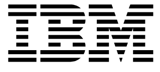


AIX XL Pascal/6000

SC09-1756–00

User's Guide

Version 2.1

Note!

Before using this information, and the product it supports, be sure to read the general information under “Notices” on page v.

Second Edition (December 1993)

This edition applies to Version 2, Release 1 of IBM AIX XL Pascal Compiler/6000 (Program 5765–245), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile to (attention: RCF Coordinator), or you can send your comments electronically to IBM. Please see “Communicating Your Comments to IBM” for a description of the methods. This page immediately precedes the Readers’ Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1990, 1993. All rights reserved.**

Note to US Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
IBM is a registered trademark of International Business Machines Corporation, Armonk, N.Y.

Contents

Notices . . . v

Chapter 1. Introduction . . . 1
How to Read Syntax Diagrams . . . 2
Pascal Industry Standards . . . 5
Related Publications . . . 6
Typographical Conventions . . . 7

Chapter 2. Features of the XL Pascal Compiler . . . 9
Compiler Installation . . . 10

Chapter 3. Compiling, Linking, and Running Programs . . . 13
Invoking the XL Pascal Compiler . . . 13
The XL Pascal Compilation Environment . . . 14
The XL Pascal Configuration File . . . 15
Input Files to the xlp Command . . . 19
Output Files from the xlp Command . . . 20
Prime Files . . . 21
Compiler Options . . . 22
Summary of the Compiler Options . . . 25
Detailed Descriptions of the Compiler Options . . . 29
Invoking the Linkage Editor . . . 40
Running a Program . . . 41
The XL Pascal Runtime Environment . . . 41

Chapter 4. Input and Output Facilities . . . 43
Environment-Determined File Names . . . 43
Opening Files for Input and Output . . . 45
File Opening Procedures . . . 50
Processing a TEXT File . . . 53
Processing a Record File . . . 61
Closing a File (CLOSE) . . . 65
Appending Data to a File . . . 66
File-Name Association . . . 66

Chapter 5. Improving Performance . . . 69
Optimization Levels . . . 69
Optimization Techniques . . . 70
Making Your Programs More Efficient . . . 71

© Copyright IBM Corp. 1990, 1993

Chapter 6. Problem Determination 73
Compiler Listings 73
Error Messages 79
Message Catalog Errors 86

Chapter 7. Interlanguage Applications 89
Interlanguage Reference Requirements 89
External Names 89
Matching Data Types 90
Character Variable Types 92
Storage of Arrays 93
Pascal Arrays, C Pointers, and Arrays 93
Routine Calls and Returned Values 94
Routine Linkage Convention 95
Interlanguage Parameter Passing Conventions 97
Enforcement of Type Matching 104

Appendix A. Example Program 105

Appendix B. ASCII Character Set 107

Appendix C. XL Pascal and the 1983 ANSI/IEEE Pascal Standard 111
Implementation-Defined Features 111
Implementation-Dependent Features 113
Error Handling 114
Extension Handling 116

Appendix D. Implementation Dependencies 117
Routines That Can Be Passed as Parameters 117
Data Types 117
Compiler Limits 118
Differences between XL Pascal and the VS Pascal Release 1 Licensed Program 119
Differences between XL Pascal and the VS Pascal Release 2 Licensed Program 121

Appendix E. Data Storage 123
Storage Classes 123
Data Size and Boundary Alignment 124
Dynamic Storage Management 129

Appendix F. Single Precision Floating-Point Overflow 133

Glossary 135

Index 141

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*), used in this publication, are trademarks or service marks of IBM Corporation in the United States or other countries:

AIX	IBM	IBMLink
POWER2	PowerPC	PROFS
RISC System/6000	RT PC	RT
System/370		

The following terms, denoted by a double asterisk (**), used in this publication, are trademarks of other companies as follows:

ANSI	American National Standards Institute, Inc.

© Copyright IBM Corp. 1990, 1993

Chapter 1. Introduction

This book describes the IBM* AIX* XL Pascal Compiler/6000, and shows how to compile, link, and run programs written in XL Pascal. It also describes how to use input and output (I/O) facilities and storage, and how to do interlanguage calls.

The exceptional (XL) family of compilers provides consistency and high performance across multiple programming languages by sharing the same code optimization technology. The XL Pascal Compiler/6000 is an optimizing compiler for the Pascal language for AIX Version 3 for the RISC System/6000*system. It allows invocation of routines written in other programming languages, and the creation of routines that can be invoked by programs written in other languages. It also provides detailed compile-time and runtime diagnostics, error recovery, and debugging facilities.

XL Pascal compiles programs written using the American National Standards Institute (ANSI) Pascal language with selected VS Pascal extensions. The publication *American National Standard Pascal Computer Programming Language*, ANSI/IEEE 770X3.97–1983 details the Pascal language standard. The XL Pascal language is described in the *Language Reference for IBM AIX XL Pascal Compiler/6000*, SC09–1757.

Note: For brevity, the IBM* AIX* XL Pascal Compiler/6000 is referred to throughout this book as *XL Pascal*. The AIX Version 3 for the RISC System/6000* is referred to as *AIX*.

Who Should Use This Book

You should use this book if you are already familiar with Pascal and want to use the XL Pascal compiler to write or maintain applications in XL Pascal. You should be familiar with the AIX Version 3 Operating System, and have some previous programming experience. If you are not familiar with the operating system, refer to *AIX Version 3.2 System User's Guide: Operating System and Devices,* GC23–2522.

This book does not contain detailed information about how to write a program in XL Pascal. If you have no prior Pascal knowledge or familiarity with a high-level programming language, you may first want to obtain any of the tutorial-style Pascal books that are commercially available.

For detailed information about Pascal and Pascal language keywords and definitions, refer to the *Language Reference for IBM AIX XL Pascal Compiler/6000* manual.

How to Use This Book

This book is organized in the order of the steps necessary to compile, link-edit, and run a program using the XL Pascal Compiler/6000.

If you have not used a compiler on the RISC System/6000 computer before, you should first read Chapter 2, “Features of the XL Pascal Compiler” and Chapter 3, “Compiling, Linking, and Running Programs.” The remaining chapters deal with more advanced topics. After you are familiar with the system and the compiler, you can use this book as a handy reference.

Use this book along with the *Language Reference for IBM AIX XL Pascal Compiler/6000*.

© Copyright IBM Corp. 1990, 1993

How This Book Is Organized

Each of the following chapters is devoted to a major concept or feature of the XL Pascal compiler:

Chapter 1, "Introduction" is the chapter you are reading now. It introduces the major elements of the compiler, the standards it conforms to, and tells you where to go for more information.

Chapter 2, "The XL Pascal Compiler" summarizes what you should know about the compiler before creating a Pascal source program.

Chapter 3, "Compiling, Linking, and Running Programs" shows how to invoke the compiler, specify compiler options, invoke the linkage editor, and run your compiled program.

Chapter 4, "Input and Output Facilities" shows how to use the XL Pascal I/O facilities under the AIX Version 3.2 operating system.

Chapter 5, "Improving Performance" shows some of the optimizations XL Pascal performs and how to make your programs more efficient.

Chapter 6, "Problem Determination" shows how to use error messages, the compiler listing, and the symbolic debugger to find and eliminate problems in your programs.

Chapter 7, "Interlanguage Applications" shows how to write function and procedure calls and external data references among the IBM AIX RISC System/6000 XL languages.

Appendix A, "Example Program" presents an example program, the listing it produces with the options specified, and the output from running it.

Appendix B, "ASCII Character Set" lists the standard ASCII characters in ascending numerical order, with the corresponding decimal and hexadecimal values and ASCII control characters with **Ctrl–** notation.

Appendix C, "XL Pascal and the 1983 ANSI/IEEE Pascal Standard" shows the implementation-defined and implementation-dependent features of XL Pascal, error handling, and extensions to the Pascal standard language.

Appendix D, "Implementation Dependencies" discusses the XL Pascal implementation dependencies, compiler limits, and other factors that affect programming. It also summarizes the differences between XL Pascal and VS Pascal.

Appendix E, "Data Storage" shows how XL Pascal data values are represented in storage, and how storage management operations are implemented.

Appendix F, "Single Precision Floating Point Overflow" shows a hardware error in the implementation of the **frsp** (floating round to single precision) instruction and how to use the **XFLAG=DD24** compiler option to avoid the error.

How to Read Syntax Diagrams

The following conventions are used in syntax diagrams:

- Keywords and reserved words are in bold uppercase letters (for example, **VAR**, **BEGIN**, and **END**). They can be written in uppercase or lowercase, but they must be spelled exactly as shown.
- Variables or identifiers that you supply are in all lowercase italics (for example, *label_dcl*).
- You enter punctuation marks, parentheses, arithmetic operators, and other nonalphabetic symbols as they are shown in the syntax.

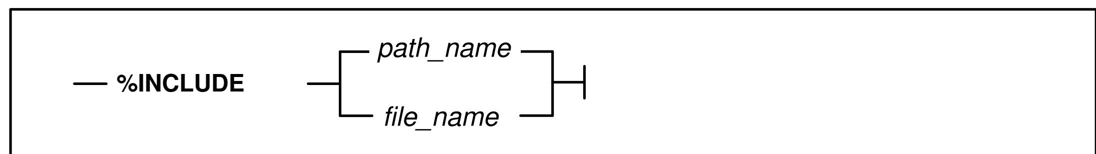
Read syntax diagrams from left to right, from top to bottom, following the path of the line:

- The following symbol indicates the beginning of a diagram: ──
- The following symbol indicates that the syntax continues on the next line: ─►
- The following symbol indicates that the syntax is continued from the previous line: ►──
- The following symbol indicates the end of the diagram: ─┤
- Syntactical units that are not complete statements start with the following symbol: ►──
- Syntactical units that are not complete statements end with the following symbol: ─►

Required and Optional Items

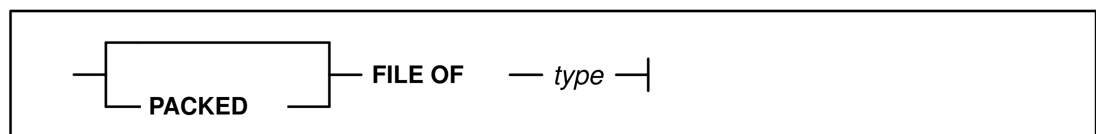
Required items are on the horizontal line (the main path).

Branching shows two paths through the syntax.



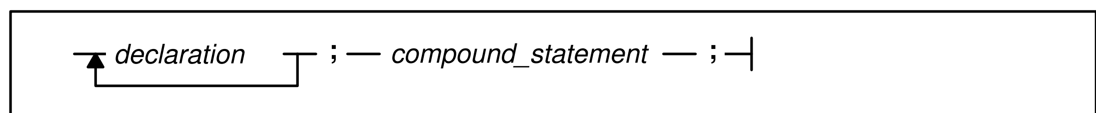
If you must choose one of three or more items, they are in a multiple choice box on the main path.

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

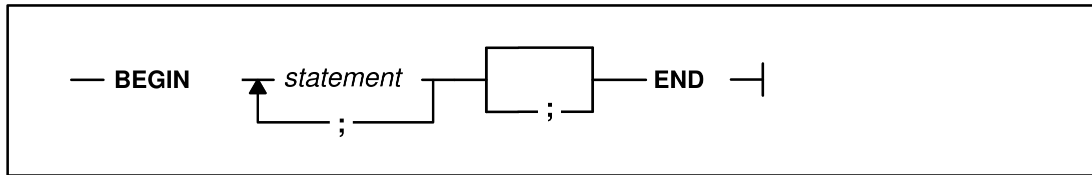


Repeatable Items

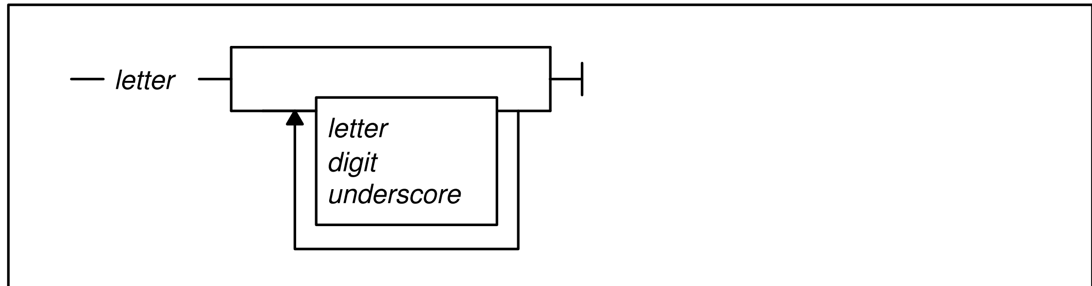
An arrow returning to the left below a line shows items that you can repeat.



Punctuation on a repeat arrow is always between the repeated items.



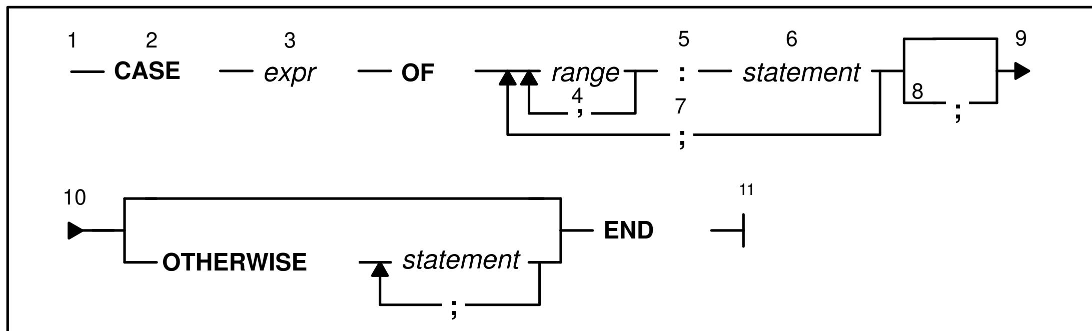
A repeat arrow below a multiple choice box indicates that you can choose one or more items in the box, but you must choose at least one.



Default Items

A heavy line is the default path.

Example



The diagram is interpreted as follows:

1. This is the start of the diagram.
2. Type the keyword **CASE**.
3. Type a valid expression followed by the word **OF**.
4. Type at least one range. For more than one range, separate each by a comma.
5. Type the colon symbol (:).
6. Type a valid statement.

7. Type the semicolon symbol (;).
8. This path is optional.
9. The diagram is continued at 10.
10. The diagram is continued from 9.
11. The end of the diagram.

The following **CASE** statements conform to the syntax shown in the syntax diagram:

```
CASE a_card.r OF
   ace:
      points := 11;
   two..ten:
      points := ORD( a_card.r ) + 1;
   OTHERWISE
      points := 10
END ;
```

```
CASE s OF
   triangle:
      area := 0.5 * side * base;
   rectangle:
      area := sidea * sideb;
   circle:
      area := 3.14159 * SQR(radius)
END ;
```

```
CASE s OF
   triangle:
      area := 0.5 * side * base
END ;
```

A Note about Examples

Examples in this book are written in a simple style. They do not attempt to conserve storage, check for errors, achieve fast run time, or demonstrate all possible uses of a language element.

Pascal Industry Standards

XL Pascal complies with the ANSI standard (commonly referred to as ANSI-83) defined in *American National Standard Pascal Computer Programming Language*, ANSI/IEEE 770X3.97–1983.

This standard is adopted by International Standards Organization (ISO) and Federal Information Processing Standards (FIPS). It also implies conformance to:

- International Standard ISO 7185–1983 (Level 0), Programming Languages, Pascal
- Federal Information Processing Standard, FIPS PUB 109, Pascal.

In this book, *Standard Pascal* or *Standard mode* refers to the ANSI-83 standard.

Related Publications

IBM Publications

- *Language Reference for IBM AIX XL Pascal Compiler/6000*, SC09–1757, includes a detailed description of the program structure, declarations, constants, data types, variables, expressions, statements, and routines in XL Pascal.
- *Installation Instructions for IBM AIX XL Pascal Compiler/6000*, GC09–1775, shows how to install the AIX XL Pascal Compiler/6000 Version 2.1.
- *VS Pascal Language Reference*, SC26–4320, provides definition of the VS Pascal programming language and its syntax.
- *VS Pascal Application Programming Guide*, SC26–4319, describes how to use the VS Pascal compiler and explains how to compile, link-edit, run, and debug VS Pascal programs.
- *AIX Version 3.2 System User's Guide: Operating System and Devices*, GC23–2522, introduces the AIX operating system to first-time users. It describes basic tasks, such as, logging in, running commands, working with files and directories, and using the information that supports the system.
- *AIX Version 3.2 Topic Index and Glossary*, GC23–2201, provides a glossary of terms used in AIX and RISC System/6000 publications. It also contains a list of some topics in the AIX and RISC System/6000 library and the books in which those topics are discussed.
- *AIX Version 3.2 Commands Reference* (4 volumes), GBOF–1802, contains descriptions and examples of the AIX commands and their available flags.
- *AIX Version 3.2 General Programming Concepts*, SC23–2205, discusses the AIX operating system from a programming perspective.
- *AIX Version 3.2 Technical Reference: Base Operating System and Extensions*, SC23–2198, describes AIX base operating system runtime services and device services.
- *AIX Version 3.2 Assembler Language Reference*, SC23–2197, describes the assembler language program implemented by the AIX Assembler Version 3.2.5.
- *Optimization and Tuning Guide for Fortran, C, and C++: AIX Version 3.2 for RISC System/6000*, SC09–1705, describes techniques you can use to improve the performance of programs compiled with the AIX compilers.

Non-IBM Publications

- *American National Pascal Computer Programming Language*, ANSI/IEEE 770X3.97–1983
- *International Standards Organization Programming Language Pascal*, (ISO) 7185–1983 (Level 0)
- *Federal Information Processing Standards Publication Pascal*, (FIPS) PUB 109
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754–1985

Typographical Conventions

Type style highlights important terms and elements of the compiler. The following kinds of information are distinguished by different typographical conventions.

Keywords and Reserved Words

Predefined identifiers, which you must write exactly as presented, are in **BOLD CAPITALS**. When used generically, these words are in lowercase. For example, the reserved word **TYPE** is in capital letters, but general references to data type are in lowercase.

New Terms

When a term is used for the first time, it is in *italics*, often followed by a brief definition. All of these terms and definitions are in the *AIX Version 3.2 Topic Index and Glossary*.

Multibyte Characters

VS mode XL Pascal permits the use of multibyte character set (MBCS) characters:

- A boldface capital **D** represents one multibyte character
- A boldface capital **B** represents one MBCS blank

Chapter 2. Features of the XL Pascal Compiler

The XL Pascal compiler is an IBM licensed program that operates within the IBM AIX RISC System/6000 environment.

Language Support

The XL Pascal compiler supports two language levels:

Standard is the Pascal ANSI-83 standard definition.

VS includes selected IBM VS Pascal Release 1 and Release 2 extensions.

The *Language Reference for IBM AIX XL Pascal Compiler/6000* gives a complete description of the XL Pascal language and extensions to the ANSI-83 Standard Pascal language.

Compiler Features

The XL Pascal compiler supports:

- Optimized object code
- VS Pascal and RT PC* VS Pascal compatibility (with some exceptions)
- Descriptive diagnostics
- Separate segment compilation
- Flexible command-line options
- Interlanguage communication among IBM AIX RISC System/6000 XL languages

Compiler Options

You can invoke the compiler with the **xlp** command and control its actions with the options provided. The compiler sets the return code to indicate the completion status of the program compilation, and also provides timing and resource usage data.

The compiler output listing has optional sections controlled by XL Pascal compiler options. By default, XL Pascal produces no listing.

Symbolic Debugger (dbx)

With the XL Pascal compiler options, you can generate debug information tables compatible with the AIX symbolic debugger, which is invoked with the command **dbx**.

Online Compiler Help

The **xlp** command with no arguments invokes online help that details available command-line options.

Migration Characteristics

The XL Pascal compiler aids code migration by providing source code compatibility with other compilers, but it does not provide object code compatibility with them.

- XL Pascal is source compatible with the following VS Pascal licensed programs, with some exceptions as outlined in Appendix D, “Implementation Dependencies”:
  - RT PC VS Pascal
  - VS Pascal Release 1 and Release 2

© Copyright IBM Corp. 1990, 1993

- The following symbol conventions may make your programs portable with VS mode:
  - @ or –> instead of ^ for pointer references and declarations
  - (* and *) or /* and */ instead of { and } for comments
  - NOT instead of the tilde (~) character
  - (. and.) instead of [and]
  - <> instead of ~= for not equal

Compiler Modes

XL Pascal offers two modes for compiling programs: Standard mode and VS mode. The **LANGLVL** compiler option sets the language mode. More than one mode is permitted in a program, but each separate compilation unit must be in a single mode. Work in the mode you need or the one you are most familiar with:

Standard Mode Uses only the ANSI-83 definition of Pascal. When you work in it, only programs that satisfy the syntax of the ANSI-83 standard are compiled.

VS Mode Is the default mode of the compiler. It allows you to compile code that uses the IBM VS Pascal Release 1 definition of Pascal, with selected features from VS Pascal Release 2.

Related Information

Invoking the compiler and using the compiler options are described in Chapter 3, “Compiling, Linking, and Running Programs”. The format of the compiler output listing is described in “Compiler Listings” on page 73. The *AIX Version 3.2 Commands Reference* describes the **dbx** symbolic debugger.

Compiler Installation

You install the compiler using the AIX **installp** command while logged on with root authority. The following list shows the files that are installed on the system and the source of the files:

- The **xlpcmp.obj installp** image contains:

  /usr/lpp/xlp/lib/xlpentry
  /usr/lpp/xlp/lib/README.xlp
  /usr/lpp/xlp/lib/xlp_prime
  /usr/lpp/xlp/lib/xlp_base_prime
  /usr/lpp/xlp/lib/xlp.pas
  /usr/lpp/xlp/lib/xlp_base.pas
  /usr/bin/xlp
  /usr/man/cat1/pascal
  /usr/lpp/xlp/lib/default_msg/xlp00.cat
  /usr/lpp/xlp/lib/default_msg/xlp01.cat
  /usr/lpp/xlp/lib/default_msg/xlp03.cat
  /usr/lpp/xlp/lib/default_msg/xlp.help

- The **xlpcmpEn_US.msg installp** image contains:

  /usr/lib/nls/msg/En_US/xlp00.cat
  /usr/lib/nls/msg/En_US/xlp01.cat
  /usr/lib/nls/msg/En_US/xlp03.cat
  /usr/lib/nls/msg/En_US/xlp.help

- The **xlprte.obj installp** image contains:

  /usr/lib/libxlp.a
  /usr/lpp/xlprtemsg/xlp02.cat

- The **xlprtemEn_US.msg installp** image contains:

  /usr/lib/nls/msg/En_US/xlp02.cat

Related Information

The *AIX Version 3.2 Commands Reference* describes the **installp** command. See the *Installation Instructions for IBM AIX XL Compiler/6000* for further information about installing XL Pascal.

Chapter 3. Compiling, Linking, and Running Programs

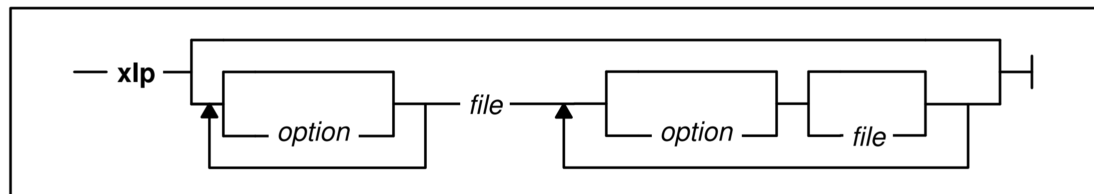
This chapter discusses the details of invoking the compiler, specifying the compiler options, invoking the linkage editor, and running your compiled program. It also describes the IBM AIX XL Pascal runtime environment.

Note: Before using the XL Pascal Compiler/6000, read the descriptions of the compiler options to understand the correct operation of the compiler.

Invoking the XL Pascal Compiler

To compile an XL Pascal source program, use the **xlp** command.

Syntax



Parameters

xlp is the command name.

option is a command line option. An option has one of the following forms:

- **–q***option* some of these options take one or more parameters.
- *option_flag* is usually a single letter preceded by an initial dash (–).

file is an input file to the compiler.

Description

The **xlp** command compiles the source files, links them with any specified object files in the order on the command line, and then produces an executable file called **a.out**. To specify an alternative name for this file, use the **–o** compiler option.

The following sequence of programs is run on each input file. Each program processes the source file according to the suffix of the file name and then sends the results to the next step in the sequence. No compiling is done if you list only object files after the **xlp** command.

1. The **xlp** command calls the compiler or the assembler:
   - If the input file has a **.pas** suffix, it calls the compiler. The default suffix can be changed with the **psuffix** attribute.
   - If the input file has a **.s** suffix, it calls the assembler. The default suffix can be changed with the **ssuffix** attribute.
2. If you did not specify the **–c** compiler option, the **xlp** command then calls the linkage editor.

© Copyright IBM Corp. 1990, 1993

Example

The command

```
xlp toplev.pas mappr.s seg1.pas seg2.o
```

has the following results:

- Files **toplev.pas** and **seg1.pas** are compiled by the compiler.
- File **mappr.s** is assembled by the assembler.
- The object (**.o**) files produced for **toplev.pas**, **seg1.pas**, and **mappr.s**, together with object code file **seg2.o**, are sent to the linkage editor, which produces the executable file **a.out**.

Related Information

The syntax of command line options is described in “Specifying Options on the Command Line” on page 23. “Input Files to the xlp Command” on page 19 describes *files* and input file suffixes. Detailed information about command line options and flags appears in “Compiler Options” on page 22.

The XL Pascal Compilation Environment

Before you compile your Pascal programs, make sure that the environment variables and the configuration file are set up to meet your needs.

Environment Variables for the Message Catalogs and Help Files

Before using the compiler, you must install the message catalogs and help files and set the following two environment variables:

LANG Specifies the national language locale name in effect for message catalogs and help files.

NLSPATH Specifies the full path name of message catalogs and help files.

You can set the **LANG** environment variable to any of the locales provided on the system. See the *AIX Version 3.2 System User's Guide: Operating System and Devices* manual for more information on configuring the National Language environment.

The locale name **En_US** is the national language code for United States English. If the appropriate message catalogs have been installed on your system, any other valid national language code can be substituted for **En_US**.

To determine the current setting of the national language on your system, use the following **echo** commands:

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

These environment variables are initialized when the operating system is installed, and may differ from the settings you want to use.

You use different commands to set the environment variables depending on the shell you are in, the Korn shell (**ksh**), Bourne shell (**bsh** or **sh**), or C shell (**csh**). To determine the current shell, use the **echo** command:

```
echo $SHELL
```

The Bourne shell path is **/usr/bin/bsh** or **/usr/bin/sh**. The Korn shell path is **/usr/bin/ksh**. The C shell path is **/usr/bin/csh**.

Setting Environment Variables from the Korn Shell or Bourne Shell

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

```
LANG=En_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
export LANG NLSPATH
```

You can use the two special variables, **%N** and **%L**, in the **NLSPATH** environment variable. The **%N** variable is the name of the catalog to be opened. The **%L** variable is the locale-specific directory containing message catalogs. The current value of the **LC_MESSAGES** locale category is used for the directory name.

To set the variables so that all users have access to them, add the commands to the file **/etc/profile**. To set them for a specific user only, add the commands to the file **.profile** in the user's home directory. The environment variables are set each time the user logs on.

Setting Environment Variables from the C Shell

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

```
setenv LANG En_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
```

In the C shell, you cannot set the environment variables so that all users have access to them. To set them for a specific user only, add the commands to the file **.cshrc** in the user's home directory. The environment variables are set each time the user logs on.

Related Information

For more information about using environment variables, see the *AIX Version 3.2 System User's Guide: Operating System and Devices* manual. For more information about the AIX national language facilities and setting **NLSPATH** and **LANG**, see the *AIX Version 3.2 General Programming Concepts* manual. For details about Korn shell commands and C shell commands, see the *AIX Version 3.2 Commands Reference*.

The XL Pascal Configuration File

The configuration file specifies information that the compiler uses when you invoke it. A configuration file consists of a series of stanzas, each of which consists of a label and a series of attribute definitions. The XL Pascal compiler uses the attribute definitions in one stanza of a configuration file.

The default configuration file used by XL Pascal is **/etc/xlp.cfg**, and the default stanza name is the name by which the compiler is invoked.

To specify a different stanza name, link the XL Pascal compiler to a different command name, as described in "Customizing Configuration File Stanzas" on page 17.

Configuration File Attributes

The configuration file contains the following attributes:

as
The path name to be used for the assembler.

asopt
List of options that, if seen on the command line, are directed to the assembler and not to the compiler. These flags override all normal processing by XL Pascal and are directed to the assembler as specified in the configuration file. The string is formatted for the AIX **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (:) if the corresponding flag takes a parameter.

crt
The path name of the object file passed as the first parameter to the linkage editor if you do not specify either the **–p** or **–pg** flag. The default is **/usr/lib/crt0.o**.

gcrt
The path name of the object file passed as the first parameter to the linkage editor if you specify the **–pg** flag. The default is **/usr/lib/gcrt0.o**.

ld
The absolute path name of the linkage editor.

ldopt
List of options that, if seen on the command line, are directed to the linkage editor and not to the compiler. These flags override all normal processing by XL Pascal and are directed to the linkage editor as specified in the configuration file. The string is formatted for the AIX **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (:) if the corresponding flag takes a parameter.

libraries
Library options, separated by commas, that XL Pascal passes as the last parameters to the linkage editor. This attribute specifies all the libraries the linkage editor is to use at link-edit time for which a corresponding profiled library does not exist. The default is **–lxlp,–lm,–lc**.

mcrt
The path name of the object file passed as the first parameter to the linkage editor if you specify the **–p** flag. The default is **/usr/lib/mcrt0.o**.

options
A string of option flags, separated by commas, to be processed by XL Pascal as if they are being entered on the command line.

osuffix
The suffix for object files. The default is **o**. The suffix cannot be the source code suffix for the XL Pascal language processor (**pas**).

proflibs
Binder library options, separated by commas, that XL Pascal uses to pick up profiled versions of the libraries when the linkage editor is called with one of the profiling flags **–p** or **–pg**.

The default is **–L/lib/profiled,–L/usr/lib/profiled**.

psuffix
The suffix for Pascal source programs. The default is **pas**.

ssuffix
The suffix for assembler programs. The default is **s**.

use
The values for attributes that are taken from the named stanza in addition to the local stanza. For single-valued attributes, values in the **use** stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the **use** stanza are added to the values from the local stanza.

xlp The path name to be used for the compiler. The default is **/usr/lpp/xlp/lib/xlpentry**.

xlpopt List of options that, if encountered on the command line, are directed to the compiler. These flags override all normal processing by XL Pascal and are directed to the compiler as specified in the configuration file. The string is formatted for the AIX **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (:) if the corresponding flag takes a parameter.

A Typical Configuration File

The configuration file in the following example contains the `xlp` stanza and the `DEFLT` stanza. The `xlp` stanza is used by the XL Pascal compiler, and the `DEFLT` stanza supplies default values for certain names.

```
* standard xlp compiler
xlp:   use   = DEFLT

* common definitions
* The "proflibs2" stanza sets the search order to locate the
* profiling
* versions of libraries before locating the non-profiling versions
*
* This stanza is used for dynamically binding with libxlp.a. If
* the intention is to bind with libxlp.a statically,two additional
* options need to be specified: -bnso,-bimport:/lib/syscalls.exp
DEFLT: xlp         = /usr/lpp/xlp/lib/xlpentry
       as          = /bin/as
       crt         = /lib/crt0.o
       mcrt        = /lib/mcrt0.o
       gcrt        = /lib/gcrt0.o
       ld          = /bin/ld
       libraries   = -lxlp,-lm,-lc
       proflibs    = -L/lib/profiled,-L/usr/lib/profiled
       options     = -T512,-H512,-qprime=xlp_prime,
-I/usr/lpp/xlp/lib,-bh:4
```

Customizing Configuration File Stanzas

Each stanza in a configuration file has a name. When an AIX program uses a configuration file, it gets attributes from the configuration file stanza whose name is the same as the program name. You can define versions of the XL Pascal compiler that have default properties different from the basic XL Pascal compiler. To define your own Pascal compilation environment, follow these steps:

1. Choose a name for your version of the compiler.
2. In the configuration file, insert a stanza with the same name as your compiler version name.
3. Copy the `xlp` stanza from the default XL Pascal configuration file **/etc/xlp.cfg** into this new stanza.
4. Change the new stanza to the properties you want for your version of the compiler.
5. Link your compiler version name to the **xlp** command using the AIX **ln** command:

   ln *xlp_name new_name*

where

xlp_name is the path name of the **xlp** command. Installation of the XL Pascal compiler usually puts this command in /**usr**/**bin**/**xlp**.

new_name is the path name of your version of the compiler.

