicates implicit typing of variable names.
CHARLEN(*num*)	–qcharlen=*num*	CHARLEN(500)	Sets maximum character variable and character constant length. (1 to 32767)
CI(num_1,num_2,...num_n)	–qci=num_1:num_2: . . . :num_n	no default value	Activates the specified **INCLUDE** compiler directives.
I4 NOI4	–qi4 –qnoi4	I4	Determines how the compiler interprets **INTEGER** and **LOGICAL** specifications and **FUNCTION** statements in which a length is not specified.
DPC NODPC	–qdpc	NODPC	Specifies how the compiler interprets basic real constants.
ONETRIP NOONETRIP	–1	NOONETRIP	Specifies whether **DO** loops in the compiled program will be executed at least once if reached.

BK_SIZE(*num*)
CN_SIZE(*num*)
ST_SIZE(*num*)
NA_SIZE(*num*)
PD_SIZE(*num*)
AUX_SIZE(*num*)
TKQ_SIZE(*num*)
TKA_SIZE(*num*)
SPILLsize(*num*)
	–NB*num*
	–NC*num*
	–ND*num*
	–NN*num*
	–NP*num*
	–NA*num*
	–NQ*num*
	–NT*num*
	–NS*num*
	–N
		BK_SIZE(50)
		CN_SIZE(1024)
		ST_SIZE(2048)
		NA_SIZE(32768)
		PD_SIZE(50)
		AUX_SIZE(8192)
		TKQ_SIZE(10000)
		TKA_SIZE(20000)
		SPILLsize(512)
			Specifies the internal compiler table sizes. The value *num* is used to calculate the size of the table.

DBCS NODBCS	–qdbcs –qnodbcs	NODBCS	Indicates whether character and Hollerith constants can contain DBCS characters.
	–F*config_fn* –F*config_fn:stanza*	–F/etc/xlf.cfg	Names an alternative configuration file for the compiler.
	–B*prefix*		Constructs substitute compiler, assembler, or linkage editor program names.
	–t*programs*		Applies the **–B** flag *prefix* to the designated *programs* (**c**, **a**, **l**).

	–W*program, option$_1$, option$_2$, . . . option$_n$*		Gives the listed options to the compiler program *program*.
	–I*dir*		Determines search path if file name in the **INCLUDE** compiler directive does not start with an absolute path.

Options Affecting the Compiler Object Code to be Produced

Compiler option (@PROCESS Syntax)	Command line flag	Default	Description
OBJect NOOBJect	–qobj –qnoobj	OBJect	Specifies whether an object file will be produced.
CHECK NOCHECK	–C	NOCHECK	Specifies whether run time checking of array bounds and character substring expressions will be performed.
RECUR NORECUR	–qrecur –qnorecur	NORECUR	Specifies whether subprograms may be called recursively.
OPTimize NOOPTimize	–O	NOOPTimize	Specifies whether code optimization during compilation is to be performed.
EXTCHK NOEXTCHK	–qextchk –qnoextchk	NOEXTCHK	Specifies whether type checking information will be set up for external names.
EXTNAME NOEXTNAME	–qextname –qnoextname	NOEXTNAME	Specifies whether compiler generated external names will be postfixed with an underscore.

IEEE(Near)
IEEE(Minus)
IEEE(Plus)
IEEE(Zero)
	–yn
	–ym
	–yp
	–yz
		IEEE(Near)	Indicates rounding mode of constant floating–point expressions at compile time.

RNDSNGL NORNDSNGL	–qrndsngl –qnorndsngl	NORNDSNGL	Specifies whether **REAL*4** floating–point expressions will be rounded to single precision.
RRM NORRM	–qrrm	NORRM	Specifies run–time rounding mode.
FOLD NOFOLD	–qfold –qnofold	FOLD	Specifies whether constant floating–point expressions are to be evaluated at compile time.

MAF NOMAF	–qmaf –qnomaf	MAF	Specifies whether multiply–add instructions will be generated.
DBG NODBG	–g	NODBG	Specifies whether debug information will be generated for use by **dbx**.
XFLAG=DD24	–qxflag=dd24		If all single precision floating point overflows must be detected, use this option. See the detailed description of this option.

Options Describing the Compiler Output

Compiler option (@PROCESS Syntax)	Command line flag	Default	Description
FIPS NOFIPS	–qfips –qnofips	NOFIPS	Specifies whether flagging of full ANSI FORTRAN 77 standard language will take place.
SAA NOSAA	–qsaa –qnosaa	NOSAA	Specifies whether SAA FORTRAN standard language flagging will take place.
SOURCE NOSOURCE	–qsource –qnosource	NOSOURCE	Specifies whether a source listing will be produced.
XREF NOXREF XREF(FULL)	–qxref –qnoxref –qxref=full	NOXREF	Specifies whether a cross–reference listing is to be produced.
ATTR NOATTR ATTR(FULL)	–qattr –qnoattr –qattr=full	NOATTR	Specifies whether an attribute listing will be produced.
LIST NOLIST	–qlist –qnolist	NOLIST	Specifies whether an object listing is to be produced.
LISTOPT NOLISTOPT	–qlistopt –qnolistopt	NOLISTOPT	Specifies whether the settings of all options are to be displayed in the listing.
STATs NOSTATs	–qstat –qnostat	NOSTATs	Specifies whether table size and timing statistics will be reported in the listing.
PHSINFO NOPHSINFO	–qphsinfo –qnophsinfo	NOPHSINFO	Specifies whether phase timing information will be displayed at the terminal.
FLAG(*sev1*,*sev2*)	–qflag=*sev1*:*sev2* –w	FLAG(L,L)	Specifies severity level of diagnostic messages to be reported.

HALT(*sev*)	–qhalt=*sev*	HALT(U)	Stops the compiler after the first phase depending on the severity of compile messages.
	–c	compile and link–edit	Requests that the object file not be sent to the linkage–editor.
	–Q+*names* –Q–*names* –Q –Q!	no inlining	Specifies whether subroutine inlining is to be performed.
	–qnoprint	listing suppression	Requests listing suppression.

Options Used for Debugging

Compiler option (@PROCESS Syntax)	Command line flag	Default	Description
	–v	no verbose	Generates compiler progress information.
	–#	no trace	Displays the same information as in **–v** without invoking the compiler.
	–p –pg	no profiling	Sets up the object file for profiling.

Options Used for the Linkage Editor

Compiler option (@PROCESS Syntax)	Command line flag	Default	Description
	–o*name*	a.out	Specifies a name for the object module.
	–l*key*	libraries listed in xlf.cfg are searched	Searches the specified library file.
	–L*dir*	standard directories	Looks in *dir* for files specified by the **–l** keys.

Conflicting Options

The following table shows:

- Compiler options which conflict
- The resolution which is performed by the compiler when such options are specified at any given level.

Option	Conflicting Options	Options Assumed
FIPS/SAA	FLAG(N,N) where N=S, E, or Q	FLAG(N,N)
HALT	OBJ	HALT
HALT	NOOBJ	NOOBJ
–qnoprint	XREF/ATTR/SOURCE/LISTOPT/LIST/STAT	–qnoprint
XREF	XREF(FULL)	XREF(FULL)
ATTR	ATTR(FULL)	ATTR(FULL)
–p	–pg	Last option specified

If more than one variation of the same option is specified (with the exception of **XREF** and **ATTR** listed above), then the setting of the option specified last will be assumed.

If a command line flag is valid for more than one compiler program (for example, compiler, linkage editor, assembler), you must specify it in `xlfopt`, `ldopt`, or `asopt` in the configuration file. The command line flags must appear in the order that they are to be directed to the appropriate compiler program.

Detailed Descriptions of the Options

The following information shows the details associated with the compiler options described in the previous tables. All options displayed in capitals can be used with the **@PROCESS** compiler directive within your FORTRAN source program. You can specify options shown in lowercase directly on the command line. The default value of that option if it is not specified in the configuration file, on the command line, or in an **@PROCESS** compiler directive is underlined.

Options Describing the Input to the Compiler

FREE | <u>FIXED</u> | –k

FREE or **FIXED** indicates whether the input source program is in free format or in fixed format.

MIXED | <u>NOMIXED</u> | –U

MIXED specifies case sensitivity. If **MIXED** is specified, the source is not folded and identifiers are case sensitive. You must enter keywords in lowercase or XL FORTRAN treats them as identifiers. If **NOMIXED** is specified, the source is folded to lowercase.

DLINES | <u>NODLINES</u> | –D

If **DLINES** is specified, the lines that have a **D** in column 1 are compiled. If **NODLINES** or nothing is specified, those lines are treated as comments.

UNDEF | <u>NOUNDEF</u> | –u

UNDEF specifies no implicit typing of variable names. It has the same effect as the **IMPLICIT NONE** statement. **NOUNDEF** specifies implicit typing.

CHARLEN(*num* | 500) | –qcharlen=*num*

CHARLEN specifies the maximum length permitted for any **CHARACTER** variable, **CHARACTER** array element, **CHARACTER** function name, **CHARACTER** entry name, **CHARACTER** named constant, or **CHARACTER** literal string (up to 32767).

CI(*num*$_1$,*num*$_2$,...*num*$_n$) | –qci=*num*$_1$:*num*$_2$:...:*num*$_n$

CI specifies the identification numbers of the **INCLUDE** files to be processed (where $1 <= num_n <= 255$ and $1 <= n <= 255$). The set of identification numbers recognized is the union of all identification numbers specified on all occurrences of the **CI** option.

I4 | NOI4 | –qi4 | –qnoi4

These options specify how the compiler interprets **INTEGER** and **LOGICAL** specification statements and **FUNCTION** statements in which a length is not specified. This option is to assist migration from 16–bit machines. The default is **I4**, which causes the compiler to interpret **INTEGER** and **LOGICAL** specifications as **INTEGER*4** and **LOGICAL*4** respectively. If **NOI4** is specified, the compiler interprets **INTEGER** and **LOGICAL** specifications as **INTEGER*2** and **LOGICAL*2** respectively.

Integer constants in the range –32768 to 32767 are interpreted as having length 2 when **NOI4** is specified and length 4 otherwise. When **NOI4** is specified, such integer constants must not be passed to dummy arguments of type **INTEGER*4** without being explicitly defined as **INTEGER*4** in a **PARAMETER** statement.

Logical constants **.TRUE.** and **.FALSE.** are interpreted as having length 2 when **NOI4** is specified and length 4 otherwise. When **NOI4** is specified, such logical constants must not be passed to dummy arguments of type **LOGICAL*4** without being explicitly defined as **LOGICAL*4** in a **PARAMETER** statement. When **NOI4** is specified, logical expressions have type size equal to the largest type size of its operands. If **I4** is specified, all logical expressions have type **LOGICAL*4**.

DPC | NODPC | –qdpc

When **DPC** is specified, all basic real constants (for example, 1.1) are treated as double precision constants.

ONETRIP | NOONETRIP | –1

ONETRIP specifies that **DO** loops in the compiled program are to be executed at least once, if reached, even if the iteration count is 0. This option provides compatibility with ANSI FORTRAN 66, whereas in ANSI FORTRAN 77, **DO** loops are not performed if the iteration count is 0. If **NOONETRIP** is specified, **DO** loops will not be executed if the iteration count is 0.

–N*tnum*

Specifies the internal compiler table sizes. The value *t* indicates the type of the table. The value *num* is used to calculate the size of the table. The default table sizes for each type are:

B 50	Maximum number of nested **IF** or **DO** blocks.
C 1024	Maximum number of constants.
D 2048	Maximum number of variables.
N 32768	Maximum number of bytes to store the names of variables.
P 50	Maximum number of subprograms.
A 8192	Maximum number of dictionary auxiliary table entries.
Q 10000	Maximum number of tokens in the source file.
T 20000	Maximum number of bytes to store the tokens.
S 512	Maximum number of spill variables.

–N without any arguments specifies that help information for the **–N***tnum* option should be displayed.

DBCS | NODBCS | –qdbcs | –qnodbcs

Indicates to the compiler whether character and Hollerith constants can contain Double Byte Character Set (DBCS) characters.

–F*config_fn* **| –F***config_fn:stanza* **| –F***/etc/xlf.cfg*

The configuration file specifies the location of various files required by the compiler. A default configuration file (**/etc/xlf.cfg**) is supplied at installation time. This option allows an alternative configuration file to be specified. It also specifies what stanza to use within the configuration file. (See "Configuration File" on page 14 for more information.)

–B*prefix*

This option is used to construct substitute compiler, assembler, or linkage editor program names. *prefix* defines part of a path name to the new programs. To form the complete path name for each program, **xlf** adds *prefix* to the standard program names. The standard program names for the compiler, assembler, and linkage editor are **xlfentry**, **as**, and **ld** respectively.

–t*programs*

Applies the **–B** flag *prefix* to the designated *programs*. *programs* can be one or more of **c**, **a**, or **l** corresponding to compiler, assembler, and linkage editor respectively.

–W*program,option$_1$,option$_2$, . . . option$_n$*

Gives the listed options to the compiler program *program*. *program* is **c**, **a**, or **l** corresponding to compiler, assembler, and linkage editor respectively. Valid compiler options are the **@PROCESS** options listed in "Summary of the XL FORTRAN Compiler Options" on page 18. For more information on the valid assembler and linkage editor options see the *AIX Commands Reference for IBM RISC System/6000*, SC23–2199.

–I*dir*

Specifies the search path if the file name in the **INCLUDE** compiler directive does not start with an absolute path. *dir* must be a valid path name (for example **/u/dir** or **/tmp** or **./subdir**). The compiler appends a / to the *dir* and then concatenates with the file name before making a search. If more than one **–I** option is specified in the command line, files are searched in the order of the *dir* as they appear on the command line.

Options Affecting the Compiler Object Code to be Produced

OBJect | NOOBJect | –qobj | –qnoobj

Specifies whether or not you want an object file produced. If **NOOBJect** is specified, only the first phase of the compiler is completed. See "Invoking the Compiler" on page 13 for information on the compiler phases.

CHECK | NOCHECK | –C

CHECK specifies that run–time checking of array bounds and character substring expressions is to be performed. **NOCHECK** turns this checking off.

RECUR | NORECUR | –qrecur | –qnorecur

If the **RECUR** option is specified for a subprogram, that subprogram may call itself recursively. If a subprogram is to be called recursively at any point in the program, the **RECUR** option must be specified for that subprogram.

OPTimize | NOOPTimize | –O

Specifies whether code optimization is to be performed during the compilation.

NOOPT	Performs no optimization of generated code.
OPT	Performs optimization using some or all of the following techniques:

- Value numbering
- Straightening
- Common expression elimination
- Code motion
- Reassociation and strength reduction
- Constant propagation
- Store motion
- Dead store elimination
- Dead code elimination
- Inlining (if the **–Q** compiler option is specified)
- Global register allocation
- Instruction scheduling.

EXTCHK | NOEXTCHK | –qextchk | –qnoextchk

EXTCHK specifies that type checking information is to be set up for common blocks and procedure references, to be later used by **ld** for the purpose of detecting mismatches across compilation units. **EXTCHK** verifies that actual arguments agree in type, passing mode, and class. It also checks actual and dummy arguments and attributes for agreement and verifies that declarations of common blocks are consistent.