Invoke your version of the compiler by using your compiler version name instead of the **xlp** command.

Note: To specify a different configuration file, stanza name, or both, use the **–F** compiler option, as described on page 29.

Example

Suppose you want command **stdp** to invoke the XL Pascal compiler with the default option **–qlanglvl=standard** and with **.pastd** as the default suffix for Pascal source files. The steps are outlined below.

1. Name this version of the compiler **stdp**.
2. Use the same default configuration file as for the **xlp** command, which is /**etc**/**xlp.cfg**.
3. Copy **xlp.cfg** to a directory under your home directory.
4. Copy the **xlp:** stanza into new stanza **stdp:**
5. In this new stanza, add the **psuffix** attribute set to **pastd**.
6. Copy the **options** attribute from the **DEFLT** stanza into the **stdp** stanza, and add **–qlanglvl=standard** to the **options** attribute.
7. Change to the directory under your home directory where you have copied the new **xlp.cfg**.
8. Use the following AIX command to put the new **stdp** command in /**usr**/**bin**, the same directory as the **xlp** command:

   ```
   ln /usr/bin/xlp ./stdp
   ```

9. To invoke the compiler with the **stdp** command, specify the full path or **./stdp**, if you are in the directory. You must also use the **–F** option with the path to the directory containing your new **xlp.cfg**:

   ```
   ./stdp –F./xlp.cfg example.pastd
   ```

With this modified configuration file, you now use the **stdp** command to run the XL Pascal compiler. The **stdp** command now treats files with the suffix **.pastd** as Pascal source files, and always sets the option **langlvl=standard**.

The following example shows your customized configuration file:

```
* standard xlp compiler
xlp:  use        = DEFLT
* new stanza with different psuffix and -qlanglvl=standard
stdp: use        = DEFLT
      psuffix    = pastd
      options    = -T512,-H512,-qprime=xlp_prime,
-I/usr/lpp/xlp/lib,-bh:4,-qlanglvl=standard

* common definitions
* The "proflibs2" stanza sets the search order to locate the
* profiling versions of libraries before locating the
* non-profiling versions
*
* This stanza is used for dynamically binding with libxlp.a. If
* the intention is to bind with libxlp.a statically, two
* additional options need to be specified:
*  -bnso,-bimport:/lib/syscalls.exp

DEFLT:  xlp        = /usr/lpp/xlp/lib/xlpentry
        as         = /bin/as
        crt        = /lib/crt0.o
        mcrt       = /lib/mcrt0.o
        gcrt       = /lib/gcrt0.o
        ld         = /bin/ld
        libraries  = -lxlp,-lm,-lc
        proflibs   = -L/lib/profiled,-L/usr/lib/profiled
        options    = -T512,-H512,-qprime=xlp_prime,
-I/usr/lpp/xlp/lib,-bh:4
```

Related Information

For more information about configuration files, see the *AIX Version 3.2 General Programming Concepts* manual. For more information about the **getopt** subroutine and its attributes, see the *AIX Version 3.2 Technical Reference: Base Operating System and Extensions* manual. The **as** and **ld** commands are described in the *AIX Version 3.2 Commands Reference*. The **–F** compiler option is described on page 29. The **–p** and **–pg** flags are described on page 30.

Input Files to the xlp Command

The kind of code to be processed determines the part of the system that is called by the **xlp** command. The input files to **xlp** are as follows.

Pascal source files Have either the default suffix **.pas**, or the suffix specified by the **psuffix** attribute in the configuration file.

The **xlp** command sends all Pascal source files with the suffix **.pas** to the compiler in the order in which they appear. If it cannot find the specified source files, XL Pascal produces an error message, and the **xlp** command proceeds to the next file if one exists.

Assembler source files Have either the default suffix **.s**, or the suffix specified by the **ssuffix** attribute in the configuration file.

The **xlp** command sends all assembler source files with the suffix **.s** to the assembler (**as**).

Object code files Have either the default suffix **.o**, or the suffix specified by the **osuffix** attribute in the configuration file.

After it compiles all the source files, the **xlp** command sends all object code files, including those produced by the compiler and assembler, to the linkage editor (**ld**). The input files are then link-edited to produce a single executable output file.

Library Files (Archive) Have the suffix **.a**.

The **xlp** command sends all the library files (**.a** files) to the linkage editor at link-edit time.

Note: Any files with unrecognized suffixes are sent to **ld**.

Output Files from the xlp Command

The output files produced by the **xlp** command include the following:

Executable files – a.out If you do not specify the **–c** compiler option, XL Pascal produces an executable file in the current directory. Its default name is **a.out**. To name the executable file explicitly, use the compiler option flag **–o***filename*, where *filename* must not have the XL Pascal language processor source file suffix (**pas**).

If you specify **–c**, XL Pascal does not produce an executable file.

Object files – *filename***.o** If you specify the **–c** compiler option, the compiler produces an object file for each of the **.pas** source files. It does not produce an executable file. The object files have the same prefix name as the source file, and appear in the current directory.

If you do not specify the **–c** compiler option, XL Pascal deletes the object files after calling the linkage editor (**ld**). If you use **–c**, the object files remain.

Object files can have either the **.o** default suffix, or a suffix specified by the **osuffix** attribute in the configuration file.

Listing files – *filename***.lst** By default, no listing appears unless you request 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**. Listing files are placed in the current directory.

Assembler files –*filename***.s** If you specify the –**S** compiler option, the compiler produces an object file for each of the **.s** source files. The object files have the same prefix name as the source file, and appear in the current directory.

Prime Files

An integral part of XL Pascal compiler operation, prime files contain precompiled declarations in the internal table format of the compiler. The compiler uses a prime file to initialize its internal tables before beginning compilation.

Using Prime Files

To use a prime file, specify its name in either a **–qprime** compiler option or a **%OPTION PRIME** directive. The XL Pascal compiler initializes its internal tables from the prime file before beginning compilation. The declarations from the prime file are placed in a scope outside your program or segment, so they are accessible throughout your source file. Your program can redefine any identifier supplied in a prime file.

XL Pascal uses a default prime file **xlp_prime**, which supplies the declarations for predefined identifiers, such as constants (for example, **MAXINT** and **MININT**), and data types (for example, **BOOLEAN**). This prime file also supplies declarations used by the compiler for the runtime environment routines that support I/O, string manipulation, and other functions. If you do not specify a **–qprime** or **–qnoprime** option, the default is **–qprime=xlp_prime**.

Using a prime file is similar to using an **%INCLUDE** compiler directive but provides faster compilation because the declarations in a prime file are precompiled. Prime files are useful in large programming projects where it is necessary to use common type declarations and external routine declarations in several program components.

You can use both a **–qprime** and a **–qprimeout** option in the same compilation. For example, the following command creates a prime file named **combo.prime** combining all of the declarations in the prime file **commrout.prime** with the declarations in file **commtypes.pas**:

```
xlp commtypes.pas –qprime=commrout.prime –qprimeout=combo.prime
```

Note: Declarations in a prime file being created are at the same scope as any declarations in a prime file it invokes. Therefore a prime file being created cannot redeclare any identifiers from an invoked prime file.

Creating Prime Files

To create new prime files, specify either a **–qprimeout** compiler option or a **%OPTION PRIMEOUT** directive. When you do this:

- Your source files can only contain declarations, with the exception of **VAR** declarations; prime files cannot contain any executable code. They can contain **%INCLUDE** directives, but not any **PROGRAM** or **SEGMENT** statements or executable routine bodies.
- The compiler generates a prime file instead of an object code file.
- The prime file you create contains all of the declarations introduced by the **–qprime** option or the **%OPTION PRIME** option, together with the declarations in your source files.
- You cannot specify **langlvl=standard** when creating prime files.

Example

The following source file **newc.pas** defines a new constant. This file does not contain any executable code:

```
(* File newc.pas : define new constant "newconst"  *)
CONST newconst = 7;      (* Define a new constant.  *)
```

Compile **newc.pas** to create a prime file named **newprime** with the following command:

```
xlp newc.pas –qprimeout=newprime
```

This command uses the default **–qprime=xlp_prime**. The prime file it creates contains everything in **xlp_prime** and your new declaration for `newconst`.

In the following example, file **primdemo.pas** contains a program that uses `newconst`:

```
(* File primdemo.pas to demonstrate "–qprime..." option *)
PROGRAM primdemo(output);
BEGIN
   WRITELN('newconst = ',newconst)
END.
```

The following command compiles **primdemo.pas** using the new prime file:

```
xlp primdemo.pas –qprime=newprime
```

Prime Files Supplied with XL Pascal

Two prime files are supplied when you install XL Pascal:

xlp_prime contains declarations for most of the predefined Pascal identifiers and for the routines in the XL Pascal library. This is the default prime file.

xlp_base_prime contains declarations for the minimum collection of predefined Pascal identifiers needed by the compiler.

The source code for **xlp_prime** is in file **xlp.pas** and the source code for **xlp_base_prime** is in file **xlp_base.pas**. To use **–qprime=xlp_base_prime**, you must also supply declarations for all library routines called by the object code generated from your source files.

Compiler Options

With XL Pascal compiler options, you can change any default settings of the compiler. You can specify compiler options in the source file or on the command line. Options specified on the command line remain in effect for all compilation units in the file, unless the compiler directive **%OPTION** overrides them. Command-line options can also appear in the configuration file and remain in effect for all compilations unless the command-line or compiler directive options override them.

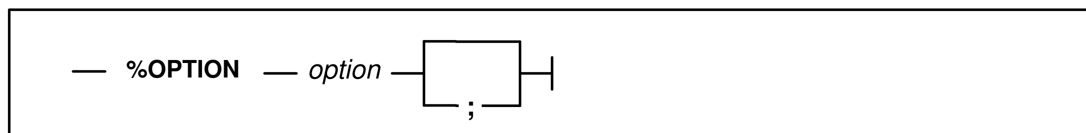
If you specify no options, the compiler:

- Folds all source code (except string literals) to lowercase
- Sets margins at 1 and 256
- Uses configuration file **/etc/xlp.cfg**
- Generates **DDNAME**s compatible with VS Pascal
- Uses prime file **xlp_prime**
- Sends object files directly to the linkage editor to produce the executable file **a.out**
- Does not produce source listing
- Writes all error messages to the standard error device
- Compiles in VS mode
- Produces warning messages

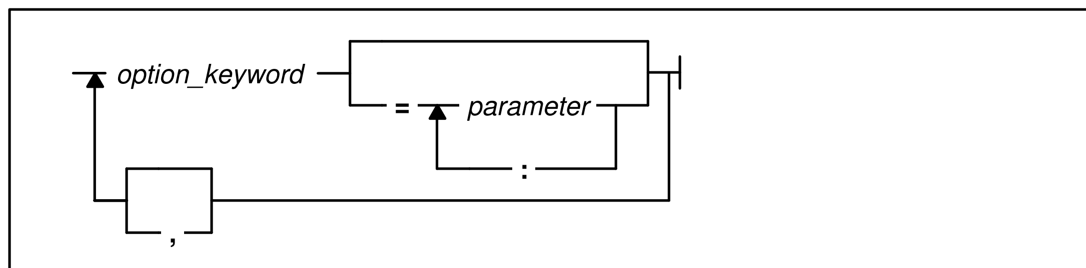
Specifying Options in the Source File

Use the **%OPTION** compiler directive in the source file to modify the options specified on the command line, or to change the default setting temporarily if no command line options are in effect.

Syntax



Option:



You can use either commas or blanks to separate options in the directive. Blanks are permitted between punctuation marks, option keywords, and parameters.

If the parameter you specify for an option contains special characters, you can specify the parameter as a string literal (delimited with single or double quotation marks).

Option settings designated with the **%OPTION** statement are effective only for the compilation unit in which the statement appears.

Specifying Options on the Command Line

Use either **–q***option* or single-letter option flags to specify compiler options on the command line. You can specify any command line option before or after file names.

For example:

```
xlp -qxref=full -c file.pas
```

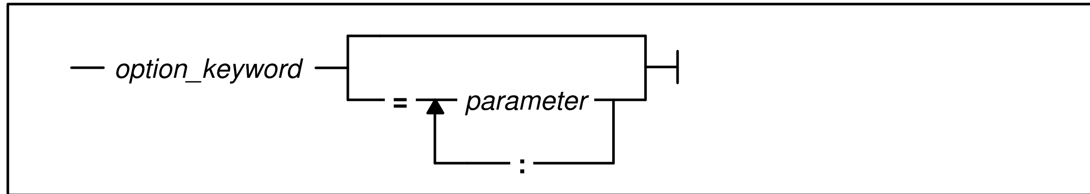
is the same as

```
xlp file.pas -qxref=full -c
```

Command Line Options with the –q Flag

Syntax

Option:



Option keywords can appear in either uppercase or lowercase, but the **–q** must be in lowercase.

For example:

```
xlp –qsource file.pas
```

is the same as

```
xlp –qSOURCE file.pas
```

More than one **–q***option* can appear in the same command line, but the options must be separated by blanks:

```
xlp file.pas –qstat –qattr=full –qalign
```

Some options take one or more parameters, which are indicated on the command line with an equal sign following the **–q***option*. Multiple parameters are separated with colons (:). No blanks can appear between punctuation marks and parameters, for example:

```
xlp –qhalt=W –qmargins=1:80 file.pas
```

Single or double quotation marks can enclose the options after the **–q**, but the opening and the closing quotation marks must be the same type. Option keywords can appear in either uppercase or lowercase, and either commas or blanks can separate options within the quoted string.

For example, you can write:

```
xlp file.pas –qlanglvl=standard –qsource
```

as either

```
xlp file.pas –q”langlvl=standard source”
```

or

```
xlp file.pas –q”langlvl=standard,source”
```

Single Letter Flags

XL Pascal supports many conventional single-letter flags in common with the other XL compilers, as well as flags directed to the AIX **ld** command. XL Pascal passes those flags to the linkage editor. All single-letter flags are case sensitive.

You can type multiple flags that do not take arguments in one string. For example, `xlp –cv file.pas` has the same effect as `xlp –c –v file.pas`.

Blanks must separate flags that take arguments. For example, the **–o** flag cannot run together with other flags:

```
xlp –otest –cv test.pas
```

Note: Though it consists of two characters, the flag **–pg** is considered a single option. It does not mean **–p –g**.

Summary of the Compiler Options

The following tables summarize the compiler options available in XL Pascal. You can enter the options in the source code using the option directive, or on the command line using the **–q** flags and single-letter flags.

The tables show:

- The syntax for the command line option in **–q***option* form, and the corresponding single-letter flag, if applicable.
- The option syntax for the **%OPTION** command. The uppercase letters in the option keyword represent its valid abbreviation. For example, **ATTR** is a valid abbreviation for **ATTRibutes**. You can spell out in full any options that allow abbreviation.
- The default value of the option if you do not specify it on the command line, in an **%OPTION** statement, or in the configuration file. This value is underlined.
- A brief description of the option's effect during compilation.
- A page reference to a more detailed description of the option.

Options Describing the Input to the Compiler

Command–Line Option	%OPTIONS Name	Description	Page
–B*prefix*		Constructs path names for substitute preprocessor, compiler, assembler, or linkage editor programs by adding *prefix* to the standard program names.	29
–F*config_file***:***stanza* **–F***config_file*		Names an alternative configuration file for **xlp**. **Default: –F/etc/xlp.cfg**	29
–I*directory*		Determines search path if file name in the **INCLUDE** directive does not start with an absolute path. **Default: –I/usr/lpp/xlp/lib**	30
–qmargins=*left:right*	**MARgins=***left:right*	Compiles only source code inside the left and right columns of the source line. **Default: –qmargins=1:256**	36
–qdbcs **–qnodbcs**	**DBCS** **NODBCS**	Allows the compiler to accept DBCS characters in literals and comments. This option is obsolete. The preferred option is **MBCS.** **Default: –qnodbcs**	32
–qmbcs **–qnombcs**	**MBCS** **NOMBCS**	Allows the compiler to accept MBCS characters in literals and comments. **Default: –qnombcs**	37
–qmixed **–qnomixed** **–U**	**MIXED** **NOMIXED**	Specifies case sensitivity. **Default: –qnomixed**	37
–t*programs*		Applies the **–B***prefix* to construct a file name for the designated *programs*.	30
–W*program***,***flags*		Passes the listed *flags* to the designated *program*.	31

Related Information

Specifying an alternative configuration file is described in "Customizing Configuration File Stanzas" on page 17.

Options Describing the Compiler Object Code to Be Produced

Command–Line Option	%OPTIONS Name	Description	Page
–g **–qnodbg**	**DBG** **NODBG**	Generates debug information for use by the symbolic debug program. **Default: –qnodbg**	32
–qarch=com **–qarch=pwr** **–qarch=pwr2** **–qarch=ppc** **–qarch=pwrx**	**ARCH=COM** **ARCH=PWR** **ARCH=PWR2** **ARCH=PPC** **ARCH=PWRX**	Specifies the architecture on which the executable program is run. **Default: –qarch=com**	31
–qcheck **–qnocheck**	**CHECK** **NOCHECK**	Specifies runtime error checking for the error conditions controlled by the **%CHECK** directive. **Default: –qcheck**	32
–qddname=unique **–qddname=compat**	**DDNAME**	Determines whether **DDNAME**s are unique names or compatible with VS Pascal. **Default: –qddname=compat**	32
–qextchk **–qnoextchk**	**EXTCHK** **NOEXTCHK**	Specifies whether to check for type conflicts in external name declarations. **Default: –qextchk**	32
–qfloat=*options*	**FLOAT=***options*	Specifies various floating-point options. Use this form in your new applications. **Default: –qfloat=nohsflt:nohssngl:norndsngl: maf:fold:norrm:nospnans:nonans:norsqrt**	33
–qflttrap=*options*	**FLTTRAP=***options*	Generates extra instructions to detect and trap floating–point exceptions.	34
–qfold **–qnofold**	**FOLD** **NOFOLD**	Specifies whether constant floating–point expressions are to be evaluated at compile time. This option is obsolete. **Default: –qfold**	35
–qfpret=std **–qfpret=return** **–qfpret=fast**	**FPRET=***options*	Sets levels of conformance with conventions for floating-point values returned from functions. **Default: –qfpret=fast**	35
–qlanglvl=vs **–qlanglvl=standard** **–qlanglvl=std**	**LANGLVL**	Determines which XL Pascal language mode is used. **Default: –qlanglvl=vs**	36
–qlog **–qnolog**	**LOG** **NOLOG**	Requests logging and reporting of floating-point exceptions. **Default: –qnolog**	36
–qmaf **–qnomaf**	**MAF** **NOMAF**	Specifies whether the compiler is to generate floating–point multiply–add instructions. **Default: –qmaf**	36
–qmaxmem=*num* **–qnomaxmem**	**MAXMEM=***num* **NOMAXMEM**	Limits the amount of memory used for local tables of memory-intensive optimizations to *num* kilobytes. **Default: –qmaxmem=2048**	36
–qopt=0 **–qopt=2** **–qopt=3** **–O** **–O2** **–O3**	**OPT=0** **OPT=2** **OPT=3**	Specifies object code optimization during compilation. **Default: –qopt=0**	37

Command–Line Option	%OPTIONS Name	Description	Page
–qprime=*file* **–qnoprime**	**PRIME=***file* **NOPRIME**	Requests that *file* be used as the prime file. **Default: –qprime=xlp_prime**	37
–qprimeout=*file* **–qnoprimeout**	**PRIMEOUT=***file* **NOPRIMEOUT**	Creates *file* as a prime file. **Default: –qnoprimeout**	38
–qptr4 **–qnoptr4**	**PTR4** **NOPTR4**	Specifies that **POINTER** data types occupy 4 bytes. **Default: –qnoptr4**	38
–qrrm **–qnorrm**	**RRM** **NORRM**	Prevents floating-point optimizations that are incompatible with runtime rounding modes Plus Infinity and Minus Infinity. **Default: –qnorrm**	38
–qspill=*n*	**SPILL=***n*	Sets the register allocation spill area as *n* bytes. **Default: –qspill=512**	38
–qtraceid **–qnotraceid**	**TRACEID** **NOTRACEID**	Requests that routine names be included in traceback tables so the **TRACE** procedure can display routine names. **Default: –qnotraceid**	39
–qxflag=dd24		Generates floating-point no-op instructions to cause detection of overflow in rounding floating-point intermediate results to single precision.	38
–yn **–ym** **–yp** **–yz**	**IEEE=***mode* **IEEE=***mode:comply*	Controls the handling of floating-point arithmetic during compilation. **Default: –yn**	36

Related Information

The **%CHECK** directive is described in the *Language Reference for IBM AIX XL Pascal Compiler/6000*. See “Enforcement of Type Matching” on page 104 for information about using **NOEXTCHK** in interlanguage calls. Some of the optimizations performed by XL Pascal are described in Chapter 5, “Performance Considerations”. Using prime files is described in “Prime Files” on page 21.

Options Describing the Compiler Output

Command–Line Option	%OPTIONS Name	Description	Page
–c		Prevents the completed object file from being sent to the AIX **ld** command for link-editing.	29
–qattr **–qattr=full** **–qnoattr**	**ATTR** **ATTR=FULL** **NOATTR**	Requests an attribute listing. **Default: –qnoattr**	31
–qflag=*sev_1***:***sev_2*	**FLAG=***sev_1*:*sev_2*	Sets minimum severity level at which diagnostic messages are reported. **Default: –qflag=w:w**	33
–qhalt=*sev*	**HALT=***sev*	Stops compilation at any compilation phase that encounters an error of specified severity or greater. **Default: –qhalt=s**	35
–qlist **–qnolist**	**LIST** **NOLIST**	Requests an object listing. **Default: –qnolist**	36

Command–Line Option	%OPTIONS Name	Description	Page
–qoption **–qnooption**	**OPTIONs** **NOOPTIONs**	Specifies that the settings of all options in effect during compilation are to be displayed in the listing. **Default: –qnooption**	37
–qquiet **–qnoquiet**	**QUIET** **NOQUIET**	Specifies that no compilation phase timings be shown at the terminal. **Default: –qquiet**	38
–qsource **–qnosource**	**SOURCE** **NOSOURCE**	Requests a source listing. **Default: –qnosource**	38
–qtune=PPC601 **–qtune=PPC** **–qtune=PWR** **–qtune=PWR2** **–qtune=PWRX**	**TUNE=***option*	Specifies the architecture system for which the executable program is optimized. The default depends on the architecture.	39
–qxref **–qxref=full** **–qnoxref**	**XREF** **XREF=FULL** **NOXREF**	Requests a cross-reference listing. **Default: –qnoxref**	40
–qwrite **–qnowrite**	**WRITE** **NOWRITE**	Enables the **%WRITE** compiler directive to produce output to the terminal. **Default: –qnowrite**	40
–S		Generates an assembler file (**.s**) for each source file.	30

Options Used for Debugging Programs

Command–Line Option	%OPTIONS Name	Description	Page
–#		Displays information about the progress of the compilation, without actually invoking the individual components.	29
–p **–pg**		Sets up the object file for profiling.	30
–v		Instructs the compiler to generate information on the progress of the compilation.	30

Related Information

For more information on options used for debugging, refer to the **DBG** option on page 32, the **FLAG** option on page 33, and the **HALT** option on page 35.

Using the **ld**, **prof**, and **gprof** commands for profiling is described in the *AIX Version 3.2 Commands Reference*.

Options Used for the Linkage Editor

Because the following are **ld** options, any conflicts are handled by the **ld** command.

Command–Line Option	%OPTIONS Name	Description	Page
–l*key*		Searches the specified library file specified by **lib***key***.a**.	30
–L*dir*		Searches *dir* for library files specified by the **–l***key* option.	30
–o*name*		Specifies a name for the object module. **Default: a.out**	30

Options Used for Performance Optimization

Command–Line Option	%OPTIONS Name	Description	Page
–qcompact **–qnocompact**	**COMPACT** **NOCOMPACT**	Reduces optimizations that increase code size. **Default: –qnocompact**	32
–qstrict **–qnostrict**	**STRICT** **NOSTRICT**	Ensures that optimizations performed by the **OPT=3** option do not alter the semantics of the program. **Default: –qnostrict**	38

Miscellaneous Options

Command–Line Option	%OPTIONS Name	Description	Page
–qwait=*num* **–qnowait**		Specifies the maximum wait time in *num* seconds for a network license token. **Default: –qnowait**	39

Detailed Descriptions of the Compiler Options

You can type the compiler options in the source code using an option directive, or on the command line using the –q flags and the single-letter flags.

Except for the **FLAG** option, if you set an option more than once in the configuration file, on the command line, or in your source, the option that you specify *last* controls the compiler. If **FLAG** options conflict, the option you set *first* takes precedence. The –**I** option takes exception to both of these rules; it has a cumulative effect, building a list of directory paths that are searched in the order in which they are specified.

Unless otherwise indicated, you can set or reset an option no later than the program prefix, that is, before the first real source language statement. You can set some options only in the command line; you can specify others anywhere in the command line or the source program. These are indicated in the descriptions that follow.

Compiler options that have no parameters represent on/off functions. For example, to turn on option **–qLIST**, you specify `–qLIST`. To turn it off, you specify `–qNOLIST`.

Note: Before using the XL Pascal Compiler/6000, read the descriptions of the compiler options to understand its correct operation and limitations.

–#
Displays information about the progress of the compilation, without actually invoking the individual components.

–B*prefix*
Constructs path names for substitute compiler, assembler, or linkage editor programs. The *prefix* defines part of a path name to the new programs. To form the complete path name for each program, **xlp** adds *prefix* to the standard program names. The standard program names for the compiler, assembler, and linkage editor are **xlpentry**, **as**, and **ld**, respectively.

–c
Prevents the completed object file from being sent to the AIX **ld** command for link-editing. With this flag, the output is an object code file (**.o**) for each source file.

–F*config_file* **| –F***config_file:stanza* **| <u>–F/etc/xlp.cfg</u>**
Specifies an alternative configuration file. The configuration file specifies the location of various files required by the compiler. Default configuration file /**etc**/**xlp.cfg** is supplied at installation time.

–I*dir* | **–I/usr/lpp/xlp/lib**

Specifies the search path if the file name in the **INCLUDE** statement does not start with an absolute path (/). The argument *dir* must be a valid path name (for example, **/home/dir** or **/tmp** or **./subdir**). The compiler appends a right slash (/) to the argument *dir*, and then concatenates it to the file name before making a search. If more than one **–I** option is specified in the command line, directories are searched in the order that the **–I** options occur.

–L*dir*
Looks in *dir* for files specified by the **–l** keys. Because this is an **ld** option, any conflicts are handled by the **ld** command.

–l*key*
Searches the specified library file, where *key* selects the file **lib***key***.a**. Because this is an **ld** option, any conflicts are handled by the **ld** command.

–o*name* | **a.out**

Specifies a name for the object module. A *name* with the XL Pascal source file suffix (**pas**), such as **file.pas**, causes an error. Because this is an **ld** option, any conflicts are handled by the **ld** command.

–p | **–pg**
Sets up the object file for profiling, where:

–p
Prepares the program so that the AIX **prof** command can generate a runtime 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.

–pg
Like **–p**, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a **gmon.out** file when the program ends normally. Use the **gprof** command to generate a runtime profile.

–S
Generates an assembler file (**.s**) for each source file, as opposed to the executable file that is generated when this option is not specified. You can assemble the resulting **.s** files to produce object files (**.o**) or an executable file (**a.out**).

You can use the **–o** option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is not valid:

```
xlp myprogram1.s myprogram2.s –oyourname.s –S
```

For more information about **.s** files, refer to the *AIX Version 3.2 Assembler Language Reference*.

–t*program*$_1$ *program*$_2$ . . . . *program*$_n$

Applies the **–B** flag instructions for constructing a file name for the designated path names, where *program* is:

c   compiler

a   assembler

l   linkage editor

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

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

Assigns the listed flags to the compiler program, where *program* is:

c compiler

a assembler

l linkage editor

ARCH | –qarch=*option*

Specifies the architecture on which the executable program is run. You can specify the architecture using the following values of *option*:

COM Produces an object that contains instructions that run on all the POWER, PowerPC*, and POWER2* hardware platforms.

PPC Produces an object that contains instructions that run on any of the 32-bit PowerPC hardware platforms.

PWR Produces an object that contains instructions that run on any of the POWER hardware platforms.

PWR2/PWRX Produces an object that contains instructions that run on the POWER2 hardware platforms.

The default is **–qarch=com**.

You can use the **–qarch=***option* option with the **–qtune**=*option.* The **–qarch=***option* option specifies the architecture for which the instructions are to be generated, and the **–qtune**=*option* option specifies the target platform for which the code is optimized. The following table summarizes the valid combinations of **–qarch**=*option* and **–qtune**=*option*.

–qarch option	Valid –qtune options	Default
COM	PWR, PWR2/PWRX, PPC601	PWR
PWR	PWR, PWR2/PWRX, PPC601	PWR
PWR2/PWRX	PWR2/PWRX	PWR2/PWRX
PPC	PPC601	PPC601

Note: PWRX is interchangeable with PWR2. However, PWR2 is preferred.

Use **–qarch=com** (the default) if you want your program to be widely distributable.

If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option.

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

Requests an attribute list consisting of all referenced objects and their attributes, giving the type and size of identifiers. If you specify **ATTR** without a parameter, the compiler reports only identifiers that are actually referenced. If you specify **ATTR=FULL**, the compiler reports all identifiers, even if they are not referenced.

<u>CHECK</u> | NOCHECK | <u>–qcheck</u> | –qnocheck

Specifies runtime error checking for the error conditions controlled by the **%CHECK** directive. The **CHECK** option lets you use **%CHECK** directives in your source code to turn checking for different conditions on and off.

NOCHECK specifies that no error conditions are checked, and tells the compiler to ignore any **%CHECK** directives in your source code.

You can use the **CHECK** and **NOCHECK** options together with **%CHECK** directives in your source code to control runtime checking selectively:

- **%OPTION CHECK** turns on all checking.
- From one **%OPTION CHECK** to the next **%OPTION NOCHECK**, you can use **%CHECK** directives to turn specific checking on or off.
- From one **%OPTION NOCHECK** to the next **%OPTION CHECK**, no checking is done, and the compiler ignores all **%CHECK** directives.

COMPACT | <u>NOCOMPACT</u> | –qcompact | <u>–qnocompact</u>

Reduces optimizations that increase code size. Some performance optimization techniques also make the program larger. Use the **–qcompact** option to reduce the expansion if your system has limited storage. Optimization options will still work with **–qcompact** in effect.

DBCS | <u>NODBCS</u> | –qdbcs | <u>–qnodbcs</u>

Allows the compiler to accept DBCS characters in literals and comments. If double-byte characters appear when the option is off, syntax errors are likely. This option is obsolete. Please use the **MBCS** option in your new applications.

DBG | <u>NODBG</u> | –g | <u>–qnodbg</u>

Generates information required by the symbolic debugger to access variables in the object program by name. **NODBG** allows only limited debugger functions.

DDNAME=*options* **| –qddname=unique | <u>–qddname=compat</u>**

Determines whether **DDNAME**s are unique names or compatible with VS Pascal, where:

UNIQUE	Generates a unique **DDNAME**. This is the default for Standard mode.
COMPAT	Generates a **DDNAME** compatible with VS Pascal. This is the default for VS mode.

For more information about **DDNAME**, refer to “File Name Association” on page 66.

<u>EXTCHK</u> | NOEXTCHK | <u>–qextchk</u> | –qnoextchk

Specifies whether to check for type conflicts in external name declarations. If you specify **EXTCHK**, the linkage editor will check external name declarations for type conflicts with the declarations of those names in other program or segment units. If you specify **NOEXTCHK**, the linkage editor will not check the declarations of external names. You can specify this option anywhere in the command line or the source file.

FLAG=*sev1:sev2* **|** **<u>FLAG=W:W</u>** **| –qflag=***sev1:sev2*

Specifies the minimum severity level of diagnostic messages to be reported, where:

sev1 No messages less than this level reported in listing

sev2 No messages less than this level displayed on terminal.

The message severities *sev1* and *sev2* can have the following values:

W Warning messages

E Error messages

S Severe error messages

Q Do not report any messages

The compiler reports messages of the specified severity level or higher. In case of conflicting options, the last option specified takes effect.

FLOAT=*options* **| –qfloat=***opt1:opt2:...:optn*

Specifies floating-point options that are to be in effect. Use this format in your new applications. The default setting is:

FLOAT(FOLD, NOHSFLT, NOHSSNGL, MAF, NONANS, NORNDSNGL, NORRM, NORSQRT, NOSPNANS).

The available suboptions (opt1, opt2,...,optn) are:

FOLD Specifies that constant floating-point expressions are to be evaluated at compile time. This is the default setting for the **FOLD**/**NOFOLD** suboption pair.

HSFLT Specifies that single-precision expressions are not to be rounded and that floating-point-to-integer conversions are performed without range checking.

Note: The **HSFLT** option is for specific applications in which floating-point computations have known characteristics. If you use this option when compiling other application programs, it may produce incorrect results without warning.

HSSNGL Specifies that single-precision expressions be rounded only when the results are stored in **SHORTREAL** memory locations.

MAF Specifies whether or not the compiler produces multiply-add instructions. Multiply-add instructions may affect the precision of floating-point intermediate results, giving better performance and better accuracy. This is the default setting for the **MAF**/**NOMAF** suboption pair.

NANS Detects runtime operations involving signaling NaNs so that you can use **–qflttrap=inv:en** to deal with exception conditions involving signaling NaNs. Use this suboption only if your program explicitly creates NaNs values.

RNDSNGL Specifies that the result of each single-precision operation be rounded to single-precision.

RRM Specifies the runtime rounding mode. This option is used if the runtime rounding mode is rounded to +infinity, –infinity, or is not known.

RSQRT Specifies whether a sequence of code that contains division by the result of a square root can be replaced by calculating the reciprocal of the square root and multiplying. This produces code that executes faster.

SPNANS Specifies that conversion of single-precision NaNs to double-precision is to be detected.

NOFOLD Specifies that the preceding **FOLD** suboption not be done.

NOHSFLT Specifies that single-precision expressions be rounded after expression evaluation, and that floating-point-to-integer conversions are checked for out-of-range values. This suboption is the default setting for the **HSFLT**/**NOHSFLT** suboption pair.

NOHSSNGL Specifies that single-precision expressions be rounded after expression evaluation. This suboption is the default setting for the **HSSNGL**/**NOHSSNGL** suboption pair.

NOMAF Specifies that the preceding **MAF** suboption not be done.

NONANS Specifies not to do the **NANS** suboption above.

NORNDSNGL Specifies that single-precision expressions be rounded only after full expressions have been evaluated. This suboption is the default setting for the **RNDSNGL**/**NORNDSNGL** suboption pair.

NORRM Specifies not to do runtime rounding as described in the **RRM** suboption above. This suboption is the default setting for the **RRM**/**NORRM** suboption pair.

NORSQRT Specifies that a sequence of code that involves division by the result of a square root is *not* to be replaced by calculating the reciprocal of the square root and multiplying. This suboption is the default setting for the **RSQRT**/**NORSQRT** suboption pair.

NOSPNANS Specifies that **SPNANS** conversion need not be detected. This suboption is the default setting for the **SPNANS**/**NOSPNANS** suboption pair.

FLTTRAP=*options* | **–qflttrap=***opt1:opt2:...:optn*
Generates extra instructions to detect and trap floating-point exceptions.

The **flttrap** option has the following suboptions:

OVerflow Generates code to detect and trap floating-point overflow when enabled.

UNDerflow Generates code to detect and trap floating-point underflow when enabled.

ZEROdivide Generates code to detect and trap floating-point division by zero when enabled.

INValid Generates code to detect and trap floating-point invalid operation exceptions when enabled.

INEXact
: Generates code to detect and trap floating-point inexact exceptions when enabled.

ENable
: Enables the specified exceptions in the prologue of the main program.

IMPrecise
: Generates code for imprecise detection of the specified exceptions. If an exception occurs, it is detected, but the exact location of the exception is not determined.

Specifying the **flttrap** option only once with no suboptions is equivalent to setting:

–qflttrap=ov:und:zero:inv:inex.

The exceptions are not automatically enabled and all floating-point operations are checked to provide precise exception-location information.

FOLD | NOFOLD | –qfold | –qnofold
Specifies whether constant floating-point expressions are to be evaluated at compile time. This option is obsolete. Use **–qfloat=fold** in your new applications.

FPRET=*options* | –qfpret=std | –qfpret=return | –qfpret=fast
Sets levels of conformance with conventions for floating-point values returned from functions, where:

STD
: Full conformance. Functions return floating-point results both in the general-purpose registers and the floating-point registers. The compiler always expects floating-point results in both these registers.

RETURN
: Partial conformance. Functions return floating-point results both in the general-purpose registers and the floating-point registers, but the compiler uses them only in floating-point registers.

FAST
: No conformance. Functions return floating-point results only in floating-point registers, and the compiler expects them in floating-point registers. This parameter improves performance.

HALT=*sev* | HALT=S | –qhalt=*sev*
Stops the compiler after any compilation phase in which the maximum severity of messages encountered equals or exceeds the specified severity. In case of conflicting options, the smallest value takes effect.

The message severity is specified as one of:

W
: Warning messages

E
: Error messages

S
: Severe error messages

U
: Unrecoverable error messages.

To get a syntax check, use **HALT=W**.

IEEE=*mode* | IEEE=*mode:comply* | <u>IEEE=NEAR:Nocomply</u> | –yn | –ym | –yp | –yz

Controls the handling of floating-point arithmetic during compilation. The first parameter specifies the rounding mode, and the second specifies whether to round intermediate results to single precision. Compliance causes all intermediate results to be rounded to single precision, which is less efficient.

Rounding mode can have the following values:

<u>NEAR</u> | –yn	Round to nearest whole number.
MINUS | –ym	Round toward minus infinity.
PLUS | –yp	Round toward plus infinity.
ZERO | –yz	Round toward zero.

The intermediate rounding option has two values:

Comply	Round intermediate results to single precision
<u>Nocomply</u>	Allow intermediate results to be double precision.

The **–y** flags set only rounding mode, not intermediate rounding.

LANGLVL=*options* | <u>–qlanglvl=vs</u> | –qlanglvl=standard | –qlanglvl=std

Determines which XL Pascal language mode is used. **DIALECT** and **IBMSET** are synonyms for **LANGLVL**. Language mode has two values:

<u>VS</u>	Accepts the IBM VS Pascal language dialect. The parameter IBM is a synonym for the VS parameter.
STANDard	Accepts the ANSI-83 Standard Pascal language dialect. The parameter **STD** is a synonym for the parameter **STANDard**.

LIST | <u>NOLIST</u> | –qlist | <u>–qnolist</u>

Requests an object listing. You can specify this option anywhere in the command line or the source file.

LOG | <u>NOLOG</u> | –qlog | <u>–qnolog</u>

Requests logging and reporting of floating-point exception and the routine in which it occurred.

<u>MAF</u> | NOMAF | <u>–qmaf</u> | –qnomaf

Specifies whether the compiler is to generate multiply-add instructions, which may affect the precision of floating-point intermediate results. This option is obsolete. Use **–qfloat=maf** in your new applications.

MARgins=*left:right* | <u>MARgins = 1:256</u> | –qmargins=*left:right*

Compiles only source code inside the left and right columns of the source line. The default is 1:256. This option can be specified anywhere in the command line or the source file.

MAXMEM=*num* | <u>MAXMEM=2048</u> | NOMAXMEM | –qmaxmem=*num* | –qnomaxmem

Limits the amount of memory used for local tables of specific, memory-intensive optimizations to *num* kilobytes. If that memory is insufficient for a particular optimization, the quality of the optimization is reduced. The **–qnomaxmen** option permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this might exceed available paging space. The default is **maxmem=2048**.

Notes:

1. The limit set by **maxmem** is the amount of memory for specific optimization phases, and not for the compiler as a whole. Tables required during entire compilation process are not affected.
2. Setting a large limit has no negative effect on the compilation of source files where the compiler needs less memory.
3. The *num* value must be non-negative.

MBCS | NOMBCS | –qmbcs | –qnombcs
Allows the compiler to accept MBCS characters in literals and comments. If multibyte characters appear when the option is off, syntax errors are likely.

MIXED | NOMIXED | –U | –qmixed | –qnomixed
Specifies the case sensitivity. If **MIXED** is specified, the source is not folded to lowercase. **NOMIXED** does fold it to lowercase. Reserved words can be in any case.

OPT=0 | OPT=2 | OPT=3 | –O | –O2 | –O3 | –qopt=0
Specifies object code optimization during compilation, where:

OPT=0 is default object code generation

OPT=2 performs extensive object code optimization.

OPT=3 performs additional optimizations that are memory intensive, compile-time intensive, and may change the semantics of the program. These optimizations are recommended when the desire for runtime speed improvements outweighs the concern for limiting compile-time resources.

This level of optimization also affects the setting the **–qfloat** option, turning on the **RSQRT** suboption by default.

The **STRICT** option on page 38 shows how to turn off the effects of **OPT=3** that may change the semantics of a program.

Notes:

1. Increasing the optimization level may or may not result in additional performance improvements, depending on whether the additional analysis detects any further optimization opportunities.
2. Compilations with optimizations may require more time and machine resources than other compilations.
3. The more the compiler optimizes a program, the more difficult it is to debug the program with a symbolic debugger.

OPTIONs | NOOPTIONs | –qoption| –qnooption
Specifies that the settings of all options in effect during compilation are to be displayed in the listing. An abbreviated list of options is always shown.

PRIME=*file* | PRIME=xlp_prime |NOPRIME| –qprime=*file* | –qnoprime
Requests that *file* be used as the prime file. The default prime file is **xlp_prime**. This option can be specified anywhere in the command line. You can also specify **PRIME** in the source file, but only in a **%OPTION** directive before all other statements in the compilation unit.

PRIMEOUT=*file* | NOPRIMEOUT | –qprimeout=*file* | –qnoprimeout

Creates *file* as a prime file. This prime file includes the declarations provided in the current source and can be used for later compilations. This option must be specified in the program prefix of the source file. Please note that the linkage editor is invoked if the **PRIMEOUT** option is specified in the source file, unless the **–c** option was specified on the command line; the linkage editor is not invoked when **–qprimeout** is specified on the command line.

PTR4 | NOPTR4 | –qptr4 | –qnoptr4

Specifies that **POINTER** data types occupy 4 bytes, containing address information only. Use this option for interlanguage calls where you require 4-byte pointers.

QUIET | NOQUIET | –qquiet | –qnoquiet

Specifies that no compilation phase timings be shown at the terminal. **NOQUIET** displays the timing of each compilation phase. The timing of the following compilation phases is displayed:

- Syntax checking
- Code generation
- Optimization
- Register allocation
- Object generation

–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.

RRM | NORRM | –qrrm | –qnorrm

Prevents floating-point optimizations that are incompatible with runtime rounding modes Plus Infinity and Minus Infinity. The default is **NORRM**, which generates code that is compatible with runtime rounding modes Nearest and Zero. This option is obsolete. Use **–qfloat=rrm** in your new applications.

SOURCE | NOSOURCE | –qsource |–qnosource

Requests a source listing. You can specify this option anywhere in the command line or the source file.

SPILLsize=*n* | SPILL=512 | –qspill=*n*

Sets the spill area size equal to *n*. The *spill area* is a storage area used to save the contents of registers. The default is 512.

STRICT | NOSTRICT | –qstrict | –qnostrict

Ensures that optimizations performed by the **OPT=3** option do not alter the semantics of a program. By default, **OPT=3** optimizations may rearrange calculations and exception-producing code so that results or exceptions are different than in unoptimized programs.

This option is intended for rare situations where the changes in program execution produce different results from unoptimized programs.

The **–qnostrict** option is only valid with the **OPT=3** option.

With **OPT=3** in effect, the following optimizations are turned on unless **STRICT** is also specified:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution, or might not occur at all. The compiler minimizes such situations.
- Floating-point operations may not preserve the sign of a zero value. In order to make certain that this sign is preserved, you also need to specify either **–qfloat=rrm** or **–qfloat=nomaf**.
- Floating-point expressions may be reassociated. For example, (2.0*3.1)*4.2 might become 2.0*(3.1*4.2) if the latter were faster, even though the result is not identical.
- The **RSQRT** suboption of the **FLOAT** option is turned on.

TRACEID | <u>NOTRACEID</u> | –qtraceid | <u>–qnotraceid</u>
TRACEID includes routine names in traceback tables so the **TRACE** procedure can display routine names. **NOTRACEID** removes routine names from traceback tables. With **NOTRACEID** the display produced by the **TRACE** procedure does not contain routine names.

TUNE | –qtune=*option*
Specifies the architecture system for which the executable program is optimized. You can specify the architecture using the following values of *option*:

PPC601 — Produces an object optimized for all the PowerPC601 processors

PPC — Produces an object optimized for the 32-bit PowerPC hardware platforms

PWR — Produces an object optimized for the POWER hardware platforms

PWR2/PWRX — Produces an object optimized for the POWER2 Power hardware platforms

You can use **–qtune=***option* with **–qarch=***option*. **–qarch=***option* specifies the architecture for which the instructions are to be generated, and **–qtune=***option* specifies the target platform for which the code is optimized. The following table summarizes the valid combinations of **–qarch=***option* and **–qtune=***option*:

–qarch option	Valid –qtune options	Default
COM	PWR, PWR2/PWRX, PPC601	PWR
PWR	PWR, PWR2/PWRX, PPC601	PWR
PWR2/PWRX	PWR2/PWRX	PWR2/PWRX
PPC	PPC601	PPC601

WAIT=*num* **| <u>NOWAIT</u> | –qwait=***num* **| <u>–qnowait</u>**
WAIT specifies the maximum wait time in *num* seconds for a network license token. **NOWAIT** specifies indefinite wait time.

WRITE | NOWRITE | –qwrite | –qnowrite
Enables the **%WRITE** compiler directive to produce output to the terminal. You can specify this option anywhere on the command line or in the source file.

XREF | NOXREF | XREF=FULL| –qxref | –qnoxref | –qxref=full
Requests a cross-reference listing. If you specify **XREF** with no parameter, the compiler reports only the identifiers actually used and their line numbers. If you specify **XREF=FULL**, the compiler reports all identifiers that appear in the program, whether they are used or not. You can specify this option anywhere in the command line or the source file.

Invoking the Linkage Editor

You can start the compiler without the **–c** option. The XL Pascal compiles the source program and calls the linkage editor program. The names of all object files created by the compiler are passed to the linkage editor along with any object files specified with the **xlp** command. The compiler also passes to the linkage editor the names of the libraries needed for the programs.

If you specify **–c** as a compiler option, XL Pascal only compiles the source program and creates object files. To link edit object files as a separate step, invoke the linkage editor explicitly with the AIX **ld** command, or issue the **xlp** command a second time without the **–c** or **–S** options, specifying the desired object file (**.o**) names.

Example

The following command invokes XL Pascal to compile `p1.pas` into an object module `p1.o`:

```
xlp p1.pas p2.o p3.o
```

Because the **–c** option is not specified in this command, the compiler invokes the linkage editor with object files `p1.o`, `p2.o`, and `p3.o`, and the libraries of runtime routines specified in the **xlp.cfg** file.

If you specify no **.pas** files, XL Pascal performs no compilation, but passes libraries and any **.o** files specified to the linkage editor to create an object program.

The **.o** file from a compilation is normally removed from the system after it passes to the linkage editor. Any **.o** files named in the **xlp** command but not produced during the current invocation are retained. In the preceding example, files `p2.o` and `p3.o` are retained, but `p1.o` is removed. If you specify the **–c** option, newly created **.o** files are also retained because **–c** prevents the linkage editor from being invoked. For example:

```
xlp p2.pas –c
```

creates `p2.o`;

```
xlp p3.pas –c
```

creates `p3.o`;

```
xlp p1.pas –c
```

creates `p1.o`, and

```
xlp p1.o p2.o p3.o –oprog
```

creates the file `prog` using `p1.o`, `p2.o`, `p3.o`, and the libraries.

Static Linking

In statically linked programs, all code is in a single executable module. Library references are more efficient because the library procedures are statically linked into the program. Static linking increases the file size of your program, and it may increase the code size in memory if other applications, or other copies of your application, are being run on the system.

You can use the command line option `-bnso -bI:/lib/syscalls.exp` when you compile your programs to create statically linked object files. This option forces the linker to place the library procedures that your program references into the program's object file. The file /**lib**/**syscalls.exp** contains the names of system routines that must be imported to your program from the system. This file must be specified for static linking.

Related Information

The **ld** command and linkage editor flags are described in the *AIX Version 3.2 Commands Reference*.

Running a Program

To run a program, enter the path name and file name of the executable object file, and any runtime options on the command line. If the **–o***name* compiler option is specified, the file name is *name*. The default file name is **a.out**.

The XL Pascal Runtime Environment

Object code produced by the XL Pascal compiler may call several runtime routines. The XL Pascal runtime environment includes a library of runtime routines and also facilities for producing runtime diagnostic messages in the national language appropriate for your system. Normally, you cannot run object code produced by the XL Pascal compiler without the runtime environment.

External Names in the Runtime Environment

Runtime routines are collected into libraries. When you use the **xlp** command without the **–c** or **–S** options, the compiler invokes the linkage editor and gives it the names of the libraries that contain runtime routines called by Pascal object code.

The names of these runtime routines are external symbols. When object code produced by the XL Pascal compiler calls a runtime routine, the **.o** object code file contains an external symbol reference to the name of the routine. A library contains an external symbol definition for the routine. The linkage editor resolves the runtime routine call with the routine definition.

Avoiding the Use of Runtime Routine Names

You should avoid using external names in your XL Pascal program that conflict with names of runtime routines. Conflict arises under two conditions:

- The name of an **EXTERNAL** routine has the same name as a runtime routine.
- The Pascal program calls an **EXTERNAL** routine with the same name as a runtime routine but does not supply a definition for the routine.

If you define an **EXTERNAL** routine with the same name as a runtime routine, your definition of that name may be used in place of the runtime routine, or it may cause a binding error. To avoid conflicts with the names of the external symbols in the XL Pascal runtime environment, the identifiers you use should not begin with the dollar sign ($) or be the same as an XL Pascal predefined identifier. You should also avoid naming an **EXTERNAL** routine **main** because XL Pascal defines an entry point **main** to start your program.

Object code produced by the XL Pascal compiler can call some routines from libraries other than the XL Pascal runtime environment.

User References to Undefined Runtime Routine Names

Do not leave names of **EXTERNAL** routines or **REF** variables undefined. If your Pascal code refers to an external name without defining it, the linkage editor will attempt to resolve the reference in the XL Pascal runtime environment or in any of the other libraries from which XL Pascal uses runtime routines. Resolution by the linkage editor may appear to be successful but the program will probably run unpredictably.

Note: XL Pascal uses some runtime routines in the C and mathematics libraries. Each of these libraries contains several names with common spellings.

AIX Shared Libraries

The runtime library included in the XL Pascal runtime 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 conflicts between user-defined external names and the names of any routines that are called by runtime routines.

For example, when you invoke the Pascal **NEW** procedure, the XL Pascal compiler generates a call to the runtime routine **$pnew**. This runtime routine in turn calls AIX system routines to allocate blocks of storage. All calls within **$pnew** are resolved among the XL Pascal runtime environment library and other libraries. This allows a Pascal user program to define and call a routine with the same name as any of the system routines called by **$pnew**, and the name will not conflict with any calls in **$pnew**.

Related Information

Shared libraries are described in the *AIX Version 3.2 Technical Reference: Base Operating System and Extensions*. The description of the **ld** command in the *AIX Version 3.2 Commands Reference* contains other details of creating and using shared libraries.

Chapter 4. Input and Output Facilities

Your program retrieves information, processes it the way you specify, and then produces the results you want. An essential part of every program you develop is reading and writing data.

XL Pascal uses input and output statements to read and write data. Input and output statements operate on *files*. A file is a named set of records stored and processed as a unit. File records are the ordered information items that program statements manipulate. An input statement reads data from a file; an output statement writes data to one.

XL Pascal supports two types of files:

- **TEXT** files consisting of character data.
- **RECORD** files containing a sequence of data items of any type. All data in the same record file must be of the same type, and each item in the file of a fixed size.

This chapter shows how to use the XL Pascal input and output (I/O) facilities under the AIX Version 3 Operating System.

Environment-Determined File Names

The name of an input or output file can be determined at program run time by using environment-determined file names. Using environment variables permits access to a different file each time you run the program. You can set environment variables two ways to open environment-determined files: on the command line or in shell scripts.

Using Environment Variables on the Command Line

Use AIX environment variables to associate a file name with a program variable, such as `infile`. The following AIX commands demonstrate the use of environment variables in the Korn shell and in the C shell. Enter these commands on the command line before invoking the program.

Note: Unless otherwise specified, the examples in this section use the Korn shell.

Syntax

Korn or Bourne Shell:

environment–name=*external–filename*; **export** *environment–name*

environment–name=' (*option*,...)' ; **export** *environment–name*

C Shell:

setenv *environment–name* ' (*option*,...)'

Parameters

environment–name is the same as the **DDNAME** of the file. In standard mode, you must use the file variable name.

external–filename is the external file name used in the AIX file system.

option is the file opening option **NAME**=*external–filename* alone, or both **NAME**=*external–filename* and **DISP**=**MOD**. You cannot use the **DISP**=**MOD** option alone.

© Copyright IBM Corp. 1990, 1993

Environment Variables in the Runtime Environment

In the Korn shell, you must use the AIX **export** command so that the system can use the environment variables in your runtime environment.

In the following example, the environment variable `infile`, which is used as a program variable, is associated with the file `file1.text`. Then the program **myprog** is run.

```
infile=file1.text; export infile
myprog
```

After the program is run, the exported file name, `infile`, remains in the AIX environment for any subsequent runs that use the same variable and AIX file. To run the same program using a different file, you must associate the new file name with the `infile` environment variable and export it again.

It is possible to run the same program both in the background and in the foreground using different files as shown in the following example:

```
infile=file1.text; export infile; myprog&
infile=file2.text; export infile
myprog
```

By entering the program invocation on the same line as the environment statements, you associate the statements on each line with its own unique AIX process. Environment variables are only known in their current environment. Therefore, `file1` is local to the first invocation of **myprog**, and `file2` is local to its second invocation.

If you define the environment variable of a file with the **DISP**=**MOD** option and use **REWRITE** to open the file, output is appended to the end of the existing file.

Options are not case sensitive, so you can enter them in uppercase or lowercase. Some examples of environment variables with open options are:

```
infile='(name=file1.text)'; export infile
infile='(name=file1.text, disp=mod)'; export infile
```

Related Information

The **DDNAME** of a file in VS mode is described on page 66.

Using Environment Variables in Shell Scripts

The easiest and most efficient method to run a program with a variable name is to use a shell script with all the necessary commands. When you invoke the shell script, each command in the file is processed sequentially.

The shell script allows you to associate an environment variable either with the same file name each time you call the script, or with a different file name each time.

Shell Script Using the Same File Name

The following example illustrates a shell script called **run1**, which associates the environment variable named `infile` with the file named `file1.text`. It also contains the command to run the program **myprog**.

The shell script **run1** contains

```
infile=file1.text; export infile
myprog
```

After the shell script is created as a file, you can use the following AIX command:

```
chmod 755 run1
```

To run it, enter:

```
run1
```

While the script is running, it is considered an AIX process. Therefore, anything that runs within the shell script is local to that process, and the content of the `infile` environment variable is unknown to other AIX processes.

Shell Script Using Different Files

To avoid having to edit the shell script whenever you use a different AIX file, you can create the shell script with a variable in place of the file name. The following version of the shell script **run1** uses a variable `$1` in place of the physical file name:

```
infile=$1; export infile
myprog
```

To run the file script using **file1.text**, enter:

```
run1 file1.text
```

When you run the shell script, `file1.text` replaces the variable `$1`. You can use the script with any file name, so that **run1** can run in the background under one file name and in the foreground under a second, as illustrated in the following example:

```
run1 file1.text&
run1 file2.text
```

Related Information

See *AIX Version 3.2 System User's Guide: Operating System and Devices* for information on shells, writing shell scripts, and using environment variables. The *AIX Version 3.2 Commands Reference* describes the **csh** command.

Opening Files for Input and Output

Before your program can read data from or write it to a file, that file must be opened. Opening a file associates a file variable with a file or device in the system and establishes whether the file variable will be used for input or output.

Options for Opening a File

All XL Pascal file opening procedures are defined with a string parameter that contains the options pertaining to the file being opened. These options determine how the file is to be opened and what attributes it is to have.

The file opening options are valid for the following XL Pascal I/O procedures:

- **RESET**
- **REWRITE**
- **TERMIN**
- **TERMOUT**
- **UPDATE**

Not all of the options apply to all open procedures. If an incorrect option is specified for a procedure, XL Pascal ignores the option.

Syntax

```
      ┌──────────────────────────────┐
 ─ '─▼─ NAME=external–filename ──────┴─ '─┤
       LRECL=n
       RECFM=c
       DISP=MOD
       UCASE
       INTERACTIVE
       DDNAME=name
      └─────────── , ─────────────────┘
```

You can use either commas or blanks to separate options.

Parameters

NAME=*external–filename*

Specifies the external name of the physical file to be associated with the file variable being opened. This name remains associated with the file variable until the file is closed. You can specify the name as a full AIX path name or, if the file is in the directory in which the program is running, as a simple file name.

The **NAME** option applies to the following procedures:

- **RESET**
- **REWRITE**
- **UPDATE**
- **TERMIN**
- **TERMOUT**

If the name of the external file is specified in one of these procedures using the **NAME** option, then that declaration overrides any subsequent attempt to use a different file or file name with the file variable.

Using the **NAME** attribute is shown in the following example:

```
CONST
   open_opt = 'NAME=/u/fred/data/tempfile.info';

VAR
   f : TEXT;
   s : STRING;

BEGIN
   .
   .
   REWRITE( f, open_opt );
   WRITELN( f, 'this goes into the file' );
   .
   .
   RESET( f, open_opt );
   READLN( f, s );
END.
```

LRECL=*n* Specifies the logical record length to be associated with an output file. The value *n* must be an integer. **LRECL** applies to the **REWRITE** and **TERMOUT** procedures.

You are not required to specify a logical record length for your files. Let the compiler assign a default value for **LRECL**.

For **TEXT** files with a variable length record format (**RECFM**=**V**), the system uses the **LRECL** value you supply. The default logical record length for **TEXT** files with fixed length record format (**RECFM**=**F**) is 256.

The logical record length of a record file must be at least large enough to contain the base component of the file; otherwise, a runtime error message is issued when you open the file. For example, a file variable declared as `FILE OF INTEGER` requires the associated physical file to have a logical record length of at least 4 bytes.

You can use the **LRECL** attribute in the **TERMOUT** procedure to determine the maximum length of the line to be written to your terminal.

RECFM=*c* Specifies a record format to be associated with an input file. Fixed format (**F**) is the only value of *c* allowed for record files. **TEXT** files can have fixed or variable length (**V**) record format. **RECFM**=**V** is the default for **TEXT** files.

RECFM applies to **REWRITE**.

You are not required to specify a record format for your files. Let the compiler assign a default value for **RECFM**.

If a file has fixed-length records and the logical record length is larger than necessary to contain the component type of the file, the extra space in each logical record is wasted.

DISP=MOD Adds data to the file instead of rewriting it completely. If the file does not exist, it is created. Usually **REWRITE** erases an existing file; this option preserves existing data. The only value for **DISP** is **MOD**, and the option applies only to **REWRITE**.

UCASE Causes text being read from a file to be translated to uppercase. This option only applies to **TERMIN**.

INTERACTIVE Indicates that the file is to be opened for input as an interactive file. You can write a prompt asking the operator for data before your program reads data. That is, the program does not read ahead when it reaches the end of a line.

If the end-of-file condition on an interactive **TEXT** file tests **FALSE**, it is still possible for the file to be empty on the next **READ** operation.

The **INTERACTIVE** attribute applies to the **RESET** procedure and is implied for **TERMIN**.

Opening a file for interactive input is shown in the following example:

```
PROGRAM interact;

VAR
   sysin : TEXT;
   data : STRING( 80 );

BEGIN
   RESET( sysin, 'INTERACTIVE' );
      (* open sysin for interactive input *)
   WRITELN( ' ENTER DATA: ' );
      (* prompt for response *)
   READLN( sysin, data );
      (* read in response *)
END.
```

DDNAME=*name*

Causes the physical file associated with the file variable to have **DDNAME** indicated by *name*. This new **DDNAME** remains associated with the file variable even if the file is closed and then opened again. It can only be changed by another call to a file open routine with the **DDNAME** attribute specified.

The name you specify can be of any length, up to the maximum for valid XL Pascal identifiers (256 characters), and can include the AIX path name. If you do not specify the **NAME** attribute as well, the **DDNAME** is used for the external file name.

DDNAME applies to the following procedures:

- **RESET**
- **REWRITE**
- **UPDATE**
- **TERMIN**
- **TERMOUT**

The **DDNAME** depends on the compiler option in effect:

- The **DDNAME**=**COMPAT** compiler option causes XL Pascal to generate **DDNAME**s that match the identifiers used in the program. Note that the operating system does not observe XL Pascal scoping rules. For example, a file variable *file* in one scope would be different from one in another scope, but both would have the **DDNAME** *file*. When **DDNAME**=**COMPAT** is in effect, the first occurrence of a file variable can be overwritten by the second occurrence of one with the same name.
- The **DDNAME**=**UNIQUE** compiler option instructs XL Pascal to generate unique **DDNAME**s based on the names used in the program. This option ensures that file variables with the same name but different scopes do not get mapped to the same external file.

To ensure that a particular file variable gets mapped to the external file you want, you must ensure that the first 32 characters of the file variable names you use are unique. To generate external file names for file variables that are not unique in the first 32 characters, use the **DDNAME**=**UNIQUE** compiler option in VS mode.

Interactions between LRECL and RECFM Open Options

Record format and logical record length can interact to cause runtime errors under the following conditions:

Text Files with Variable Length Records:

For **TEXT** files with **RECFM**=**V**, a runtime warning is generated if you exceed the logical record length currently in effect. The line is broken up according to the **LRECL**, as shown in the following example:

```
.
.
REWRITE( infile, 'RECFM=V,LRECL=5' );
WRITELN( infile, 'thisismorethan5see?' )
.
.
```

The previous lines put the following into `infile`:

```
'thisi'
'smore'
'than5'
'see?'
```

Note: No padding occurs with a variable record format, regardless of the logical record length.

Text Files with Fixed Length Records:

With fixed-format files, no new lines are supplied. If you do not specify **LRECL** for **TEXT** files with **RECFM**=**F**, the **LRECL** is set to a default of 256. A **WRITELN** applied to a fixed-format file results in blank padding to fill out the line to the value of **LRECL**.

If you attempt to write a line of data longer than the **LRECL** permits, a warning message is issued and the program continues running. Writing to a **TEXT** file with **RECFM**=**F** and **LRECL**=5 is shown in the following example:

```
.
.
REWRITE( infile, 'RECFM=F,LRECL=5' );
WRITE( infile, 'ab' );
WRITE( infile, 'cde' );
WRITE( infile, 'c' );
.
.
```

The previous lines put the following into `infile:`

```
'ab cde c  '
```

Record Files:

XL Pascal does not allow variable length record format for record files. If you specify **RECFM**=**V** for a record file, you are notified at run time that the record format is being changed to **RECFM**=**F**, so the program can continue running.

Logical record length has the following effect on record files:

LRECL	Effect
Not specified	Default **LRECL** is set to hold exactly one element of the file. No extra padding is done

Smaller than file elements	XL Pascal issues a runtime error message and changes **LRECL** to let the program run
Larger than file elements	Each record in the file is padded with blanks to to specified **LRECL**

Using the INTERACTIVE Option with RESET

Because **RESET** performs an implicit read operation to fill a file buffer, it is not well suited for files you intend to associate with interactive input. For example, if the file you want to open is assigned to your terminal, you must enter a line of data when the file is opened. You may want to do this if your program displays prompt messages before it reads data.

To avoid this problem, you can open a file for interactive input by specifying **INTERACTIVE** in the options string of **RESET**. No initial read operation is performed on files opened this way. The file pointer has the value **NIL** until the first file operation (**GET** or **READ**). If the file is a **TEXT** file, the end-of-line condition is initially **TRUE**.

Related Information

The end-of-file condition for **TEXT** files is described on page 60.

The end-of-line condition for **TEXT** files is described on page 59.

The **DDNAME** compiler option is described on page 32. See "File-Name Association" on page 66 for more information on external file names. You can set **NAME** and **DISP** attributes with environment variables, as described on page 43.

File Opening Procedures

The following sections describe the XL Pascal procedures for opening files for input and output.

The options for opening a file are described in "Opening Files for Input and Output" on page 45.

Opening a File for Input (RESET)

To explicitly open a file for input, use the procedure **RESET**. A call to **RESET** has the form

```
RESET ( f, options )
```

Where	Represents
f	a file variable
options	an optional string containing the open options

The **RESET** procedure allocates a buffer, reads the first logical record, and positions the file pointer at the beginning of the buffer. For **TEXT** file *f*, the statement **RESET**(*f*) implies that *f@* points to the first character of the file.