EXTNAME | NOEXTNAME | –qextname | –qnoextname

EXTNAME specifies that all compiler generated external names, with the exception of main program names, are postfixed with an underscore. The **EXTNAME** option is to aid in porting RT code to the RISC System/6000 computer if it contains the following:

- C, Assembler, or Pascal routines, referenced from FORTRAN, that contain an underscore at the end of the routine name
- C, Assembler, or Pascal routines, calling FORTRAN routine that contains an underscore at the end of the routine name
- Subroutines or functions in FORTRAN source code that are named **main, MAIN,** or with system command or subroutine names
- C, Assembler, or Pascal external or global data items that are shared with a FORTRAN routine that contains an underscore at the end of the data name.

IEEE(Near | Minus | Plus | Zero)| –yn | –ym | –yp | –yz

Specifies the rounding of constant floating–point expressions at compile time:

n	round to nearest
m	round toward minus infinity
p	round toward plus infinity
z	round toward zero.

RNDSNGL | NORNDSNGL | –qrndsngl | –qnorndsngl

Specifies strict adherence to the IEEE standard in that the result of each single precision floating–point operation will be rounded to single precision.

RRM | NORRM | –qrrm

Specifies the run–time rounding mode. This option is used if the run–time rounding mode is round to +infinity, –infinity, or is not known.

FOLD | NOFOLD | –qfold | –qnofold

FOLD specifies that constant floating–point expressions are to be evaluated at compile time.

MAF | NOMAF | –qmaf | –qnomaf

Specifies whether or not the compiler is to generate multiply–add instructions. Multiply–add instructions may affect the precision of floating–point intermediate results, by giving better performance and better accuracy.

DBG | NODBG | –g

Specifies that debug information is to be generated for use by the symbolic debugger.

XFLAG=DD24 | –qxflag=dd24

Generates floating point no–op instructions to cause detection of overflow in rounding floating point intermediate results to single precision. See Appendix F, "Single Precision Floating Point Overflow" for more information.

Options Describing the Compiler Output

FIPS | NOFIPS | –qfips | –qnofips

Specifies whether or not Full ANSI FORTRAN 77 standard language flagging is to be performed.

SAA | NOSAA | –qsaa | –qnosaa

Specifies whether or not SAA FORTRAN standard language flagging is to be performed.

SOURCE | NOSOURCE | –qsource | –qnosource

Specifies whether or not the source listing is to be produced. Specifying **SOURCE** as an option implies that a listing will be produced, unless **–qnoprint** is specified. No listing is produced by default. Parts of the source can be selectively printed by using **SOURCE** and **NOSOURCE** throughout the program.

XREF | NOXREF | XREF(FULL) | –qxref | –qnoxref | –qxref=full

Specifies whether or not a cross–reference listing is to be produced. If only **XREF** is specified, only identifiers that are used will be reported. If **XREF(FULL)** is specified, all identifiers that appear in the program, whether used or not, will be reported. Both **XREF** and **XREF(FULL)** imply that a listing will be produced, unless **–qnoprint** is specified. No listing is produced by default.

ATTR | NOATTR | ATTR(FULL) | –qattr | –qnoattr | –qattr=full

Specifies that an attribute listing is to be produced (consisting of the attributes of names). If only **ATTR** is specified, only identifiers that are referenced will be reported. If **ATTR(FULL)** is specified, all identifiers, whether referenced or not, will be reported. Both **ATTR** and **ATTR(FULL)** imply that a listing will be produced unless **–qnoprint** is specified. No listing is produced by default.

LIST | NOLIST | –qlist | –qnolist

Specifies whether or not an object code listing is to be produced. **LIST** implies that a listing will be produced, unless **–qnoprint** is specified. No listing is produced by default.

LISTOPT | NOLISTOPT | –qlistopt | –qnolistopt

Specifies whether or not the settings of all options are to be displayed as part of the listing. (The default is to display the settings of specified options only.) **LISTOPT** implies that a listing will be produced, unless **–qnoprint** is specified. No listing is produced by default.

STATs | NOSTATs | –qstat | –qnostat

Specifies whether or not table size statistics and timing statistics are to be reported as part of the listing. **STAT** implies that a listing will be produced, unless **–qnoprint** is specified. No listing is produced by default.

PHSINFO | NOPHSINFO | –qphsinfo | –qnophsinfo

Specifies whether or not phase timing information is to be displayed.

FLAG(*sev1,sev2*) | FLAG(L,L) | –qflag=*sev1:sev2* | –w

Specifies the severity level of diagnostic messages to be reported:

sev1: message level reported on listing
sev2: message level reported on terminal.

where *sev1/sev2* is **I**, **L**,**W**, **E**, **S**, or **Q** meaning informational, language, warning, error, or severe error, and **Q** means do not report any messages at all.

Both *sev1* and *sev2* must be specified. Messages of the specified severity level or higher will be reported. When **–w** is specified on the command line, it is equivalent to specifying **FLAG(E,E)** (that is, warning and informational messages will be suppressed).

HALT(*sev*) | HALT(U) | –qhalt=*sev*

Stops the compiler after the first phase if the maximum severity of messages encountered at compile time equals or exceeds the specified severity.

sev	is **I**, **L**, **W**, **E**, **S**, or **U** meaning informational, language, warning, error, severe error, or unrecoverable error.

–c

Does not send the completed object file to the **ld** command for link–editing. With this flag, the output is a **.o** file for each source file.

***–Q+names* | *–Q–names* | –Q | –Q!**

Inlining is performed if possible. Instead of the normal subprogram call, inline code is produced for references to subprograms that are fewer than 100 intermediate instructions, and are in the same file as the calling program. *+names* and *–names* specify the names of subprograms that are and are not, respectively, to be inlined. The **–Q** flag without any list will cause all appropriate subprograms to be inlined. –**Q!** causes no inlining to be done. **OPT** must be specified for inlining to take effect. If inlining has been specified, the following options cannot be specified in an **@PROCESS** statement unless that statement occurs before the first compilation unit: **OBJect**, **HALT**, **MAF**, **RNDSNGL**, **LIST**, **PHSINFO**, and the table size options.

–qnoprint

Suppresses the production of listings. If there are any options specified that cause a listing to be produced, the **–qnoprint** option sends the listing file to **/dev/null**, which effectively overrides all listing options.

Options Used for Debugging

–v

Instructs the compiler to generate information on the progress of the compilation.

–#

Displays the same information as with **–v** without invoking the compiler.

–p | –pg

Sets up the object file for profiling.

–p prepares the program so that the AIX **prof** command can generate a run–time profile. The compiler produces code that counts the number of times each routine is called. If programs are sent to **ld**, the compiler replaces the startup routine with one that calls the monitor subroutine at the start and writes a **mon.out** file when the program ends normally. You can use the **prof** command to generate a run–time profile.

–pg is like **–p**, but invokes a run–time recording mechanism that keeps more extensive statistics and produces a **gmon.out** file when the program ends normally. You can then use the **gprof** command to generate a run–time profile.

For more information on profiling, the **ld**, **prof**, and **gprof** commands, see the *AIX Commands Reference for IBM RISC System/6000*, SC23–2199.

Options Used for the Linkage Editor

The following are common **ld** options that can be specified in the **xlf** command. Other **ld** options can also be specified.

–o*name* | a.out

Specifies a name for the object module. This is an **ld** option, so conflicts are handled by **ld**.

–l*key*

Searches the specified library file, where *key* selects the file **lib***key***.a**. This is an **ld** option, so conflicts are handled by **ld**.

–L*dir*

Looks in *dir* for files specified by the **–l** keys. This is an **ld** option, so conflicts are handled by **ld**.

Invoking the Linkage Editor

If you specify **–c** as a compiler option, XL FORTRAN only compiles the source program and creates an object file. To perform the linkage editor phase, invoke the linker using the **ld** command, or issue the **xlf** command a second time without the *–c* option and specifying the desired object file (**.o**) names.

Note: See the *AIX Commands Reference for IBM RISC System/6000*, SC23–2199. for a description of the linkage editor and link–edit flags.

Running the Program

You can run the program by entering the path name and file name of the executable object file and any desired run–time parameters on the command line. If the **–o***name* compiler option has been specified, the file name will be *name*. The default file name is **a.out**.

The following environment variables are recognized at run time:

- **xrf_messages**

  If **xrf_messages** is set to "no", run–time messages are suppressed. If you do not set it or you specify anything else, run–time messages are issued to standard error.

- **TMPDIR**

  This environment variable allows you to choose a directory to place any temporary files that may be created when when opening a file whose **STATUS='SCRATCH'**. If **TMPDIR** has not been set, temporary files will be placed in */tmp*.

The following is an example of how to set environment variables:

- From the C shell:
```
setenv xrf_messages no
setenv TMPDIR /u/joe/temp
```

- From the Bourne shell:
```
xrf_messages=no
 TMPDIR=/u/joe/temp
export xrf_messages TMPDIR
```

See "Environment Variables" on page 43 for more information. Also, for more information on **a.out** see the *AIX Commands Reference for IBM RISC System/6000*, SC23–2199.

FORTRAN Exception Handling

The following cases will cause an exception at run time:

1. Fixed–point division by zero
2. Character substring expression out of bounds if you specify the **CHECK** option at compile time
3. Array subscript out of bounds if you specify the **CHECK** option at compile time
4. The flow of control in the program reaches a location for which a semantic error with severity of **S** was issued when the program was compiled.

If you call **SIGNAL** (`CALL SIGNAL(SIGTRAP,xl__trce)`) to install the default exception handler before the exception occurs, a diagnostic message and a traceback showing the offset number of each routine called leading to the exception are written to standard error after the exception occurs. If the exception handler is not installed, a core image file is produced.

You can then use the **dbx** symbolic debugger to determine the error. **dbx** provides a specific error message describing the cause of the exception. The **dbx where** subcommand or **dbx trace** subcommand provides a complete traceback showing the FORTRAN source line number and the sequence of instructions leading to the exception. See the *AIX Commands Reference for RISC System/6000*, SC23–2199 for information about the **dbx** debugger.

The XL FORTRAN Run Time Environment

Object code produced by the XL FORTRAN compiler may call several run–time subprograms. The XL FORTRAN Run Time Environment includes a library of run–time subprograms (**libxlf.a**) and also facilities for producing run–time diagnostic messages in the national language appropriate for your system. Normally, you cannot run object code produced by the FORTRAN compiler without the FORTRAN Run Time Environment.

External Names in the Run Time Environment

Run–time subprograms are collected into libraries. When you use the **xlf** command without the **–c** option, the compiler invokes the linkage editor and gives it the names of the libraries that contain run–time subprograms called by FORTRAN object code.

The names of these run–time subprograms are external symbols. When object code produced by the FORTRAN compiler calls a run–time subprogram, the **.o** object code file contains an external symbol reference to the name of the subprogram. A library contains an external symbol definition for the subprogram. The linkage editor resolves the run–time subprogram call with the subprogram definition.

You should avoid using names in your XL FORTRAN program that conflict with names of run–time subprograms. Conflict can arise under two conditions:

- The name of subroutine, function, or common block defined in a FORTRAN has the same name as a library subprogram.
- The FORTRAN program calls a subroutine or function with the same name as a library subprogram but does not supply a definition for the called subroutine or function.

Avoiding the Use of Run–Time Subprogram Names

If you define a subroutine, function, or common block with the same name as a run–time subprogram, your definition of that name may be used in place of the run–time subprogram, or it may cause a link–edit error. To avoid conflicts with the names of the external symbols in the XL FORTRAN library, the names you use should not begin with the underscore (_) or the number sign (#), and should not be names that are the same as the service and utility subprograms that are provided in the library. (See Appendix B of the *Reference Manual for IBM AIX XL FORTRAN Compiler/6000* for a list of these subprograms.) You should also avoid naming a subroutine or function **main** since XL FORTRAN defines an entry point **main** to start your program.

Object code produced by the XL FORTRAN compiler can call some run–time routines from libraries other than the XL FORTRAN Run Time Environment.

References to Undefined Run–Time Subprogram Names

You should not leave names of subroutines or functions undefined. If your FORTRAN code calls a subroutine or function without defining it, the linkage editor will attempt to resolve the reference in the XL FORTRAN Run Time Environment or in any of the other libraries that XL FORTRAN uses for run–time routines. Resolution by the linkage editor may appear to be successful but the program result may be unpredictable.

Note: XL FORTRAN uses some subprograms from the C (**libc.a**) and mathematics (**libm.a**) libraries. Each of these libraries contains several subprograms with common spellings. (For example, **read** and **write**.)

AIX Shared Libraries

The run–time library included in the XL FORTRAN Run Time Environment is an AIX shared library. Shared libraries are processed by the linkage editor to resolve all references to external names. This limits the possibility of problems between user–defined subroutine and function names and the names of any routines that are called by run–time subprograms.

For example, when you invoke the FORTRAN **PAUSE** statement, the XL FORTRAN compiler generates a call to the run–time subroutine **#PAUSECHR**. This run–time subprogram in turn calls AIX system routines to write the given character constant to standard output. All calls within **#PAUSECHR** are resolved within the XL FORTRAN Run Time Environment and other libraries. This allows a FORTRAN program to define and call a routine with the same name as any of the system routines called by **#PAUSECHR**, and that name will not conflict with any calls in **#PAUSECHR**.

The description of the **ld** command in the *AIX Commands Reference for RISC System/6000*, SC23–2199 contains further details about creating and using shared libraries.

Chapter 4. Input/Output

The following chapter discusses support for the IBM AIX Version 3 for RISC System/6000 file system by the XL FORTRAN compiler and Run Time Environment.

File Formats

XL FORTRAN implements files in the following manner:

- Sequential unformatted files:

  The requirements on these files make it unlikely that you will read or write them by any means other than FORTRAN input/output. A 4–byte integer containing the length of the record precedes and follows each record.

- Sequential formatted files:

  The XL FORTRAN input/output system breaks sequential formatted files into records while reading, by using each new–line character as a record separator.

  On output, the input/output system writes a new–line character at the end of each record. Programs can also write new–line characters for themselves. This practice is not recommended, because the effect is that the single record that appears to be written is treated as more than one record when being read or back spaced over.

- Direct files:

  XL FORTRAN simulates direct files with AIX files containing a single record, whose length is a multiple of the record length of the file. You must specify, in an **OPEN** statement, the record length (**RECL**) of the direct file. XL FORTRAN uses this record length to distinguish records from each other.

  For example, the third record of a direct file of record length 100 bytes would start at byte 200 of the single record of an AIX file and end at byte 299.

  If the length of the record of a direct file is greater than the total amount of data you want to write to the record, XL FORTRAN pads the record on the right with blanks (X'20').

File Names

A valid AIX file name must have a full path name of total length <=2048 characters, with each file name being <=256 characters long (though you need not specify the full path name).

You must specify a valid AIX file name in such places as:

- The **FILE**= specifier of the **INQUIRE** statement
- The **FILE**= specifier of the **OPEN** statement
- The **INCLUDE** compiler directive.