If the **RESET** operation is performed on an open file, the file is closed and then reopened.

Example

```
PROGRAM doreset;

VAR
   sysin : TEXT;
   c : CHAR;

BEGIN
   (* open sysin for input *)
   RESET( sysin, 'NAME=sysin.text' );
   (* get first character of file *)
   c := sysin@;
END.
```

Opening a File for Output (REWRITE)

Use the **REWRITE** procedure to open a file for output. A call to the procedure has the form:

```
REWRITE( f, options )
```

Where	Represents
f	a file variable
options	an optional string containing the open options

REWRITE positions the file pointer at the beginning of an empty buffer. Files already open are closed before being reopened.

Examples

Opening a TEXT File with REWRITE

```
PROGRAM dorewrite;

VAR
   sysprint : TEXT;

BEGIN
   REWRITE( sysprint );
   WRITELN( sysprint, 'MESSAGE' );
END.
```

Opening a Record File with REWRITE

```
PROGRAM dorewrite;

VAR
   outfile : FILE OF INTEGER;
   i : INTEGER;

BEGIN
   REWRITE( outfile, 'NAME=/tmp/foo.file' );
   (* open the file    *)
   i := 3;
   outfile@ := i
   PUT(outfile);   (* write out an integer value *)
END.
```

Opening a File for Terminal Input (TERMIN)

Use the **TERMIN** procedure to open a **TEXT** file for interactive input directly from the keyboard. No initial input/output operation is performed on files opened interactively until a **READ**, **READLN**, or **GET** statement is encountered.

Unless you specify an input file, a **READ**, **READLN**, or **GET** operation gets its data from the predefined file **INPUT**, which comes from the AIX standard input. The default standard input is the terminal, but you can redirect the data from a file. If you apply **TERMIN**, the **READ**, **READLN**, or **GET** is sent directly to the terminal device (**TTY**) for input. You cannot redirect the input from a file.

A call to the procedure has the following form:

```
TERMIN( f, options )
```

Where	Represents
f	a **TEXT** file variable
options	an optional string containing the open options

Note: The **TERMIN** procedure opens the file with the **INTERACTIVE** attribute.

Opening a File for Terminal Output (TERMOUT)

Use the **TERMOUT** procedure to open a **TEXT** file for terminal output.

Usually, a **WRITE**, **WRITELN**, or **PUT** operation places its data into the predefined file **OUTPUT**, which goes to the AIX standard output. The default standard output is the terminal, but you can redirect the data to a file. If you apply **TERMOUT**, **WRITE**, **WRITELN**, or **PUT** output is displayed directly on the terminal device (**TTY**). You cannot redirect the output to a file.

A call to the procedure has the form

```
TERMOUT( f, options )
```

Where	Represents
f	a **TEXT** file variable
options	an optional string containing the open options

Example

```
PROGRAM dotermio;

VAR
   ttyin, ttyout : TEXT;
   i : INTEGER;

BEGIN
   (* open terminal files input and output *)
   TERMIN( ttyin );
   TERMOUT( ttyout );
   WRITELN( ttyout, 'ENTER DATA:' ); (* write a prompt message*)
   READLN( ttyin, i );               (* read in the response  *)
   .
   .
END.
```

Opening a File for Updating (UPDATE)

Use the **UPDATE** procedure to open a record file for reading and writing data. In this mode, you can read records, modify them, and then replace them. A call to the procedure has the form:

```
UPDATE( f, options )
```

Where	Represents
f	a record file variable
options	an optional string containing the open options

On a call to **UPDATE**, a file buffer is allocated and the first record of the file is read into it. A subsequent **GET, PUT, READ, WRITE**, or **SEEK** operation causes the contents of the buffer to be stored in the file at the location from which it was read.

Each **GET** operation reads successive records in the file. A **PUT** operation writes the record back to the location from which the last **GET** obtained it.

Example

```
PROGRAM doupdate;

VAR
   f : FILE OF RECORD
         name : STRING( 30 );
         age : 0..99;
       END;

BEGIN
   UPDATE( f );
   WHILE NOT EOF( f ) DO
   (* update each record by incrementing age *)
      BEGIN
         f@.age := f@.age + 1;
         PUT( f );
         GET( f );
      END;
END.
```

Related Information

Redirecting input is described in *AIX Version 3.2 System User's Guide: Operating System and Devices*. The **INTERACTIVE** attribute is described in "Options for Opening a File" on page 45.

Processing a TEXT File

This section describes how to read data from and write it to a **TEXT** file.

Reading Data from a TEXT File (GET)

Use the **GET** procedure to read data from a file. A call to the procedure has the form:

```
GET( f )
```

Where	Represents
f	a **TEXT** file variable

When applied to an input **TEXT** file, **GET** causes the file pointer to be incremented by one character position. If the file pointer is at the last position of a logical record, the **GET** operation causes the end-of-line condition to become **TRUE**, and the file pointer to be positioned to a blank.

If, before the call to **GET**, the end-of-line condition is **TRUE**, the file pointer is positioned at the beginning of the next logical record. Conversely, if the file pointer is positioned to the end of the last logical record of a **TEXT** file before the call to **GET**, the end-of-file condition becomes **TRUE**.

Attempting to use **GET** on a **TEXT** file that has not been opened for input causes the file to be opened implicitly, as if **RESET** had been called. When this happens, the file pointer skips the first character of the file and is positioned at the second character.

Example

```
PROGRAM doget;

VAR
   infile : TEXT;
   c1, c2 : CHAR;
   .
   .

BEGIN
   RESET( infile );  (* get first character of infile  *)
   c1 := infile@;
   GET( infile );    (* get second character of infile *)
   c2 := infile@;
   .
   .
END.
```

Related Information

The end-of-file condition for a **TEXT** file is described on page 60.

The end-of-line condition for a **TEXT** file is described on page 59.

Writing Data to a TEXT File (PUT)

Use the **PUT** procedure to write data to a file. A call to the procedure has the form:

```
PUT( f )
```

Where	Represents
f	a **TEXT** file variable

Before issuing a **PUT** operation, you must open the file for output or update it, and ensure that the associated output buffer contains the data to be written.

When applied to a **TEXT** file opened for output, the **PUT** procedure increases the file pointer by one character position. If, before the call, the number of characters in the current logical record is equal to the logical record length of the file (**LRECL**), the file pointer is positioned within the associated buffer to begin a new logical record.

When the file buffer is filled, the buffer is written to the associated physical file. The file pointer is then positioned to the beginning of the buffer so that it can be refilled on subsequent calls to **PUT**.

Example

```
PROGRAM doput;

VAR
   outfile : TEXT;
   c : CHAR;

BEGIN
   REWRITE( outfile );
   outfile@ := c;
   PUT( outfile );    (* write out value of c *)
END.
```

Reading Data from a TEXT File (READ)

The **READ** procedure reads data from a **TEXT** file. A call to the procedure has either of the following forms:

```
READ( f, v )
```

```
READ( f, v : n )
```

Where	Represents
f	an optional **TEXT** file variable. If you omit the file variable *f* and the comma following it, the file **INPUT** is assumed.
v	one or more variables of the following types: **CHAR** (or subrange) **GCHAR** **GSTRING** **INTEGER** (or subrange) **PACKED ARRAY**[1..*n*] **OF CHAR** **PACKED ARRAY**[1..*n*] **OF GCHAR** **REAL** **SHORTREAL** **STRING**
n	an optional field length that is an **INTEGER** expression.

If the file pointer is not set when **READ** is called, an initial **GET** operation is performed. This happens when a file is opened interactively.

If you call **READ** for a closed file, the file is opened for input by an implicit call to **RESET**.

Examples

```
PROGRAM doread;

VAR
   infile : TEXT;
   r : ARRAY[1..10] OF RECORD
          name : STRING( 25 );
          age : 0..99;
          weight : REAL;
       END;
   i : 1..10;

BEGIN
   RESET( infile );
   FOR i := 1 TO 10 DO
      WITH r[i] DO
         BEGIN
            READ( infile, name, age );
            READ( infile, weight );
            READLN( infile );
         END;
END.
```

The following example shows the **READ** procedure with length qualifiers. Given this input stream from file **INPUT**:

```
951239999991000.00JUNK
```

the program `readzip` produces the following output:

```
ZIP =       95123
MAN =      999999
BALANCE = 1000.00
```

Immediately after the **READ** statement in `readzip` is processed, the file **INPUT** is positioned to the character `'N'`.

```
PROGRAM readzip;

VAR
   zip : 0..99999;
   man : 0..999999;
   balance : REAL;

BEGIN
   READ( zip : 5, man : 6, balance : 9 );
   WRITELN( 'ZIP = ', zip );
   WRITELN( 'MAN = ', man );
   WRITELN( 'BALANCE = ', balance : 8 : 2 );
END.
```

Related Information

The **INTERACTIVE** file opening option is described on page 47. Using the **INTERACTIVE** option with the **RESET** procedure is described on page 50.

Reading Data from a TEXT File (READLN)

A call to **READLN** is identical to **READ** and performs the same function, except that after the data is read, all remaining characters in the logical record are skipped. The procedure **READLN** applies to **TEXT** files only. **READLN** implicitly opens a closed **TEXT** file.

Unless the end-of-file is reached, **READLN** normally causes the next logical record to be read. The file pointer is positioned at the beginning of the buffer containing the record. For **TEXT** files opened with the **INTERACTIVE** attribute (which occurs (1) when the predefined files **INPUT**, **OUTPUT**, and **STDERR** are implicitly opened with the **INTERACTIVE** attribute in VS Mode, (2) as the result of a call to **RESET** with the **INTERACTIVE** attribute specified, or (3) when the **TERMIN** procedure is used), the file pointer is positioned after the end of the logical record, and the end-of-line condition is set to **TRUE**.

Example

```
PROGRAM copy;

VAR
   infile,
   outfile : TEXT;
   buff : STRING( 100 );

BEGIN
    (* open infile for input and outfile for output *)
   RESET( infile );
   REWRITE( outfile );
   WHILE NOT EOF( infile ) DO
      BEGIN
         (* read each line from infile *)
         READ( infile, buff );
         (* write out the first 100 characters *)
         (* of each line to outfile            *)
         WRITELN( outfile, buff );
         READLN( infile ); (* skip characters after column 100*)
                          (* in each line                    *)
      END;
END.
```

Related Information

The end-of-file condition for **TEXT** files is described on page 60.

The end-of-line condition for **TEXT** files is described on page 59.

The **INTERACTIVE** file opening option is described on page 47.

Writing Data to a TEXT File (WRITE)

The **WRITE** procedure writes data to a **TEXT** file beginning at the current position of the file pointer. A call to the procedure has the form:

```
WRITE( f, e )
```

or

```
WRITE( f, e : n )
```

or

```
WRITE( f, e : n1 : n2 )
```

Where	Represents
f	an optional **TEXT** file variable. If you omit the file variable *f*, the file **OUTPUT** is assumed.
e	one or more expressions of the following types: **BOOLEAN** **CHAR** (or subrange) **GCHAR** **GSTRING** **INTEGER** (or subrange) **PACKED ARRAY**[1..*n*] **OF CHAR** **PACKED ARRAY**[1..*n*] **OF GCHAR** **REAL** **SHORTREAL** **STRING**
n, n1, n2	optional field lengths that are **INTEGER** expressions.

If **WRITE** is called for a closed file, the file is opened implicitly for output.

If, during a call to **WRITE**, the length of the logical record being produced becomes longer than the logical record length (**LRECL**) of the **TEXT** file, a runtime error message is generated.

Example

```
PROGRAM dowrite;

VAR
   outfile : TEXT;
   r : ARRAY[1..10] OF RECORD
          name : STRING( 25 );
          age : 0..99;
          weight : REAL;
       END;
   i : 1..10;

BEGIN
   REWRITE( outfile );     (* open outfile for output *)
   FOR i := 1 TO 10 DO
      WITH r[i] DO
         BEGIN
            WRITE( outfile, name : -30, age : 3, ' ' );
            WRITE( outfile, weight );
            WRITELN( outfile );
         END;
END.
```

Writing Data to a TEXT File (WRITELN)

A call to **WRITELN** has the same form as a call to **WRITE** and performs the same function, except that it completes the current logical record so that the next output operation can begin a new logical record. The **WRITELN** procedure applies to **TEXT** files only.

If the record format of the file is fixed (**RECFM**=**F**), **WRITELN** fills the remainder of the current record with blanks. For variable length records (**RECFM**=**V**), no padding occurs, regardless of the logical record length.

WRITELN implicitly opens a closed **TEXT** file.

Example

```
PROGRAM doublespace;

VAR
   filein,
   fileout : TEXT;
   buff : STRING( 255 );

BEGIN
   REWRITE( fileout );
   RESET( filein );
   WHILE NOT EOF( filein ) DO
      BEGIN
         READLN( filein, buff );
         WRITELN( fileout, buff );
         WRITELN( fileout );   (* insert a blank line *)
      END;
END.
```

The PAGE Procedure

The **PAGE** procedure writes an ASCII form feed character (`X'OC'`) to a **TEXT** file.

A call to the procedure has the form:

```
PAGE( f )
```

Where	Represents
f	an optional **TEXT** file variable. The default is **OUTPUT**.

The **PAGE** procedure checks whether **WRITE** has written anything to the file since the most recent **WRITELN**. If it has, **PAGE** does an implicit **WRITELN** on the file, and then writes the form feed character to the file.

Example

```
PROGRAM dopage;

VAR
   print : TEXT;

BEGIN
   .
   .
   REWRITE( print );
   .
   .
   PAGE( print );  (* start a new page *)
END.
```

End-of-Line Condition

The end-of-line condition occurs on a **TEXT** file opened for input when the file pointer is positioned after the end of a logical record. To test for this condition, use the **EOLN** function.

The end-of-line condition becomes true when **GET** is called for a file positioned at the last character of a logical record, or if a call to **READ** consumes all of the characters of the current logical record.

The file pointer always points to a blank character when the end-of-line condition occurs.

The **EOLN** function applies only to **TEXT** files.

Example

```
PROGRAM getlrecl;

VAR
   sysin : TEXT;
   cnt : 0..32767;
BEGIN
   RESET( sysin );  (* compute the length of the first logical*)
   cnt := 0;        (* record of sysin                        *)
   WHILE NOT EOLN( sysin ) DO
      BEGIN
         cnt := cnt + 1;
         GET( sysin );
      END;
   WRITELN( cnt );  (*write out the total length of the first *)
                    (* record                                 *)
END.
```

End-of-File Condition

The end-of-file condition becomes true for a **TEXT** file when one of the following occurs:

- **RESET** is called and the file is empty.
- The file is open for output.
- **READ** or **READLN** is called and all characters of the last logical record are consumed.
- **GET** is called when the file pointer is positioned at the end of the last logical record of the file. That is, the end-of-line condition is true.

To test for this condition, use the **EOF** function.

When the end-of-file condition is true for a file, any calls to **GET**, **READ**, or **READLN** on that file result in a runtime error.

Example

```
PROGRAM getnumrec;

VAR
   sysin : TEXT;
   cnt : 0..32767;

BEGIN
   RESET( sysin );  (* compute the number of logical *)
   cnt := 0;        (* records in file sysin         *)
   WHILE NOT EOF( sysin ) DO
      BEGIN
         cnt := cnt + 1;
         READLN( sysin );
      END;
   WRITELN( cnt );    (* write out the number of records *)
END.
```

The COLS Function

The **COLS** function returns the position of the next character to be written to an output file. **COLS** applies only to **TEXT** files. A call to the procedure has the form:

```
COLS( f )
```

Where	Represents
f	a **TEXT** file variable

Example

The following example shows how to use **COLS** to force output to a specific column:

```
IF tab > COLS( f ) THEN
 WRITE( f, ' ' : tab-COLS( f ) ) ;
```

Processing a Record File

This section describes how to read data from and write it to a record file.

Reading Data from a Record File (GET)

Use the **GET** procedure to read data from a file. A call to the procedure has the form:

```
GET( f )
```

Where	Represents
f	a record file variable

Each call to **GET** reads the next logical record into the buffer referenced by the file pointer. The end-of-file condition becomes true when no more records are in the file.

If the record file is not open for input or update before doing a **GET** operation, a runtime error message is issued.

Example

The program `getrecf` in the following example loops through the file, reading each record and printing out the name and age fields:

```
PROGRAM getrecf;

VAR
   f : FILE OF RECORD
          name   : STRING( 25 );
          age    : 0..99;
          weight : REAL;
          sex    : (male,female);
       END;

BEGIN
   RESET( f );    (* open f for input *)
   WHILE NOT EOF( f ) DO
      BEGIN
         WRITE( ' Name : ', f@.name );
            (* print the fields and read *)
         WRITE( ' Age  : ', f@.age : 3 );
            (* the next record *)
         WRITELN;
         GET( f );
      END;
END.
```

Related Information

The end-of-file condition for a record file is described on page 64.

Writing Data to a Record File (PUT)

Use the **PUT** procedure to write data to a file. A call to the procedure has the form:

```
PUT( f )
```

Where	Represents
f	a record file variable

The **PUT** procedure writes the file record assigned to the output buffer by the file pointer to the associated physical file. Each call to **PUT** produces one logical record.

If the file is not open for output or update before **PUT** is called, a runtime error message is issued.

Example

The program `putrecf` in the following example builds up an output structure for a record file and writes the data to the file.

```
PROGRAM putrecf;

VAR
   f : FILE OF RECORD
         name : STRING( 25 );
         age : 0..99;
         weight : REAL;
         sex : ( male, female );
      END;

BEGIN
   REWRITE( f );     (* open f for input    *)
   f@.name := 'John F. Doe';
   f@.age := 36;
   f@.weight := 100.0;
   f@.sex := male;
   PUT( f );         (* write data to f     *)
END.
```

Reading Data from a Record File (READ)

The statement

```
READ( f, v )
```

is equivalent to

```
v := f@;
GET( f );
```

Where	Represents
f	a record file variable
v	a variable of the same type as the components of the record file

If file *f* is not open for input or update when **READ** is called, a runtime error message is issued.

Using the **READ** procedure on record files is shown in "Writing Data to a Record File (WRITE)" on page 63.

Writing Data to a Record File (WRITE)

The statement

```
WRITE( f, e )
```

is equivalent to

```
f@ := e;
PUT( f );
```

Where	Represents
f	a record file variable
e	a variable of the same type as the components of the record file

If file *f* is not open for output or update when **WRITE** is called, a runtime error message is issued.

Example of Using the READ and WRITE Procedures on Record Files

The program `rdwrtrec` in the following example copies `infile` to `outfile` and loops through the files, reading and writing.

```
PROGRAM rdwrtrec;

TYPE
   rec = RECORD
            name : STRING( 25 );
            age  : 0..99;
            sex  : ( male, female );
         END;
VAR
   infile  : FILE OF rec;
   outfile : FILE OF rec;
   buffer  : rec;

BEGIN
   RESET( infile );
   REWRITE( outfile );
   WHILE NOT EOF( infile ) DO
      BEGIN
         READ( infile, buffer );
         WRITE( outfile, buffer );
      END;
END.
```

Relative Record Access (SEEK)

You can search records of a record file in random order with the **SEEK** procedure. The **SEEK** procedure positions a file pointer to a specific element within a record file. A call to the procedure has the form:

```
SEEK( f, n )
```

Where	Represents
f	a record file previously opened with **RESET**, **REWRITE**, or **UPDATE.**
n	a positive integer expression that corresponds to a record number. The number of the first record is 1.

A call to **GET** or **PUT** operates on the *n*th record of the file. Each subsequent call to **GET** or **PUT** will operate on subsequent records.

If you use **SEEK** to position a file pointer beyond the end of a file, the file is extended to accommodate the new file pointer position.

Note: The **SEEK** procedure does not perform an I/O operation.

Example

Using the **SEEK** procedure to search records randomly is illustrated in the following example. The program `goseek` writes out records from `recfile` in the order specified by the index entries in `idxfile`.

```
PROGRAM goseek;

TYPE
   rec = RECORD
            name : STRING( 25 );
            age  : 0..99;
            sex  : ( male, female );
         END;
   idx = RECORD
            recno : 0..MAXINT;
         END;

VAR
   recfile : FILE OF rec;
   idxfile : FILE OF idx;

BEGIN
   RESET( idxfile );               (* open files for input   *)
   RESET( recfile );
   WHILE NOT EOF( idxfile ) DO
      BEGIN
         SEEK( recfile, idxfile@.recno );
            (* search recfile for the relative record number *)
            (* given by the current index value              *)
         GET( recfile );
         WRITELN( OUTPUT, recfile@.name );
         GET( idxfile );           (* get the next index     *)
      END;
END.
```

End-of-File Condition

The end-of-file condition becomes true for a record file when one of the following occurs:

- **RESET** is called for an empty file.
- The file is opened for output.
- No more records remain in the file on a call to **GET** or **READ**.

To test for this condition, use the **EOF** function.

Any calls to **GET** or **READ** for a file with an end-of-file condition of **TRUE** result in an error message.

Closing a File (CLOSE)

Use the **CLOSE** procedure to close a file explicitly. A call to the procedure has the form:

```
CLOSE( f )
```

Where	Represents
f	a file variable

Closing a file affects the data that ends up in the corresponding external file. If two or more file variables map to the same external file, the external file contains only data associated with the last file closed. When files are closed implicitly, the order in which they are closed is often unpredictable, making it difficult to determine the contents of an external file. Using file names that are not unique within 32 characters or allowing the default external naming convention to take effect can give unpredictable results.

Files are closed implicitly under the following conditions:

- At the end of run time
- When you use one of the following procedures to open a file:

  RESET
  REWRITE
  TERMIN
  TERMOUT
  UPDATE

Files declared in the body of a routine are closed implicitly when the routine returns to its caller.

If you are uncertain about the order in which files are implicitly closed in your program, use **CLOSE** to explicitly close them.

Example

This example shows how to use the **CLOSE** procedure:

```
PROGRAM doclose( OUTPUT );

VAR
   fstk : ARRAY[1..4] OF TEXT;
   filename : STRING( 8 );
   i : 1..4;

BEGIN
   filename := 'TEST ';
   FOR i := 1 TO 4 DO
      BEGIN
         filename[5] := CHR( i + ORD( '0' ) );
         REWRITE( fstk[i], 'FILE = '||filename );
         WRITELN( fstk[i], 'Test #', i : 1 );
         CLOSE( fstk[i] );
      END;
END.
```

Appending Data to a File

You can append data to an existing file by opening it for output with a call to **REWRITE** and specifying **DISP**=**MOD** in the open parameters.

Example

```
PROGRAM ex1;

VAR
   myfile : TEXT;
   s1, s2 : STRING;

BEGIN
   REWRITE( myfile );
   WRITELN( myfile, 'line ONE' );
   RESET( myfile );
   READLN( myfile, s1 );
   IF s1 <> 'line ONE' THEN
      WRITELN( 'error' );
   REWRITE( myfile, 'DISP=MOD' );
   WRITELN( myfile, 'line TWO' );

   RESET( myfile );
   READLN( myfile, s1 );
   IF s1 <> 'line ONE' THEN
      WRITELN( 'error' );
   READLN( myfile, s2 );
   IF s2 <> 'line TWO' THEN
      WRITELN( 'error' );
 END.
```

File-Name Association

Whenever you use **RESET**, **REWRITE**, or **UPDATE** to open a file for input or output, the file variable is associated with an AIX data file. This section shows how the name of the data file is determined and how you can control the association between your Pascal file variables and AIX data file names.

The DDNAME of a File

The **DDNAME** of an opened file variable is a name you can use to determine the external name of the data file associated with that file variable. The **DDNAME** of an open file is determined in one of three ways:

1. If the file variable was opened with a **DDNAME** option, the name given by that option is the **DDNAME**.
2. If the file variable was opened, but not with a **DDNAME** option, the compiler generates a **DDNAME** for you in the following cases:
   - For any file variable that is a record field, an array element, or a pointer target, the compiler concatenates a sequence number, an underscore, and the letters **pas** to form the **DDNAME**.

For example, in VS mode with compiler option **DDNAME=COMPAT**, the statement:

```
REWRITE( a[2,j] )
```

would generate a **DDNAME** like

```
0_pas
```

or:

```
13_pas
```

In standard mode, or in VS mode with compiler option **DDNAME=UNIQUE**, the statement:

```
RESET( p->r.f )
```

would generate a **DDNAME** in a similar way.

- In VS mode with compiler option **DDNAME=UNIQUE**, the compiler generates a **DDNAME** for every file variable that has a name; that is, it is not a record field, an array element, or a pointer target. The name consists of a sequence number prefix, an underscore, and the actual file variable name. All characters of the name are significant.

  For example, the statement:

  ```
  RESET( myfile1 )
  ```

  would generate a **DDNAME** like

  ```
  5_myfile1
  ```

  or :

  ```
  17_myfile1
  ```

3. In all other cases, the name of the file variable is the **DDNAME**. This applies only in standard mode or in VS mode with compiler option **DDNAME=COMPAT**, and only to a file variable that is not a record field, an array element, or a pointer target. The compiler does not generate a **DDNAME** for you. In VS mode with compiler option **DDNAME=COMPAT**, the **DDNAME** is unique in the first 32 characters of the file variable name.

Data File Name

A data file is associated with a file variable when the file variable is opened. The name of the data file associated with an open file variable is determined in one of three ways:

1. If you use a **NAME** file opening option to open the file variable, the name of the data file is the name you specify in the **NAME** option.
2. If you do not use a **NAME** file opening option to open the file variable, but an AIX environment variable exists whose name is the same as the **DDNAME** of the file variable, the name of the data file is the value of that environment variable.
3. If no environment variable exists whose name is the same as the **DDNAME**, the name of the data file is the **DDNAME**.

   Note: Because the **DDNAME** in standard mode is generated by the compiler and is therefore unpredictable, you should use the same file variable name as the environment variable in standard mode.

Related Information

RESET, **REWRITE**, and **UPDATE** are described in "File Opening Procedures" on page 50. The **DDNAME** and **NAME** file opening options are described on page 45. The **DDNAME** compiler option is described on page 32. Using environment variables to supply file names through **DDNAME**s is described in "Environment-Determined File Names" on page 43.

Chapter 5. Improving Performance

This chapter discusses the performance features of XL Pascal. It summarizes the XL Pascal optimization levels, describes some of the optimizations XL Pascal performs, and outlines some programming techniques that you can use to take advantage of the optimization features of the compiler.

Optimization is the process of improving the object code generated by the compiler. Some optimizations make the object code smaller, while others allow it to execute faster. Optimization requires additional compile time, but usually results in reduced run time.

Although optimization makes your program run faster, it does not change what it does or its overall design. Careful choice of an algorithm for each task is your best strategy for optimal performance. The optimizing feature of the compiler is no substitute for an efficient algorithm.

Optimization Levels

XL Pascal compile-time options specify whether optimization is performed. Three optimization levels are available: **OPT**=0, **OPT**=2, and **OPT**=3.

The **OPT**=0 compiler option generates default object code without optimizing the program. It is the recommended level of optimization for a program you are compiling to check syntax or debugging. It provides the fastest compile time but a less efficient program run. The compiler may perform some minor optimizations.

When you request the **OPT**=2 option, XL Pascal performs the optimization techniques shown in the following sections.

When you request the **OPT**=3 option, XL Pascal performs the **OPT**=2 optimizations and additional optimizations that may:

- Require more machine resources during compilation
- Take longer to compile
- Change the semantics of the program

Use the **OPT**=3 option when runtime performance is a critical factor, and machine resources can accommodate the extra compile-time work.

Optimization is accomplished by control and data flow analysis for the entire program. 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.

In loops, optimization operates to prevent movement of any code that might cause an exception unless the exception will occur anyway. For example, in the following loop, code evaluating the expression `n/k` could be moved outside the loop, because it is invariant for each iteration of the loop:

```
FOR j:=1 TO n DO
   IF (K<>0) THEN
      m[j]:=n/k;
```

However, the code will not be moved because `K` could be 0. Invariant computations involving floating-point arithmetic or integer division (including the **MOD** function) are not moved out of a loop.

© Copyright IBM Corp. 1990, 1993

Optimization Techniques

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

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

Common Expression Elimination
Evaluates an expression once and saves its value instead of recalculating several times. This is done even for intermediate expressions within expressions. For example, if your program contains the following statements, where `C` and `D` are variables, the common expression `C+D` may be saved from its first evaluation for use in determining the value of `F`:

```
A := C + D;
    ...
F := C + D + E;
```

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. Code motion accomplishes this.

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

Strength Reduction
Replaces less efficient instructions with more efficient ones.

Constant Folding
Combines constants used in an expression, and creates new ones.

For example:

```
A := 3 * SQR( 20 );
```

would be compiled as:

```
A := 1200;
```

All operators are eligible for constant folding, but only functions that can be used in constant expressions are eligible for constant folding. An expression like `SUBSTR('abc',1,2)` still requires a library call.

Dead Code Elimination
Eliminates unnecessary code. Other optimization techniques may cause code to become dead.

Global Register Allocation
Allocates variables and expressions to available hardware registers by coloring.

Instruction Scheduling
Reorders instructions to minimize execution time.

Related Information

Refer to the *Optimization and Tuning Guide for Fortran, C, and C++: AIX Version 3.2 for RISC System/6000* for techniques you can use for improve the performance of programs compiled with the AIX compilers.

Debugging Optimized Code

Avoid using **dbx** to debug code that has been optimized. If you must debug optimized code, use caution with debugging techniques that rely on examining values in storage. A common expression evaluation may have been deleted or moved. Variables assigned to a register do not appear in storage.

Programs compiled with no optimization may appear to work differently when compiled with **OPT**=2. This is often caused by program variables that have not been initialized. If a program that worked with **OPT**=0 fails when compiled with **OPT**=2, you should look at the cross-reference listing. Check for variables that are used but never set, and for program logic that allows a variable to be used before being set.

Making Your Programs More Efficient

This section contains programming suggestions to take advantage of the optimization features in making your programs more efficient.

Boolean Short Circuiting

XL Pascal makes the evaluation of Boolean expressions with **AND** (&) and **OR** (|) more efficient by *short-circuiting*. That is, the right operand of these expressions is not evaluated if the result of the operation can be determined by evaluating the left operand. The evaluation of the expression is always from left to right.

You can take advantage of Boolean short-circuiting two ways:

- Put *guard expressions* at the beginning of Boolean expressions. They check that other operations can be done, as illustrated in the following example:

  ```
  IF ( p <> NIL ) & ( p@.q = key1 ) THEN
     (* process record p@                          *)
  ELSE
     (* p does not point to a record that can be processed *)
  ```

  is equivalent to:

  ```
  IF ( p <> NIL) THEN
     IF (p@.q = key1) THEN
        (* process record p@                          *)
     ELSE
        (* p does not point to a record that can be processed*)
  ELSE
     (* p does not point to a record that can be processed   *)
  ```

  The expression `(p <> NIL)` guards the statement `(p@.q = key1)`, assuring that `p@.q` is not evaluated if `p = NIL`. Because you need only one copy of the code at `(* p does not point to a record that can be processed *)`, the logic of the program is also simplified.

- For faster execution, put operands determining the value of a Boolean expression early in your code. For example, an expression consisting of several operands joined by **AND** is **FALSE** if any one of the operands is **FALSE**. If you know that one of the operands is usually **FALSE** and the others are as likely to be **FALSE** as **TRUE**, put the one that is usually **FALSE** first.

The following example illustrates this technique.

```
IF test_1( x ) AND  test_2( y ) AND test_3( x, y ) THEN
   (* process x and y   *)
ELSE
   (* do something else *)
```

If `test_1(x)` is more likely to be **FALSE** than `test_2(y)` or `test_3(x,y)`, the program is likely not going to need to evaluate `test_2(y) AND test_3(x,y)`. To make the processing of the **IF** statement more efficient, make `test_1(x)` the first operand.

Value, VAR, and CONST Parameter Passing

The data type of the parameter passed to a routine affects the efficiency of the parameter passing convention you choose.

Scalar Type Parameters

Passing by value is the default, and is the most efficient way to pass scalar type parameters. If the routine changes the value of the actual parameter, pass it by **VAR**. Passing a scalar type parameter by **CONST** is usually of no advantage.

Structured Type Parameters

For structured types, passing by **CONST** or **VAR** is more efficient than passing by value. If the routine does not change any of the components of a structured type parameter, pass it by **CONST**. If the routine changes any of the components of a structured parameter, you must pass it by **VAR**. Because the called routine must make a copy of the actual parameter, passing a structured parameter by value is less efficient.