Preconnected Files

The system preconnects units 0, 5, and 6 when a program starts:

- Unit 0 is the standard error
- Unit 5 is the standard input for sequential formatted input/output
- Unit 6 is the standard output for sequential formatted input/output.

All other units are also preconnected when run time begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, and XL FORTRAN does not create them unless you use their units. The default connection is for sequential formatted input/output.

Note: Because unit 0 is preconnected for standard error, you cannot use it for the following statements: **OPEN**, **CLOSE**, **ENDFILE**, **BACKSPACE**, **REWIND**, and direct input/output.

File Positioning

ANSI X3.9–1978 does not specify the initial position of a file that is explicitly opened for sequential input/output. Therefore, XL FORTRAN has adopted the following conventions.

On an explicit **OPEN** (by an **OPEN** statement):

- If the file **STATUS** is **NEW** or **SCRATCH**, the file is positioned at the beginning.
- If **STATUS** = **'OLD'** is specified, the file is positioned at the end.
  - If the next operation is **WRITE**, the next record is appended to the file.
  - If the next operation is **READ**, the file is repositioned to the beginning so that the first record is read.
- If **STATUS** = **'UNKNOWN'** is specified, and the file exists, the file is positioned as if **STATUS** = **'OLD'** were specified. Otherwise, the file is positioned as if **STATUS** = **'NEW'** were specified.

The implementation of implicit **OPEN** is equivalent to an explicit **OPEN** with **STATUS** = **'NEW'** (that is, the file is positioned at the beginning).

- If the first input/output operation on the file is **READ**, it will read the first record of the file.
- If the first input/output operation on the file is **WRITE**, it will overwrite the first record of the file.

Therefore, to append to an existing file, the file must be explicitly opened with an **OPEN** statement with **STATUS** = **'OLD'** specified before the **WRITE** statement is performed.

Chapter 5. Optimization

The following chapter discusses optimization techniques used by XL FORTRAN. It also outlines some programming techniques that you can employ to take advantage of the optimization features of the compiler.

Optimization Levels

Optimization requires additional compile time, but usually results in reduced run time.

XL FORTRAN allows you to select whether or not you want optimization to be performed at compile time. The **NOOPT** compiler option (which is the default) disables the optimization of your program. The **OPT** option performs the optimization techniques outlined below:

- Value numbering
- Straightening
- Common expression elimination
- Code motion
- Reassociation and strength reduction
- Constant propagation
- Store motion
- Dead store elimination
- Dead code elimination
- Inlining (if the **–Q** compiler option is specified)
- Global register allocation
- Instruction scheduling.

NOOPT is the recommended level of optimization for a program you are debugging, or compiling to check syntax. It provides the fastest compile time, but the least efficient run time. The compiler may perform some minor optimizations.

OPT performs control and data flow analysis for the entire program unit. This analysis allows optimizations such as common expression elimination, strength reduction, code motion, and global register assignment. Particular attention is paid to innermost loops and to subscript address calculations. Variables are retained in registers where possible to eliminate unnecessary loads and stores.

Optimization does not move any code out of a loop that might cause an exception unless the exception will occur anyway. For example, in the loop:

```
      DO 10 J=1,N
        IF (K.NE.0) M(J)=N/K
10    CONTINUE
```

code evaluating the expression `N/K` could be moved outside the loop, because it is invariant for each iteration of the loop. However, it will not be moved because `K` could be 0.

Optimization Techniques

Several techniques are used by the optimizer:

Value Numbering

Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening

Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common Expression Elimination

In common expressions the same value is recalculated in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
10      A=C+D
        .
        .
        .
20      F=C+D+E
```

the common expression `C+D` is saved from its first evaluation at statement `10`, and is used at statement `20` in determining the value of `F`.

Code Motion

If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Reassociation

Reassociation rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

Strength Reduction

Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant Propagation

Constants used in an expression are combined and new ones generated. Some mode conversions are done and compile–time evaluation of some intrinsic functions takes place.

Store Motion

Store motion moves store instructions out of existing loops.

Dead Store Elimination

The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead Code Elimination

The compiler may eliminate code for calculations found to be unnecessary. Other optimization techniques may cause code to become dead.

Inlining

Inlining causes all program units within a single source file to be compiled at one time. It replaces subprogram calls with the actual code of the subprogram.

Global Register Allocation

Variables and expressions are allocated to available hardware registers by coloring.

Instruction Scheduling

Reorders instructions to minimize execution time.

Debugging Optimized Code

Debugging optimized programs presents special problems. Changes made by optimization can be confusing.

Use debugging techniques that rely on examining values in storage with caution. A common expression evaluation may have been deleted or moved. Note that variables may have been assigned to a register, and will not appear in storage.

Programs that appear to work properly when compiled with no optimization may fail when compiled with **OPT**. This is often caused by program variables that have not been initialized. If a program that worked with **NOOPT** fails when compiled with **OPT**, it is a good idea to look at the cross–reference listing. Check for variables that are fetched but never set, and for program logic that allows a variable to be used before being set.

Optimized code can fail if a FORTRAN program contains invalid code. For example, if the FORTRAN program passes an actual argument that also appears in a common block in the called routine, or if two or more dummy arguments are associated with the same actual argument.

Increasing Optimization of Your Program

The following section contains suggestions on how to use the optimization features.

Optimization Recommendations

- Use **NOOPT** during program development for syntax checking, testing, and debugging purposes. Debugging programs with dbx is straightforward, with none of the side effects of optimization.
- Use **OPT** once a program has been debugged. If the program is to be run more than once, or if the program takes more than a few CPU seconds to run, the optimization savings at run time may exceed the cost of compiling with **OPT** in effect.
- More virtual storage, system page space, and longer compilation times are required for optimization. Depending on the complexity and number of loops in the program (opportunities for optimization), and the number of identifiable elementary expressions, the compilation time may increase greatly. You may have to compile larger or complex programs with **NOOPT** if they fail to compile with **OPT** because of storage exhaustion.

Programming Recommendations

The following section contains programming suggestions to take advantage of the optimization features.

Input/Output

Optimization has little effect on the run time of input/output statements. Here are some guidelines to improve input/output run–time performance:

- Unformatted input/output takes less processing time and uses less storage than formatted input/output. Unformatted input/output also maintains the precision of the data items being processed.
- Use an implied **DO** in input/output statements instead of a **DO** loop containing the input/output statement. For example:

```
      DIMENSION A(200)
      WRITE(3) (A(I),I=1,100)
```

  is more efficient than:

```
      DIMENSION A(200)
      DO I=1,100
        WRITE(3) (A(I))
      END DO
```

  If the entire array is to be read or written in the storage order, use the array name without the implied **DO** adjusting the format appropriately. For example:

```
      DIMENSION A(200)
      WRITE(3,10) (A(I),I=1,200)    ! This writes the contents of A.
10    FORMAT (' ',10F6.2)
```

Variables

- Certain variables cannot always be optimized:
  - Variables in input statements and in **CALL** statement argument lists are less likely to be optimized.
  - Variables in common blocks cannot be optimized across subroutine calls.

  Do not use **DO** loop indexes in any of the above ways.
- Each reference to a variable in a common block requires that the address of the common block be in a register. This is the basis for the following recommendations:
  - Minimize the number of common blocks. Group concurrently referenced variables into the same common block. For example:

```
      COMMON /X/ A
      COMMON /Y/ B
      COMMON /Z/ C
      A=B+C                    ! Three registers required

      COMMON /Q/ A,B,C
      A=B+C                    ! One register required
```

  - Place scalar variables before arrays in a given common block. For example:

```
      COMMON /Z/ X(50000),Y
      X(1)=Y                   ! Two registers required

      COMMON /Z/ Y,X(50000)
      X(1)=Y                   ! One register required
```

  - Place small arrays before large ones. All the scalar variables and the first few arrays can then be addressed through one address constant. The subsequent larger arrays probably each need a separate address constant.
  - Assign frequently referenced scalar variables in a common block to a local variable. References to the local variable will not require the common block address to be in a register. Be sure to assign the value back to the common block variable at the end of processing.

- **REAL*8** values are handled more efficiently than **REAL*4** values.
- Items that are incorrectly aligned in common blocks adversely affect the performance of the program.

Subroutine Arguments

- Entry into a subprogram associates actual arguments with the dummy arguments in the referenced subprogram or **ENTRY** statement. Therefore, all appearances of these arguments in the whole subprogram become associated with actual arguments. New values will not be transmitted for arguments not listed in the **ENTRY** statement.

  The only way to guarantee that you will get the current value of an argument is to have the argument listed on the **ENTRY** statement through which you invoke the subprogram.
- Large numbers of actual arguments can be dealt with more efficiently if they are in common blocks.

Constant Operands

- Define constant operands as local variables. The compiler recognizes only local variables as having a constant value. (Operands in common or in an argument list can change, and cannot be optimized as fully.)

Arrays

- Expand some smaller arrays to match the dimensions of the arrays they interact with. If the arrays in a subprogram, block of code, loop, or nest of loops have the same shape, the compiler calculates one subscript and uses it for all the arrays. The compiler can maintain one index for all the arrays defined as having the same dimensions.
- Subscripting of adjustable dimensioned arrays requires additional indexing computations. Using an adjustable dimensioned array as a dummy argument, requires an additional calculation on each entrance into the subprogram. To lessen the amount of extra processing, use the following technique:
  - If indexing can be varied in the low–order dimensions, make the adjustable dimensions of an array the high–order dimensions. This reduces the number of computations needed for indexing the array, as shown:

```
SUBROUTINE EXEC(Z,N)     ! Computation not required
REAL*8 Z(9,N)
Z(I,5)=A

SUBROUTINE EXEC(Z,N)     ! Computation (I*N) required
REAL*8 Z(N,9)
Z(5,I)=A
```

- Initialize large arrays using a **DO** loop. You get faster overall run time and use less storage than if you initialize using a **DATA** statement. For example, the following statements:

```
DOUBLE PRECISION A(5000)
DATA A/5000*1.0D0/
```

  generate 40000 bytes of object module information. The 5000 copies of `1.0` are placed in the object module, placed in the load module, and fetched into storage when you run the program.

Expressions

- If components of an expression are duplicate expressions, code them either at the left end of the expression, or within parentheses. For example:

```
A=B*(X*Y*Z)          ! Duplicates recognized
C=X*Y*Z*D
E=F+(X+Y)
G=X+Y+H

A=B*X*Y*Z           ! No duplicates recognized
C=X*Y*Z*D
E=F+X+Y
G=X+Y+H
```

The compiler can recognize `X*Y*Z` and `X+Y` as duplicate expressions because they're either coded in parentheses or coded at the left end of the expression.

- When components of an expression in a loop are constant, code the expressions either at the left end of the expression, or within parentheses. If `C`, `D`, and `E` are constant and `V`, `W`, and `X` are variable, the following examples show the difference in evaluation:

```
V*W*X*(C*D*E)     ! Constant expressions recognized
C+D+E+V+W+X

V*W*X*C*D*E      ! Constant expressions not recognized
V+W+X+C+D+E
```

Critical Loops

- If your program contains a short, heavily–referenced **DO** loop, consider expanding the code to be a straight sequence of statements. For example:

```
A(1)=B(K+1)*C(M+1)
A(2)=B(K+2)*C(M+2)
A(3)=B(K+3)*C(M+3)
A(4)=B(K+4)*C(M+4)
A(5)=B(K+5)*C(M+5)
```

would run faster than:

```
DO I=1,5
  A(I)=B(K+I)*C(M+I)
END DO
```

Conversions

- Avoid forcing the compiler to convert numbers between integer and floating–point internal representations. Conversions require several instructions, including some double precision floating–point arithmetic. For example:

```
      X=1.0             ! No conversions needed
      DO 10 I=1,9
        A(I)=A(I)*X
        X=X+1.0
10    CONTINUE

      DO 10 I=1,9          ! Multiple conversions needed
        A(I)=A(I)*I
10    CONTINUE
```

When you must use mixed–mode arithmetic, code the fixed–point and floating–point arithmetic in separate computations as much as possible.

Arithmetic Constructions

- In subtraction operations, if only the negative is required, change the subtraction operations into additions, as follows:

```
      Z=—2.0                  ! Efficient
      DO 10 I=1,9
        A(I)=A(I)+Z*B(I)
10    CONTINUE

      DO 10 I=1,9                ! Inefficient
        A(I)=A(I)—2.0*B(I)
10    CONTINUE
```

- In division operations, do the following:
  - For constants, use the following construction:

    ```
    X*(1.0/3.0)
    ```

    rather than the construction `X/3.0`. Note that division by a constant of an exact power of 2 is changed to a multiplication by the reciprocal power by the compiler. For example, `X/2.0` is changed to `X*0.5`.
  - For a variable used as a denominator in several places, use the same technique.

IF Statements

- Use a block or logical **IF** statement rather than an arithmetic **IF** statement. If you must use an arithmetic **IF** statement, try to make the next statement one of the branch destinations.
- In block or logical **IF** statements, if your tests involve a series of **.AND.** or **.OR.** operators, try to:
  - Put the simplest tested conditions in the leftmost positions.
  - Put complex conditions (such as tests involving function references) in the rightmost positions.
  - Put tests most likely to be decisive in the leftmost positions.

Chapter 6. Problem Determination

This chapter describes some methods you can use for debugging your programs:

- Error messages
- Compiler listings
- The symbolic debugger.

Error Messages

The following sections discuss environment variables, compile–time messages, and run–time messages.

Environment Variables

The message catalogs must be installed and the environment variables **LANG** and **NLSPATH** must be set to a language for which the message catalog has been installed before the compiler will execute. See “Environment Variables” on page 14 for more information.

If the following message is issued during compilation, there has been an error opening the appropriate message catalog:

```
Error occurred while initializing the message system in file:
msgfile
```

where *msgfile* is the name of the message catalog that the compiler was unable to open. This message is only issued in English.

You should then verify that the message catalogs and the environment variables are in place and correct. If the message catalog or environment variables are not correct, compilation can continue, but all non–diagnostic messages will be suppressed and the following message will be issued instead:

```
No message text for msgno
```

where *msgno* is the XL FORTRAN internal message number. This message is only issued in English.

If the following message is issued during run time, there has been an error opening the run–time message catalog:

```
1525–100 An error occurred while opening the message catalog
xrfmsg.cat. The program will continue but only the error message
numbers will be displayed.
```

This message is only issued in English. You should then verify that the message catalogs and the environment variables are in place and correct. If the message catalog or environment variables are not correct, execution can continue, but only the error message numbers will be displayed.

To determine which XL FORTRAN message catalogs are installed on your system, list all of the file names using the following commands:

```
/usr/lpp/msg/language–code/xlf*.cat  (compile–time messages)
/usr/lpp/msg/language–code/xrfmsg.cat (run–time messages)
```

where `language–code` is one of the national language codes.

Compile–Time Messages

XL FORTRAN displays compile–time diagnostic messages on the terminal (standard error), and in the source listing, if you request a listing using the **LIST**, **SOURCE**, **XREF**, **ATTR**, **LISTOPT**, or **STATs** compiler option. You can also control the diagnostic messages issued, according to their severity, using the **FLAG** option. The maximum number of errors that can be issued per source line is 100.

In addition to the diagnostic message issued, the source line and a pointer to the position in the source line at which the error was detected is printed or displayed, if you specify the **SOURCE** compiler option. If **NOSOURCE** is in effect, the file number (if it is an include file), the line number, and column position of the error are displayed with the message.

The return code at the end of compilation is set to 0 if the highest severity level of all errors diagnosed is E, W, L, or I, or less than *halt_sev* if the **HALT** compiler option has been specified. Otherwise, the return code is set to one of the following values:

1	A severe or unrecoverable error has been detected that is not one of the others listed here.
40	An option error has been detected.
41	A configuration file error has been detected.
250	An out–of–memory error has been detected. The **xlf** command cannot allocate any more memory for its use.
251	A signal received error has been detected. A fatal error or interrupt signal is received.
252	A file–not–found error has been detected.
253	An input/output error has been detected. Cannot read or write files.
254	A fork error has been detected. Cannot create a new process.
255	An error has been detected while executing a process.

The format of a compile–time diagnostic message is:

15*cc–nnn 'message text'*

Where:

15	indicates an XL FORTRAN compiler message.
cc	is the component number, as follows: **11–20** indicates a FORTRAN specific message. **00** indicates a code generation or optimization message. **01** indicates an XL common message.
nnn	is the message number.
'message text'	is the text describing the error.

The severity levels of the messages are:

(U) Unrecoverable error	Internal compiler error. This error should be reported to your IBM service representative.
(S) Severe error	– Conditions exist which cannot be corrected by the compiler. An object file is produced; however, you are advised not to attempt to run the program. – An internal compiler table has overflowed. Processing of the program is discontinued and no object file is produced. This error condition can typically be corrected through the use of an option. – An include file does not exist. Processing of the program is discontinued and no object file is produced.

(E) Error	Conditions exist which can be corrected by the compiler with some degree of confidence that the program will run correctly.
(W) Warning	Warning message.
(L) Language	Language level message: FIPS or SAA warning message.
(I) Informational	Note to the programmer concerning conditions found during compilation.

Run–Time Messages

The format of a run–time diagnostic message is:

1525–*nnn* '*message text*'

Where:

1525	indicates an XL FORTRAN run–time message.
nnn	is the message number.
'*message text*'	is the text describing the error.

You can investigate errors that occur during the execution of a program using the symbolic debugger **dbx** which is available on the IBM AIX RISC System/6000.

Error messages will be issued during execution of a program if:

- An input/output error is detected.
- An exception error is detected, and a call to **SIGNAL** to install the default exception handler was specified before the exception occurred. A run–time exception will occur in the following cases:
  - An array subscript or character substring expression is out of range the **CHECK** compiler option was specified at compile time.
  - The flow of control in the program reaches a location for which a semantic error with severity of **S** was issued when the program was compiled
  - A fixed–point division by zero occurs.

Input/Output Errors

If the error detected is an input/output error, and you have specified **IOSTAT**=*integer_variable* on the input/output statement in error, the **IOSTAT** variable will be assigned the following value:

- The negation of the message number if an end–of–file condition exists
- The message number in all other cases.

If you have installed the XL FORTRAN run–time message file on the system on which the program is executing, then a message number and message text is issued to the terminal (standard error). If this run–time message file is not installed on the system, only the message number appears.

Run–Time Exceptions

For run–time exception errors, if you have installed the exception handler before the exception occurs, a message and a traceback is displayed. **dbx** can then be used to examine the location of the exception.

The exception handler supplied to perform these functions is called **xl__trce**. To install the exception handler use the following command sequence in your FORTRAN source:

```
include 'fexcp.h'            !definitions for exception handler
call signal(SIGTRAP,xl__trce) !install exception handler
```

Compiler Listings

The listing produced by the compiler (if you specify the appropriate compiler option) consists of a combination of the following sections:

- Header Section
- Options Section
- Source Section (optional)
- Attribute and Cross–Reference Section (optional)
- Object Section (optional)
- File Table Section
- Compilation Statistics Section (optional)
- Compilation Unit Epilogue Section
- Compilation Epilogue Section.

A heading identifies each major section of the listing. Greater than symbols precede the section heading so that you can easily locate the beginning of a section.

```
>>>>> section name
```

The following is a simple programming example to demonstrate the sections of a listing:

```
IBM AIX XL FORTRAN Compiler/6000  Version 01.01.0000.0000 —— user16.f 12/28/89
12:50:24

>>>>> OPTIONS SECTION <<<<<
***   Options In Effect   ***
         LIST          SOURCE        STATS
         XREF( FULL )

>>>>> SOURCE SECTION <<<<<
          1 |       PROGRAM MAIN
          2 |       INTEGER I
          3 |       DO 10 I=1,10
          4 |          WRITE(6,*) I
          5 |10     CONTINUE
          6 |       STOP
          7 |       END

>>>>> ATTRIBUTE AND CROSS REFERENCE SECTION <<<<<


IDENTIFIER                              DEF         CROSS REFERENCE
NAME                             FILE   LINE   COL
i                                   0      2   15   0–3.13@  0–4.21
main                                0      1   15

>>>>> OBJECT SECTION <<<<<


GPR's set/used:  ss–s ssss ssss s——   ———— ———— ———— ——ss
```

```
FPR's set/used:  ssss ssss ssss ss--  ---- ---- ---- ----
 CR's set/used:  ss-- --ss

    0| 000000                               PDEF     main
    0| 000000                               PROC
    0| 000000 mfspr   7C08 02A6     1     -MFSPR   r0=LR
    0| 000004 stm     BFC1 FFF8     2     -STM     (r1,-8)=r30-r31
    0| 000008 st      9001 0008     1     -ST      (r1,8)=r0
    0| 00000C stu     9421 FFC0     1     -STU     r1=(r1,-64)
    0| 000010 l       83E2 0008     1     L        r31=.&main$(r2,0)
    3| 000014 cal     3860 0001     1     LI       r3=1
    3| 000018 st      907F 0000     1     ST       i(r31,0)=r3
                                      CL.0:
    4| 00001C cal     3860 0006     1     LI       r3=6
    4| 000020 bl      4BFF FFE1     0     CALL
r3=#LDSO,1,r3,#LDSO",cr0",cr1",cr6",cr7",r0",r4"-r12",fp0"-fp13",mq"
    4| 000024 cror    4DEF 7B82     0
    4| 000028 l       807F 0000     1     L        r3=i(r31,0)
    4| 00002C bl      4BFF FFD5     0     CALL
r3=#LDINTO,1,r3,#LDINTO",cr0",cr1",cr6",cr7",r0",r4"-r12",fp0"-fp13",mq"
    4| 000030 cror    4DEF 7B82     0
    4| 000034 cal     3860 0000     1     LI       r3=0
    4| 000038 cal     38A0 0001     1     LI       r5=1
    4| 00003C oril    60A4 0000     1     LR       r4=r5
    4| 000040 bl      4BFF FFC1     0     CALL
r3=#EO,3,r3,r4,r5,#EO",cr0",cr1",cr6",cr7",r0",r4"-r12",fp0"-fp13",mq"
    4| 000044 cror    4DEF 7B82     0
                                      ?10:
    5| 000048 l       807F 0000     1     L        r3=i(r31,0)
    5| 00004C ai      3063 0001     2     AI       r3=r3,1
    5| 000050 st      907F 0000     1     ST       i(r31,0)=r3
    5| 000054 cmpi    2C83 000A     1     C        cr1=r3,10
    5| 000058 bc      4085 FFC4     3     BF       CL.0,cr1,0x2/gt
                                      CL.1:
    6| 00005C cal     3BC0 0000     1     LI       r30=0
    6| 000060 oril    63C3 0000     1     LR       r3=r30
    6| 000064 bl      4BFF FF9D     0     CALL
r3=#ATESTOP,1,r3,#ATESTOP",cr0",cr1",cr6",cr7",r0",r4"-r12",fp0"-fp13",m
q"
    6| 000068 cror    4DEF 7B82     0
    7| 00006C bl      4BFF FF95     0     CALL
#CLOSEALL,0,#CLOSEALL",cr0",cr1",cr6",cr7",r0",r3"-r12",fp0"-fp13",mq"
    7| 000070 cror    4DEF 7B82     0
    7| 000074 oril    63C3 0000     1     LR       r3=r30
                                      CL.2:
    7| 000078                               PEND
    7| 000078 l       8001 0048     1     -L       r0=(r1,72)
    7| 00007C ai      3021 0040     1     -AI      r1=r1,64
    7| 000080 mtspr   7C08 03A6     1     -MTSPR   LR=r0
    7| 000084 lm      BBC1 FFF8     2     -LM      r30-r31=(r1,-8)
    7| 000088 bcr     4E80 0020     2     -RET     LR
       Straight-line exec time   31
     | 00008C                         Tag Tables
     | 00008C 00000000
     | 000090 00012041
     | 000094 80020001
     | 000098 0000008C
     | 00009C 0004
     |        main
```

```
      | 0000A8                        Constant Area Starts Here
      | 0000A8 000ACC50
      | 0000AC                        End Of Code Csect

Instruction count is 35

** main   === End of Compilation 1 ===

>>>>> FILE TABLE SECTION <<<<<


                                          FILE CREATION        FROM
FILE NO   FILENAME                      DATE        TIME        FILE    LINE
     0    user16.f                      12/28/89    12:49:36


>>>>> COMPILATION STATISTICS SECTION <<<<<


Compiler table statistics                                  Used         Total
    Procedure list..............................    256        261888
    Computation.................................    312        524287
    Symbolic register...........................    137          4097
    Dictionary..................................     60          2049
    Integer value...............................    154          2049
    Real value..................................      0          1025
    Procedure descriptor........................      1            50
    Name and constant...........................    565         32768
    Constant....................................      4          1024


>>>>> COMPILATION EPILOGUE SECTION <<<<<


FORTRAN Summary of Diagnosed Conditions

TOTAL   UNRECOVERABLE  SEVERE       ERROR     WARNING    INFORMATIONAL
              (U)        (S)         (E)        (W)           (I)
    0          0          0           0          0             0


    Source records read.....................................          7
    Compilation start............................ 12/28/89  12:50:24
    Compilation end.............................. 12/28/89  12:50:24
    Elapsed time.........................................00:00:00
    Total cpu time.......................................    0.800
    Virtual cpu time.....................................    0.000
1501-510  Compilation successful for file user16.f.
1501-543  Object file created.
```

Another listing is shown in Appendix A, “Sample Program”.

Header Section

The listing file has a header section containing the following:

- A compiler identifier consisting of:
  - Compiler name
  - Version number
  - Release number
  - Modification number.
- Source file name
- Date of compilation
- Time of compilation.

It is the first line in the listing and appears only once in the file. The header section is always present in a listing.

Options Section

The options section is always present in a listing, and is repeated for each compilation unit. This section indicates the specified options that are in effect for the compilation unit. This information is useful when you have conflicting options. Finally, if you specify the **LISTOPT** compiler option, this section lists the settings for all options.

Source Section

The source section contains the input source lines with a line number and, optionally, a file number. The file number indicates the source file (or include file) that the source line has come from. All main file source lines (those that are not from an include file) do not have the file number printed. Each include file has a file number associated with it and source lines from include files have that file number printed. The file number appears on the left, the line number appears to its right, and the text of the source line is to the right of the line number. XL FORTRAN numbers lines relative to each file. The source lines and the numbers associated with it appear only if the **SOURCE** compiler option is in effect. Parts of the source can be selectively printed by using **SOURCE** or **NOSOURCE** throughout the program.

The source section also contains error messages interspersed with the code, as they would appear cn the terminal (standard error) during compilation. If **NOSOURCE** is in effect, and you request a listing, the source section contains only messages. If there are no messages, the source section will be empty.

Error Messages

If the **SOURCE** option is in effect, the error messages are interspersed with the source listing. The error messages generated during the compilation process contain:

- The source line
- A line of indicators which point to the columns that are in error
- The error message which consists of:
  - The 4–digit component number
  - The number of the error message
  - The severity level of the message
  - The text describing the error.

For example:

```
          2 |          equivalence (i,j,i)
            ...........................a.
a – 1514–092: (E) Same name appears more than once in an equivalence
group.
```

If the **NOSOURCE** option is in effect, the error messages are all that appear in the source section, and they contain:

- The line number and column position of the error (and the file number if the line in error is in an **INCLUDE** file)
- The error message which consists of:
  - The 4–digit component number
  - The number of the error message
  - The severity of the message
  - The text of the error.

For example:

```
3.15   1513–039: (S) Number of arguments is not permitted for
INTRINSIC function abs.
```

Attribute and Cross–Reference Section

This section provides information about the variables used in the compilation unit. It is present if the **XREF** or **ATTR** compiler option is in effect. Depending on the options in effect, this section contains all or part of the following information about the variables used in the compilation unit:

- Name of the symbolic variable
- Attributes of the variable (if **ATTR** is in effect). Attribute information includes the type and the storage class of the variable, relative address of the variable, dimensions of the variable (if an array), and the alignment of the variable (if **ATTR** is in effect).
- File, line, and column numbers on which you define a variable and coordinates to indicate where you have referenced or modified the variable. If the variable is initialized, the coordinates are marked with a `*`. If the variable is set, the coordinates are marked with a `@` . If the variable is referenced, the coordinates are not marked.

  Storage class may be one of the following:

  Program
  Function
  Subroutine
  Entry
  External Subprogram
  Static
  BSS
  Common
  Common block
  NAMELIST
  Automatic
  Reference Parameter
  Value Parameter.

  Type may be one of the following:

  Logical
  Integer
  Real
  Complex
  Character.

If you specify the **FULL** suboption with **XREF** or **ATTR**,XL FORTRAN reports all variables in the compilation unit. If you do not specify this suboption, only the variables you actually use appear.

Object Section

XL FORTRAN produces this section only when the **LIST** compiler option is in effect. It contains the object code listing, which shows the source line number, the instruction offset in hexadecimal, the assembler mnemonic of the instruction, and the hexadecimal value of the instruction. On the right side, it also shows the cycle time of the instruction and the intermediate language of the compiler. Finally, the total cycle time (straight–line execution time) and the total number of machine instructions produced is displayed. This section is repeated for each compilation unit.

File Table Section

This section contains a table showing the file number and file name for each main source file and include file used. It also lists the line number of the main source file at which the include file is referenced. This section is always present.

Compilation Statistics Section

This section appears in the listing only if the **STAT**s compiler option is in effect. It provides compilation statistics, such as various compiler table sizes and usage.

Compilation Unit Epilogue Section

This is the last section of the listing for each compilation unit. It contains the diagnostics summary, and indicates whether or not the unit compiled successfully. The compilation unit epilogue section is not present in the listing if the file contains one compilation unit.

Compilation Epilogue Section

The above sections are repeated for each compilation unit when more than one compilation unit is present. At completion, XL FORTRAN presents a summary of the compilation in terms of number of source records read, compilation start, compilation end, total compilation time, total cpu time, and virtual cpu time. This section is always present in a listing.

The Symbolic Debugger

XL FORTRAN supports the **dbx** symbolic debugger. See the *AIX Commands Reference for IBM RISC System/6000,* SC23–2199 for detailed information.

Chapter 7. Interlanguage Calls

XL FORTRAN permits you to call subroutines written in other languages from your program. This chapter assumes you are familiar with the syntax of the languages you will be using, and gives details on how to perform interlanguage calls from your FORTRAN program.

Programming Conventions

XL FORTRAN has adopted the following conventions to allow or assist interlanguage calls:

- Programs and symbolic names are folded to lowercase by default.
- Both the underscore (_) and currency symbol ($) are valid characters in names, and you can use either as the first character of a name.

  Note: Names that begin with _ are reserved names in C and in the XL FORTRAN library. It is recommended that you do not use _ as the first character of a name. Also, using the $ as the first character in external names can cause problems, because AIX uses $ as the first character in a shell variable name.
- Names can be up to 250 characters long.
- There are two common parameter passing modes: by value and by reference. (These are explained in detail for FORTRAN in "%VAL and %REF" on page 56.)

For compatibility with C language usage, XL FORTRAN uses the following backslash escapes in character strings:

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

Figure 2. Backslash Escapes

If you are porting your application from the RT, and your application makes use of interlanguage calls, the **EXTNAME** compiler option may be necessary. See "Options Affecting the Compiler Object Code to be Produced" on page 25 for information about this option.

Programming Tips

The following describes programming tips for writing XL FORTRAN procedures which interact with routines written in other languages:

- Character data:

  C functions expect that strings are terminated by the null character. XL FORTRAN does not append the null character to the end of character literals. It is the programmer's responsibility to concatenate the null character to the end of any character data which is passed to C routines.

- Floating–point functions and arguments:

  In C, an implicit conversion takes place for float values which are passed as arguments on a call to a C function having no prototype visible, and the called C function expects that all incoming float arguments have been converted to double. As XL FORTRAN does not perform a conversion on **REAL*4** quantities passed by value, it is recommended that **REAL*4** values not be passed as arguments to C functions that are not declared as function prototypes.

- Input/Output:

  To improve performance, the XL FORTRAN run–time library has its own buffers and its own handling of these buffers. Therefore, it is not recommended that XL FORTRAN routines and routines of other languages perform input/output on the same data files within the same executable program. However, if such mixing of input/output is required, the data file should be both opened and explicitly closed within XL FORTRAN routines before any input/output operations are performed on that file by routines written in another language. If any XL FORTRAN routines contain WRITE statements, and these routines are used in an executable program in which the main program is not written in XL FORTRAN, then the programmer must explicitly **CLOSE** the data file in an XL FORTRAN routine to ensure that the buffers are flushed.

  These restrictions do not apply to **READ** or **WRITE** statements using logical units 5 or 6, which are preconnected to standard in and standard out, or to **PRINT** statements.

Corresponding Data Types

The following table indicates the data types available in theXL FORTRAN, Pascal, and C languages.

FORTRAN Data Types	**Pascal Data Types**	**C Data Types**
INTEGER*1	PACKED –128..127	signed char
INTEGER*2	PACKED –32768..32767	signed short
INTEGER*4	INTEGER	signed int
REAL REAL*4	SHORTREAL	float
REAL*8 DOUBLE PRECISION	REAL	double
REAL*16		
COMPLEX COMPLEX*8		structure of 2 floats

COMPLEX*16 DOUBLE COMPLEX		structure of 2 doubles
COMPLEX*32		
LOGICAL*1	PACKED 0..255	unsigned char
LOGICAL*2	PACKED 0..65535	unsigned short
LOGICAL*4		unsigned int
CHARACTER	CHAR	char
CHARACTER*n	PACKED ARRAY[1..n] OF CHAR	char[n]
Dimensioned variable	ARRAY	

Figure 3. Correspondence of Data Types among FORTRAN, Pascal, and C

Character Variable Types

Most numeric data types have counterparts across languages but there are some differences between the data types. The most difficult aspect of interlanguage calls is passing character, string, or text variables between languages.

The only character type in FORTRAN is **CHARACTER**, which is stored as a set of contiguous bytes, one character per byte. The length of a FORTRAN character variable or character array element is determined at compile time and is therefore static. Character lengths are returned by the FORTRAN intrinsic function **LEN**.

When you pass a FORTRAN character data item as a parameter, the address of the beginning of the character is passed along with a parameter that is the length of the character string. The parameter is added to the end of the declared parameter list.

Pascal's character–variable data types are **STRING**, **PACKED ARRAY OF CHAR**, **GSTRING**, and **PACKED ARRAY OF GCHAR**. The **STRING** data type has a 4 byte string length aligned on a double word boundary followed by a set of contiguous bytes, one character per byte. The dynamic length of the string can be determined using the **LENGTH** function. Packed arrays of **CHAR**, like FORTRAN's **CHARACTER** type, are stored as a set of contiguous bytes, one character per byte.

Character strings in C are typically stored as arrays of the type **char**. The **char** data type stores one character per byte; therefore, an array of **char** is stored exactly like a FORTRAN **CHARACTER** variable or a Pascal **PACKED ARRAY OF CHAR**. When an array of **char** is used to represent a character string in C, the end of the string is indicated by the null character.

How Arrays Are Stored

XL FORTRAN stores array elements in ascending storage units in column–major order. C and Pascal store array elements in row–major order. The following example shows how a two–dimensional array declared by A(3,2) in FORTRAN, C, and Pascal is stored:

	FORTRAN Element Name	Pascal Element Name	C Element Name
Lowest storage unit	A(1,1)	A[1,1]	A[0,0]
	A(2,1)	A[1,2]	A[0,1]
	A(3,1)	A[2,1]	A[1,0]
	A(1,2)	A[2,2]	A[1,1]
	A(2,2)	A[3,1]	A[2,0]
Highest storage unit	A(3,2)	A[3,2]	A[2,1]

%VAL and %REF

To call subprograms written in languages other than FORTRAN (for example, user–written C programs, or AIX system routines), the actual arguments may need to be passed by a method different from the default method used by FORTRAN. The form of an actual argument can be changed by using the **%VAL** and **%REF** keywords in the argument list of a **CALL** statement or function reference. These keywords specify the way the actual argument should be passed to the subprogram.

The argument list keywords are:

%VAL This keyword can be used with actual arguments that are **CHARACTER*1**, logical, integer, real, or complex expressions. It cannot be used with actual arguments that are array names, procedure names, or character expressions of length greater than 1 byte. Hexadecimal, binary, octal, and Hollerith constants are passed as if they were of the type **INTEGER*4**. If the actual argument is a **CHARACTER*1**, it is padded on the left with zeros to a 32–bit value.

This keyword causes the actual argument to be passed as 32–bit intermediate values. If the actual argument is an integer that is shorter than 32 bits, it is sign–extended to a 32–bit value. If the actual argument is a logical that is shorter than 32 bits, it is padded on the left with zeros to a 32–bit value. If the actual argument is of type real or complex with a length greater than 32 bits, it is passed as multiple 32–bit intermediate values.

%REF This keyword causes the actual argument to be passed by reference. (The address of the actual argument is passed.) This is the default for FORTRAN.

Note that, if the actual argument is of character data type, only the address of the actual argument is passed, whereas a character actual argument passed without the **%REF** function is passed as the address and the length of the character argument. If such a character argument is being passed to a C routine, the string must be terminated with a null character, so that the C routine can determine the length of the string.

Subroutine Linkage Convention

The "subroutine linkage convention" describes the machine state at subroutine entry and exit. This scheme allows routines that are compiled separately in the same or different languages to be linked and executed when called. The information on Subroutine Linkage and System Calls in the User's Guide for IBM AIX XL FORTRAN Compiler/6000 is the base reference on this topic, and should be consulted for further details.

The linkage convention for the IBM AIX XL FORTRAN Compiler/6000 provides a fast and efficient subroutine linkage by passing parameters in registers, taking full advantage of the large number of floating–point registers (FPRs) and general–purpose registers (GPRs), and minimizing the saving and restoring of registers on subroutine entry and exit. The linkage convention allows for parameter passing and return values to be in FPRs, GPRs, or both. (GPRs are also referred to as registers.)

Register Usage

If a register is not designated as saved during the call, its contents may be changed during the call. Conversely, if a register is saved, its contents must be preserved across the call. The following table lists registers and their functions.

Register	Preserved Across Calls	Use
0	no	Stack pointer.
1	yes	TOC pointer.
2	yes	
3	no	1st word of arg list; return value 1.
4	no	2nd word of arg list; return value 2.
5	no	3rd word of arg list; return value 3.
6	no	4th word of arg list; return value 4.
7	no	5th word of arg list; return value 5.
8	no	6th word of arg list; return value 6.
9	no	7th word of arg list; return value 7.
10	no	8th word of arg list; return value 8.
11	no	DSA pointer to internal procedure (Env).
12	no	
13–31	yes	

Figure 4. General Purpose Register Usage

The following table lists floating–point registers and their functions. The floating–point registers are double precision (64 bits).

Register	Preserved Across Calls	Use
0	no	
1	no	FP parameter 1, function return 1.
2	no	FP parameter 2, function return 2.
. . .	. . .	
13	no	FP parameter 13, function return 13.
14–31	yes	

Figure 5. Floating–Point Register Usage

The following table lists special purpose register conventions.

Register	Preserved Across Calls	Use
Condition register		
Bits 0–7 (CR0,CR1)	no	
Bits 8–19 (CR2,CR3,CR4)	yes	
Bits 20–23 (CR5)	yes	Reserved for system use. Never set or changed.
Bits 24–31 (CR6,CR7)	no	
Link register	no	
Count register	no	
MQ register	no	
XER register	no	
FPSCR register	no	

Figure 6. Special Purpose Register Usage

Stack

The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current "top" of the stack.

A stack frame is the portion of the stack used by a single procedure. You can consider the input parameters as being part of the current stack frame. In a sense, each output argument belongs to both the caller's and the callee's stack frames. In either case, the stack frame size is best defined as the difference between the caller's stack pointer and the callee's.

The storage map of a typical stack frame is shown below.

In this diagram, the current routine has acquired a stack frame which allows it to call other functions. If no calls are made, and there are no local variables or temps, then the function need not allocate a stack frame. It can still use the register save area at the top of the caller's stack frame, if needed.

The stack frame is double word aligned. The FPR save area and the parameter area (P1, P2, ..., Pn) are also double word aligned. Other areas require word alignment only.

```
                      RUN—TIME STACK
                IBM AIX XL FORTRAN Compiler/6000
  Low                   |                    |  Stack grows at
  Addresses             |                    |  this end.
                        |--------------------|
Callee's stack  —> 0    |     Back chain     |
pointer            4    |     Saved CR       |
                   8    |     Saved LR       |
               12—16    |     Reserved       |  <—— LINK AREA (callee)
                  20    |     SAVED TOC      |
                        |--------------------|
 Space for P1—P8        |        P1          |    OUTPUT ARGUMENT AREA
 is always reserved     |        ...         |  <—— (Used by callee to
                        |        Pn          |      construct argument
                        |--------------------|      list)
                        |     Callee's       |
                        |     stack          |  <—— LOCAL STACK AREA
                        |     area           |
                        |--------------------|
                        |                    |  (Possible word wasted
                        |--------------------|     for alignment.)
 —8*nfprs—4*ngprs —>    |   Caller's GPR     |  Rfirst = R13 for full
   save                 |    save area       |           save
                        |   max 19 words     |  R31
                        |--------------------|
         —8*nfprs —>    |   Caller's FPR     |  Ffirst = F14 for a
                        |    save area       |           full save
                        |   max 18 dblwds    |  F31
                        |--------------------|
Caller's stack  —> 0    |     Back chain     |
 pointer           4    |     Saved CR       |
                   8    |     Saved LR       |
               12—16    |     Reserved       |  <—— LINK AREA (caller)
                  20    |     Saved TOC      |
                        |--------------------|
Space for P1—P8   24    |        P1          |    INPUT PARAMETER AREA
is always reserved      |        ...         |  <——(Callee's input
                        |        Pn          |     parameters found
                        |--------------------|      here. Is also
                        |     Caller's       |     caller's arg area.)
                        |     stack          |
  High                  |     area           |
  Addresses             |                    |
```

Link area

This area consists of six words, and is at offset zero from the caller's stack pointer on entry to a procedure. The first word contains the caller's back chain (stack pointer). The second word is where the callee saves the Condition Register (CR) if needed. The third word is where the callee's PROLOG code saves the Link Register if necessary. The fourth word is reserved for the **SETJMP**, **LONGJMP** processing, and the fifth word is reserved for future use. The last word (word 6) is reserved for use by the Global Linkage routines which are used when calling out–of–module routines (for example, in shared libraries).

Input Parameter Area

This is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double word aligned, and is located on the stack directly following the caller's link area. This area is at least 8 words in size. If more than 8 words of parameters are expected, they would have been stored as register images starting at positive offset 56 from the incoming stack pointer.

Register Save area

This area is double word aligned, and provides the space needed to save all non–volatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved above the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. The system defined stack floor includes the maximum possible save area (18*8 for FPRs + 19*4 for GPRs). Locations at a numerically lower address than stack floor should not be accessed.

A callee needs only to save the non–volatile registers that it actually uses. Register 31 is always saved in the highest addressed word of the particular save area.

Local stack area

This is the space allocated by the callee procedure for local variables, and temporaries.

Output Parameter Area

The parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures called from the procedure that owns this stack frame.

This area is at least 8 words long regardless of the length or existence of any argument list. If more than 8 words are being passed, an extension list is constructed beginning at offset 56 from the current stack pointer.

Parameter Passing

The IBM AIX RISC System/6000 linkage convention takes advantage of the large number of registers available. The linkage convention passes arguments in both GPRs and FPRs. The GPRs and FPRs available for argument passing are specified in two fixed lists: R3–R10 and FP1–FP13.

When there are more argument words than available parameter GPRs and FPRs the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers. Space for more than 8 words of arguments (float and non–float) must be reserved on the stack even if all the arguments were passed in registers.

The size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, it is convenient to consider them as forming a list in this area, each one occupying one or more words.

For call by reference (as in FORTRAN), the address of the parameter is passed in a register. The following refers to call by value, as in C or as in FORTRAN when **%VAL** is used. For purposes of their appearance in the list, arguments are classified as floating–point values or nonfloating–point values:

- Each nonfloating–point scalar argument requires one word and appears in that word exactly as it would appear in a GPR. It is right justified, if language semantics specify, and is word aligned.
- Each floating–point value occupies one word, while float double occupies two successive words in the list.
- Structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a fullword and occupy `(sizeof(struct X)+3)/4` fullwords with any padding at the end. A structure which is smaller than a word will be left justified within its word or register. Larger

structures can occupy multiple registers, and may be passed partly in storage and partly in registers.

- Other aggregate values are passed "val–by–ref", that is, the compiler actually passes their address and arranges for a copy to be made in the invoked program.
- A procedure or function pointer is passed as a pointer to the routine's function descriptor; its first word contains its entry point address. (See "Pointers to Functions" on page 62 for more information.)

Argument passing rules (by value)

Following is an example of a call to a function:

```
         f(%val(l1), %val(l2), %val(l3), %val(d1), %val(f1),
           %val(c1), %val(d2), %val(s1), %val(cx2))

where: l denotes integer*4 (full word integer)
       d denotes real*8 (double precision)
       f denotes real*4 (real)
       s denotes integer*2 (half word integer)
       c denotes character (one character)
       cx denotes complex*16 (double complex)
```

```
                               STORAGE MAPPING OF
 WILL BE PASSED IN:          PARM AREA ON THE STACK
                              ___________________
  R3                      0 |         l1         |
                            |____________________|
  R4                      4 |         l2         |
                            |____________________|
  R5                      8 |         l3         |
                            |____________________|
                         12 |                    |
  FP1 (R6,R7 unused)        |         d1         |
                         16 |                    |
                            |____________________|
  FP2 (R8 unused)        20 |         f1         |
                            |____________________|
  R9                     24 |////|////|////| c1  |<— right justified (if
                            |____________________|   language sematics
                         28 |                    |    specify)
  FP3 (R10 unused)          |         d2         |
                         32 |                    |
                            |____________________|
  STACK                  36 |////|////|    s1    |<— right justified (if
                            |____________________|   language semantics
  FP4 (8 bytes           40 |                    |    specify)
   reserved in              |     cx2 (real)     |
   stack)                44 |                    |
                            |____________________|
  FP5 (8 bytes           48 |                    |
   reserved in              |  cx2 (imaginary)   |
   stack)                52 |                    |
                            |____________________|
```

From the above illustration we state the following rules:

- The parameter list is a conceptually contiguous piece of storage containing a list of words. For efficiency, the first 8 words of the list are not actually stored in the space reserved for them, but passed in GPR3–GPR10. Further, the first 13 floating–point value parameters are not stored in the space reserved for them or passed in GPRs, but are passed in FPR1–FPR13.

- If the called procedure wishes to treat the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C) then the parameter registers are stored in the space reserved for them in the stack.
- A register image is stored on the stack.
- The argument area ($P_1 ... P_n$) must be large enough to hold the largest parameter list.

Function Calls

A routine has two symbols associated with it: a function descriptor (*name*) and an entry point (*.name*). When a call is made to a routine, the compiler branches to the name point directly. Excluding the loading of parameters (if any) in the proper registers, calls to functions are expanded by compilers to the following two instruction sequence:

```
BL     .foo                      # Branch to foo
CROR   15,15,15                  # Special NOP
```

The linkage editor will do one of 2 things when it sees a **BL**:

1. If `foo` is imported (not in the same module), then the linkage editor will change the **BL** to `.foo` to a **BL** to `.glink` (global linkage routine) of `foo`, and insert the `.glink` into the module. Also, if a **NOP** instruction (`CROR 15,15,15`) immediately follows the **BL** instruction, the linkage editor will replace the **NOP** instruction with the **LOAD** instruction `L R2, 20(R1)`.
2. If `foo` is bound in the same module as its caller, and a **LOAD** instruction (`L R2,20(R1)`) immediately follows the **BL** instruction, then it will replace the **LOAD** instruction with a **NOP** (`CROR 15,15,15`).

Note: For any export, the linkage editor will insert the procedure's descriptor into the module.

Pointers to functions

Function pointer is the name given to a data type whose values range over procedure names. Variables of this type appear in several programming languages such as C and FORTRAN. In FORTRAN, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Support is provided for the use of function pointers in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a three word object. The first word contains the address of the entry point of the procedure, the second word has the address of the TOC of the module in which the procedure is bound, and the third word is the environment pointer for languages such as Pascal, and PL/I. There is only one function descriptor per entry point. It is bound into the same module as the function it identifies if the function is external. The descriptor has an external name, which is the same as the function name but with a different storage class that uniquely identifies it. It is this descriptor name that is used in all import or export operations.

Function values

Functions return their values according to type:

- **INTEGER** and **LOGICAL** of any length are returned (right justified) in R3.
- Floating–point values are returned in FP1–FP13.
- A single or double complex value is returned in FP1 and FP2.

- Character strings are returned in a buffer allocated by the caller. The address and the length of this buffer are passed in R3 and R4 respectively as hidden parameters. This means that the first explicit parameter word will be in R5 and that all subsequent parameters are moved to the next word.

Stack Floor

The stack floor is a system defined address below which the stack may not grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

There are other system invariants related to the stack that must be maintained by all compilers and assemblers, These are:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32767 bytes, care must be taken to ensure that its value is changed in a single instruction. This ensures that there is no timing window in which a signal handler would either overlay the stack data, or erroneously appear to overflow the stack segment.

Stack Overflow

The linkage convention for IBM AIX XL FORTRAN Compiler/6000 requires no explicit inline check for overflow. The operating system uses a storage protect mechanism to detect stores past the end of the stack segment.

Prolog/Epilog

On entry to a procedure, some or all of the following steps may have to be done:

1. Save the link register at offset 8 from the stack pointer if necessary.
2. If any of the CR bits 8–19 (CR2, CR3, CR4) are used then save the CR at displacement 4 from the current stack pointer.
3. Save any non–volatile FPRs used by this procedure in the caller's FPR save area. There is a set of routines named **_savef14**, **_savef15**, ... **_savef31** which may be used.
4. Save all non–volatile GPRs used by this procedure in the caller's GPR save area.
5. Store back chain and decrement stack pointer by the size of the stack frame. Note that if a stack overflow occurs, it will be known immediately when the store of the back chain is done.

On exit from a procedure, some or all of the following steps may have to be performed:

1. Restore all GPRs saved.
2. Restore stack pointer to the value it had on entry.
3. Restore link register if necessary.
4. Restore bits 8–19 of the CR if necessary.
5. If any FPRs were saved then restore them using **_restf***n* where *n* is the first FPR to be restored.
6. Return to caller.

Traceback

The compiler supports the traceback mechanism, which is required by the IBM AIX Version 3 for RISC System/6000 symbolic debugger in order to unravel the call or return stack. Each module has a traceback table in the text segment at the end of its code. This table contains information about the module including the type of module as well as stack frame and register information.

Type Encoding/Checking

Early error detection (prior to execution) is a key objective of the AIX Linkage Convention. Execution time errors are hard to find, and a good number of them are caused by mismatching subroutine interfaces or conflicting data definition. To help uncover such errors early, a scheme has been defined by which compilers will encode information about all external symbols (data and programs). This information will then be checked at bind or load time for consistency if the **EXTCHK** option has been specified.

Sample Program – FORTRAN Calling C

The following example is to illustrate how program units written in different languages can be combined to create a single program. It also shows the methods used for parameter passing between FORTRAN and C subroutines with different data types as arguments.

```
      PROGRAM EXAMPLE
      INTEGER*4 I1, I3, I4, I6, IRET, CFUNC1, CFUNC3, CFUNC4, CFUNC5
      INTEGER*4 IARG1 /1/
      INTEGER*2 IARG2 /2/
      INTEGER*1 IARG3 /3/
      CHARACTER*1 CHAR_ARG1 /'a'/, CHAR_ARG2 /'b'/
      REAL*8 R_ARG1 /1.0/, R_ARG2 /2.0/, R_ARG3 /3.0/, CFUNC2, R2
      COMPLEX*16 CX_ARG1 /(1.0,1.0)/, CX_ARG2/(1.0,2.0)/,
     +           CX_ARG3 /(3.0,3.0)/
      LOGICAL*4 LARG1 /T/, LARG2 /F/, LARG3 /F/
C call C function using integer arguments
      I1 = CFUNC1(IARG1, %VAL(IARG2), %REF(IARG3), %VAL(X"0004"))
C
C call C function using real arguments
      R2 = CFUNC2(R_ARG1, %VAL(R_ARG2), %REF(R_ARG3), %VAL(4.0D0))
C
C call C function using complex arguments
      I3 = CFUNC3(%VAL(CX_ARG2), %VAL((5.0D0, 6.0D0)))
C
C call C functions using logical arguments
      I4 = CFUNC4(LARG1, %VAL(LARG2), %REF(LARG3), %VAL(.TRUE.))
C
C call C functions using character arguments
      I6 = CFUNC5(%VAL(CHAR_ARG1), %REF(CHAR_ARG2), %VAL('c'))
C
C check values returned
      IF ( (I1.NE. 1) .OR. (R2.NE.1.0) .OR. (I3.NE.1) .OR. (I4.NE.1)
     +     .OR. (I6.NE.1) ) PRINT *,"PROGRAM NOT CORRECT"
       END
```

```
/* C Program                                                         */
/* C function that receives integer values                           */
cfunc1( ivar1, ivar2, ivar3, ivar4)
int *ivar1, ivar4;
short ivar2;
char *ivar3;
{
int retval;
   if (*ivar1!=1 || ivar2!=2 || *ivar3!=3 || ivar4!=4)
   {
      retval = -1;
   }
   else retval = 1;
   return(retval);
}
/* C function that receives real values                              */
cfunc2(rvar1, rvar2, rvar3, rvar4)
double *rvar1, rvar2, *rvar3, rvar4;
{
double retval;
   if ( *rvar1!=1.0 || rvar2!=2.0 || *rvar3!=3.0 || rvar4!=4.0)
   {
      retval=-1.0e0;
   }
   else retval = 1.0;
   return(retval);
}
/* C function that receives FORTRAN complex values as real           */
cfunc3(rvar1, rvar2, rvar3, rvar4)
double rvar1, rvar2, rvar3, rvar4;
{
int retval;
   if ( rvar1!=1.0 || rvar2!=2.0 || rvar3!=5.0 || rvar4!=6.0 )
      retval=-1;
   else retval = 1;
   return(retval);
}
/* C function that receives FORTRAN logical values                   */
cfunc4(lvar1, lvar2, lvar3, lvar4)
int *lvar1, lvar2, *lvar3, lvar4;
{
int retval;
    if (*lvar1!=0x01 || lvar2!=0x00 || *lvar3!=0x00 || lvar4!=0x01)
       retval = -1;
    else retval = 1;
    return(retval);
}
/* C function that receives FORTRAN character values                 */
cfunc5(chvar1, chvar2, chvar3)
char chvar1, *chvar2, chvar3;
{
int retval;
   if (chvar1!='a' || *chvar2!='b' || chvar3!='c')
      retval = -1;
   else retval = 1;
   return(retval);
}
```

Appendix A. Sample Program

The following sections outline a sample program, the listing it would produce with the **SOURCE** compiler option specified, and the output from running the program.

Source File

```
      PROGRAM CALCULATE
c
c Program to calculate the sum of up to n values of x**3
c where negative values are ignored.
c Stop if the sum is negative.
c
      READ(5,*) N
      SUM=0
      DO 10 I=1,N
         READ(5,*) X
         IF (X.GE.0) THEN
            Y=X**3
            IF (SUM.GE.0) THEN
               SUM=SUM+Y
            ELSE
               GOTO 20
            END IF
         END IF
10    CONTINUE
20    CONTINUE
      WRITE(6,*) 'This is the sum:',SUM
      STOP
      END
```

Sample Listing

If you specify the **SOURCE** compiler option, XL FORTRAN produces the following listing:

```
IBM AIX XL FORTRAN Compiler/6000  Version 01.01.0000.0000 —— user20.f 11/26/03
05:42:40

>>>>> OPTIONS SECTION <<<<<
***   Options In Effect   ***
         SOURCE

>>>>> SOURCE SECTION <<<<<
          1 |      PROGRAM CALCULATE
          2 |c
          3 |c Program to calculate the sum of up to n values of x**3
          4 |c where negative values are ignored.
          5 |c Stop if the sum is negative.
          6 |c
          7 |      READ(5,*) N
          8 |      SUM=0
          9 |      DO 10 I=1,N
         10 |         READ(5,*) X
         11 |         IF (X.GE.0) THEN
```

```
        12 |             Y=X**3
        13 |             IF (SUM.GE.0) THEN
        14 |                SUM=SUM+Y
        15 |             ELSE
        16 |                GOTO 20
        17 |             END IF
        18 |          END IF
        19 |10     CONTINUE
        20 |20     CONTINUE
        21 |       WRITE(6,*) 'This is the sum:',SUM
        22 |       STOP
        23 |       END
    ** calculate   === End of Compilation 1 ===

>>>>> FILE TABLE SECTION <<<<<


                                              FILE CREATION          FROM
FILE NO   FILENAME                           DATE        TIME        FILE     LINE
     0    user20.f                           07/17/89    23:24:55


>>>>> COMPILATION EPILOGUE SECTION <<<<<


FORTRAN Summary of Diagnosed Conditions

TOTAL   UNRECOVERABLE  SEVERE       ERROR      WARNING     INFORMATIONAL
              (U)        (S)         (E)         (W)            (I)
    0         0          0           0           0              0