Any pointer, including the **STRINGPTR** and **GSTRINGPTR** type, is a structured object. To change the target of the pointer but not the pointer itself, you can pass a pointer parameter by **CONST**.

VALUE Initializations

Use a **VALUE** declaration to initialize a **STATIC** or **DEF** variable rather than using an assignment statement at the beginning of a routine. The linkage editor performs this initialization if you use a **VALUE** declaration.

Note: If a routine modifies a **STATIC** or **DEF** variable, the next time the routine is called, the variable will have the new value, not that specified in the **VALUE** declaration.

Chapter 6. Problem Determination

The compiler listing, error messages, and the symbolic debugger help you find and correct problems in your program. This chapter describes how to use them to determine where an error in your program may lie.

Compiler Listings

Depending on the compiler options you specify, the compiler produces a listing that consists of a combination of the following sections:

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

A heading identifies each major section of the listing. Angle brackets precede this heading, allowing you to locate easily the beginning of a section. The heading looks like this:

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

This simple programming example demonstrates the sections of a listing:

```
IBM AIX RISC System/6000 XL Pascal  Version 02.01 --- sample.pas
10/25/93 13:38:24


>>>>>  Options in effect:
        ASCII ATTRIBUTES=FULL CHECK FLOAT=MAF:FOLD NOFLTTRAP
        NOINLINE LANGLVL=VS
        LIST MAXMEM=2048 NATIVE NOOPTIMIZE SOURCE STRICT TRACEID
        XCOFF XREF=FULL

>>>>>  Source Listing:

          1 |program sample ;
          2 |var
          3 |i: integer ;
          4 |begin
          5 |   (* Write out the numbers from 1 to 5 *)
          6 |    for i := 1 to 5 do
          7 |       begin
          8 |          write(i) ;
          9 |       end ;
         10 |    writeln ;
         11 |end.
```

© Copyright IBM Corp. 1990, 1993

```
>>>>>  Input Files:

   M sample.pas( line 0 )


>>>>>  Object Listing:
%EJECT 0

GPR's set/used:  ss-s ssss ssss s---   ---- ---- ---- -sss
FPR's set/used:  ssss ssss ssss ss--   ---- ---- ---- ----
 CR's set/used:  ss-- --ss

   | 000000                          PDEF     sample
   | 000000                          PROC
  1| 000000 mfspr  7C08 02A6    1    -MFSPR   r0=LR
  1| 000004 stm    BFA1 FFF4    3    -STM     (r1,-12)=r29-r31
  1| 000008 st     9001 0008    1    -ST      (r1,8)=r0
  1| 00000C stu    9421 FFB0    1    -STU     r1=(r1,-80)
  1| 000010 l      83E2 0000    1    L        r31=.+sample(r2,0)
  6| 000014 cal    3860 0001    1    LI       r3=1
  6| 000018 st     9061 0038    1    ST       #1(r1,56)=r3
  6| 00001C cmpi   2C83 0005    1    C        cr1=r3,5
  6| 000020 bc     4185 0060    3    BT       CL.4,cr1,0x2/gt
                                    CL.5:
                                    CL.1:
  1| 000024 l      83A2 001C    1    L        r29=./GlAuto/(r2,0)
  6| 000028 l      8061 0038    1    L        r3=#1(r1,56)
  6| 00002C st     907D 0000    2    ST       i(r29,0)=r3
  1| 000030 l      83C2 0014    1    L        r30=.output(r2,0)
  8| 000034 oril   63C3 0000    2    LR       r3=r30
  8| 000038 ai     30BF 0060    1    AI       r5=r31,96
  8| 00003C cal    3880 0001    1    LI       r4=1
  8| 000040 bl     4BFF FFC1    0    CALL $pckopen,3,output",r3,r4
,r5,$pckopen",cr[0167]",r0",r3"-r12",fp0"-fp13",mq",lr",xer"-ffsr
",ffcr

  8| 000044 cror   4DEF 7B82    0
  8| 000048 oril   63A4 0000    1    LR       r4=r29
  8| 00004C oril   63C3 0000    1    LR       r3=r30
  8| 000050 l      8084 0000    1    L        r4=i(r4,0)
  8| 000054 cal    38A0 000C    1    LI       r5=12
  8| 000058 cal    38C0 0000    1    LI       r6=0
  8| 00005C bl     4BFF FFA5    0   CALL Spfwrti,4,output",r3,r4
,r5,r6,$pfwrti",cr[0167]",r0",r3"-r12",fp0"-fp13",mq",lr",xer"-ff
sr",ffcr
```

```
 8| 000060 cror   4DEF 7B82    0
                                   CL.2:
 6| 000064 l      8061 0038    1    L         r3=#1(r1,56)
 6| 000068 cmpi   2C83 0005    2    C         cr1=r3,5
 6| 00006C bc     4186 0014    3    BT        CL.6,cr1,0x4/eq
 6| 000070 l      8061 0038    1    L         r3=#1(r1,56)
 6| 000074 ai     3063 0001    2    AI        r3=r3,1
 6| 000078 st     9061 0038    1    ST        #1(r1,56)=r3
 6| 00007C b      4BFF FFA8    0    B         CL.1
                                   CL.6:
                                   CL.7:
                                   CL.3:
                                   CL.4:
 1| 000080 l      83C2 0014    1    L         r30=.output(r2,0)
10| 000084 oril   63C3 0000    2    LR        r3=r30
10| 000088 ai     30BF 0060    1    AI        r5=r31,96
10| 00008C cal    3880 0001    1    LI        r4=1
10| 000090 bl     4BFF FF71    0    CALL $pckopen,3,output",r3,r4
,r5,$pckopen",cr[0167]",r0",r3"-r12",fp0"-fp13",mq",lr",xer"-ffsr
",ffcr

10| 000094 cror   4DEF 7B82    0
10| 000098 oril   63C3 0000    1    LR        r3=r30
10| 00009C bl     4BFF FF65    0    CALL
$pfwrtln,1,output",r3,$pfwrtln"
,cr[0167]",r0",r3"-r12",fp0"-fp13",mq",lr",xer"-ffsr",ffcr
10| 0000A0 cror   4DEF 7B82    0
                                CL.0:
 1| 0000A4 cal    3860 0004    1    LI        r3=4
 1| 0000A8 cal    38A0 0000    1    LI        r5=0
 1| 0000AC oril   60A4 0000    1    LR        r4=r5
 1| 0000B0 bl     4BFF FF51    0    CALL
$plstmng,3,r3,r4,r5,$plstmng"
,cr[0167]",r0",r3"-r12",fp0"-fp13",mq",lr",xer"-ffsr",ffcr

1| 0000B4 cror   4DEF 7B82    0
                               CL.8:
11| 0000B8 l      8001 0058    1    L         r0=#stack(r1,88)
11| 0000BC mtspr  7C08 03A6    2    LLR       lr=r0
11| 0000C0 ai     3021 0050    1    AI        r1=r1,80
11| 0000C4 lm     BBA1 FFF4    3    LM
r29,r30,r31=#stack(r1,-12)
11| 0000C8 bcr    4E80 0020    0    BA        lr

    Straight-line exec time   53
   | 0000CC                     Tag Tables
   | 0000CC 00000000
   | 0000D0 00022041
   | 0000D4 80030001
   | 0000D8 000000CC
   | 0000DC 0006
```

```
       sample
 0002E8                        Constant Area Starts Here
 0002E8 186D9BE0 696E7075 74202020 20202020
 0002F8 20202020 20202020 20202020 20202020
 000308 20202020 696E7075 74202020 20202020
 000318 20202020 20202020 20202020 20202020
 000328 20202020 00000000 00000000 00000000
 000338 FFFFFFFF 8A000400 00000000 00000000
 000348 186D9BE0 6F757470 75742020 20202020
 000358 20202020 20202020 20202020 20202020
 000368 20202020 6F757470 75742020 20202020
 000378 20202020 20202020 20202020 20202020
 000388 20202020 00010101 00000000 00000000
 000398 FFFFFFFF 8A000080 00000000 00000000
 0003A8 186D9BE0 73746465 72722020 20202020
 0003B8 20202020 20202020 20202020 20202020
 0003C8 20202020 73746465 72722020 20202020
 0003D8 20202020 20202020 20202020 20202020
 0003E8 20202020 00020101 00000000 00000000
 0003F8 FFFFFFFF 8A000080 00000000 00000000
 000408                        End Of Code Csect
 Instruction count is 51

>>>>>  CROSS REFERENCE LISTING

=================================================================
IDENTIFIER          ATTRIBUTES
(File#-Line#)
=================================================================
i                   CLASS = global auto  TYPE = integer  SIZE = 4
(0-3)               ALIGN = double word
       REFERENCES:  *0-6   0-8

sample              CLASS = automatic   TYPE =    SIZE = 0
(0-1)

>>>>> Compilation Epilog:
Compiler was created 93/10/09 23:21:31.

Diagnostics Issued:
        Total errors           :0
        Maximum Severity       :0
```

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

Header Section

The listing file always has a header section containing the following:

- A compiler identifier consisting of:
  - compiler name
  - version number
  - release number
- Source file name
- Date of compilation
- Time of compilation

The header section appears only once in the file as the first line in the listing.

Options Section

The options section is always present in a listing. It lists only the options in effect for the compilation. With the **OPTIONs** compiler option set, this section lists the settings for all options. Refer to page 37 for more information on the **OPTIONs** compiler option.

Source Section

The source section contains the input source lines, each with a line number and a file number, if one exists. The source section also contains error messages interspersed with the code, as they would appear on the terminal during compilation. The source lines and the numbers associated with them appear only if the **SOURCE** compiler option is in effect. Refer to page 38 for more information on the **SOURCE** compiler option.

Related Information

"Correcting Compile-Time Errors" on page 81 describes the format of error messages in the source section. "Compile-Time Error Messages" on page 79 describes file and line numbering.

Cross-Reference and Attribute 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 variable.
- Attributes of the variable (if **ATTR** is in effect). Attributes information includes the type and the storage class of the variable.

  Storage class can be any one of the XL Pascal variable types:

  automatic
  dynamic
  external
  global automatic
  parameter
  static
  type

  Type can be any one of the XL Pascal data types:

  alfa
  alpha
  array
  boolean
  char
  file
  function
  gchar
  gstring
  gstringptr
  integer
  pointer
  procedure
  real
  record
  set
  shortreal
  space
  string
  stringptr
  text

- Dimensions of the variable (if an array), the size in bytes, and the alignment of the variable (if **ATTR** is in effect).

- File and line numbers on which your program defines, references, or modifies the variable. If the variable is initialized or set, the coordinates are marked with an asterisk (*), for example, `TEST 0–10, 0–20, *0–30`. If the variable is referenced, the coordinates are not marked.

Specifying the **FULL** suboption with **XREF** or **ATTR** causes XL Pascal to report all variables in the compilation unit. Specifying no suboption displays only the variables you use.

File Table Section

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

Object Section

This section is produced 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 run time) and the total number of assembler instructions produced is displayed. XL Pascal repeats this section for each compilation unit.

Compilation Epilog Section

This section is always present in a listing. It is the last section of the listing for each compilation. It displays the date and time the compiler was created, contains the diagnostics summary, and says whether the compilation was successful.

Error Messages

The compiler issues messages for all errors it detects during compilation. Compiled object code and runtime environment routines issue error messages for errors they detect at run time.

Compile-Time Error Messages

Compile-time error messages go to **STDOUT**. With the appropriate options set, they can also be displayed in the listing file in which the errors were detected.

This type of message describes the error that was detected and its location. The format of a compile-time error message is

```
ff . ll |===> (s) 15cc-nnn message-text
```

Where	Is
ff	source file number
ll	line number within the source file where the error was detected
s	severity level
15	product number, indicating that the XL Pascal compiler issued the error message
cc	component number, indicating the component of the compiler that issued the message: **30** Pascal-specific message, indicating a syntax error or a violation of Pascal requirements **00** code generation error message **01** IBM AIX RISC System/6000 XL general compiler message
nnn	unique error message number
message–text	message text describing the error, including identifiers and line numbers associated with it

Include files are numbered sequentially in the order in which **%INCLUDE** directives are processed. The top-level source file has no source file number. XL Pascal numbers source lines separately for each file.

The error severity level is a number from 0 to 8, indicating how severe the error is and what the compiler does about it. The return code value of the compiler is the highest error message severity code in the program.

Message Classes and Compiler Response

Message Class	Return Code	Explanation	Compiler Response
Informational	0	Note or suggestion to programmer about conditions found during compilation. Not an error.	Compilation continues
Warning	1	A minor error has been detected that should be corrected. The error does not interfere with compilation.	Compilation continues
Error	2	The compiler detects an error that will cause a runtime trap if the program is run.	Compilation continues
Severe error	3	The program has an error that prevents object code generation.	Compilation continues but object code is not produced
Unrecoverable error	4-8	A compiler limit has been exceeded, a compiler error has occurred, or a source file cannot be found.	The compiler halts

Example

This XL Pascal compile-time error message:

```
3 . 26 |===> (3) 1530-267 XMIN is already defined on line 5.
```

means:

- The error was detected in the third include file on line 26.
- The error is a level-3 error, so compilation continues but the compiler does not generate object code.
- The error message was issued by the XL Pascal compiler (15) and is specific to Pascal (30).
- The error message number is 267.
- The message text indicates that the identifier **XMIN** was defined more than once: on line 5 in the top-level source file, and again on the line where the error was detected.

Note: The identifier **XMIN** appears in uppercase in the error message, but it can be in any combination of uppercase and lowercase letters in the source file.

Correcting Compile-Time Errors

XL Pascal displays compile-time diagnostic messages to standard error. Messages are placed in the source listing if you request a listing using the **LIST**, **SOURCE**, **XREF**, **ATTR**, or **STAT** compiler option. Compile-time error messages generally mean program errors that prevent the program from running in its present form. These error messages identify the file and line numbers in the source program where the error is detected. A *column mark line* appears above the error message in the listing, but not in the standard error file displayed after compilation. The symbol _|_ marks the column at which the error is first detected.

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

- Source line
- Column mark line
- Error message, which consists of:
  - File and line number of the error
  - Severity of the message
  - Number of the error message
  - Text of the error

For example:

```
20 | i := max('a', 3);
                   _|_
20 |===> (3) 1530-210: Arguments to max not conformable.
```

If the **NOSOURCE** option is in effect and you request, for example, an object listing (with the **LIST** option), only error messages appear in the source section of the listing. They contain:

- Column mark line
- Error message, which consists of:
  - File and line number of the error
  - Severity of the message
  - Number of the error message
  - Text of the error

For example:

```
                    _|_
20 |===> (3) 1530-210: Arguments to max not conformable.
```

There are two kinds of compile-time errors: *syntax* errors and *semantic* errors. Syntax errors are caused by incorrect punctuation or misuse of reserved words or operators. Semantic errors result from incorrect definition or use of constants and identifiers.

Correcting Syntax Errors

Syntax error message texts include one of the following phrases:

- Syntax error, unexpected symbol
- Unexpected reserved word
- Unexpected keyword
- Unexpected end-of-file

They also include the unexpected symbol or identifier, and the column mark line shows which column contains it.

The XL Pascal compiler does all syntax checking on one pass. It detects errors based on the language objects it encounters during its single pass through the source code, marking points where it detected unexpected symbols. XL Pascal issues a diagnostic if it finds something it did not expect based on what it has processed to that point.

Because XL Pascal syntax analysis processes combinations of several input symbols at once, an error might be detected one symbol before or after the one identified in the error message. In the following example of a syntax error, XL Pascal detects a missing semicolon at the end of the **VAR** statement only after it has read the **BEGIN** symbol.

```
PROGRAM error ;

VAR
   sysprint : TEXT    (* semicolon missing here  *)

BEGIN
   REWRITE(sysprint) ;
   WRITELN(sysprint,'Here's a message from XL Pascal') ;
END.
```

Compiling the program results in the following syntax error messages:

```
>>>>>  Source Listing:
    1 |PROGRAM error ;
    2 |
    3 |VAR
    4 |   sysprint : TEXT     (* semicolon missing here *)
    5 |
    6 |BEGIN
     _|_
    6 |===> (3) 1530-144: Syntax error: unexpected symbol BEGIN.
    7 |   REWRITE(sysprint) ;
    8 |   WRITELN(sysprint,'Here's a message from XL Pascal') ;
                         _|_
    8 |===> (3) 1530-144: Syntax error: unexpected symbol ,.
                                                     _|_
    8 |===> (3) 1530-126: String ended by end of line.
    9 |END.
         _|_
    9 |===> (3) 1530-145: Unexpected end of file.
```

The compiler recovers from a syntax error by finding the next error-free part of the program and starting again there. In the program **error**, the compiler cannot compile any of the program past the line with the mismatched single quotation marks, and indicates that it was unable to recover from a syntax error by issuing the following message:

```
Unexpected end of file.
```

Correcting other syntax errors usually gets rid of `Unexpected end of file` messages.

Correcting Semantic Errors

Semantic errors usually result from using an identifier or constant incorrectly, as shown in the following example:

```
>>>>>  Source Listing:
   1 |PROGRAM error ;
   2 |
   3 |TYPE this_type = no_type ;
                         _|_
   3 |===> (3) 1530-372: Type NO_TYPE is undefined.
   4 |
   5 |VAR
   6 |   my_var : no_type ;
   7 |      another_var : no_type ;
   8 |STATIC more_var : no_type ;
   9 |
  10 |BEGIN
  11 |   REWRITE(my_var) ;
       _|_
  11 |===> (3) 1530-205: Argument to rewrite must be of type FILE.
  12 |   WRITELN(my_var,'This has no type') ;
               _|_
  12 |===> (3) 1530-166: Identifier not in proper context.
  13 |END.
```

For each error detected, the error message describes the problem, and the column mark line shows which symbol is incorrect.

One error can cause more than one message. In the example, the type declaration for `this_type` in line 3 has an error message. At line 5, the variable `my_var` is declared as type `no_type`, and XL Pascal issues an error message the first time the variable `my_var` is used at line 12. Correcting the type declaration prevents the other error messages.

Semantic error messages are generally not repeated. As shown in the example, only the first variable declaration with type `no_type` causes an error message stating that the type name is not valid. The other **VAR** declaration and the **STATIC** declaration that use the type `no_type` do not have error messages.

Runtime Error Messages

Runtime error messages are issued either by compiled object code or by runtime environment routines. They are put in the standard error file.

The format of an XL Pascal runtime error message is:

`1530-nnn message-text`

Where	**Is**
1530	the combined product and component number indicating that the error message is from XL Pascal object code or an XL Pascal runtime environment routine
nnn	the unique error message number
message–text	the message text describing the error.

Some runtime error messages are simply warnings indicating that an error was detected but the program can continue running. For example, the following shows that the operand of **DISPOSE** was either **NIL** or not currently pointing to a Pascal dynamic variable:

```
1530-022 DISPOSE operand NIL or invalid.
```

The **DISPOSE** operation is unsuccessful, but the program continues running.

Other runtime errors are not recoverable and the program ends immediately. For example, the following shows that the program tried to access a subscripted variable with an incorrect subscript value:

```
1530-101 Subscript out of range.
```

Correcting Runtime Errors

Runtime error messages generally mean that the program will run, but it does not run correctly. It may end abnormally, or it may run but give incorrect output.

Runtime Checking Errors

If checking is activated with the **%CHECK** compiler directive, a runtime trap occurs when the compiler detects an error. When this happens, the program ends. The following errors cause a runtime trap:

Subscript error
The value of an array subscript is outside the bounds allowed for the subscript.

Subrange error
The value assigned to a subrange variable is outside the bounds allowed for the subrange.

NIL pointer
Attempting to reference a variable from a pointer with the value NIL.

CASE label
The expression of a **CASE** statement has a value other than any of the specified **CASE** labels, and there is no **OTHERWISE** clause.

String truncation
Attempting to assign a value to a string with more characters than the maximum length of the string.

Assertion failure
Processing an **ASSERT** statement with an associated Boolean expression value of **FALSE**.

String subscript out of bounds
An indexing operation was attempted on a string with a length greater than the current length of the string.

Function value
A function was returned without assigning a result value.

Example

```
VAR
   st3 : STRING( 3 );
   st5 : STRING( 5 );
   ch : CHAR;
   i : INTEGER;
   .
   .

FUNCTION
   f : INTEGER;
BEGIN
   RETURN;
   f := 1;
END;
.
.
st5 := 'abcde';
st3 := st5;                     (* string truncation *)
ASSERT FALSE > TRUE             (* assertion failure *)
st3 := 'ab';
ch := st3[3];     (* string subscript out of bounds *)
i  := f;          (* function not assigned a result *)
```

Using the Symbolic Debugger

XL Pascal supports the **dbx** symbolic debugger. Use the **–g** compiler option when compiling to use the symbolic debugger.

Using Traceback Facilities

If you use one of the traceback facilities and your program causes a runtime trap, the program halts.

XL Pascal can produce a traceback when there is a runtime trap. A traceback shows the sequence of routine calls that led to the trap. To get a traceback, call the predefined procedure **xl__trap** sometime before the trap occurs. If your program causes a trap any time after calling **xl__trap**, XL Pascal produces a traceback. If your program does not cause a trap, **xl__trap** has no effect.

Note: Be careful to spell **xl__trap** correctly. There are two underscore characters together.

Example

Given the following program:

```
PROGRAM traceback;
BEGIN
   xl__trap;
   assert( FALSE );
END.
```

If you compile without the **–g** option, the following output is produced:

```
Trap encountered:
SIGTRAP - Trace trap

Traceback:

Offset 0x00000014 in procedure traceback

--- End of call chain ---
```

If you compile with the **–g** option, the following output is produced:

```
Trap encountered:
SIGTRAP - Trace trap

Traceback:

Offset 0x00000014 in procedure traceback, line 6 in file p.pas

--- End of call chain ---
```

When you run the program within **dbx**, an error message is written to standard error. By specifying the **–g** option at compilation time, the line number of the instruction causing the trap and the file name are also written to standard error. Use the **where** subcommand in **dbx** to get a complete traceback written to standard error showing the line number and the sequence of instructions leading up to the trap.

Related Information

The **dbx** symbolic debugger and its subcommands are described in the *AIX Version 3.2 Commands Reference*. The **–g** compiler option is described on page 32. The format of runtime messages is described in “Runtime Error Messages” on page 83.

Message Catalog Errors

The XL Pascal Compiler and runtime environment produce all of their messages from message catalogs using the AIX National Language Support facilities. Before you can use these facilities, the AIX environment variables **LANG** and **NLSPATH** should be set to a language for which the XL Pascal message catalogs were installed on your system.

Default Messages

The default U.S. English message catalogs are in the following directories:

```
/usr/lpp/xlp/lib/default_msg (compile-time messages)
/usr/lpp/xlprtemsg           (runtime messages)
```

These directories contain the most current message catalogs in U.S. English. They are the default message catalogs that are shipped with the compiler and runtime licensed program products.

The file names of the message catalogs are the same for all supported international languages.

If the specified language message catalog cannot be opened to display messages, the corresponding default U.S. English message catalog is used.

If the specified language message catalog can be opened, but a particular message cannot be found, the missing message is retrieved from the corresponding default U.S. English message catalog. Messages which can be found appear in the specified language.

If compile-time messages are appearing in U.S. English when they should be in another language, verify that the correct message catalogs are installed and that the environment variables are set accordingly. For runtime messages, also ensure that your program calls the **setlocale** routine.

To determine the XL Pascal message catalogs that were installed, use the following command to display a list:

```
lslpp -h xlp*.msg
```

Related Information

See "Environment Variables for the Message Catalogs and Help FIles" on page 14 for more information on **LANG** and **NLSPATH**. For more information about the AIX national language facilities, see the *AIX Version 3.2 General Programming Concepts*.

Chapter 7. Interlanguage Applications

The AIX Operating System for IBM RISC System/6000 system and the XL Pascal compiler support routine calls and data references among routines written in different languages. This chapter shows how to write Pascal references to external variables defined in other XL languages. It also discusses how to write Pascal routines that call code written in those languages and that can be called from such routines.

This chapter assumes you are familiar with the syntax of the languages you will be using. The elements of the languages are not defined here. Refer to the appropriate XL language reference manuals for detailed information.

The Pascal language described here is IBM AIX XL Pascal. The Fortran language is IBM AIX XL Fortran, and the C language is IBM AIX XL C.

Interlanguage Reference Requirements

Code written in XL Pascal must meet four requirements to use data defined in other XL languages or to call routines written in other XL languages:

1. The name of a variable or routine defined in another language must be accessible in Pascal as an external name.
2. An external variable defined in another language must be declared in Pascal with a matching data type.
3. An actual parameter passed to a routine written in another language must be declared in the calling program with a data type that matches the corresponding formal parameter. It must also be passed with the same parameter-passing mode as the corresponding formal parameter.
4. The type of value returned by a function written in another language must match the type of value expected by the calling program.

External Names

This section discusses how your XL Pascal program gains access to external routines and variables, and how to refer to external names that contain uppercase and lowercase letters.

External Routine Names

To call an external routine, Pascal code must include an appropriate **FUNCTION** or **PROCEDURE** declaration containing an **EXTERNAL** directive. For example, the following procedure declaration gives a Pascal routine access to external procedure `getscr`:

```
PROCEDURE getscr( var img : rc ); EXTERNAL;
```

External Variable Names

To gain access to an external variable, Pascal code must include an appropriate **REF** or **DEF** declaration for the variable. The declaration in the following example gives a Pascal routine access to external variable `xaxis_symb`:

```
DEF xaxis_symb : integer;
```

© Copyright IBM Corp. 1990, 1993

A **REF** declaration is preferable to **DEF** because **REF** does not create the variable if no other definition of it exists, whereas a **DEF** creates the variable even if it has not been previously defined. For example, if you misspell the name of the external variable, **DEF** creates a variable with the incorrect name, but a **REF** declaration generates an undefined symbol diagnostic message when the program is link-edited and loaded.

Mixed-Case External Names

By default, all external names used in XL Pascal programs are folded to lowercase regardless of how they are written in the source code. Letters are folded to lowercase, and all other characters remain intact. For example, external names written as **Z_axis2**, **Z_Axis2**, or **Z_AXIS2** all become references to the external name **z_axis2**.

If the program being called requires it, your program can preserve mixed uppercase and lowercase in external names: with the **MIXED** compiler option or with the linkage editor **rename** command. The **rename** command is equivalent to the **–brename** linkage editor option.

The MIXED Compiler Option

Specifying **MIXED** makes all identifiers case sensitive. You can set this option in the Pascal source code with the **%OPTION MIXED** directive, or on the **xlp** command line with the **–U** option or the **–qmixed** option.

Linkage Editor Rename Command

The **rename** command in the linkage editor (**ld**) converts a reference to one name into a reference to another name. Both names can be mixed case. For example, the following linkage editor command converts a Pascal reference to the lowercase external name `z_axis2` into a reference to the mixed-case name `Z_Axis2`:

```
rename z_axis2 Z_axis2
```

The **rename** command is useful if you want only certain selected Pascal identifiers to be in mixed case. To make all identifiers in your program case sensitive, use the **MIXED** compiler option.

Related Information

The **MIXED** compiler option is described on page 37. The linkage editor **rename** command is described with the AIX **ld** command in the *AIX Version 3.2 Commands Reference*.

Matching Data Types

Matching data types in different languages have the same internal representation. A data object can be processed by code written in different languages only if all of the declarations for the object specify matching types. This section describes the correspondence between data types in XL Pascal and other XL languages.

Matching Data Types among XL Languages

When passing data between programs written in different languages, you must know which data types have counterparts across languages, and where no equivalent representation exists. You should avoid using Pascal data types not available in other languages.

The following table shows the correspondence among the data types available in the XL Pascal, XL Fortran, and XL C languages. Blank cells indicate that no applicable matching data type exists.

XL Pascal Data Types	XL Fortran Data Types	XL C Data Types
ARRAY	Dimensioned variable (transposed)	array pointer (*) to type
BOOLEAN		
CHAR	CHARACTER*1	char
Enumeration		enumeration
FILE	No corresponding type	
Functional Parameter	Dummy procedure	pointer (*) to function
GCHAR		wchar_t
GSTRING(n)		
GSTRINGPTR		
INTEGER	INTEGER INTEGER*4	signed long int
PACKED -32768..32767	INTEGER*2	short signed int
PACKED 0..65535	LOGICAL*2	short unsigned int
PACKED -128..127	INTEGER*1	signed char
PACKED 0..255	LOGICAL*1	unsigned char
PACKED ARRAY[1..n] OF CHAR	CHARACTER*n	char array
PACKED RECORD	Sequence derived type	
POINTER (with NOPTR4)	no corresponding type	
POINTER (with PTR4)	Integer POINTER	*type
REAL	REAL*8 DOUBLE PRECISION	double
RECORD	Derived type	struct
RECORD variant		union
SET		
SHORTREAL	REAL REAL*4	float
SPACE		
STRING(n)		
STRINGPTR		
TEXT		

Creating Matching Data Types

In some cases, defining a new data type in XL Pascal to match an otherwise unmatched type of another language is possible, as shown in the following examples:

- Pascal does not have a predefined type to match the Fortran **COMPLEX** type. To define a Pascal matching type for **COMPLEX**, use a Pascal **RECORD** type with two **SHORTREAL** fields.

- C does not have a predefined type that matches a Pascal **POINTER** type. To define a C matching type for a **POINTER**, use a C **struct** with one C pointer field and one **int** field. However, if the **–qptr4** option was used on the Pascal file, no changes are required in C.

Character Variable Types

Most numeric data types have counterparts across languages, but character and string data types generally do not. This section describes the conventions for passing character data between XL Pascal and the other XL languages.

Pascal and Fortran

Both Pascal and Fortran have the same internal representation for fixed strings. In Fortran, the only character type is **CHARACTER*N**, which corresponds to the Pascal **PACKED ARRAY**[1..*n*] **OF CHAR**. Both are stored as sets of contiguous bytes, one character per byte. Like the Pascal fixed string, the length of a Fortran character variable or character array element is determined at compile time, and is therefore static. Fortran does not have a data type to match the Pascal varying-length **STRING** type.

Pascal and C

Pascal and C use different internal representations for character stings. In Pascal, you can declare strings several different ways, each of which has a different internal representation. C has one internal representation of character strings but has several ways to declare them.

C Strings and char Arrays

C represents a character string in a sequence of consecutive bytes with an ASCII null character (hexadecimal `'00'XC`) in the byte immediately following the last character in the string. Storing a C string in a fixed-length array of **char** requires that one of the array elements contain the ASCII null to mark the end of the string content.

The C analog of a Pascal fixed-length character string is an array of **char** whose size is the length of the Pascal string. The C analog of a Pascal varying-length character string is a **struct** containing two fields:

- A **short int** giving the *current* length of the string
- A **char** array whose size is the *maximum* length of the Pascal string

Pascal Arrays of CHAR

Pascal represents fixed-length and varying-length character strings in a fixed-length array of characters.

Fixed-length strings The size of the array is the length of the string, and all elements of the array are character positions in the string.

Varying-length strings An additional 2-byte packed subrange value gives the current length of the string. The fixed-length array is large enough for the maximum length of the string.

The Pascal analog of a C character string of indeterminate length is a **PACKED ARRAY OF CHAR** containing an ASCII null (`'00'XC`) in the element following the actual string content. If a Pascal program does not insert the ASCII null element, C code cannot find the end of the string.

Note: XL Pascal will not treat an ASCII null as the end of a string unless the Pascal program explicitly looks for the null.

Storage of Arrays

Fortran arrays have a user-defined lower bound. Arrays in C have a lower bound of 0. XL Pascal and XL C store elements of multidimensional arrays in *row-major order*; that is, in consecutive locations as their rightmost subscript increases. Conversely, Fortran arranges array elements in ascending storage units in *column-major order*, with the elements stored in consecutive locations as their leftmost subscript increases.

Example

The following table shows how a two-dimensional array declared in Pascal by

```
a:ARRAY[1..3,1..2] OF INTEGER
```

is stored in all three XL languages.

Language	XL Pascal	XL Fortran	XL C
Array Declaration	a:ARRAY[1..3,1..2] OF INTEGER	INTEGER a(2,3)	int a[3][2]
Elements	a[1,1] a[1,2] a[2,1] a[2,2] a[3,1] a[3,2]	a(1,1) a(2,1) a(1,2) a(2,2) a(1,3) a(2,3)	a[0][0] a[0][1] a[1][0] a[1][1] a[2][0] a[2][1]

Pascal Arrays, C Pointers, and Arrays

A Pascal **POINTER** occupies 8 bytes. It contains the address of an object and additional information about the size, alignment, and allocation status of the object. However, when the **–qptr4** option is turned on, the **POINTER** will occupy 4 bytes. When you specify **–qptr4**, only the address of the object is included. A Pascal **POINTER** type has no matching analog in C, unless the **–qptr4** option is specified.