    Source records read.......................................      23
    Compilation start.............................. 11/26/03  05:42:40
    Compilation end................................ 11/26/03  05:42:40
    Elapsed time..............................................00:00:00
    Total cpu time............................................   0.300
    Virtual cpu time..........................................   0.400
1501-510  Compilation successful for file user20.f.
1501-543  Object file created.
```

Output Produced

Given the input `5 37 22 -4 19 6`, the following output would appear when the sample program runs: `68376.00000`

Appendix B. ASCII/EBCDIC Character Set

XL FORTRAN uses the ASCII character set as its collating sequence.

This appendix lists the standard ASCII and EBCDIC characters in numerical order with the corresponding decimal and hexadecimal values. The table indicates the control characters with "`Ctrl–`" notation. For example, the horizontal tab (`HT`) appears as "`Ctrl–I`", which you enter by simultaneously pressing the `Ctrl` key and `I` key.

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
0	00	Ctrl–@	NUL	null	NUL	null
1	01	Ctrl–A	SOH	start of heading	SOH	start of heading
2	02	Ctrl–B	STX	start of text	STX	start of text
3	03	Ctrl–C	ETX	end of text	ETX	end of text
4	04	Ctrl–D	EOT	end of transmission	SEL	select
5	05	Ctrl–E	ENQ	enquiry	HT	horizontal tab
6	06	Ctrl–F	ACK	acknowledge	RNL	required new–line
7	07	Ctrl–G	BEL	bell	DEL	delete
8	08	Ctrl–H	BS	backspace	GE	graphic escape
9	09	Ctrl–I	HT	horizontal tab	SPS	superscript
10	0A	Ctrl–J	LF	line feed	RPT	repeat
11	0B	Ctrl–K	VT	vertical tab	VT	vertical tab
12	0C	Ctrl–L	FF	form feed	FF	form feed
13	0D	Ctrl–M	CR	carriage return	CR	carriage return
14	0E	Ctrl–N	SO	shift out	SO	shift out
15	0F	Ctrl–O	SI	shift in	SI	shift in
16	10	Ctrl–P	DLE	data link escape	DLE	data link escape
17	11	Ctrl–Q	DC1	device control 1	DC1	device control 1
18	12	Ctrl–R	DC2	device control 2	DC2	device control 2
19	13	Ctrl–S	DC3	device control 3	DC3	device control 3
20	14	Ctrl–T	DC4	device control 4	RES/ ENP	restore/enable presentation
21	15	Ctrl–U	NAK	negative acknowledge	NL	new–line
22	16	Ctrl–V	SYN	synchronous idle	BS	backspace
23	17	Ctrl–W	ETB	end of transmission block	POC	program–operator communications
24	18	Ctrl–X	CAN	cancel	CAN	cancel
25	19	Ctrl–Y	EM	end of medium	EM	end of medium
26	1A	Ctrl–Z	SUB	substitute	UBS	unit backspace
27	1B	Ctrl–[	ESC	escape	CU1	customer use 1

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
28	1C	Ctrl–\	FS	file separator	IFS	interchange file separator
29	1D	Ctrl–]	GS	group separator	IGS	interchange group separator
30	1E	Ctrl–^	RS	record separator	IRS	interchange record separator
31	1F	Ctrl–_	US	unit separator	IUS/ITB	interchange unit separator / intermediate transmission block
32	20		SP	space	DS	digit select
33	21		!	exclamation point	SOS	start of significance
34	22		"	straight double quotation mark	FS	field separator
35	23		#	number sign	WUS	word underscore
36	24		$	dollar sign	BYP/INP	bypass/inhibit presentation
37	25		%	percent sign	LF	line feed
38	26		&	ampersand	ETB	end of transmission block
39	27		'	apostrophe	ESC	escape
40	28		(	left parenthesis	SA	set attribute
41	29		)	right parenthesis		
42	2A		*	asterisk	SM/SW	set model switch
43	2B		+	addition sign	CSP	control sequence prefix
44	2C		,	comma	MFA	modify field attribute
45	2D		–	subtraction sign	ENQ	enquiry
46	2E		.	period	ACK	acknowledge
47	2F		/	right slash	BEL	bell
48	30		0			
49	31		1			
50	32		2		SYN	synchronous idle
51	33		3		IR	index return
52	34		4		PP	presentation position
53	35		5		TRN	
54	36		6		NBS	numeric backspace
55	37		7		EOT	end of transmission
56	38		8		SBS	subscript
57	39		9		IT	indent tab
58	3A		:	colon	RFF	required form feed
59	3B		;	semicolon	CU3	customer use 3
60	3C		<	less than	DC4	device control 4
61	3D		=	equal	NAK	negative acknowledge

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
62	3E		>	greater than		
63	3F		?	question mark	SUB	substitute
64	40		@	at symbol	SP	space
65	41		A			
66	42		B			
67	43		C			
68	44		D			
69	45		E			
70	46		F			
71	47		G			
72	48		H			
73	49		I			
74	4A		J		¢	cent
75	4B		K		.	period
76	4C		L		<	less than
77	4D		M		(	left parenthesis
78	4E		N		+	addition sign
79	4F		O		|	logical or
80	50		P		&	ampersand
81	51		Q			
82	52		R			
83	53		S			
84	54		T			
85	55		U			
86	56		V			
87	57		W			
88	58		X			
89	59		Y			
90	5A		Z		!	exclamation point
91	5B		[	left bracket	$	dollar sign
92	5C		\	left slash	*	asterisk
93	5D		]	right bracket	)	right parenthesis
94	5E		^	hat, circumflex	;	semicolon
95	5F		_	underscore	¬	logical not
96	60		'	grave	–	subtraction sign
97	61		a		/	right slash
98	62		b			
99	63		c			

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
100	64		d			
101	65		e			
102	66		f			
103	67		g			
104	68		h			
105	69		i			
106	6A		j		¦	split vertical bar
107	6B		k		,	comma
108	6C		l		%	percent sign
109	6D		m		_	underscore
110	6E		n		>	greater than
111	6F		o		?	question mark
112	70		p			
113	71		q			
114	72		r			
115	73		s			
116	74		t			
117	75		u			
118	76		v			
119	77		w			
120	78		x			
121	79		y		`	grave
122	7A		z		:	colon
123	7B		{	left brace	#	number sign
124	7C		|	logical or	@	at symbol
125	7D		}	right brace	'	apostrophe
126	7E		~	similar, tilde	=	equal
127	7F		DEL	delete	"	straight double quotation mark
128	80					
129	81				a	
130	82				b	
131	83				c	
132	84				d	
133	85				e	
134	86				f	
135	87				g	
136	88				h	

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
137	89				i	
138	8A					
139	8B					
140	8C					
141	8D					
142	8E					
143	8F					
144	90					
145	91				j	
146	92				k	
147	93				l	
148	94				m	
149	95				n	
150	96				o	
151	97				p	
152	98				q	
153	99				r	
154	9A					
155	9B					
156	9C					
157	9D					
158	9E					
159	9F					
160	A0					
161	A1				~	similar, tilde
162	A2				s	
163	A3				t	
164	A4				u	
165	A5				v	
166	A6				w	
167	A7				x	
168	A8				y	
169	A9				z	
170	AA					
171	AB					
172	AC					
173	AD					
174	AE					

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
175	AF					
176	B0					
177	B1					
178	B2					
179	B3					
180	B4					
181	B5					
182	B6					
183	B7					
184	B8					
185	B9					
186	BA					
187	BB					
188	BC					
189	BD					
190	BE					
191	BF					
192	C0				{	left brace
193	C1				A	
194	C2				B	
195	C3				C	
196	C4				D	
197	C5				E	
198	C6				F	
199	C7				G	
200	C8				H	
201	C9				I	
202	CA					
203	CB					
204	CC					
205	CD					
206	CE					
207	CF					
208	D0				}	right brace
209	D1				J	
210	D2				K	
211	D3				L	
212	D4				M	

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
213	D5				N	
214	D6				O	
215	D7				P	
216	D8				Q	
217	D9				R	
218	DA					
219	DB					
220	DC					
221	DD					
222	DE					
223	DF					
224	E0				\	left slash
225	E1					
226	E2				S	
227	E3				T	
228	E4				U	
229	E5				V	
230	E6				W	
231	E7				X	
232	E8				Y	
233	E9				Z	
234	EA					
235	EB					
236	EC					
237	ED					
238	EE					
239	EF					
240	F0				0	
241	F1				1	
242	F2				2	
243	F3				3	
244	F4				4	
245	F5				5	
246	F6				6	
247	F7				7	
248	F8				8	
249	F9				9	
250	FA				|	vertical line

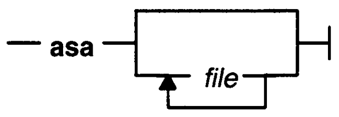
Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
251	FB					
252	FC					
253	FD					
254	FE					
255	FF				EO	eight ones

Appendix C. FORTRAN Specific AIX Commands

You can use the following two IBM AIX Version 3 for RISC System/6000 commands with XL FORTRAN files.

asa Command

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



The **asa** command processes either the files whose names are given as arguments, or the standard input stream, if no file name is supplied. The first character of each line is assumed to be a control character. The control characters and their meanings are:

First Character of Record	Vertical Spacing before Printing
Blank	Single new–line before printing (blank character).
0	Double new–line before printing.
1	New page before printing.
+	Overprint previous line.

Lines beginning with characters other than the defined control characters are treated as if they begin with a blank character. If any such lines appear, an appropriate diagnostic appears on standard error. The first character of a line is not printed.

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

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:

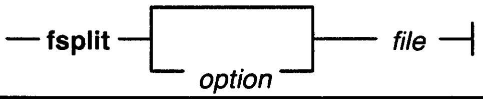
```
a.out | asa > lpr
```

directs the output produced by the program `a.out`, properly formatted and paginated, to the line printer `lpr`.

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

fsplit Command

The **fsplit** command splits specified FORTRAN source program files into several files.



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
- Program
- Subroutine.

The naming conventions for the resulting files are as follows:

- If the program or subprogram is explicitly named *name*, the resulting file is *name*.**f**. For example, if you use the **fsplit** command on the following program:

```
      program splitup
      ...
      call split2
      ...
      stop
      end
c
      subroutine split2
      ...
      return
      end
```

  the resulting files are `splitup.f` and `split2.f`.

- If you have an unnamed block data subprogram, the resulting file will be **blockdata1.f**.
- If you do not have a program or subprogram statement, the resulting file will be **main.f**.

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.
s	Strip FORTRAN input lines to 72 or fewer characters, and remove trailing blanks.

Appendix D. XL FORTRAN Internal Limits

The following table lists the internal XLFORTRAN compiler limits. The numbers within parentheses in the table correspond to the notes which follow the table.

Language Feature	Limit
Maximum length of a free–form statement	6600
Maximum number of symbolic names in a program unit	(1)
Maximum Hollerith constant length	255
Maximum number of computed **GOTO** statement numbers	999
Maximum **DO** loop and implied **DO** loop nesting level	(2)
Maximum block **IF** statement nesting level	(2)
Maximum number of nested **INCLUDE** compiler directives	16
Maximum number of parentheses groups in a format	500
Maximum numeric format field width	2000
Maximum character format field width	32767
Maximum number of times a format code can be repeated	32767

Figure 7. Internal FORTRAN Compiler Limits

Notes on compiler limits:

1. The number of symbolic names is limited by the size of the dictionary. The default dictionary size is 2048, but may be modified through the **ST_SIZE** option.
2. The maximum **DO** loop and block **IF** nesting level has a default value of 50, but may be modified through the **BK_SIZE** option.

Appendix E. Migration Considerations

This appendix outlines some items for consideration if you are porting source code to XL FORTRAN. The XL FORTRAN compiler provides no object code compatibility on the IBM AIX RISC System/6000 computer, so you must recompile any source code before it can be run.

Compatibility with the ANSI Standard

XL FORTRAN is fully compatible with the ANSI X3.9–1978 standard (Full ANSI FORTRAN 77). If you specify the **FIPS** compiler option, all features which are extensions to the ANSI standard are flagged with a warning message.

Compatibility with SAA FORTRAN

XL FORTRAN is fully compatible with SAA FORTRAN Release 1.0. If you specify the **SAA** compiler option, all features which are extensions to SAA FORTRAN Release 1.0 are flagged with a warning message.

Compatibility with RT PC FORTRAN 77

XL FORTRAN is source code compatible with RT PC FORTRAN 77 with the exception that character constants outside **DATA** statements are not followed by a null character as they are in RT PC FORTRAN 77. The purpose of this feature is to provide compatibility with the C language. XL FORTRAN allows you to append the null character to character constants.

Compatibility with VS FORTRAN (S/370)

This section lists the major differences between XL FORTRAN and VS FORTRAN, both in terms of the language specification and the implementation.

Language differences

Extended precision data types

XL FORTRAN handles any extended precision real or complex data as double precision real or complex data respectively, since the IBM AIX RISC System/6000 computer does not support extended precision floating point data. Such data is allocated appropriate storage (16 and 32 bytes respectively) to maintain equivalence relationships.

Floating–point representation

XL FORTRAN uses the ANSI/IEEE binary floating–point representation. This results in different ranges of representable values for real data:

Compiler	Data	Length (bytes)	Absolute minimum	Absolute maximum	Precision
XL	Real	4	0.117 549 4 E– 37	0.340 282 4 E+ 39	Approximately 8 decimal digits
VS	Real	4	0.539 760 5 E– 78	0.723 700 5 E+ 76	Approximately 6 decimal digits
XL	Real	8	0.222 507 4 D–307	0.179 769 3 D+309	Approximately 17 decimal digits
VS	Real	8	0.539 760 5 D– 78	0.723 700 6 D+ 76	Approximately 15 decimal digits
XL	Real	16	0.222 507 4 D–307	0.179 769 3 D+309	Approximately 17 decimal digits
VS	Real	16	0.539 760 5 Q– 78	0.723 700 6 Q+ 76	Approximately 32 decimal digits

Note: Ranges specified are for normalized real values.

Keyed access input/output

The IBM AIX RISC System/6000 computer does not provide the facilities required for keyed access input/output. Therefore, XL FORTRAN does not include the VS FORTRAN statements for keyed access input/output:

- **READ** with keyed access
- **WRITE** with keyed access
- **REWRITE**.

Asynchronous input/output

XL FORTRAN does not provide the asynchronous input/output statements:

- Asynchronous **READ**
- Asynchronous **WRITE**
- **WAIT**.

Note also that VS FORTRAN only provides asynchronous I/O under MVS.

BLANK specifier

XL FORTRAN assumes a default of '**NULL**' for the **BLANK** specifier when reading from a unit for which an **OPEN** statement was not specified. The default for VS FORTRAN is '**ZERO**'.

INCLUDE statement

XL FORTRAN allows apostrophes and quotation marks as delimiters (as well as parentheses).

Debug statements

XL FORTRAN does not include the VS FORTRAN debug statements:

- **AT**
- **DEBUG**
- **END DEBUG**
- **DISPLAY**
- **TRACE OFF**
- **TRACE ON**.

Backslash in character and Hollerith constants

For compatibility with C language usage, XL FORTRAN treats the backslash as an escape character. Two consecutive backslashes within a character or Hollerith constant represent the backslash character.

Hexadecimal constants: truncation

The truncation of hexadecimal constants by XL FORTRAN is always on the left, but in VS FORTRAN the truncation is on the left for numeric data and on the right for character data. For example:

```
CHARACTER C /Z4122/          ! 'A' (EBCDIC) in VS FORTRAN,
                             ! 'B' (ASCII) in XL FORTRAN
INTEGER*2 I /Z000001/        ! 1 in VS FORTRAN, 1 in XL FORTRAN
```

Intrinsic functions

XL FORTRAN does not provide the following intrinsic functions:

- **COTAN**
- Extended precision intrinsic functions.

Extended error handling subroutines

XL FORTRAN does not provide the extended error handling subroutines available in VS FORTRAN:

- **ERRMON**
- **ERRSAV**
- **ERRSET**
- **ERRSTR**
- **ERRTRA**.

Service and utility subprograms

XL FORTRAN does not provide the service and utility subprograms available in VS FORTRAN:

- **DVCHK**
- **OVERFL**
- **DUMP/PDUMP**
- **CDUMP/CPDUMP**
- **EXIT**
- **SDUMP**
- **XUFLOW**.

Implementation differences

Character passing to subprograms

Characters are passed in two words, one containing the address of the string and the other containing the length. The words containing the lengths of character arguments appear after all words containing addresses. This facilitates interlanguage calls.

For example:

```
CHARACTER C1,C2
INTEGER   I1,I2
CALL SUBR(C1,I1,C2,I2)
```

In the above example, the call to **SUBR** would result in the following parameter list being set up:

```
addr(C1),addr(I1),addr(C2),addr(I2),len(C1),len(C2)
```

VS FORTRAN uses a shadow parameter to pass the length of the string. Only the register containing the address appears in the call.

Normalization by adding zero

```
X = X + 0
```

This assignment is eliminated through optimization performed by the XL FORTRAN compiler back end. In contrast, VS FORTRAN interprets this assignment as a request to normalize the floating–point number X.

Compatibility with RT PC VS FORTRAN

The XL FORTRAN compiler is source code compatible with RT PC VS FORTRAN, except for:

- Mode compiler options (**IBM**, **R1**, **AN**, **VX**)
- The following feature of R1 mode:
  - Character constants outside **DATA** statements are followed by a null character.
- The following features of VX mode:
  - **BYTE** data type
  - **VIRTUAL** statement
  - Extended range of a **DO** loop
  - **TYPE** and **ACCEPT** statements
  - Logicals or integers in arithmetic or logical expressions
  - **Q** edit–descriptor for character count editing
  - **NAMELIST** statement allowed anywhere before use
  - *'rn* record number specifier in direct input/output statements
  - Certain intrinsic functions (for example, trigonometric degree functions).

Appendix F. Single Precision Floating Point Overflow

The following information is based on the IBM RISC System/6000 hardware technical reference manual description of floating point arithmetic. It describes an error in the implementation of the **frsp** (floating round to single precision) instruction which may affect some single precision floating point results. The **XFLAG=DD24** compiler option generates instructions to avoid the error as described here.

The frsp Instruction

The Floating Round to Single Precision (**frsp** or **frsp.**) instruction may produce incorrect results when all of the following conditions are met:

1. The **frsp** is dependent on a previous floating point arithmetic operation. Dependent means that it uses the target register of the arithmetic operation as the source register.
2. Less than two nondependent floating point arithmetic operations occur between the **frsp** and the operation on which it is dependent.
3. The magnitude of the double precision result of the arithmetic operation is less than 2**128 before rounding.
4. The magnitude of the double precision result after rounding is exactly 2**128.

If the error occurs, the magnitude of the result placed in the target register is 2**128:

```
X'47F0000000000000'    or    X'C7F0000000000000'
```

This is not a valid single precision value. A single precision store of this value will store the same value, plus or minus infinity, as if the **frsp** had executed correctly. But the result in the target register is the double precision representation of 2**128.

Effects of Compiler Option XFLAG=DD24

One way to avoid this error is to insure that two nondependent floating point operations are placed between a floating point arithmetic operation and the dependent round to single precision. The target registers for these operations should not be the same register that is a source register for the **frsp**.

If you use the **XFLAG=DD24** option, the compiler detects the first two conditions above that are necessary for this error. The compiler inserts two no–op **lrfl** instructions between the nondependent floating point operation and the dependent **frsp**. That eliminates one of the conditions necessary for the error.

The **XFLAG=DD24** option only affects the results of floating point calculations where the magnitude of the double precision result is less than 2**128, the result is rounded to single precision, and the magnitude of the rounded result is exactly 2**128. The effect of the **XFLAG=DD24** option is that the rounded result is always treated as a single precision infinity, and not as a valid double precision value 2**128.

The extra no–op **lrfl** instructions inserted by the **XFLAG=DD24** option may degrade the performance of the compiled program. These extra **lrfl** instructions are most likely to be inserted if you compile with the **OPT** and **–qrndsngl** compiler options.

Glossary

This is a glossary of commonly used terms in the *User's Guide for IBM AIX XL FORTRAN Compiler/6000* and the *Reference Manual for IBM AIX XL FORTRAN Compiler/6000*. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*, SC20–1699.

alphabetic character
A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as $ and _) allowed by a particular language.

alphanumeric
Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII)
The code developed by ANSI for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII character set consists of 7–bit control characters and symbolic characters.

American National Standards Institute (ANSI)
An organization sponsored by the Computer and Business Equipment Manufacturers Association through which accredited organizations create and maintain voluntary industry standards.

argument
A parameter passed between a calling routine and a called routine.

arithmetic constant
A constant of type integer, real, double precision, or complex.

arithmetic expression
One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator
A symbol that directs a compiler to perform an arithmetic operation. The arithmetic operators for XL FORTRAN are:

+	`addition`
–	`subtraction`
*	`multiplication`
/	`division`
**	`exponentiation`

array
A variable that contains an ordered group of data objects. All objects in an array have the same data type.

array declarator
The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element
A single data item in an array, identified by the array name followed by one or more integer expressions called subscript expressions indicating its position in the array.

array name
The name of an ordered set of data items.

assignment statement
An operation that assigns stores the value of the right operand in the storage location of the left operand.

binary
Pertaining to a system of numbers to the base two; the binary digits are 0 and 1.

binary constant
A constant that is made of one or more binary digits.

blank common
An unnamed common block.

block data subprogram
A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

character constant
A string of one or more alphabetic characters enclosed in single or double quotation marks.

character expression
A character constant or variable, a character array element, a character substring, a character valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

character operator
A symbol which represents an operation, such as concatenation (//)., to be performed on character data.

character string
A sequence of consecutive characters.

character substring
A contiguous portion of a character string.

character type
A data type that consists of alphanumeric characters. See also **data type**.

collating sequence
The sequence in which characters are ordered within the computer for sorting, combining, or comparing. The collating sequence for IBM AIX Version 3 for RISC System/6000 is ASCII.

comment
A language construct for the inclusion of text in a program that has no impact on the execution of the program.

common block
A storage area that may be referred to by a calling program and one or more subprograms.

compilation time
The time during which a source program is translated from a high–level language to a machine language program.

compilation unit
A portion of the computer program that is sufficiently complete to be compiled correctly.

compile
To translate a program written in a high–level programming language into a machine language program.

compiler
A program that translates instructions written in a high–level programming language into machine language.

compiler directive
A statement that controls what the compiler does rather than what the user program does.

complex constant
An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

complex number
A number consisting of a ordered pair of real numbers, expressible in the form a+bi, where a and b are real numbers and i squared equals minus one.

complex type
In XL FORTRAN, a data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected unit
In XL FORTRAN, a unit that is connected to a file in one of three ways: explicitly via the OPEN statement to a named file, implicitly via an OPEN statement to an unnamed file, or implicitly by a READ or WRITE statement to a unit for which no OPEN statement has been specified.

constant
A data item with a value that does not change. Contrast with variable. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and untyped data (hexadecimal, octal, and binary).

continuation line
A line of a source statement into which characters are entered when the source statement cannot be contained on the previous line or lines.

control statement
A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as IF, or an imperative statement such as STOP.

data
1. A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means.
2. In FORTRAN, data includes constants, variables, arrays, and character strings.

data item
A unit of data to be processed. Includes constants, variables, array elements, or character strings.

data type
The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character

debug
To detect, locate, and remove mistakes from a program.

default value
A value stored in the system that is used when no other value is specified.

digit
A character that represents a non–negative integer. For example, any of the numerals from 0 through 9.

DO loop
A range of statements invoked repetitively by a DO statement. See also **range of a DO loop**.

DO variable
A variable, specified in a DO statement, that is initialized or increased prior to each occurrence of the statement or statements within a DO range. It is used to control the number of times the statements within the range are executed. See also **range of a DO loop**.

double precision constant
A processor approximation to the value of a real number that occupies 8 consecutive bytes of storage and may assume a positive, negative, or zero value. The precision is greater than that of type real.

dimension
The attribute of size given to arrays and tables

dummy argument
A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

EBCDIC
Extended binary–coded decimal interchange code. A code developed for the representation of textual data. EBCDIC consists of a set of 256 eight–bit characters.

edit
1. To modify the form or format of data; for example, to insert or remove characters such as for dates or decimal points.
2. To check the accuracy of information that has been entered, and to indicate if an error is found.

embedded blanks
Blanks that are surrounded by any other characters.

executable program
A program that can be executed as a self–contained procedure. It consists of a main program and, optionally, one or more subprograms or non–FORTRAN–defined external procedures, or both.

executable statement
A statement that causes an action to be taken by the program; for example, to calculate, to test conditions, or to alter normal sequential execution.

existing file
A file that has been defined and, conceptually, resides on the storage medium.

existing unit
A valid unit number that is system specific.

exponent
A number, indicating to which power another number (the base) is to be raised.

expression
A language construct for computing a value from one or more operands, such as literals, identifiers, array references, and function calls.

external routine
A procedure or function called from outside the program in which the routine is defined.

field
An area in a record used to contain a particular category of data.

file
A sequence of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

floating–point constant
A constant representation of a floating–point number expressed as an optional sign followed by one or more digits an including a decimal number. See also **floating–point number**.

floating–point number
A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating–point base to a power indicated by the second numeral.

fold
To translate the lowercase characters of a character string into uppercase.

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 data
Data that is transferred between main storage and an input/output device according to a specified format. See also **list–directed data** and **unformatted data**.

FORTRAN (FORmula TRANslation)
A high–level programming language used primarily for scientific, engineering, and mathematical applications.

function
A routine that returns the value of a single variable and that usually has a single exit. See also, **function subprogram**, **intrinsic function**, and **statement function**.

function reference
The appearance of an intrinsic function name or a user function name in an expression.

function subprogram
In XL FORTRAN, a subprogram headed by a FUNCTION statement.

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).

hexadecimal constant
A constant, usually starting with special characters, that contains only hexadecimal digits.

Hollerith constant
A string of any characters capable of representation in the processor and preceded with nH, where n is the number of characters in the string.

implied DO
An indexing specification (similar to a DO statement, but without specifying the word DO) with a list of data elements, rather than a set of statements, as its range.

input
Data to be processed.

input/output (I/O)
Pertaining to either input or output, or both.

input/output list
A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

integer
A positive or negative whole number or zero.

integer constant
A string of decimal digits containing no decimal point.

integer expression
An arithmetic expression whose values are of integer type.

integer type
An arithmetic data type that consists of integer values.

intrinsic function
In XL FORTRAN, a function that is supplied with the run time environment that performs mathematical, character, bit manipulation, or logical operations.

I/O
See **input/output**.

I/O list
See **input/output list**.

keyword
A specified sequence of characters that are significant to the compiler in a particular context. No sequence of characters is reserved in all contexts.

length specification
A source language specification of the number of bytes to be occupied by a variable or an array element.

letter
An uppercase or lowercase character from the set A through Z.

link–editing
To create a loadable computer program by means of a linkage editor.

linkage
The part of the program that passes control and parameters between separate portions of the computer program.

linkage editor
A program that resolves cross–references between separately compiled or assembled object modules and 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.

list–directed
An input/output specification that uses a data list instead of a FORMAT specification.

list–directed data
Data that is transferred between main storage and an input/output device according to the length and type of variables in the input/output list. See also **formatted data** and **unformatted data**.

literal
A symbol or a quantity in a source program that is itself data, rather than a reference to data.

logical constant
A constant with a value of either true or false.

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
A symbol that represents an operation on logical expressions:

`.NOT.`	`(logical negation)`
`.AND.`	`(logical conjunction)`
`.OR.`	`(logical union)`
`.EQV.`	`(logical equivalence)`
`.NEQV.`	`(logical nonequivalence)`
`.XOR.`	`(logical nonequivalence)`

logical primary
A primary that can have the value true or false.

logical type
A data type that contains values true and false.

looping
A sequence of instructions performed repeatedly until an ending condition is reached. Usually controlled by a DO statement.

main
The default name given to a main program by the compiler if the main program was not named by the programmer.

main program
The first program unit to receive control when a program is run. Contrast with **subprogram**.

message
An error indication, or any brief information that a program writes to standard error or a queue.

name
A sequence of 1 to 250 letters or digits, the first of which must be a alphabetic, that identifies a data object.

named common
A separate common block consisting of variables, and arrays, and given a name.

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
A DO loop or DO statement whose range is entirely contained within the range of another DO statement.

nonexecutable program unit
A block data subprogram.

nonexecutable statement
A statement that describes the characteristics of a program unit, of data, of editing information, or of statement functions, but does not cause any action to be taken by the program.

nonexisting file
A file which does not physically exist on any accessible storage medium.

numeric
Pertaining to any of the digits 0 through 9.

numeric character
Synonym for digit.

numeric constant
A constant that expresses an integer, real, or complex number.

octal
Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

octal constant
A constant that is made of octal digits.

one–trip DO–loop
A DO loop that is executed at least once, if reached, even if the iteration count is equal to 0.

output
The result of processing data.

pad
To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

parameter
A variable that is given a constant value for a specified application and that either is used as input or controls the actions of the procedure or program.

preconnected file
A unit or file that was defined at installation time. For example, standard input and standard output are preconnected files.

predefined convention
The implied type and length specification of a data item, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, $, and _ imply type real of length 4.

primary
An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure
A sequenced set of statements that can be used at one or more points in one or more computer programs, that is usually given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, external functions, statement functions, and intrinsic functions.

program
1. A sequence of instructions suitable for processing by a computer. Processing can include the use of an assembler, compiler, interpreter, or translator to prepare the program for execution. See **source program**.
2. To design, write, and test computer programs.

program unit
A main program or a subprogram.

random access
An access method in which records can be read from, written to, or removed from a file in any order.

range of a DO loop
Those statements that physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed.

real constant
A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

real number
A number, containing a decimal point, stored in fixed–point or floating–point format.

real type
An arithmetic data type that can approximate the values of real numbers.

record
An aggregate that consists of data objects, possibly with different attributes, that usually have identifiers attached to them.

relational expression
An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression. The result is a value that is true or false.

relational operator
The words or symbols used to express a relational condition or a relational expression:

`.GT.`	`greater than`
`.GE.`	`greater than or equal to`
`.LE.`	`less than or equal to`
`.EQ.`	`equal to`
`.NE.`	`not equal to`

relative record number
A number that specifies the location of a record in relation to a base position in the file containing it.

run
To cause a program, utility, or other machine function to be performed.

scale factor
A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

scope
The portion of a program within which a declaration applies.

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.

source program
A set of instructions that are written in a programming language and that must be translated to machine language before the program can be executed.

specification statement
One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies information required to allocate data storage.

statement
A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

statement function
A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression, and which can be used as a substitute for the expression throughout the program.

statement function definition
A statement that defines a statement function. Its form is a statement function, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement label
A number from one through five decimal digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO, or to refer to a FORMAT statement.

statement number
See **statement label**.

subprogram
In XL FORTRAN, a subprogram that has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. Contrast with **main program**.

subroutine
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. See also **function subprogram** and **statement function**.

subscript
A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity
In XL FORTRAN, a component of a subscript. A subscript quantity is an integer or real constant, variable, or expression.

substring
A part of a character string.

symbolic name
In a programming language, a unique name used to represent an entity such as a file or data item. See also **name**.

syntax
The rules for the construction of a statement.

System Application Architecture (SAA) FORTRAN
A superset of the ANSI X3.9 – 1978 FORTRAN 77 standard.

type declaration
The specification of the type and, optionally, the length of a constant, variable, array, or function using an explicit type specification statement. Contrast with **predefined convention**.

unformatted record
A record that is transmitted unchanged between internal and external storage. See also **formatted record** and **list–directed data**.

unit
A means of referring to a file to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit identifier
The number that specifies an external unit or internal file. The number can be one of the following:
1. An integer expression whose value must be zero or positive.
2. An asterisk (*) that corresponds to unit 5 for input or unit 6 for output.
3. The name of a character array, character array element, or character substring for an internal file.

variable
1. A quantity that can assume any of a given set of values.
2. A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution.

zero suppression
The substitution of blanks for leading zeros in a number. For example, 00057 becomes 57 when using zero suppression.

Reader's Comment Form

Jser's Guide for IBM AIX XL FORTRAN Compiler/6000

3C09–1257–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, 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.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date	________
Your Name	________
Company Name	________
Mailing Address	________
	________
	________
Phone No.	( ) ________ Area Code

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 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold Fold

Cut or Fold Along Line

Fold and Tape Please Do Not Staple Fold and Tape

SC09-1257-00