The C pointer type (*) is an address occupying 4 bytes. A C program can use a pointer in two ways:

- As the address of an isolated object, which corresponds to a Pascal **VAR** or **CONST** parameter whose type matches the target object of the C pointer.
- As the address of a generic element in an array, which corresponds to a Pascal **VAR** or **CONST** parameter that is an array whose type matches the target object of the C pointer.

Examples

The following examples show how to convert between XL Pascal 8-byte pointers and XL C 4-byte pointers, assuming the **–qptr4** option is not used.

```
SEGMENT POINTERS;

TYPE
   BOTH_PTRS = RECORD (*overlap Pascal pointer with C pointer *)
                  CASE INTEGER OF
                     1 : ( pp : POINTER );
                     2 : ( cp : INTEGER;
                           cf : INTEGER );
               END;  (* endcase *)

(* Converting a Pascal pointer to an integer  *)
(* to be used as a C pointer                  *)

FUNCTION PTR_P2C( ppi : POINTER ) : INTEGER; EXTERNAL;

FUNCTION PTR_P2C;

VAR
   gp : BOTH_PTRS;

BEGIN (* PTR_P2C *)
   gp.pp := ppi;
   PTR_P2C := gp.cp;
END;  (* PTR_P2C *)

(* Converting an integer representing a C pointer *)
(* to a Pascal pointer                            *)

FUNCTION PTR_C2P( cpi : INTEGER ) : POINTER; EXTERNAL;

FUNCTION PTR_C2P;

VAR
   gp : BOTH_PTRS;

BEGIN (* PTR_C2P *)
   gp.cp := cpi;
   gp.cf := MAXINT; (*defeats Pascal addressing range check*)
   PTR_C2P := gp.pp;
END;  (* PTR_C2P *)
```

Routine Calls and Returned Values

Pascal code that calls an external routine must declare that routine as an **EXTERNAL FUNCTION** or an **EXTERNAL PROCEDURE**. This section describes how the language of the external routine and the type of the return value determine the corresponding declaration of the routine in the calling program.

Function Calls

Declaring an external routine in the Pascal code as a **FUNCTION** requires the routine to return a value whose type matches the type of value that the calling routine expects.

Pascal External Routines
You must define the external routine as a **FUNCTION**. In the calling routine, the **FUNCTION** declaration corresponding to the external routine must be the same type as the external routine.

Fortran External Routines
You must define the external routine as a **FUNCTION**. The type of the external routine must match the type in the **FUNCTION** declaration in the Pascal program.

C External Routines
All C routines are defined as functions. The type of the function must match the type in the **FUNCTION** declaration in the Pascal program. If the function definition in C does not specify a return value type, it returns a C **int** that matches the Pascal **INTEGER** type. The C type **void** is equivalent to **int** for all interlanguage references.

Procedure Calls

Declaring an external routine as a **PROCEDURE** causes a Pascal program to ignore any value returned by the called routine.

Pascal External Routines
You must define the external routine as a **PROCEDURE**.

Fortran External Routines
You can define the external routine as a **SUBROUTINE** or a **FUNCTION**. If it is a function, a Pascal routine can call it as a procedure and ignore the returned value.

C External Routines
You must define all C routines as functions. A Pascal routine can call a C function as a procedure and ignore its returned value.

Routine Linkage Convention

XL Pascal uses the AIX common linkage convention. For every routine call, the compiler forms a sequence of parameter words from the types and modes of the routine parameters. AIX common linkage convention uses the parameter word sequence and the type of the routine return value to define the following details of routine linkage:

- Register contents
- Register usage (both general and floating-point registers)
- Stack contents
- Object code format

The other XL languages also use the AIX common linkage convention. In general, for code written in one XL language to call a routine written in another language, both must form identical parameter word sequences for the routine call.

Parameter Linkage

XL Pascal passes parameters as a sequence of parameter words. The parameter word sequence for each formal parameter is determined from left to right by the formal parameter list of the called routine.

Each parameter is passed as one or two parameter words. The first word is either the value or the address of the actual parameter. If the actual parameter requires a descriptor, the second word is the value of the descriptor.

First Parameter Word

The first parameter word is determined by the mode and type of the formal parameter.

Pass by Value (no VAR or CONST Specified)

For a scalar type whose value fits in one 32-bit word, the first parameter word is the value of the actual parameter.

For a **REAL** parameter, the 64-bit value is treated as two words containing a 64-bit floating-point value.

For a parameter that is not a scalar, the first parameter word is the address of the actual parameter. The called routine makes a local copy of the actual parameter and processes its local copy. A **RECORD** or **ARRAY** parameter is passed this way, even if its entire value fits in one word.

Pass by VAR or CONST

The first parameter word is the address of the actual parameter.

Procedural Parameters

The first parameter word is the address of the routine descriptor of the actual parameter.

Second Parameter Word

The second parameter word is a *descriptor* word whose value is determined by the type of the formal parameter. A descriptor gives additional information about the size of the actual parameter or the sizes of its components. It is used only if the actual parameter requires a descriptor. If the type of the formal parameter does not require any descriptor information, XL Pascal uses no second parameter word.

Descriptor words are required for two types of formal parameters:

Conformant string

The length of the string, including the leading two bytes that contain the current length of the string. This is 2 plus the declared maximum length of the string.

Variant record

The size of the record. If the record was created by **NEW** with tag values, the size is determined by the active variants for those tag values. For any tag values not specified, the size of the largest variant is used.

Examples of Parameter Word Sequences

For this routine declaration:

```
PROCEDURE d1 ( p1:INTEGER; VAR p2:CHAR; p3:STRINGPTR);
```

XL Pascal forms the following parameter word sequence:

1. The value of the first actual parameter, an **INTEGER**.
2. The address of the second actual parameter, a **CHAR**.
3. The address of the third parameter, a **STRINGPTR**, which occupies 8 bytes and cannot be passed as one word. Procedure `d1` makes a local copy of the parameter and processes that local copy.

For this routine declaration:

```
FUNCTION f ( r:recptr; procedure pp(VAR x:REAL);
VAR
   s:STRING);    (* recptr is a variant record type *)
```

XL Pascal forms the following parameter word sequence:

1. The address of the first actual parameter, a variant record of the type `recptr`. Function `f` makes a local copy of the parameter and processes that local copy.
2. The size of the first actual parameter, which is determined by the calling program from the tag values that are specified when the record is created.
3. The address of the descriptor for the procedure that is passed as the second actual parameter.
4. The address of the third actual parameter, a varying-length string.
5. The size of the third actual parameter, which is 2 plus the declared maximum length of the actual parameter.

The NONPASCAL Routine Directive

Within each XL language, the AIX common linkage convention is controlled by the compiler options and varies according to whether certain kinds of information are duplicated both in the stack and in parameter registers.

To have an XL Pascal routine call an external routine written in a language that uses a different linkage variation, your Pascal code should use the **NONPASCAL** directive in the **EXTERNAL** declaration for that routine.

The **NONPASCAL** directive causes the compiler to generate code that satisfies all of the AIX common linkage convention variations. Calls to **NONPASCAL** routines may generate less efficient code than other calls, but **NONPASCAL** works with all variations of the AIX common linkage convention.

Related Information

Refer to the *AIX Version 3.2 Assembler Language Reference* for details of the AIX common linkage convention.

Interlanguage Parameter Passing Conventions

This section describes the general correspondence of formal parameter definitions across XL languages.

Calls between XL Pascal and XL C

XL C is case sensitive. The name of a routine or external variable can include both uppercase and lowercase letters, exactly as it appears in the C source code. Pascal by default is not case sensitive. To make Pascal external names match C external names, use the techniques described in “Mixed-Case External Names” on page 90.

Parameter Handling in C and Pascal

Pointers A C pointer (*) is an address that occupies 4 bytes. A Pascal **POINTER** type occupies 8 bytes. It contains the address of an object and additional information about the size, alignment, and allocation status of the object. However, when the **–qptr4** option is turned on, the **POINTER** will occupy 4 bytes. When you specify **–qptr4**, only the address of the object is included. “Pascal Arrays, C Pointers, and Arrays” on page 93 explains in detail the differences between C and Pascal pointers.

Parameter Passing Modes

All C parameters are passed by value. Pascal parameters by default are passed by value, but **VAR** and **CONST** are passed by reference. A C parameter that is not a pointer (without *) generally corresponds to a default Pascal parameter of the matching type.

A C pointer type by default is a one-word address that generally corresponds to a Pascal **VAR** or **CONST** parameter, or a Pascal **ARRAY**, depending on how the C code uses it. The type of the Pascal parameter matches the target type of the C pointer.

Character Strings

C and Pascal use different internal representations for character strings, as described in “Character Variable Types” on page 92.

External Variables

A C **extern** variable corresponds to a Pascal **DEF** variable of matching type.

Function Return Values

A C function return type that is not a pointer (without *) corresponds to a Pascal function value of matching type.

A C pointer (*) function return type can correspond to a Pascal **ARRAY** of matching type, depending on how the C code uses it. If it does not correspond to a Pascal **ARRAY**, no matching type exists.

Passing Character Strings to C from Pascal

C functions frequently pass character strings as C **char*** parameters. Use the following steps to write a Pascal function call that passes a **char*** parameter to a C function:

1. Write the Pascal **FUNCTION** declaration with a **VAR CHAR** formal parameter. This is passed as one parameter word containing the address of a **CHAR** and matches a C **char*** parameter.
2. Create the string in a Pascal **PACKED ARRAY**[1..*n*] **OF CHAR** object, where *n* is at least one greater than the length of the string.
3. Insert an ASCII null (`'00'xc`) in the next array element immediately following the actual string content, which is necessary for the C code to detect the end of the string.
4. Pass the first element of the array as the actual parameter.

Creating the string in a Pascal **STRING** or as the target of a **STRINGPTR** still permits C code to process the string. If the C code modifies the string length by moving the null, the change is not reflected in the current length of the Pascal string.

Receiving Character Strings from C in Pascal

If C code calls a Pascal routine and passes a character string to it, the string should be passed as a C **char** array to match a Pascal **PACKED ARRAY**[1..*n*] **OF CHAR**.

A C program can call a Pascal routine and pass a character string to it as a C **char*** parameter. Use the following steps to write an XL Pascal routine to accept the character string this way:

1. Write the Pascal routine with a **VAR PACKED ARRAY**[1..*n*] **OF CHAR** parameter corresponding to the C **char***, where *n* is a constant at least one greater than the length of the C string.
2. Since these are not matching parameter types, you may have to compile the Pascal routine with the **NOEXTCHK** option.

3. Do not access elements of the **PACKED ARRAY OF CHAR** parameter past the first element containing the ASCII null.

Note: Your XL Pascal routine may not be portable to other Pascal compilers.

Examples

The following examples show Pascal code that follows C conventions for routine declarations, parameter passing, and string manipulation. All routines are functions, parameters are passed by value, a C **char*** parameter is equivalent to a Pascal **VAR CHAR** parameter, and all strings are terminated by ASCII null characters.

The following Pascal program `p_call_c` calls a C routine and passes an integer value and the character string from Pascal to C:

```
PROGRAM p_call_c( OUTPUT );

CONST
   null_char = '00'XC;

TYPE
   string_buffer = PACKED ARRAY[1..256] OF CHAR;

VAR
   i, rc : INTEGER;
   c_string : string_buffer;

FUNCTION c_callby_p( i : INTEGER; VAR c : CHAR ) : INTEGER;
   EXTERNAL;

BEGIN
   i := 7;
   c_string := 'String passed from Pascal to C' || null_char;
   WRITELN( 'Pascal: p_call_c prepares to call c_callby_p.' );
   rc := c_callby_p( i, c_string[1] );
   RETCODE( rc )
END.
```

The following C routine `c_callby_p` is called with two parameters, an **int** value, and a C-style **char*** string. It prints the input values, calls the Pascal function `p_callby_c`, and passes it different values of the same types, returning the string:

```
"String passed from C to Pascal."
```

The Pascal function `p_callby_c` has the same interface as `c_callby_p`. It prints the input values it receives from the C routine.

```
#include <stdio.h>

extern int p_callby_c(int,char*);

int c_callby_p(int ip,char* ps)
{
  int rc;
  printf("C: c_callby_p receives integer value %5d\n",ip);
  printf("C: c_callby_p receives string '%32s'\n",ps);
  ip += 1;
  rc = p_callby_c(ip,"String passed from C to Pascal.\n");
  return(rc);
}
```

The following example shows the Pascal segment containing the function `p_callby_c`. Both the function and the segment have the same name, which is the Pascal way of compiling an external routine separate from a program.

```
SEGMENT p_callby_c;

CONST
   null_char = '00'XC;

TYPE
   string_template = PACKED ARRAY[1..256]OF CHAR;

FUNCTION p_callby_c(pi:INTEGER;VAR s:string_template) :
   INTEGER; EXTERNAL;
FUNCTION p_callby_c;

VAR
   i : INTEGER;
   c_string : STRING;

BEGIN
   WRITELN( 'Pascal: p_callby_c receives integer value ', pi );
   c_string := '';
   FOR i := 1 TO HBOUND( s ) DO
      IF s[i] = null_char THEN
         BEGIN
         c_string := substr( str( s ), 1, i );
         LEAVE
      END;
   WRITELN( 'Pascal: p_callby_c receives character string
       "'||c_string||'"' );
   p_callby_c := 0
END;
```

The following commands compile these programs into a single executable file **p_call_c**:

```
xlc c_callby_p.c -oc_callby_p.o -c

xlp p_call_c.pas p_callby_c.pas c_callby_p.o -op_call_c
```

The command p_call_c runs the program to produce the following output:

```
Pascal: p_call_c prepares to call c_callby_p.
C: c_callby_p receives integer value     7
C: c_callby_p receives string '  String passed from Pascal to C'
Pascal: p_callby_c receives integer value           8
Pascal: p_callby_c receives character string "String passed from
C to Pascal."
```

Calls between XL Pascal and XL Fortran

The following are the major differences in parameter handling between Fortran and Pascal.

Parameter Handling in Fortran and Pascal

Arrays

A Fortran dimensioned variable corresponds to a Pascal array of matching type in which the dimension declarations are transposed.

Fortran stores arrays so that they appear to be transposed in Pascal. For example, an external array declared in Fortran as:

```
INTEGER a (5,20)
```

should be transposed in the Pascal declaration:

```
DEF a: ARRAY[ 1..20 , 1..5 ] of INTEGER;
```

The element referred to as a(I,J) in a Fortran program is the same as the element referred to in a Pascal program as a[J,I].

Fortran array dimensions always have a user-defined lower bound, but Pascal **ARRAY** index types can be declared with arbitrary lower bounds. "Storage of Arrays" on page 93 describes how Fortran and Pascal arrays differ in storage.

Parameter passing modes

A Fortran parameter is by default passed by reference. If a Fortran routine call passes an argument with the **%VAL** keyword, the argument is passed by value.

The default parameter passing mode in Pascal is by value, which is the opposite of Fortran. Parameters declared **VAR** or **CONST** are passed by reference in Pascal.

Examples

The following examples show Pascal code calling Fortran and being called by Fortran. The conventions for routine declarations, parameter passing, and string manipulation are similar to those used for interlanguage calls between Pascal and C as shown on page 99.

The following Pascal program `p_call_f` calls Fortran function `f_callby_p` and passes an integer value and the following null-terminated character string to it:

```
PROGRAM p_call_f(OUTPUT);

CONST
   null_char = '00'xc;

TYPE
   string_buffer = PACKED ARRAY[1..256] OF CHAR;

VAR
   i,rc:INTEGER;
   f_string:string_buffer;

FUNCTION f_callby_p(VAR i:INTEGER; VAR c:string_buffer):
   INTEGER;EXTERNAL;

BEGIN
   i := 7;
   (* Form string in C format, ended by null character. *)
   f_string := 'String passed from Pascal to Fortran.' ||
      null_char;
   WRITELN('Pascal: p_call_f prepares to call
   f_callby_p.');
   rc := f_callby_p(i,f_string);
   RETCODE(rc)
END.
```

The Fortran function `f_callby_p` in the following example is called with two parameters, an integer value, and a character string. It prints the input values, calls the Pascal function `p_callby_f`, passes it different values of the same types, and returns the string:

```
'String passed from Fortran to Pascal.'
```

The Pascal function `p_callby_f` has the same interface as `f_callby_p`. It prints the input values it receives from the Fortran function. The integer parameters are passed by **VAR** because Fortran requires parameters passed by reference.

```
       integer function f_callby_p(ip,ps)
       integer ip
       character*256 ps
       character*256 as
     1 format('Fortran: f_callby_p receives integer ',i5)
       write(6,1)ip
     2 format('Fortran: f_callby_p receives string ''',a40,'''')
       write(6,2)ps
       ip = ip + 1
C  Form output string in C format, ending with null character.
       as = 'String passed from Fortran to Pascal.\0'
       f_callby_p = p_callby_f(ip,%ref(as))
       end
```

The following example shows the Pascal segment containing the function `p_callby_f`. Both the function and the segment have the same name.

```
SEGMENT p_callby_f;

CONST
   null_char = '00'xc;
TYPE
   string_template = PACKED ARRAY[1..256]OF CHAR;

FUNCTION p_callby_f( VAR pi : INTEGER; VAR
   s : string_template ) : INTEGER; EXTERNAL;
FUNCTION p_callby_f;
VAR
   i : INTEGER;
   c_string : STRING;
BEGIN
   WRITELN('Pascal: p_callby_f receives integer ',pi);
   c_string := '';
   FOR i := 1 TO HBOUND(s) DO
      IF s[i] = null_char THEN
         BEGIN
            c_string := substr(str(s),1,i); LEAVE
         END;
   WRITELN('Pascal: p_callby_f receives string
"'||c_string||'"');
   p_callby_f := 0
END;
.
```

The following commands compile the programs to produce the executable file **p_call_f**:

```
xlf f_callby_p.f -of_callby_p.o -c

xlp p_call_f.pas p_callby_f.pas f_callby_p.o -lxlp -lxlf
-op_call_f
```

The **xlp** command includes **ld** command options that bind both the XL Pascal and XL Fortran runtime environments.

The command `p_call_f` runs the program to produce the following output:

```
Pascal: p_call_f prepares to call f_callby_p.
Fortran: f_callby_p receives integer     7
Fortran: f_callby_p receives string String passed from Pascal to
Fortran.
Pascal: p_callby_f receives integer           8
Pascal: p_callby_f receives character string "String passed from
Fortran to Pascal."
```

Enforcement of Type Matching

The XL compilers work together with the linkage editor to enforce the requirements for matching data type.

Linkage Editor Type Checking

For each external variable or routine declaration, the compiler calculates a 32-bit *hash signature*, which is calculated from the following:

- External variable types
- Numbers and types of formal parameters
- Function value types

The hash signature for each external variable and routine is included in the object file produced by the compiler.

All XL compilers use the same utility to calculate these hash signatures. The hashing algorithm is based on the internal representations of the data types. Different XL compilers produce the same hash signature for matching data types.

The linkage editor compares the hash signatures of all references to each external symbol. If two separately compiled object files do not have matching declarations for any external symbol, the hash signatures are different. The linkage editor issues a diagnostic indicating that the declarations are mismatched.

File Types Do Not Match

In general, file types do not match between different XL languages. Each of the languages has its own file handling operations, so a file opened by code written in one language generally cannot be correctly read from, written to, or closed by code written in another language.

All routines in a multilanguage application can read from standard input and write to standard output. A Pascal routine can read from the predefined file **INPUT** and write to the predefined file **OUTPUT** even if other languages are also using standard input and standard output. For any other file, the file should be opened, read from, written to, and closed in one language.

Suppressing Type Checking with the NOEXTCHK Option

Two type declarations written in different languages can be mismatched in the linkage editor even if their internal representations are the same. This is especially likely if the types contain strings or files, components which are processed differently by each language.

To suppress type checking by the linkage editor, compile your Pascal file with the **NOEXTCHK** option. With this option, the compiler produces a special *no check* hash signature for all external variables and routines. The no check hash signature prevents the linkage editor from checking references to external symbols.

Related Information

A full description of the **NOEXTCHK** option is on page 32.

Appendix A. Example Program

This appendix outlines a sample program, the listing it would produce with the compiler options specified, and the output from running it.

Source File

The following source file (**sample.pas**) contains a small program that prints the numbers from 1 to 5:

```
program sample ;
var
i: integer ;
begin
  (* Write out the numbers from 1 to 5 *)
  for i := 1 to 5 do
     begin
        write(i) ;
     end ;
  writeln ;
end.
```

Example Listing

If you specify the **SOURCE** compiler option with the following command, XL Pascal produces the listing (**sample.lst**) shown on page 106:

```
xlp -qsource sample.pas
```

© Copyright IBM Corp. 1990, 1993

```
IBM AIX RISC System/6000 XL Pascal  Version 02.01.0000.0000 ---
sample.pas 10/25/93 13:16:32


>>>>>  Options in effect:
        ASCII CHECK FLOAT=MAF:FOLD NOFLTTRAP NOINLINE LANGLVL=VS
        MAXMEM=2048 NATIVE NOOPTIMIZE SOURCE STRICT TRACEID XCOFF

>>>>>  Source Listing:

         1 |program sample ;
         2 |var
         3 |i: integer ;
         4 |begin
         5 |  (* Write out the numbers from 1 to 5 *)
         6 |   for i := 1 to 5 do
         7 |      begin
         8 |         write(i) ;
         9 |      end ;
        10 |   writeln ;
        11 |end.

>>>>>  Input Files:

   M sample.pas( line 0 )


>>>>>  Compilation Epilog:

Compiler was created 93/10/09 23:21:31.

Diagnostics Issued:
        Total errors           :0
        Maximum Severity       :0
```

Output Produced

The following output is displayed when the sample program runs:

```
          1           2           3           4           5
```

Appendix B. ASCII Character Set

XL Pascal uses the American National Standard Code for Information Interchange (ASCII) character set as its collating sequence.

The following table lists the standard ASCII characters in ascending numerical order, with their corresponding decimal and hexadecimal values. It also shows the control characters with **Ctrl–** notation. For example, the carriage return (ASCII symbol `CR`) appears as `Ctrl-M`, which you enter by simultaneously pressing the **Ctrl** key and the **M** key.

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning
0	00	Ctrl-@	NUL	null
1	01	Ctrl-A	SOH	start of heading
2	02	Ctrl-B	STX	start of text
3	03	Ctrl-C	ETX	end of text
4	04	Ctrl-D	EOT	end of transmission
5	05	Ctrl-E	ENQ	enquiry
6	06	Ctrl-F	ACK	acknowledge
7	07	Ctrl-G	BEL	bell
8	08	Ctrl-H	BS	backspace
9	09	Ctrl-I	HT	horizontal tab
10	0A	Ctrl-J	LF	new line
11	0B	Ctrl-K	VT	vertical tab
12	OC	Ctrl-L	FF	form feed
13	0D	Ctrl-M	CR	carriage return
14	0E	Ctrl-N	SO	shift out
15	0F	Ctrl-O	SI	shift in
16	10	Ctrl-P	DLE	data link escape
17	11	Ctrl-Q	DC1	device control 1
18	12	Ctrl-R	DC2	device control 2
19	13	Ctrl-S	DC3	device control 3
20	14	Ctrl-T	DC4	device control 4
21	15	Ctrl-U	NAK	negative acknowledge
22	16	Ctrl-V	SYN	synchronous idle
23	17	Ctrl-W	ETB	end of transmission block
24	18	Ctrl-X	CAN	cancel
25	19	Ctrl-Y	EM	end of medium
26	1A	Ctrl-Z	SUB	substitute
27	1B	Ctrl-[	ESC	escape
28	1C	Ctrl-\	FS	file separator

© Copyright IBM Corp. 1990, 1993

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning
29	1D	Ctrl-]	GS	group separator
30	1E	Ctrl-^	RS	record separator
31	1F	Ctrl-_	US	unit separator
32	20		SP	digit select
33	21		!	exclamation point
34	22		”	double quotation mark
35	23		#	number sign
36	24		$	dollar sign
37	25		%	percent sign
38	26		&	ampersand
39	27		’	apostrophe
40	28		(	left parenthesis
41	29		)	right parenthesis
42	2A		*	asterisk
43	2B		+	addition sign
44	2C		,	comma
45	2D		–	subtraction sign
46	2E		.	period
47	2F		/	right slash
48	30		0	
49	31		1	
50	32		2	
51	33		3	
52	34		4	
53	35		5	
54	36		6	
55	37		7	
56	38		8	
57	39		9	
58	3A		:	colon
59	3B		;	semicolon
60	3C		<	less than
61	3D		=	equal
62	3E		>	greater than
63	3F		?	question mark
64	40		@	at symbol
65	41		A	
66	42		B	

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning
67	43		C	
68	44		D	
69	45		E	
70	46		F	
71	47		G	
72	48		H	
73	49		I	
74	4A		J	
75	4B		K	
76	4C		L	
77	4D		M	
78	4E		N	
79	4F		O	
80	50		P	
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	
91	5B		[	left bracket
92	5C		\	left slash
93	5D		]	right bracket
94	5E		^	hat, circumflex, caret
95	5F		_	underscore
96	60		‘	grave accent
97	61		a	
98	62		b	
99	63		c	
100	64		d	
101	65		e	
102	66		f	
103	67		g	
104	68		h	

Decimal Value	**Hex Value**	**Control Character**	**ASCII Symbol**	**Meaning**
105	69		i	
106	6A		j	
107	6B		k	
108	6C		l	
109	6D		m	
110	6E		n	
111	6F		o	
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	
122	7A		z	
123	7B		{	left brace
124	7C				logical or
125	7D		}	right brace
126	7E		~	similar, tilde
127	7F		DEL	delete

Appendix C. XL Pascal and the 1983 ANSI/IEEE Pascal Standard

The XL Pascal Compiler complies with the requirements of ANSI/IEEE 770X3.97–1983. This implies conformance to the following standards:

- International Standards Organization (ISO) 7185–1983 (Level 0), Programming Languages, Pascal
- Federal Information Processing Standard (FIPS) PUB 109, Pascal

This appendix describes the following features of XL Pascal:

- Implementation-defined features
- Implementation-dependent features
- Error handling
- Extensions

Implementation-Defined Features

The implementation-defined features of XL Pascal constitute extensions to Pascal as it is specified by ANSI/IEEE 770X3.97–1983.

VS Mode Extensions

- The characters allowed in string types are the printable characters of the ASCII character set plus the space character. Depending on the system and terminal being used, different characters may be displayed.
- The character set used by the data type **CHAR** is the ASCII character set. Depending on the system and terminal being used, different characters may be displayed.
- The ordinal values used for each character in the **CHAR** data type are the values in the ASCII character set. Depending on the system and terminal being used, certain ordinal values may correspond to different characters.
- The subset of real numbers used by the XL Pascal **REAL** and **SHORTREAL** data types is the set of floating-point values occupying 8 and 4 bytes respectively, as described by ANSI/IEEE Standard 754–1985. The accuracy of real arithmetic is determined by RISC System/6000 floating-point operations.
- The value of the predefined constant **MAXINT** is 2147483647.
- The actions taken by the file-handling procedures and the time of their performance are specified as follows:
  - **REWRITE**(*f*) erases the external file bound to *f* and opens the file for output. These operations are performed immediately when running **REWRITE**.
  - **PUT**(*f*) places the value of the file buffer into a special buffer. This buffer is written to the external file when the special buffer becomes full, the routine or program containing file *f* ends, or the procedures **CLOSE**, **RESET**, **REWRITE**, **TERMIN**, **TERMOUT**, and **UPDATE** are applied to file *f*.

© Copyright IBM Corp. 1990, 1993

  - **RESET**(*f*) opens the external file bound to *f* for input. This occurs immediately when running **RESET**. The first element in the external file bound to *f* is immediately assigned to the file buffer, unless the file is interactive. The file pointer of an interactive file has the value **NIL** until the first **GET** or **READ** is done on the file. At this time, the file buffer contains the first element in the file.
  - **GET**(*f*) causes the file pointer to point to the next unread element of *f* (if any), and updates the file buffer with that element. This action occurs immediately on processing **GET**.
- The default length for integer output is 12.
- The default length for Boolean output is 10.
- Boolean variables are returned in all uppercase (**TRUE**, **FALSE**).
- The default length for real output is 20.
- The exponent character for real output is E.
- Files listed as program parameters are bound to external files in the same way as any other file.
- Applying **RESET** and **REWRITE** to the **TEXT** files **INPUT** and **OUTPUT** yields the same results as applying them to any other file.

AIX RISC System/6000 System Extensions

- Three exponent digits are used for real output.
- The number of significant characters in both internal and external identifiers is 256.
- No diagnostics are produced for multiple **DEF** variables on the same identifier.
- The number of significant characters in a program name is 256.
- XL Pascal supports ASCII single-byte character set (SBCS) data and multibyte character set (MBCS) data in comments and literal constants targeted for an MBCS output device.
- Multiple **VALUE** declarations on a structured **STATIC** or **DEF** variable are allowed if there is not more than one specification of the value of any scalar component of the structured variable.
- Restrictions on the **SPACE** data type:
  - If the **%CHECK** compiler option is off, references or assignments beyond the end of a **SPACE** variable may cause unpredictable results.
  - If **%CHECK SUBSCRIPT** is on and the computed address of the **SPACE** index expression lies outside the storage occupied by the space, a runtime diagnostic message is issued when the program is running in **dbx**.
  - Passing an element of a **SPACE** by reference is valid.
- Routines can be nested to at least 20 levels.
- The result of a negative shift count is zero for <<, >>, or a shift count greater than 32 for either shift left or shift right.
- The maximum number of set elements is 256, and the maximum set size is 32 bytes.
- When the expression in an **ASSERT** statement is not **TRUE**, and the program is running in **dbx**, a diagnostic message is issued, and the program stops.

- The following are the supported keyword names of the XL Pascal file opening options:

  NAME
  LRECL
  RECFM
  DISP
  UCASE
  INTERACTIVE
  DDNAME

- The pointer variable passed as a parameter of the **MARK** procedure is set to the address of the associated heap control block. Do not use the returned pointer as a base of a dynamic variable.
- **RELEASE** and **DISPOSE** set their parameter variables to **NIL**.
- The size of the string returned by the **ITOHS** function is 8 characters.
- The longest supported token returned by the **TOKEN** routine is 16 characters.
- The value returned by the **CLOCK** function is the number of microseconds the program has been running.
- The **PARMS** routine returns a conformant string.
- The **RETCODE** procedure sets the completion code.

Implementation-Dependent Features

Because the following implementation-dependent features of XL Pascal are used frequently, they are not flagged:

- The order-of-evaluation of these items is not predefined:
  - Array indexes.
  - Member-designators in a set constructor.
  - Component expressions of a member designator in a set constructor.
  - Operands of a binary operator (such as A + B). If the value of a relational expression can be determined after the evaluation of the first operand, the second operand is not evaluated.
  - Actual parameters in a procedure or function call, including accessing and binding of the actual parameter.
  - The left side and the right side of an assignment statement.
  - File parameters of the predefined procedures **READ** and **WRITE**.
  - Array parameters of the predefined procedures **PACK** and **UNPACK**.
- Applying the **PAGE** procedure to a file causes the system to produce the ASCII form feed character at the point where **PAGE** is applied.
- The binding of program parameters that are not declared as files depends on how they are declared.

Error Handling

The ANSI standard for Pascal requires a complying system to document its treatment of errors. The following list describes how XL Pascal handles the errors listed in the ANSI-83 Pascal Standard. It corresponds to the error-handling list in Appendix D of that standard.

This discussion assumes that all compile-time and runtime checking has been turned on. If the checking is off, some errors normally detected are not detected. If an error is severe enough to stop processing, further errors may not be detected.

An error stated to be *generally detected* may be left undetected under some conditions. Errors stated to be detected without qualification are always detected.

1. Array subscripting errors are detected.
2. Accessing a component of an inactive variant is not detected (for **RECORD** data type).
3. **NIL** pointer dereferencing errors are detected.
4. Undefined pointer dereferencing errors are not detected, except in special circumstances.
5. Removing the identifying value of an identified (pointer-qualified) variable while a reference to that variable exists is not detected.
6. Altering the value of a file variable while a reference to its buffer variable exists is not detected.
7. Assignment compatibility errors between actual and formal ordinal value parameters are detected.
8. Assignment compatibility errors between actual and formal set value parameters are detected.
9. Using the **PUT**, **WRITE**, **WRITELN**, and **PAGE** procedures on files not open for output is generally detected.
10. Using the **PUT**, **WRITE**, **WRITELN**, and **PAGE** procedures on undefined files for output is generally detected.
11. With **LANGLVL=STANDARD**, it is not possible to use the **PUT**, **WRITE**, **WRITELN**, and **PAGE** procedures on files with the end-of-file condition not true before the call. For **LANGLVL=VS**, the **UPDATE** procedure and random access I/O allow the end-of-file condition to be false.
12. Using the **PUT** procedure with an undefined buffer variable is not detected.
13. Using the **RESET** procedure on an undefined file causes an error if the file cannot be bound successfully.
14. Using the **GET**, **READ**, and **READLN** procedures on files not open for input is generally detected.
15. Using the **GET**, **READ**, and **READLN** procedures on undefined files is generally detected.
16. Attempting to use the **GET**, **READ**, and **READLN** procedures with the end-of-file condition true is detected.
17. For **READ**, assignment compatibility errors are detected.
18. For **WRITE**, assignment compatibility errors are detected.

19. Activating a variant part of a variable created with the **NEW** procedure of the form (*p,t1,...,tn*) where the variant is in the same variant part as, but different from, one of the specified variants is not detected.
20. Using the **DISPOSE**(*p*) procedure on a variable created with the **NEW** procedure of the form (*p,t1,...,tn*) is not detected.
21. Using the **DISPOSE** procedure of the form (*p,t1,...,ty*) on a variable created with the **NEW** procedure of the form (*p,t1,...,tx*), where *x* is not equal to *y*, is not detected.
22. Using the **DISPOSE** procedure of the form (*p,t1,...,tn*) on a variable with active variants different from (*t1,...,tn*) is not detected.
23. Using the **DISPOSE** procedure on a **NIL** pointer is detected.
24. Using the **DISPOSE** procedure on an undefined pointer is generally detected.
25. Accessing variables created by the **NEW** procedure of the form (*p,t1,...,tn*) using the identified variable in a factor, an assignment statement, or as an actual parameter, is not detected.
26. Assignment compatibility errors on the ordinal variable in the **PACK** procedure are detected.
27. Accessing undefined elements of the unpacked array in the **PACK** procedure is not detected.
28. Exceeding the index of the unpacked array in the **PACK** procedure is detected.
29. Assignment compatibility errors on the ordinal variable in the **UNPACK** procedure are detected.
30. Having undefined elements in the packed array in the **UNPACK** procedure is not detected.
31. Exceeding the index of the unpacked array in the **UNPACK** procedure is detected.
32. Applying the **SQR** function to a value whose square does not exist is not generally detected.
33. Applying the **LN** function to a value less than or equal to zero is not generally detected.
34. Applying the **SQRT** function to a negative value is not detected (no error message is displayed); the result is NaN.
35. Applying the **TRUNC** function to a value that would produce an integer with too large a magnitude is generally detected.
36. Applying the **ROUND** function to a value that would produce an integer with too large a magnitude is generally detected.
37. Applying the **CHR** function to a value with no character value is detected.
38. Applying the **SUCC** function to a value with no successor is not detected.
39. Applying the **PRED** function to a value with no predecessor is not detected.
40. Using the **EOF** function on an undefined file is not detected.
41. Using the **EOLN** function on an undefined file is not detected.
42. Using the **EOLN** function on a file with the end-of-file condition true is not detected.
43. Using undefined variables in an expression is not detected.
44. Real division by zero is not generally detected.

45. Integer division by zero is not generally detected.
46. Using the **MOD** operator with a second operand less than or equal to zero is detected.
47. Performing integer operations or functions in violation of the mathematical rules for integer arithmetic is not generally detected.
48. Undefined function result errors are detected.
49. Assignment compatibility errors with ordinal types are detected.
50. Assignment compatibility errors with set types are detected.
51. **CASE** statement index errors are detected.
52. Assignment compatibility errors between **FOR** loop indexes and initial values are detected.
53. Assignment compatibility errors between **FOR** loop indexes and final values are detected.
54. Attempting to read an integer from a **TEXT** file where a sequence of characters does not form a signed number, after skipping preceding spaces and end-of-line conditions, is detected.
55. Type compatibility for integers read from **TEXT** files is checked and detected.
56. Attempting to read a number from a **TEXT** file where a sequence of characters does not form a signed integer, after skipping preceding spaces and end-of-line conditions, is detected.
57. Using the **READ** or **READLN** procedure on a file with the buffer variable undefined is detected. Trying to read past the end of a file can cause this error.
58. Using the form *e : TotalWidth : FracDigits* or *e : TotalWidth* in the **WRITE** and **WRITELN** procedures for **TEXT** files where *TotalWidth* or *FracDigits* is less than 1, is detected. This error applies to **LANGLVL**=**STANDARD** only.

Extension Handling

When you specify **LANGLVL**=**STANDARD**, you cannot use any extensions to the ANSI-83 Standard Pascal. When you specify the **LANGLVL**=**VS** compiler option, you can use all of the extensions described in the *Language Reference for IBM AIX XL Pascal Compiler/6000*.

Appendix D. Implementation Dependencies

This appendix discusses the various implementation-specific characteristics of XL Pascal:

- Routines That Can be Passed as Parameters
- Data Types
- Compiler Limits
- Differences between XL Pascal and:
  - VS Pascal Release 1
  - VS Pascal Release 2

Note: Compiler limits are approximations; your program might deviate from the maximum limits depending on its complexity.

Routines That Can Be Passed as Parameters

Standard mode XL Pascal does not allow any predefined routines to be passed as actual parameters.

In VS mode, it allows the predefined routines listed in the following table to be passed as actual routine parameters to another routine:

ARCTAN	EXP	LTOKEN	SIN
CLOCK	GTOSTR	PARMS	SQRT
COLS	HALT	PICTURE	STOGSTR
COS	ITOHS	RANDOM	TOKEN
DATETIME	LN	RETCODE	TRACE

Data Types

INTEGER Data Type

The largest integer that can be represented is 2147483647. This is the highest value that can be represented in a 32-bit word and is equal to the predefined constant **MAXINT**.

The minimum value of the type **INTEGER** is equal to the predefined constant **MININT**, which has the value –2147483648.

Representation of Floating-Point Constants

Constant Name	Decimal Approximation	Exact Representation
MAXREAL	1.797693134862E+308	'7FEFFFFFFFFFFFFF'XR
MINREAL	4.940656458412E-324	'0000000000000001'XR
EPSREAL	2.220446049250E-016	'3CB0000000000000'XR
MAXSREAL	3.402823466385E+038	'47EFFFFFE0000000'XR
MINSREAL	1.401298464325E-045	'36A0000000000000'XR
EPSSREAL	1.192092895508E-007	'3E80000000000000'XR

© Copyright IBM Corp. 1990, 1993

Notes

- XL Pascal represents hexadecimal floating-point numbers with the syntax `'...'XR`.
- **MAXREAL** is the largest finite double-precision floating-point number that can be represented.
- **MINREAL** is the smallest positive finite double-precision floating-point number that can be represented.
- **EPSREAL** is the smallest positive double-precision floating-point number that, when added to 1, is detectable. This value is often needed in numerical computations involving converging series.
- **MAXSREAL** is the largest finite single-precision floating-point number that can be represented.
- **MINSREAL** is the smallest positive finite single-precision floating-point number that can be represented.
- **EPSSREAL** is the smallest positive single-precision floating-point number that, when added to 1, is detectable. Like **EPSREAL**, this value is often needed in numerical computations involving converging series.

Related Information

See the *Language Reference for IBM AIX XL Pascal Compiler/6000* for more information on hexadecimal floating-point numbers.

SET Data Type

Given a **SET** data type of the form `SET of a..b`, where *a* and *b* express the lower and upper bounds of the base scalar type, the following conditions must hold true:

```
ORD(a) >= 0
ORD(b) <= 255
```

Compiler Limits

Routine Nesting

The compiler requires temporary storage for each incompletely processed construct. Such storage has a maximum of 255 tokens. This size affects the limits to cumulative nesting of the following items:

- Routines
- Looping statements
- Structured data types
- Expressions

Many factors affect the maximum cumulative nesting of these constructs. Among the more important are the length of the parameter list and the syntactic complexity of the routine. Although it is difficult to state exactly the maximum number of nesting levels for routines, XL Pascal supports routine nesting of at least 20 levels.

Identifiers

Identifiers cannot span multiple lines, but XL Pascal permits identifiers up to the length of the source line. Because XL Pascal accepts a source line of up to 256 characters, the longest identifier can be 256 characters.

External procedures and **DEF** and **REF** variables should not start with the **$** character. This character is used internally by XL Pascal for runtime routine names.

Size Limitations

XL Pascal programs are subject to the following size limitations:

- Variables are limited to 2**28 bytes
- A maximum of 131072 identifiers is allowed in a program
- A maximum of 32767 routines is allowed
- The maximum length of a constant string is 32767 bytes
- A maximum of 255 **%INCLUDE** directives is allowed
- A maximum of 10 levels of **%INCLUDE** nesting is allowed

Differences between XL Pascal and the VS Pascal Release 1 Licensed Program

Additions

The following features of XL Pascal are additions to VS Pascal Release 1.

- VS mode and standard mode correspond to VS Pascal language level **EXTENDED** and **STANDARD** respectively.
- A new predefined file, **STDERR**, exists in VS mode. This file is connected to standard error, file descriptor two.
- The *caret* (^) as a pointer symbol is new to VS mode and standard mode.
- Because the ASCII character set has no *logical not* (¬) symbol, the functions assigned to that character in XL Pascal are assigned to the *tilde* (~).
- Several new predefined constants are defined for VS mode:
  - **MAXCHAR**
  - **MINSREAL**
  - **MAXSREAL**
  - **EPSREAL**
  - **EPSSREAL**
- XL Pascal supports an alternative method for specifying integer constants:

  ```
  base # digit[digit].
  ```
- The plus sign (+) can be used as a concatenation operator.
- XL Pascal supports multibyte (MBCS) character data or Extended character data, including:
  - Data type **GCHAR** and data type **GSTRING**
  - Mixed string support routines
  - Application of the string functions to the data type **GCHAR** and data type **GSTRING**

- The **NONPASCAL** function/procedure directive is an extension to VS Pascal release 1 and is also part of VS Pascal Release 2.
- The **DISP**=**MOD** option in the open string provides an additional function not available in VS Pascal but which was available in RT VS Pascal using the format documented here.

Omissions

The following features of VS Pascal are omitted in XL Pascal.

- The **LINECOUNT** compiler option is not supported. Pagination can be done using a filter, such as **pr**, after the listing file has been generated.
- The **CMS** runtime routine is not supported since XL Pascal has all of the system facilities you need to issue system calls.
- The **PDSIN** and **PDSOUT** procedures are not supported, since the concept of a partitioned data set does not exist on the AIX Operating System for IBM RISC System/6000.
- The following runtime options are not supported:

  COUNT
  DEBUG
  HEAP=*n*
  NOSPIE
  ONERROR
  SETMEM
  STACK=*n*

- The following open options are not supported:

  ASIS
  BLKSIZE=*n*
  MEMBER=*name*
  NOCC

- When you use the **LRECL** open option, the 4-byte length descriptor is not required for variable length files.

Implementation Definitions

The following implementation definitions are different from those on VS Pascal and may affect some programs. They are basically rules used by the XL Pascal compiler but not defined for the language and thus are implementation defined.

- The size of an identifier or literal is limited to 256 characters rather than 100 as supported by VS Pascal.
- A packed array of characters may not be assigned to a larger packed array, as supported by VS Pascal.
- Variables are mapped to storage differently from VS Pascal in order to achieve proper results on RISC System/6000 hardware without affecting performance. These differences fall into two classes:

Internal data representation:

- Characters are coded in ASCII rather than EBCDIC
- Real values is represented in RISC System/6000 floating-point format rather than System/370* floating point

Storage requirements and relative positioning of variables:
The following minor differences exist between XL Pascal and VS Pascal:

- All XL Pascal pointers require 8 bytes, unless **–qptr4** is specified, in which case they only require 4 bytes. All VS Pascal pointers require only 4 bytes.
- All variants at a given level of an XL Pascal record begin at the same offset from the start of a record. In VS Pascal the offset for each variant is determined by its alignment requirement.

- You cannot pass a **CONST** parameter to **DISPOSE**.
- XL Pascal detects and handles errors differently from VS Pascal:
  - The precise timing of error detection varies. Some errors detected at run time by one compiler may be caught at compile time by the other.
  - The presentation and recovery of runtime errors is different. The details of this support are described in Chapter 6, “Problem Determination”.
- **GCHAR** and **GSTRING** are not part of VS Pascal Release 1. The definition supported by XL Pascal is consistent with that of VS Pascal Release 2.
- Because of differences in the underlying operating systems, XL Pascal does not restrict the modification of **DEF** and **REF** data when a program is operated in an environment where multiple processes use the same program.
- XL Pascal accepts the directives **MAIN** and **REENTRANT** wherever VS Pascal accepts them. They have no any semantic affect on the program, nor do they impose any restrictions on your programs.

Differences between XL Pascal and the VS Pascal Release 2 Licensed Program

XL Pascal incorporates most of the features added to VS Pascal in Release 2. Several features of VS Pascal Release 2 are either not implemented or implemented differently in XL Pascal. The VS Pascal implementation of these features is described in *VS Pascal Language Reference Release 2*.

Differences

The following VS Pascal Release 2 features are implemented differently in XL Pascal.

- XL Pascal supports Multibyte Character Set (MBCS) data, including the **GCHAR** and **GSTRING** types and functions that process **STRING** types containing mixed single-byte and multibyte character strings. XL Pascal uses AIX MBCS characters instead of the System/370 method of shift-in and shift-out characters used in Release 2 of VS Pascal.
- XL Pascal supports AIX debugging facilities through the **DBG** and **–qdbg** compiler options. VS Pascal Release 2 uses its **DEBUG** option to support System/370 debugging facilities.
- XL Pascal implements traceback facilities through the predefined routine **xl__trap**. It does not implement the **ONERROR** facilities used by VS Pascal Release 2.

Omissions

The following VS Pascal Release 2 features are not implemented in XL Pascal.

- XL Pascal does not implement the **GENERIC** procedure directive. That feature was added to VS Pascal Release 2 only to provide an interface to certain System/370 software products. XL Pascal uses the **NOEXTCHK** compiler option and the linkage editor **rename** command to support calls to one polymorphic routine with several actual parameter specifications.
- XL Pascal does not implement the conditional compilation facilities provided in VS Pascal Release 2 through **%SELECT**, **%WHEN**, and **%ENDSELECT**.
- XL Pascal does not implement the **HEADER** compiler option or the **%UHEADER** directive.

Appendix E. Data Storage

This appendix describes how the different types of storage classes and data values in XL Pascal are stored. It also describes how dynamic storage is managed.

Storage Classes

Storage classes determine where and when storage for a value is allocated, when it is initialized, and when it is deallocated.

Automatic Variables

Automatic variables are declared in the **VAR** section of procedures or functions. Automatic storage is not initialized. A runtime stack frame is created for each procedure or function invocation that has not yet returned to its caller. Storage for automatic variables is allocated in the stack frame of the routine invocation. It is freed when the routine call returns and its stack frame is freed.

Global Automatic Variables

Global automatic variables are declared by **VAR** declarations at the outermost level of a program or segment, not inside any procedure or function. Global automatic storage is not initialized. A runtime stack frame is created for the invocation of the top level program. Storage for global automatic variables is allocated in the stack frame of the program. It remains in existence while the program is running.

The linkage editor maps global automatic variables of a segment into the same storage as those of the program. XL Pascal performs no type checking of global automatic storage.

Static Variables

Static variables are declared by **STATIC** declarations. Storage for them is allocated in the static section of the object code file for the program or segment in which the **STATIC** declaration appears. The names used in this section are not known outside the scope where the variable is declared.

You can initialize static variables with **VALUE** statements. Initial values are contained in the object code file and are loaded with the object code. If you do not use **VALUE** to supply initial values, the values of such variables are not initialized. All references to a given variable address the same storage and get the last value set to that variable.

External Variables

External variables are declared in **DEF** declarations or referenced through **REF** declarations. Storage for external variables is allocated for the first **DEF** declaration of the variable as a separate named section of the object code file. All **REF** declarations and subsequent **DEF** declarations of the variable use the same storage. It remains in existence while the program is running.

You can initialize external variables with **VALUE** statements. Initial values are contained in the object code file and are loaded into the external variable when the object code file is loaded.

© Copyright IBM Corp. 1990, 1993

Any object code file can specify an initial value for any component of an external variable. If two or more object code files specify initial values for any components of an external variable, only the initial values from the first such object code file are used.

Constant Storage

Storage for constants declared by **CONST** declarations is allocated in the constant section of the object code file for the program or segment in which the **CONST** declaration appears.

Data Size and Boundary Alignment

A variable is assigned storage and aligned according to its declared type. In general, a halfword value, which occupies 16 bits, is aligned on a halfword boundary. Values larger than a halfword are aligned on a fullword boundary. A fullword value occupies 32 bits. Values that can fit into a single byte of 8 bits are aligned on a byte boundary.

Whatever the size of the data element in question, the most significant data bit is always in the first byte of the total number of bytes needed to represent that object.

This section describes the data size and boundary alignments requirements for storing different types of XL Pascal values.

The Predefined Data Types

The following table shows the storage requirements and boundary alignments of variables declared with a predefined type.

Data Type	Size in Bytes	Alignment
ALFA	8	Byte
ALPHA	16	Byte
BOOLEAN	1	Byte
CHAR	1	Byte
GCHAR	2	Halfword
GSTRING(length)	length*2+2	Halfword
INTEGER	4	Fullword
SHORTREAL	4	Fullword
REAL	8	Doubleword
STRING(length)	length+2	Halfword
STRINGPTR	8	Fullword
TEXT	144	Fullword

Enumerated Data Types

An enumerated variable with 256 or fewer possible distinct values occupies 1 byte and is aligned on a byte boundary. One defining more than 256 values occupies a halfword and is aligned on a halfword boundary. Whether or not the enumerated variable is packed, XL Pascal and VS Pascal both use the minimum amount of space for enumerated data.

Subrange Scalar Data Types

A subrange scalar data type not specified as packed is mapped exactly the same way as the scalar type on which it is based.

For packed subranges, XL Pascal assigns the smallest number of bytes required to represent a value of the scalar type on which it is based.

Note: Using packed subranges may slow the run time of a program. In particular, a packed subrange whose size is 3 bytes may incur a runtime penalty. For subranges that are not packed, the subrange variable is the same as a variable of the base type of the subrange.

The following table defines the number of bytes required for different ranges of integers, given the type definition `t` as

```
TYPE
   t = PACKED i..j;
```

For ranges other than those listed, use the first range that encloses the desired range. For example, the range 100..250 would be mapped the same way as the range 0..255.

Storage Mapping of Packed Subrange Scalars:

Range of i..j	Size in Bytes	Alignment
0..255	1	Byte
-128..127	1	Byte
-32768..32767	2	Halfword
0..65535	2	Halfword
-8388608..8388607	3	Byte
0..16777215	3	Byte
Otherwise	4	Fullword

Record Data Types

The alignment of a record is the strictest alignment requirement of its component fields. The alignment of a record field is determined from its type. For byte, halfword, or word alignment, the alignment of the field is the same as the alignment required by the type of the field. A field type that requires doubleword alignment gets word alignment.

The first field of a record starts with the alignment determined for the record type. In records that are not packed, padding bytes are inserted between fields so that each succeeding field satisfies its alignment requirement, unless specific offsets are specified. No padding is done for packed records; each succeeding field begins in the byte following the previous field.

The alignment of a variant part is the strictest alignment requirement of the fields in the variant part. All variants have the same alignment as the variant part, and the first fields of all variants start at the same byte. Padding bytes are inserted before the variant part of records that are not packed to satisfy their alignment requirements.

If the element type is a named record or an array type that was previously declared, the format of such an element is determined by its packing characteristics.

Examples

The following example shows alignment of records. The **REAL** field `float` gets word alignment instead of doubleword alignment because it is a record field. The strongest alignment in record `M` is word, so `M` is word aligned. The first field `ch1` starts with word alignment. Three padding bytes are inserted after field `ch1` to put field named `tag` at offset 4 with word alignment. The variant part has word alignment, so three padding bytes are inserted after field `ch2`. All of the variants start at offset 12.

```
M : RECORD          (* word aligned *)
       ch1 : CHAR ;     (* offset 0      *)
       tag : INTEGER ; (* offset 4, integer requires word*)
       ch2 : CHAR ;     (* offset 8     *)
       CASE TAG: OF
          1: ( int1 : INTEGER );(* offset 12  *)
          2: ( ch3 : CHAR  );  (* offset 12   *)
          3: ( float: REAL  ); (* offset 12, word aligned *)
    END ;
```

The following example shows alignment of previously declared records. Record `r1` is not packed and has padding supplied between `r1c` and `r1i`. Record `r2` is packed, so no alignment padding is supplied between its elements. The padding remains within the two fields of type `r1`, but none exists between the fields of `r2`.

```
TYPE
   r1 = RECORD
           r1c : CHAR ;
           r1i : INTEGER ;
        END ;

   r2 = PACKED RECORD
           r2c  : CHAR ;         (* offset 0 *)
           r2ra : r1 ;           (* offset 1 *)
           r2rb : r1 ;           (* offset 9 *)
        END ;
```

In the following example, the contained records are declared in place, so they take on the packed attribute from the declaring structure.

```
r3 = PACKED RECORD
        r3c   : CHAR ;        (* offset 0 *)
        r3ra  : RECORD        (* offset 1 *)
                   r3rac : CHAR ;     (* offset 1 *)
                   r3rai : INTEGER ;  (* offset 2 *)
                END ;
        r3rb  : RECORD        (* offset 6 *)
                   r3rbc : CHAR ;     (* offset 6 *)
                   r3rbi : INTEGER ;  (* offset 7 *)
                END ;
     END ;
```

ARRAY Data Types

In unpacked arrays, each element is aligned on the next available proper boundary for the element type. Any padding between elements is also repeated after the last element.

For example, consider the following type definition:

```
TYPE
   a = ARRAY [ s ] OF t
```

where type `s` is a simple scalar and `t` is any type.

A variable declared with this type definition would be aligned on the boundary required for type `t`. The amount of storage occupied by this variable is given by the following expression for any type *t* except an unpacked record type:

```
( ORD( HIGHEST( s ) )-ORD( LOWEST( s ) ) + 1 ) * SIZEOF( t )
```

Padding is added to unpacked record types as required between elements, so that each element is aligned on a boundary that meets the requirements of the record type.

In packed arrays, elements are adjacent with no padding between elements.

Note: This may result in unaligned data items within the array, which could slow processing.

A multidimensional array is mapped as an array of arrays. For example, the following two array definitions would be mapped identically in storage:

```
ARRAY [ i..j, m..n ] OF t
```

and

```
ARRAY [ i..j ] OF
   ARRAY [ m..n ] OF t
```

If the element type is a named record or array type that was previously declared, the format of such an element is determined by its packing characteristics.

FILE Data Types

All file variables require 148 bytes, and are aligned to a fullword boundary.

Pointer Data Types

All pointers, including **STRINGPTR** and **GSTRINGPTR**, require 8 bytes and are aligned on a word boundary.

SET Data Types

The **SET** data types are represented internally as a string of bits with one bit position for each integer value from 0 to the largest ordinal value of an element of the base type of the set. For example, a **SET** type of the form

SET of *a*..*b*

is represented by a bit string that includes one bit for each integer value from 0 to **ORD**(*b*). If some value *c* is a member of the set, the bit in position **ORD**(*c*) is set to 1; otherwise, the bit is set to 0.

The length of the bit string representing the set depends on the base type of the set and whether the set is packed.

Unpacked Sets of Integer Subranges

An unpacked set over an integer subrange is represented by a string of 256 bits occupying 32 bytes with full word alignment. This string has bit positions representing all potential ordinal values from 0 to 255 although some of the bit positions may not be needed.

For example, a set definition

```
SET of 100..200
```

is represented by 32 bytes containing 256 bit positions for the potential ordinal values 0 to 255, but bit positions 0 through 99 and 201 through 255 are not used.

In the following example, all of the sets occupy 256 bits:

```
VAR
   a : SET of CHAR;
   b : SET of '0'..'9';
   c : SET of 0..255;
   d : SET of 10..20;
```

Packed Sets and Sets of Enumerated Types

Packed sets, and sets with an enumerated base type or a base type that is a subrange of an enumerated type occupy the minimum number of bytes needed to allocate one bit for each integer value from 0 to the largest ordinal value of an element of the base type of the set.

The representation of a packed set or a set of enumerated values is determined by *m*, where

```
m := ORD ( HIGHEST (base-type of the SET) );
```

The following table shows the size and alignment of a packed set of *base–type* as a function of `m`.

Storage mapping of a Packed Set:

Range of m	Size in Bytes	Alignment
0 <= m <= 7	1	Byte
8 <= m <= 15	2	Halfword
16 <= m <= 23	3	Byte
24 <= m <= 31	4	Fullword
32 <= m <= 255	(m+7) DIV 8	Byte

A set definition

```
PACKED SET of 100..200
```

requires bit positions for each value from 0 to 200. This set is represented by 26 bytes containing 208 bit positions for the potential ordinal values 0 to 207, but bit positions 0 through 99 and 201 through 207 are not used. The following example shows the storage mapping of other packed sets:

```
VAR
   e : PACKED SET of c1..c8;         (* 8 bits  *)
   f : PACKED SET of 10..20;         (* 24 bits *)
   g : PACKED SET of 0..31;          (* 32 bits *)
```

The set definition

```
SET OF BOOLEAN
```

is represented by one byte containing 8 bit positions, but only bit positions 0 (**FALSE**) and 1 (**TRUE**) are used. The following example shows the storage mapping of other sets with an enumerated base type:

```
h : SET of (c1,c2,c3,c4,c5,c6,c7, (* 16 bits *)
      c8,c9,c10,c11,c12,c13,
      c14,c15,c16);
i : SET of c1..c8;                (* 16 bits *)
```

SPACE Data Types

A variable declared as **SPACE** is aligned on a word boundary and occupies the number of bytes indicated in the length specifier of the type definition. For example, the variable `s` in the following declaration occupies 1000 bytes of storage.

```
VAR s : SPACE [1000] of INTEGER;
```

String Types

XL Pascal string data types are **STRING**, **PACKED ARRAY OF CHAR**, **GSTRING**, and **PACKED ARRAY OF GCHAR**.

Packed arrays of **CHAR** are fixed-length character arrays stored as a set of contiguous bytes, one character per byte. All of the elements in the array are character positions in the string, and the size of the array is the length of the string.

The **STRING** data type is a varying-length string that has a 2-byte packed subrange string length aligned on halfword boundary followed by a fixed-length array large enough for the maximum length of the string. The dynamic length of the string can be determined using the **LENGTH** function.

Packed arrays of **GCHAR** and variables of type **GSTRING** consist of multibyte characters.

Anonymous Types

Data types declared without a type name are said to be *anonymous*. A field of a packed record type that is an anonymous subrecord is implicitly packed. An anonymous array type that is a field of packed record is not implicitly packed.

Examples

```
TYPE
   rt1 = RECORD ... END;
   t1 = ARRAY[1..5] of rt1;
   t2 = PACKED ARRAY[1..5] of rt1;
   r1 = PACKED RECORD
           f1 : rt1;
           f2 : t1;      (* Not packed                     *)
           f3 : t2;      (* Packed because t2 declared packed *)
           f4 : ARRAY[1..5] of rt1;    (* Not packed         *)
           f5 : RECORD ... END;        (* PACKED -- a ”new”   *)
                                       (* anonymous field     *)
                     (* type of a packed record is packed *)
        END;
```

Dynamic Storage Management

This section describes how the Standard Pascal **NEW** and **DISPOSE** routines are implemented in XL Pascal. It also describes the AIX RISC System/6000 implementation of the following VS Pascal routines:

- **MARK**
- **RELEASE**
- **NEWHEAP**
- **QUERYHEAP**
- **USEHEAP**
- **DISPOSEHEAP**

Implementation of NEW and DISPOSE by malloc and free

XL Pascal uses the AIX subroutine **malloc** to implement the Pascal predefined **NEW** routine, and the AIX subroutine **free** to implement the **DISPOSE** routine.

NEW and malloc

A call to **NEW** is compiled into a call to a runtime environment routine that calls **malloc** to allocate a block of storage. This block contains both the Pascal dynamic variable and a fixed-size *dynamic block header*. The dynamic block header contains the size of the storage area.

Calling **NEW** with a list of case constants specifying tag values allocates to the dynamic variable just enough storage for the variant specified by each such case constant. For any variant part that does not have a case constant in the **NEW** invocation, **NEW** allocates enough storage for the largest variant. The case constants given in the **NEW** invocation are used only to determine the size of storage to be allocated by **malloc**. The case constant values are not saved.

DISPOSE and free

A call to **DISPOSE** is compiled into a call to a runtime environment routine that calls **free** to deallocate the block of storage containing the dynamic variable that was passed to **DISPOSE**. Any case constant values passed to **DISPOSE** are ignored. **DISPOSE** always frees exactly the storage allocated by **NEW**.

Heaps

A *heap* is a subset of the Pascal dynamic variables allocated by **NEW** but not yet freed by **DISPOSE**. A heap is implemented by XL Pascal as a doubly-linked list of the dynamically allocated blocks containing the Pascal dynamic variables in the heap. The dynamic block header contains the links and is located in a head cell for the heap.

The *active* heap is initially a default heap implemented by a default head cell. **NEW** allocates a new dynamic variable and links its dynamic block at the right-hand end of the currently active heap. The XL Pascal multiple heap management routines are implemented the following way:

NEWHEAP creates a new heap head cell with an empty list of dynamic blocks.

USEHEAP changes the currently active heap.

QUERYHEAP locates the head cell of the currently active heap.

DISPOSEHEAP frees all of the dynamic variables in a heap by freeing all of the dynamic blocks accessible to the heap head cell. If the argument to **DISPOSEHEAP** is the currently active heap, the default heap becomes active again.

Subheaps

A *subheap* is a subset of the dynamic variables in a heap. Subheaps are implemented by keeping a *mark level counter* in the dynamic header. This counter gives the number of **MARK** operations that have not been released. The head cell of a heap has a mark level of zero.

The predefined Pascal procedures **MARK** and **RELEASE** operate on subheaps and are implemented in the following way:

MARK creates an empty dynamic variable called a *mark block* and increments the mark level in the mark block. A call to **NEW** allocates a dynamic variable with the same mark level as the previous right-hand end of the currently active heap.

RELEASE starts at a mark block and frees all of the dynamic blocks from that mark block to the right up to, but not including, the heap head cell. **RELEASE** and **DISPOSEHEAP** both call the same runtime environment routine to free the dynamic variables you specify. Whereas **DISPOSEHEAP** frees entire heaps including the head cell, **RELEASE** frees subheaps but does not include the head cell.

DISPOSE, DISPOSEHEAP, and RELEASE Validity Checking

The Run Time Environment routines that support **DISPOSE**, **DISPOSEHEAP**, and **RELEASE** clear the link fields in the dynamic headers of dynamic blocks as they are freed. These routines also check for local integrity of the list representing the heap involved in the operation. This checking in general detects **DISPOSE**, **DISPOSEHEAP**, and **RELEASE** operations on inappropriate operands.

Validity checking is subject to the following conditions:

- For all three operations, the dynamic block involved must be linked to apparently valid dynamic blocks to the right and left
- For **DISPOSE**, the mark level must be the same as that of the block to the left, and the disposed block must not be a head cell
- For **DISPOSEHEAP**, the block must be a head cell
- For **RELEASE**, the mark level must increase by one from the dynamic block to the left

Related Information

See the *AIX Version 3.2 Technical Reference: Base Operating System and Extensions* manual for a description of the **malloc** subcommand and the **free** subcommand. See the *Language Reference for IBM AIX XL Pascal Compiler/6000* for a description of the XL Pascal routines that handle dynamic variables.

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 that are necessary for this error. The compiler inserts two no-op **lrfl** instructions between the nondependent floating-point operation and the dependent **frsp**. This 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 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 **OPTIMIZE** or **OPT**=2 compiler option.

© Copyright IBM Corp. 1990, 1993

Glossary

This is a glossary of commonly used terms in the *User's Guide for IBM AIX XL Pascal Compiler/6000*, and the *Language Reference for IBM AIX XL Pascal Compiler/6000*. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*, SC20–1699. It supplements the *AIX Version 3.2 Topic Index and Glossary*.

A

actual parameter
The actual value passed to a routine. Contrast with *formal parameter*.

allocate
To reserve a resource for use in performing a specific task. Contrast with *deallocate*.

alphabetic character
Any one of the uppercase letters A through Z, lowercase letters a through z, and the special characters $ and _.

alphanumeric
Pertaining to a character set containing 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 communications 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.

anonymous type
A data type defined without a type name.

ANSI
See *American National Standards Institute*.

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; 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 XL Pascal to perform an arithmetic operation..

array element
A single data item in an array.

assignment compatible
Indicates whether the type of a value allows it to be assigned to a variable. See also *compatible types*.

automatic variable
A variable allocated on entry to a routine and deallocated on the subsequent return. An automatic variable is declared with the **VAR** declaration. Contrast with *static variable*.

B

base scalar type
The data type on which another type is based. See also *data type*.

binary operator
An operator that represents an operation on exactly two operands. Compare with *unary operator*.

binder
See *linkage editor*.

C

CASE label
A value or range of values that comes before a statement in a **CASE** statement branch. When the selector is evaluated to the value of a **CASE** label, the statement following the **CASE** label is processed.

collating sequence
A specified arrangement for the order of characters in a character set. The collating sequence for AIX RISC System/6000 characters is ASCII.

column mark line
In an error message, the line that contains the symbol _|_ to indicate the column of code where the error was detected.

compatible types
Different data types that can be operands for the same operation.

© Copyright IBM Corp. 1990, 1993

compilable unit
An independently compilable piece of code. There are two types of compilable units in XL Pascal: the program unit and the segment unit.

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

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

component
One part of a structured type or value, such as an array element or record field.

conformant string
A string whose declared length does not match that of a formal parameter.

constant
(1) An unvarying quantity. (2) A value that is either a literal or an identifier associated with a value in a **CONST** declaration.

constant expression
An expression that can be completely evaluated by the compiler at compile time and used as a constant value.

constant folding
Performing operations whose operands are all constants at compilation time, and treating the results as constants.

control statement
A statement that alters the continuous sequential invocation of statements. It may be a conditional statement, such as **IF**, or an imperative statement, such as **DO**.

current heap
The area of storage where dynamic variables allocated by calls to **NEW** reside. Other heaps can exist at the same time, but only one is current.

D

dangling else
A condition arising as a result of nesting an **IF** statement in the **IF** part of an **IF**–**ELSE** statement. The **ELSE** is associated with the closest **IF**, in this case, the inner one. Placing an empty **ELSE** statement in the nested statement prevents misinterpretation by forcing the outer **ELSE** to associate with the outer **IF**.

data type
(1) The properties and internal representation that characterize data and functions. (2) One of several Pascal data classes that defines the permissible values a variable belonging to it may assume.

DBCS
See *Double-Byte Character Set*.

DDNAME
A name that can be used to determine the external name of the date file associated with a file variable.

deallocate
To release a resource assigned to a specific task. Contrast with *allocate*.

default
A value, attribute, or option that is assumed when no alternative is specified.

dereferenced pointer
An expression using –> or @ operator to locate a dynamic variable from a pointer.

descriptor
In information retrieval, a parameter word used to categorize or index information.

digit
A graphic character that represents an integer. For example, one of the characters 0 through 9.

directory
A type of file containing the names and controlling information for other files or other directories.

Double-Byte Character Set (DBCS)
A set of characters in which each character is represented by 2 bytes of storage.

dynamic block header
A data structure used by XL Pascal to link dynamic variables that are in the same heap.

dynamic string
See *string.*

dynamic variable
A variable allocated under programmer control. Explicit allocations and deallocations are required; the predefined procedures **NEW** and **DISPOSE** are provided for this purpose.

E

element
The component of an array, subrange, enumeration or set. The data type of the element is the *element type*.

embedded blanks
Blanks surrounded by any other characters.

enumerated scalar type
A scalar defined by enumerating the elements of the type. Each element is represented by an identifier.

expression
A notation that represents a value; a constant or a reference appearing alone, or in combinations of constants and/or references with operators. AN expression can be arithmetic , character, logical, or relational.

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

external variable
A variable accessible in another compilation unit.

F

field
A component of a record.

file
A sequence of records stored and processed as a unit. Files located in internal storage are internal files; files on an input/output device are external files.

file pointer
An identifier indicating the location of an item of data in an input/output buffer.

fixed part (of a record)
The part of a record common to all instances of a particular record type.

fold
To translate the lowercase characters of a character string into uppercase or vice versa. See also *constant folding*.

formal parameter
A parameter declared in a routine heading. It specifies what can be passed to a routine as an actual parameter.

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.

function
(1) A named expression that calculates a single value. (2) A routine called by coding its name on the right side of an expression. The routine passes a result back to the caller through the routine name.

G

guard expressions
Expressions placed at the beginning of Boolean expressions to check that other operations can be done.

H

hashing
(1) A method of transforming a search key into an address for the purpose of storing and retrieving items of data. (2) Encoding a character string as a fixed-length bit string for comparison. The encoding may not be unique.

hash signature
The fixed-bit character string resulting from hashing a character string. Character strings can be compared quickly by comparing their hash signatures.

heap
A collection of dynamically allocated variables.

hexadecimal
Pertaining to a system of numbers to the base 16; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

I

identifier
(1) The name of a declared item. (2) A lexical unit that names a language object, for example, the names of variables, arrays, records, labels, and procedures.

IEEE
Institute of Electrical and Electronics Engineers.

index
The selection mechanism applied to an array to identify an element of the array.

input
(1) Data to be processed. (2) A predefined standard file definition.

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

interactive
Pertaining to the exchange of information between people and a computer.

internal routine
A routine functioning only within the lexical scope in which it was declared.

invocation stack
A list of programs linked together as a result of programs calling other programs with the **CALL** instruction, or implicitly from some other event, within the same job. Also known as *program stack*.

I/O
See *input/output*.

K

keyword
A predefined identifier representing a language construct that can be redefined in a declaration. Contrast with *reserved word*.

L

lexical level
The depth to which routines are nested within one another, which determines the scope of the identifiers declared within those routines.

lexical scope
The portion of a program or segment unit in which a declaration applies. An identifier declared in a routine is known within that routine and all nested routines. If a nested routine declares an item with the same name, the outer item is not available in the nested routine.

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

linkage
The coding that passes control and parameters between two routines.

linkage editor
A computer program for creating load modules from one or more object modules or load modules by resolving cross-references among the modules and, if necessary, adjusting addresses. It may also check consistency of data references among the modules.

literal string
A character string whose value is given by the characters themselves.

logical file
A description of how data is to be presented to or received from a program. This type of data base file contains no data, but it defines formats for one or more physical files.

M

mark block
A dynamic block header that designates a subheap within a heap.

MBCS
See *Multibyte Character Set*.

mixed string
A string consisting of a mixture of MBCS characters and single-byte characters.

Multibyte Character Set (MBCS)
A set of characters in which each character is represented by more than one byte of storage. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be expressed in a single byte, are represented in this character set.

N

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), or to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

O

object
Anything that exists in, and occupies space in, storage and on which operations can be performed, such as, programs, files, and libraries.

object program
A set of instructions in machine-runnable form. The object program is produced by a compiler from a *source program*.

offset
(1) The distance from the beginning of an object to the beginning of a particular field. (2) The selection mechanism in the **SPACE** data type, an element of which is selected by placing an integer value in brackets. The origin of a **SPACE** is zero.

operand
A value manipulated by an operator.

operator
A token that specifies the algebraic or logical processes that can be performed on one or more values.

optimize
To improve the speed of a program or reduce the use of storage during processing.

ordinal type
A type whose members can be counted to indicate position.

output
(1) The result of processing data. (2) A predefined standard file definition.

P

pad
(1) To fill parts of a character string with a specified pattern. (2) To leave unused bytes between components of a structured type in order to satisfy alignment requirements.

parameter
A value specified to a command, program, or routine. The value is either used as input or controls the actions of the command, program, or routine.

pass-by-CONST
The parameter passing mechanism by which a copy of the variable is passed to the called routine. The called routine is not permitted to modify the formal parameter.

pass-by-read-only-reference
Synonym for *pass-by-CONST*.

pass-by-read/write-reference
Synonym for *pass-by-VAR*.

pass-by-value
The parameter passing mechanism by which a copy of the value of the actual parameter is passed to the called routine. Called routines that modify the formal parameter do not affect the corresponding actual parameter.

pass-by-VAR
The parameter passing mechanism by which the address of a variable is passed to the called routine. Called routines that modify the formal parameter do not affect the corresponding actual parameter.

pointer type
Defines variables containing addresses and other information about dynamic variables.

prime file
A file containing precompiled declarations in the internal table format of the XL Pascal compiler. Prime files are used to initialize the compiler's internal tables before compilation begins.

procedure
A routine, called by coding its name as a statement, that does not pass a result back to the caller.

program unit
The name of a compilable block of code that gains initial control when the compiled program is called.

R

real number
A number that contains a decimal point and is stored in fixed-point or floating-point format.

recursive routine
A routine that can call itself or be called by another routine called by the recursive routine.

register
A storage device having a specified capacity such as bit, byte, or computer word, and usually intended for a special purpose.

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

relational operator
Any of the set of operators that compare values.

reserved word
An identifier that is predefined in the language and syntax rules of XL Pascal and that cannot be redefined.

result
An entity produced by the performance of an operation.

routine
A block of statements called and processed as a unit. The two types of routines are functions and procedures. See also *function* and *procedure*.

S

scalar
Pertaining to a single data item.

scalar type
A type that defines a variable containing a single value at run time. **CHAR**, **BOOLEAN**, **INTEGER**, **REAL**, **SHORTREAL**, enumerated types, and subranges are scalar types. Contrast with *structured type*.

scope
See *lexical scope*.

segment unit
An independently compilable unit of XL Pascal code containing routines linked with the program unit. See also *program unit*.

selector
The term in a **CASE** statement that, once evaluated, determines which of the possible branches of the **CASE** statement are processed.

semantic error
A compile-time error caused by incorrect definition of constants and identifiers. See also *syntax error*.

short circuiting
The evaluation of Boolean expressions with **AND** (&) and **OR** (|) such that the right operand is not evaluated if the result of the operation can be determined by evaluating the left operand. The evaluation of the expression is always from left to right.

side effect
An undesirable result caused when a function or procedure alters the values of nonlocal variables.

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

spill area
A storage area used to save the contents of registers.

statement
A language construct that represents a step in a sequence of actions or a set of declarations.

static variable
A variable that is allocated as soon as a program starts running and remains allocated until the program stops. Contrast with *automatic variable*.

string
(1) A sequence of characters. the length of the sequence can vary at run time. Synonymous with *dynamic string*.
(2) An object of the predefined type **STRING**.

structured programming
A technique for organizing computer programs in hierarchical modules, making programs easier to debug, modify, and replace. Typically, all modules have a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the structure.

structured type
Any of several data types that define variables having multiple values; for example, records and arrays. Each value is generally referred to as a component. See also *component*. Contrast with *scalar type*.

subheap
Part of a heap delimited by a call to **MARK**. Subheaps are treated in a stack-like manner within a heap.

subrange scalar type
A type that defines a variable whose value is restricted to some subset of values of a base scalar type.

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

syntax error
A compile-time error caused by incorrect punctuation or misuse of reserved words. See also *semantic error*.

T

tag field
The field of a record that defines the structure of the variant part. See also *variant part*.

top-down
An approach to problem solving that starts at the highest level of abstraction and proceeds toward the lowest level.

type
See *data type*.

type compatibility
See *compatible types*.

type definition
The specification of a data type. The specification can be in a type declaration or in the declaration of a variable.

type identifier
The name given to a declared type.

U

unary operator
An operator that represents an operation on one operand only. Compare with *binary operator*.

unit
See *compatible unit*.

V

variant part
The portion of a record that can vary from one instance of the record to another. The variant portion consists of alternate sequences of fields that share the same physical storage.

Index

Symbols

.cshrc file, 15

.lst files, 20

.o files
  –c compiler option, 29
  invoking, 14, 40
  output files, 20

.pas files
  input files, 19
  invoking, 14, 40

.profile file, 15

.s files, 20

%CHECK directive, 32, 84

%INCLUDE directive, 21, 119

%OPTION directive, 21, 22

–# compiler option, 29

–B flag, 29

–c compiler option
  description, 29
  invoking the linkage editor, 40
  xlp command, 13

–F flag, 29

–g compiler option, 32, 85

–I flag, 30

–O compiler option, 37

–o flag, 20

–p compiler option, 16, 30

–pg compiler option, 16, 30

–q command line options, 23

–qlanglvl compiler option, 18

–qprime compiler option, 21

–qprimeout compiler option, 21

–qsource compiler option, 105

–S option, 30

–t flag, 30

–U compiler option, 37

–v compiler option, 30

–W flag, 31

–y flag, 36

/etc/profile file, 15

A

a.out file
  –o flag, 30
  invoking the compiler, 13
  output file, 20

active heap, 130

AIX
  commands
    –c compiler option, 29
    as, 16
    dbx, 9
    export, 44
    gprof, 30
    installp, 10
    ld, 16, 28, 40
    ln, 17
    prof, 30
  shared libraries, 42
  subroutines, 130

AIX Version 3 for RISC System/6000. *See* AIX

algorithm, 69

alignment, boundary. *See* boundary alignment

American National Standard Code for Information Interchange. *See* ASCII character set

American National Standards Institute (ANSI), 1

anchor–pointing. *See* short–circuiting

anonymous types, storage and alignment, 129

ANSI standard Pascal
  AIX RISC System/6000 extensions, 112
  compiler modes, 10
  industry standard, 5
  LANGLVL compiler option, 36
  language support, 9
  publication, 6
  VS mode extensions, 111
  XL Pascal compiler compliance
    error handling, 114–116
    extension handling, 116
    implementation–defined features, 111–113
    implementation–dependent features, 113

ANSI–83. *See* ANSI standard Pascal

appending data to a file, 66

ARCH compiler option, 31

array
  element, 66
  storage, 93

© Copyright IBM Corp. 1990, 1993

ARRAY data types, 126

arrays, 101

as configuration file attribute, 16

ASCII character set, 107

asopt configuration file attribute, 16

assembler source files, input to xlp command, 13, 20

assembler, source files, suffix, 16

assertion failure, 84

associating file name, 66

ATTR compiler option, 31

attribute section, compiler listing, 77

automatic variables, storage, 123

B

Boolean, expressions, 71

BOOLEAN data type, 21

boundary alignment, 124–129

Bourne shell, 14

bsh command, 14

C

C language
  array storage, 93
  case sensitivity, 97
  char arrays, 92
  character variable types, 92
  matching data types, 90
  pointers, 93
  routine calls, 94
  strings, 92

C shell, 14

CASE label error, 84

case sensitivity, 90, 97

character strings, 98

character variable types, 92

CHECK compiler option, 32

CLOSE procedure, 65

code motion, 70

COLS function, 60

column–major order, 93

command line
  configuration file attributes, 16
  running a program, 41
  specifying compiler options, 23
  using environment varaibles, 43
  xlp command, 13

commands, 14–17
  See also AIX commands

common expression elimination, 70

COMPACT compiler option, 32

compatibility, 9

compile–time error messages, 79, 80

compiler
  directives
    %CHECK, 32
    %INCLUDE, 21
    %OPTION, 22, 23
    %OPTION PRIME, 21
    %OPTION PRIMEOUT, 21
  error response, 80
  features, 9
  help, 9
  installation, 10
  invoking, 13
  limits, 118, 119
  listings, 73
  mode, 10
  options
    –c, 13, 29
    –p, 16, 30
    –pg, 16, 30
    –qlanglvl, 18
    –qprime, 21
    –qprimeout, 21
    –S, 30
    debugging, 28
    default, 25
    describing compiler output, 27
    detailed descriptions, 29–40
    input to the compiler, 25
    LANGLVL, 10
    linkage editor, 28
    object code, 26
    overview, 22
    SOURCE, 105
    specifying in the source file, 23
    specifying on the command line, 23
    xlp command, 9
  output files, 20

compiling a program, 14

configuration file
  attributes, 16–21
  customizing, 17

CONST parameter, 93

constants
  folding, 35, 70
  predefined, 21, 117
  storage, 124

control characters, 107

conventions
  parameter passing, 97, 101
  routine linkage
    NONPASCAL routine directive, 97
    overview, 95
    parameter linkage, 95
  source code symbols, 10

correcting
  compile–time errors, 81
  runtime checking errors, 84
  semantic errors, 83
  syntax errors, 81

creating matching data types, 91

cross reference section, compiler listing, 77

crt configuration file attribute, 16

csh command, 14

CTRL key, 107

customizing configuration files, 17

D

data file name, 67

data storage, 123

data types
  BOOLEAN, 21
  boundary alignment, 124
  character variables. *See* matching data types
  implementation dependencies
    INTEGER, 117
    REAL, 117
    SET, 118
    SHORTREAL, 117
  matching. *See* matching data types
  storage and alignment
    anonymous types, 129
    ARRAY types, 126
    enumerated types, 124
    FILE, 127
    overview, 124
    pointer types, 127
    predefined types, 124
    record types, 125
    SET, 127
    SPACE, 129
    string, 129
    subrange scalar types, 125

DBCS compiler option, 32

DBG compiler option, 32

dbx
  See also symbolic debugger
  command, 9
  optimized code, 71
  problem determination, 85
  where subcommand, 86

DDNAME
  compiler option, 32, 66
  data file name, 67
  file opening option, 48
  open file variable, 66

dead code elimination, 70

debugging optimized code, 71

DEF declaration, 89

default
  compiler options, 22, 25
  configuration file, 15–19
  DDNAME compiler option, 32
  executable file name, 20, 41
  IEEE compiler option, 36
  input, 52
  LRECL open option, 47
  MARGINS compiler option, 36
  NORRM compiler option, 38
  OPT compiler option, 37
  optimization level, 37
  output, 52
  Pascal source file suffix, 19, 20
  prime file, 21
  RECFM open option, 47
  rounding mode, 36
  SPILL compiler option, 38
  stanza name, 15

descriptor word, 96

detailed descriptions of compiler options, 29–40

diagnostic messages. *See* see error messages

DIALECT compiler option, 36

DISP=MOD file opening option, 47, 66

DISPOSE procedure, 130

DISPOSEHEAP procedure, 130

double precision, intermediate rounding, 36

dynamic
  block header, 130
  storage management, 129–132

E

echo command, 14

end–of–file condition, 60, 64

end–of–line condition, 59

enumerated data types, storage and alignment, 124

environment variables
  data file name, 67
  LANG, 14, 86
  NLSPATH, 14, 86

opening files, 43, 44
runtime environment, 44
setting, 14

EOF function, 60, 64

EOLN function, 59

epilog section, compiler listing, 78

EPSREAL constant, 118

EPSSREAL constant, 118

error
detection, 114–116
runtime, 84

error messages
compile time
classes, 80
compiler response, 80
correcting, 81
description, 79
semantic errors, 83
syntax errors, 81
description, 79–86
FLAG compiler option, 33
HALT compiler option, 35
runtime, 83, 84
using the symbolic debugger, 85

executable files
output from xlp command, 13, 20
running a program, 41

export command, 44

expressions, guard, 71

EXTCHK compiler option, 32

external name, 66, 90

EXTERNAL routines, 41

external variables, 98, 123

F

Federal Information Processing Standard. *See* FIPS

file
.cshrc, 15
.profile, 15
/etc/profile, 15
/etc/xlp.cfg, 15
a.out, 20
configuration
attributes, 16–21
customizing, 17
data types, 104
description, 43
executable, 13
gmon.out, 30
input, 13, 19
mon.out, 30
name
associating with external name, 66
environment–determined, 43
unique, 65, 67
object, 13
open options
DDNAME, 48, 67
DISP=MOD, 47, 66
INTERACTIVE, 47, 50
LRECL, 47, 49
NAME, 46, 67
overview, 45
RECFM, 47, 49
UCASE, 47
opening procedures
overview, 50
RESET, 50
REWRITE, 51, 66
TERMIN, 52
TERMOUT, 52
UPDATE, 53
output, 20
prime, 21
RECORD, 43
source, 13
TEXT, 43
xlp_base_prime, 22
xlp_prime, 21, 22

FILE data types, 127

file table section, compiler listing, 78

FIPS (Federal Information Processing Standard), 5, 6

first parameter word, 96

fixed–length strings, 92

FLAG compiler option, 33

FLOAT compiler option
description, 33
suboptions, 33–36

floating round to single precsion (frsp). *See* frsp instruction

floating–point
arithmetic, compiler option, 36
constant expressions, 35
data implementation, 117
exception, 36
function return values, 35
intermediate rounding, 133
overflow detection, 38
rounding mode, 38

FLTTRAP compiler option
description, 34
suboptions, 34–37

FOLD compiler option, 35

folding, 35, 70

formal parameter
  definitions across XL languages, 97
  interlanguage reference requirements, 89
  routine linkage convention, 95
  type checking, 104

Fortran
  array storage, 93
  character variable types, 92
  matching data types, 90
  routine calls, 94

FPRET compiler option, 35

free subroutine, 130

frsp instruction, 133

function calls, 94

function return value, 84

function return values, 98

G

gcrt configuration file attribute, 16

GET procedure, 53, 61

getopt subroutine, 16

global automatic variables, 123

global register allocation, 70

gprof command, 30

guard expressions, 71

H

HALT compiler option, 35

hash signature, 104

head cell, 130

header section, compiler listing, 76

heaps, 130

help, online, 9

I

IBMSET compiler option, 36

identifiers, in syntax diagrams, 2

IEEE compiler option, 36

implementation
  defined features of XL Pascal, 111
  dependent features of XL Pascal, 113

improving program performance, 71

industry standards, 5

input files, 13, 19

INPUT predefined file, 104

input procedures
  RESET, 50
  TERMIN, 52
  UPDATE, 53

installing the compiler, 10

installp command, 10

instruction scheduling, 70

instructions, multiply–add, 36

INTEGER data type, 117

INTERACTIVE file opening option, 47, 50

interlanguage communication
  array storage, 93
  character variable types, 92
  compiler feature, 9
  external names, 89, 90
  matching data types, 90
  reference requirements, 89
  routine calls, 94

intermediate rounding, 36, 38

International Standards Organization. *See* ISO

invoking, 13, 40

ISO (International Standards Organization), 5, 6

K

keywords, 2, 7

Korn shell, 14

ksh command, 14

L

LANG environment variable, 14

LANGLVL compiler option, 10, 36

language mode, 36

language support, 9

ld
  command
    See also linkage editor
    –c compiler option, 29
    –L flag, 30
    –l flag, 30
    –o flag, 30
    compiler options, 28
    invoking the linkage editor, 40
    rename subcommand, 90
    single letter flags, 24
  configuration file attribute, 16

ldopt configuration file attribute, 16

libraries configuration file attribute, 16

limits, compiler. *See* compiler limits

linkage convention, routine. *See* routine linkage convention

linkage editor
  See also ld command
  configuration file attribute, 16
  data type checking, 104
  invoking, 40
  rename subcommand, 90
  xlp command, 13

linking, static, 41

LIST compiler option, 36

list files, 20

listings
  overview, 73
  sections
    attributes, 77
    cross reference, 77
    epilog, 78
    file table, 78
    header, 76
    object, 78
    options, 77
    source, 77

ln command, 17

LOG compiler option, 36

LRECL file opening option, 47, 49

M

MAF compiler option, 36

making programs more efficient, 71

malloc subroutine, 130

MARGINS compiler options, 36

mark block, 131

MARK procedure, 131

matching data types
  creating, 91
  enforcement of type matching, 104
  file types, 104
  interlanguage communication, 90
  list and summary, 91

MAXMEM compiler option, 36

MAXREAL constant, 118

MAXSREAL constant, 118

MBCS (multibyte character set), 7

MBCS compiler option, 37

mcrt configuration file attribute, 16

message
  catalogs, 86
  classes, 80
  diagnostic. *See* see error messages
  error. *See* error messages

migration, 9

MINREAL constant, 118

MINSREAL constant, 118

MIXED compiler option, 37, 90

mode, 10

multibyte character set. *See* MBCS

multiple heap routines, 130

multiply–add instructions, 36

N

name, data file, 43, 67

NAME file opening option, 46, 67

national language support, 41, 86

NEW procedure, 42, 130

NEWHEAP procedure, 130

NIL pointer error, 84

NLSPATH environment variable, 14

no–check hash signature, 104

NOEXTCHK compiler option, 104

NONPASCAL routine directive, 97

NOSOURCE compiler option, 81

O

object
  code
    input to xlp command, 20
    optimization, 37, 69
    xlp command, 13
  compiler listing section, 78
  listing, 36

online help, 9

open options
  DDNAME, 48, 67
  DISP=MOD, 47, 66
  INTERACTIVE, 47, 50
  LRECL, 47, 49
  NAME, 46, 67
  overview, 45
  RECFM, 47, 49
  UCASE, 47

opening files
- appending data to a file, 66
- for input, 50, 52
- for output
  - REWRITE, 51, 66
  - TERMOUT, 52
- for updating, 53
- input procedures, 50, 52
- output procedures, 51, 52
- overview, 45
- procedures, 50, 53

OPT compiler option, 37, 69

optimization, 37, 69, 70

optimized code, debugging, 71

OPTION compiler option, 37

options
- –S, 30
- compiler
  - –c, 13, 29
  - –p, 16, 30
  - –pg, 16, 30
  - –qlanglvl, 18
  - –qprime, 21
  - –qprimeout, 21
  - default, 25
  - detailed descriptions, 29–40
  - LANGLVL, 10
  - listing section, 77
  - overview, 22
  - SOURCE, 105
  - specifying in the source file, 23
  - specifying on the command line, 23
  - xlp command, 9
- configuration file attribute, 16
- opening files, 45

osuffix configuration file attribute, 16

output files, 20

OUTPUT predefined file, 104

output procedures
- REWRITE, 51, 66
- TERMOUT, 52

P

packed sets, 128

PAGE procedure, 59

parameter
- linkage
  - descriptor word, 96
  - first parameter word, 96
  - overview, 95
  - second parameter word, 96
- passing
  - improving program performance, 72
  - interlanguage conventions, 97, 101
  - modes, 98, 101
  - routines, 117
- word, 96

Pascal
- array storage, 93
- arrays of CHAR, 92
- character variable types, 92
- language, 1, 9
- matching data types, 90
- pointers, 93
- routine calls, 94
- source files
  - input to xlp command, 13, 19
  - suffix, 16
- strings, 92

pointer data types, 127

pointer target, 66

pointers, 97

portability, 9

predefined
- constants, 117
- data types, 124

PRIME compiler option, 37

prime file, 21

PRIMEOUT compiler option, 38

problem determination
- compiler listings, 73
- error messages, 79
- message errors, 86
- traceback facilities, 85
- using the symbolic debugger, 85

procedure calls, 95

procedures, opening files
- overview, 50
- RESET, 50
- REWRITE, 51, 66
- TERMIN, 52
- TERMOUT, 52
- UPDATE, 53

processing
- a record file
  - end–of–file condition, 64
  - EOF function, 64
  - GET procedure, 61
  - PUT procedure, 62
  - READ procedure, 62
  - SEEK procedure, 63
  - WRITE procedure, 63
- a TEXT file
  - COLS function, 60
  - end–of–file condition, 60
  - end–of–line condition, 59
  - EOF function, 60
  - EOLN function, 59
  - GET procedure, 53
  - PAGE procedure, 59

PUT procedure, 54
READ procedure, 55
READLN procedure, 57
WRITE procedure, 57
WRITELN procedure, 58

prof command, 30

profiling, runtime, 30

proflibs configuration file attribute, 16

psuffix configuration file attribute, 16

PTR4 compiler option, 38

publications, 6

PUT procedure, 54, 62

Q

QUERYHEAP procedure, 130

QUIET compiler option, 38

R

random record access, 63

READ procedure, 55, 62

reading data
from a record file, 61, 62
from a TEXT file
GET procedure, 53
READ procedure, 55
READLN procedure, 57

READLN procedure, 57

reassociation, 70

RECFM file opening option, 47, 49

record data types, 125

record field, 66

REF, variables, 89

REF variables, 42

related documentation, 6

relative record access, 63

RELEASE procedure, 131

reserved words, 2, 7

RESET procedure
DDNAME, 66
opening a file for input, 50
opening files, 45

REWRITE procedure
appending data to a file, 66
DDNAME, 66
opening files, 45, 51

RISC System/6000, 1

rounding mode, 38

routine calls, 94

routine linkage convention
formal parameter, 95
NONPASCAL routine directive, 97
overview, 95
parameter linkage, 95

routines, parameter passing, 117

row–major order, 93

RRM compiler option, 38

RT PC VS Pascal, 9

running programs, 41

runtime
checking errors, 84
environment, 41, 86
messages, 83
profile, 30

S

scalar type parameters, 72

second parameter word, 96

SEEK procedure, 63

semantic errors, 83

SET data type, 118, 127

setenv command, 15

setting environment variables, 14

sh command, 14

shared libaries, 42

short–circuiting, 71

single precision, intermediate rounding
frsp error, 133
IEEE compiler option, 36
XFLAG compiler option, 38

single–letter flags, 24

source code symbol conventions, 10

SOURCE compiler option
compile–time errors, 81
description, 38
sample program, 105

source files, 13, 23

source section, compiler listing, 77

SPACE data types, 129

specifying compiler options, 23

SPILLSIZE compiler option, 38

ssuffix configuration file attribute, 16

standard mode
See also VS mode
ANSI standard Pascal, 5
compiler option, 10

DDNAME=UNIQUE, 32
file name association, 67
language support, 9

standard Pascal, 5, 111

standards, industry. *See* industry standards

static linking, 41

static variables, storage, 123

storage
anonymous types, 129
ARRAY data types, 126
arrays, 93
boundary alignment, 124
classes, 123
data types, 124
dynamic, 129–132
enumerated data types, 124
FILE data types, 127
packed sets, 128
pointer data types, 127
predefined data types, 124
record data types, 125
SET data types, 127
SPACE data types, 129
string data types, 129
subrange scalar data types, 125
unpacked sets, 127
variables, 123

straightening, 70

strength reduction, 70

STRICT compiler option, 38

STRING data type, 92

string data types, 129

string subscript out of bounds error, 84

string truncation error, 84

structured type parameters, 72

subheaps, 130

subrange error, 84

subrange scalar data types, 125

subscript error, 84

suppressing data type checking, 104

symbolic debugger
dbx command, 9
optimized code, 71
problem determination, 85

syntax diagrams, how to read, 2

syntax errors, 81

T

TERMIN procedure, 45, 52

TERMOUT procedure, 45, 52

traceback facilities, 85

TRACEID compiler option, 39

TUNE compiler option, 39

type matching. *See* matching data types

typographical conventions, 7

U

UCASE file opening option, 47

unique file names, 65, 67

unpacked sets, 127

UPDATE procedure
DDNAME, 66
description, 53
opening files, 45

use configuration file attribute, 16

USEHEAP procedure, 130

V

validity checking, 131

VALUE initialization, 72

value numbering, 70

VAR parameter, 93

variables
environment, 14
in syntax diagrams, 2
maximum size, 119
storage, 123

varying–length strings, 92

VS mode
compiler option, 10
DDNAME=COMPAT, 32, 67
DDNAME=UNIQUE, 67
extensions to standard Pascal, 111
language support, 9

VS Pascal
compatibility, 9
differences from XL Pascal, 119–121
LANGLVL compiler option, 36
language extensions, 1
migration, 9
omissions from XL Pascal, 120, 122

W

WAIT compiler option, 39

word, parameter. *See* parameter word

WRITE compiler option, 40

WRITE procedure, 57, 63

WRITELN procedure, 58

writing data
  to a record file, 62, 63
  to a TEXT file
    PUT procedure, 54
    WRITE procedure, 57
    WRITELN procedure, 58

X

XFLAG compiler option, 38, 133

XL Pascal
  compiler, 1, 9
  language, 1, 9
  prime files, 22
  runtime environment, 41, 86

xl__trap procedure, 85

xlp
  command
    calling run–time routines, 41
    invoking, 13, 40
    online help, 9
    output files, 20
  configuration file attribute, 17

xlp_prime file, 21, 37

xlpopt configuration file attribute, 17

XREF compiler option, 40

Communicating Your Comments to IBM

IBM AIX XL Pascal Compiler/6000
User's Guide

Version 2.1

Publication number SC09–1756–00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage–paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416–448–6161
  - Other countries: (+1)–416–448–6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: **torrcf@vnet.ibm.com**
  - IBMLINK: **toribm(torrcf)**
  - IBM/PROFS: **torolab4(torrcf)**
  - IBMMAIL: **ibmmail(caibmwt9)**

Readers' Comments – We'd Like to Hear from You

IBM AIX XL Pascal Compiler/6000
User's Guide

Version 2.1

Publication Number SC09–1756–00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall Satisfaction	☐	☐	☐	☐	☐

How satisfied were you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone Number

Readers' Comments—We'd Like to Hear from You
SC09–1756–00

Fold and Tape | **Please Do Not Staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IIBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Cut or Fold Along Line

Fold and Tape | **Please Do Not Staple** | Fold and Tape

SC09–1756–00